Refactoring: Ejercicio 3

Integrantes:

- Manuel Tierno (21340/6)
- Matias Ramos Giacosa (21318/9)

Formato:

- **Bad Smell encontrado**
- Fracción de código que contiene el Bad Smell
- **Refactoring aplicado**
- Fracción de código tras aplicar el Refactoring

Secuencia de Refactoring:

Feature Envy
Desde una instancia de la clase Empresa se le piden datos a una instancia de la clase GestorNumerosDisponibles para realizar operaciones

```
public boolean agregarNumeroTelefono(String str) {  7 usages
    boolean encontre = guia.getLineas().contains(str);
    if (!encontre) {
        guia.getLineas().add(str);
        encontre= true;
        return encontre;
    }
    else {
        encontre= false;
        return encontre;
    }
}
```

Move Method hacia la clase GestorNumerosDisponibles

```
public boolean agregarNumeroTelefono(String str) {  7 usages
    boolean encontre = getLineas().contains(str);
    if (!encontre) {
        getLineas().add(str);
        encontre= true;
        return encontre;
    }
    else {
        encontre= false;
        return encontre;
    }
}
```

Switch Statement
Large method
El método registrarUsuario es largo y tiene un condicional que pregunta por tipos

```
public Cliente registrarUsuario(String data, String nombre, String tipo) {  5 usages
    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;
}
```

Replace Conditional with Polymorphism: Subclases de Cliente, ClienteJuridico y ClienteFisico y creación de constructores
Decompose Conditional: Separación del método registrarUsuario en metodos especificos para cada tipo de cliente

```
public ClienteJuridico registrarClienteJuridico(String nombre, String cuit) {
    String tel = this.obtenerNumeroLibre();
    ClienteJuridico var = new ClienteJuridico(nombre, tel, cuit);
    clientes.add(var);
    return var;
}

public ClienteFisico registrarClienteFisico(String nombre, String dni) {  no usa
    String tel = this.obtenerNumeroLibre();
    ClienteFisico var = new ClienteFisico(nombre, tel, dni);
    clientes.add(var);
    return var;
}
```

```java
public abstract class Cliente {  13 usages  2 inheritors  Matute *
    public List<Llamada> llamadas = new ArrayList<~>();  2 usages
    private String nombre;  3 usages
    private String numeroTelefono;  3 usages

    public Cliente(String nombre, String numeroTelefono) {  2 usages  new *
        this.nombre = nombre;
        this.numeroTelefono = numeroTelefono;
    }

    public abstract String getTipo();  2 usages  2 implementations  new *

    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; }
}
```

```java
public class ClienteFisico extends Cliente {  3 usages  new *
    private String dni;  1 usage

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

    public String getTipo(){  2 usages  new *
        return "fisica";
    }
}
```

```java
public class ClienteJuridico extends Cliente {  3 usages  new *
    private String cuit;  1 usage

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

    public String getTipo(){  2 usages  new *
        return "juridica";
    }
}
```

Switch Statement

Se aplica un condicional para preguntar por el tipo de cliente

```
public double calcularMontoTotalLlamadas(Cliente cliente) {  4 usages  Matute
    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;
}
```

Replace conditional with Polymorphism: Se eliminan las constantes de descuento y se le delega el cálculo a Cliente

```
public double calcularMontoTotalLlamadas(Cliente cliente) {  4 usages  Matute *
    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;
        }

        auxc = cliente.calcularDescuento(auxc);

        c += auxc;
    }
    return c;
}
```

```
@Override  1 usage  new *
public Double calcularDescuento(Double costo) {
    return costo;
}
```

```java
@Override  1 usage  new *
public Double calcularDescuento(Double costo) {
    return costo * 0.85;
}
```

Switch Statement

```java
public double calcularMontoTotalLlamadas(Cliente cliente) {  4 usages  Matute *
    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;
        }

        auxc = cliente.calcularDescuento(auxc);

        c += auxc;
    }
    return c;
}
```

```java
public Llamada registrarLlamada(Cliente origen, Cliente destino, String t, int duracion) {  8 usages  Matute
    Llamada llamada = new Llamada(t, origen.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
    llamadas.add(llamada);
    origen.llamadas.add(llamada);
    return llamada;
}
```

Replace Conditional with Polymorphism: Se crea una jerarquia de Llamada con LlamaNacional y LlamadaInternacional y se delega el caculo del costo a las mismas. Se separa el registro de llamada en los tipos de llamada específicos

```java
public double calcularMontoTotalLlamadas(Cliente cliente) {  4 usages
    double c = 0;
    for (Llamada l : cliente.llamadas) {
        double auxc = 0;
        auxc += l.calcularCosto();
        auxc = cliente.calcularDescuento(auxc);
        c += auxc;
    }
    return c;
}
```

```java
public abstract class Llamada {  7 usages  2 inheritors  Matute *  1 related problem
    private String origen;  2 usages
    private String destino;  2 usages
    private int duracion;  2 usages

    public Llamada(String origen, String destino, int duracion) {  2 usages  Matute *
        this.origen= origen;
        this.destino= destino;
        this.duracion = duracion;
    }

    public abstract double calcularCosto();  1 usage  2 implementations  new *

    public String getRemitente() {  no usages  Matute
        return destino;
    }

    public int getDuracion() {  4 usages  Matute
        return this.duracion;
    }

    public String getOrigen() {  no usages  Matute
        return origen;
    }
}
```

```java
public class LlamadaInternacional extends Llamada{  3 usages  new *

    public LlamadaInternacional(String origen, String destino, int duracion) {  1 usage  new *
        super(origen, destino, duracion);
    }

    public double calcularCosto(){  1 usage  new *
        return this.getDuracion() * 150 + this.getDuracion() * 150 * 0.21 + 50;
    }
}
```

```java
public class LlamadaNacional extends Llamada {  3 usages  new *

    public LlamadaNacional(String origen, String destino, int duracion) {  1 usage  new *
        super(origen, destino, duracion);
    }

    public double calcularCosto(){  1 usage  new *
        return this.getDuracion() * 3 + this.getDuracion() * 3 * 0.21;}

}
```

Duplicate Code

```
public double calcularCosto(){  1 usage  new *
    return this.getDuracion() * 150 + this.getDuracion() * 150 * 0.21 + 50;
}
```

```
public double calcularCosto(){  1 usage  new *
    return this.getDuracion() * 3 + this.getDuracion() * 3 * 0.21;}
```

Pull Up Method: Se generaliza el cálculo del costo de la llamada en la superclase Llamada se crea el método getCostoPorSegundo

```
public double calcularCosto() {  3 usages  2 overrides  new *
    return this.getDuracion() * this.getCostoPorSegundo() + this.getDuracion() * this.getCostoPorSegundo() * 0.21;

}

public abstract double getCostoPorSegundo();  2 usages  2 implementations  new *

public String getOrigen() {  no usages  new *
    return origen;
}
```

```
@Override  3 usages  new *
public double calcularCosto(){
    return super.calcularCosto() + 50;
}

@Override  2 usages  new *
public double getCostoPorSegundo(){
    return 150;
}
}
```

```
@Override  3 usages  new *
public double calcularCosto(){
    return super.calcularCosto();
}

@Override  2 usages  new *
public double getCostoPorSegundo(){
    return 3;
}

}
```

Dead Code

```
public class Empresa {  2 usages   Matute *
    private List<Cliente> clientes = new ArrayList<~>();  4 usages
    private List<Llamada> llamadas = new ArrayList<~>();  2 usages
    GestorNumerosDisponibles guia = new GestorNumerosDisponibles();  9 usages
```

Remove Dead Code: La lista de llamadas solo tiene operaciones en las que se le agregan llamadas, no interviene en ninguna otra operación

```
public class Empresa {  2 usages   Matute *
    private List<Cliente> clientes = new ArrayList<~>();  4 usages
    GestorNumerosDisponibles guia = new GestorNumerosDisponibles();  9 usages
```

Romper encapsulamiento

```
public abstract class Cliente {  15 usages  2 inheritors   Matute *
    public List<Llamada> llamadas = new ArrayList<~>();  3 usages
    private String nombre;  3 usages
    private String numeroTelefono;  3 usages
```

Encapsulate Field

```
public abstract class Cliente {  15 usages  2 inheritors   Matute *
    private List<Llamada> llamadas = new ArrayList<Llamada>();
    private String nombre;  3 usages
    private String numeroTelefono;  3 usages
```

Feature Envy

```
public double calcularMontoTotalLlamadas(Cliente cliente) {
    double c = 0;
    for (Llamada l : cliente.getLlamadas()) {
        double auxc = 0;
        auxc += l.calcularCosto();
        auxc = cliente.calcularDescuento(auxc);
        c += auxc;
    }
    return c;
}
```

Move Method a clase Cliente

```
public double calcularMontoTotalLlamadas() {  4 usages  new *
    double c = 0;
    for (Llamada l : getLlamadas()) {
        double auxc = 0;
        auxc += l.calcularCosto();
        auxc = calcularDescuento(auxc);
        c += auxc;
    }
    return c;
}
```

Reinventar la rueda

```
public double calcularMontoTotalLlamadas() {  4 usages  new *
    double c = 0;
    for (Llamada l : getLlamadas()) {
        double auxc = 0;
        auxc += l.calcularCosto();
        auxc = calcularDescuento(auxc);
        c += auxc;
    }
    return c;
}
```

Substitute Algorithm usando la API streams para reemplazar la estructura for each

```
public double calcularMontoTotalLlamadas() {  4 usages  new *
    return getLlamadas().stream().mapToDouble(l -> this.calcularDescuento(l.calcularCosto())).sum();
}
```

Nombre no descriptivo

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

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

Rename Method

```
public int cantidadDeClientes() { return clientes.size(); }

public boolean existeCliente(Cliente persona) { return clientes.contains(persona); }
```

Nombre no descriptivo

```
public class GestorNumerosDisponibles {  3 usages  Matute
    private SortedSet<String> lineas = new TreeSet<~>();  8 usages
    private String tipoGenerador = "ultimo";  2 usages
```

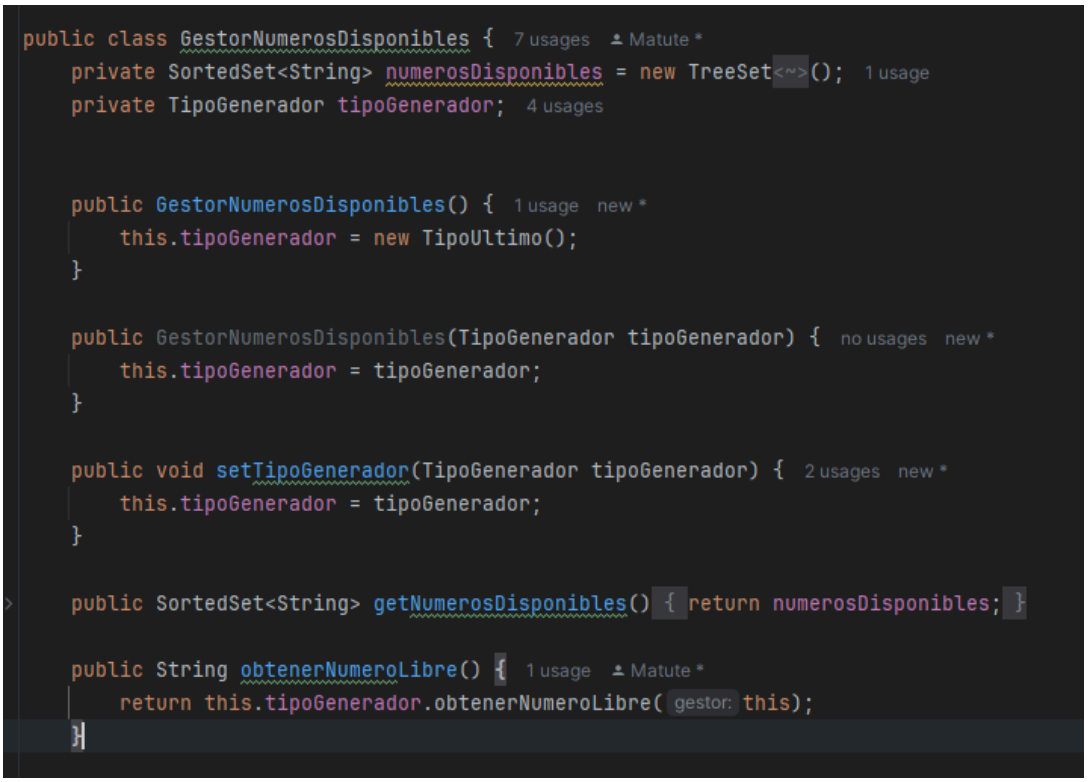
Rename Field

```
public class GestorNumerosDisponibles {  8 usages  Matute
    private SortedSet<String> numerosDisponibles = new TreeSet<~>();  1 us
```

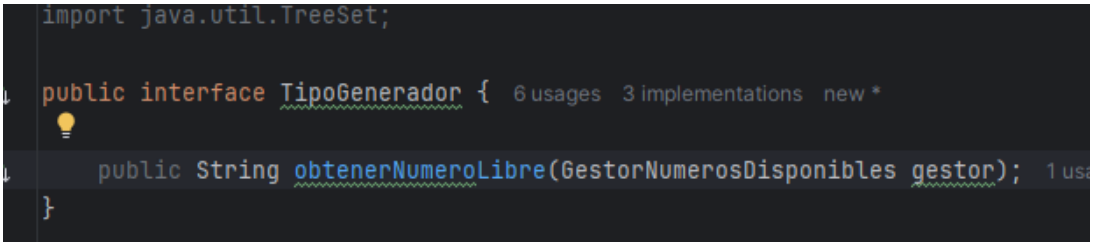
Switch Sentence

```
    public String obtenerNumeroLibre() {  1 usage  Matute
        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;
    }
```

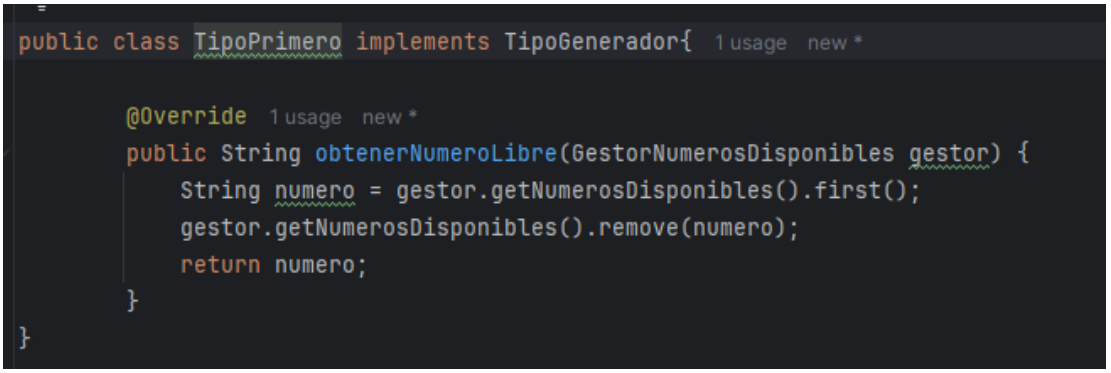
Replace Conditional with Polymorphism: Se descompone el condicional en una jerarquía de tipos de generadores utilizando el patrón strategy y definiendo el atributo tipoGenerador como uno de los generadores especificos



```
public class GestorNumerosDisponibles {  7 usages  Matute *
    private SortedSet<String> numerosDisponibles = new TreeSet<~>();  1 usage
    private TipoGenerador tipoGenerador;  4 usages


    public GestorNumerosDisponibles() {  1 usage  new *
        this.tipoGenerador = new TipoUltimo();
    }

    public GestorNumerosDisponibles(TipoGenerador tipoGenerador) {  no usages  new *
        this.tipoGenerador = tipoGenerador;
    }

    public void setTipoGenerador(TipoGenerador tipoGenerador) {  2 usages  new *
        this.tipoGenerador = tipoGenerador;
    }

    public SortedSet<String> getNumerosDisponibles() { return numerosDisponibles; }

    public String obtenerNumeroLibre() {  1 usage  Matute *
        return this.tipoGenerador.obtenerNumeroLibre( gestor: this);
    }
```



```
import java.util.TreeSet;

public interface TipoGenerador {  6 usages  3 implementations  new *

    public String obtenerNumeroLibre(GestorNumerosDisponibles gestor);  1 us
}
```



```
public class TipoPrimero implements TipoGenerador{  1 usage  new *

        @Override  1 usage  new *
        public String obtenerNumeroLibre(GestorNumerosDisponibles gestor) {
            String numero = gestor.getNumerosDisponibles().first();
            gestor.getNumerosDisponibles().remove(numero);
            return numero;
        }
}
```

```java
public class TipoRandom implements TipoGenerador{  1 usage  new *

        @Override  1 usage  new *
        public String obtenerNumeroLibre(GestorNumerosDisponibles gestor) {
            String numero = new ArrayList<String>(gestor.getNumerosDisponibles())
                    .get(new Random().nextInt(gestor.getNumerosDisponibles().size()));
            gestor.getNumerosDisponibles().remove(numero);
            return numero;
        }
}
```

```java
public class TipoUltimo implements TipoGenerador{  1 usage  new *

    @Override  1 usage  new *
    public String obtenerNumeroLibre(GestorNumerosDisponibles gestor) {
            String numero = gestor.getNumerosDisponibles().last();
            gestor.getNumerosDisponibles().remove(numero);
            return numero;
    }
}
```

Duplicate Code

```java
import java.util.TreeSet;

public interface TipoGenerador {  6 usages  3 implementations  new *

    public String obtenerNumeroLibre(GestorNumerosDisponibles gestor);  1 us
}
```

```java
public class TipoPrimero implements TipoGenerador{  1 usage  new *

        @Override  1 usage  new *
        public String obtenerNumeroLibre(GestorNumerosDisponibles gestor) {
            String numero = gestor.getNumerosDisponibles().first();
            gestor.getNumerosDisponibles().remove(numero);
            return numero;
        }
}
```

```java
public class TipoRandom implements TipoGenerador{  1 usage  new *

        @Override  1 usage  new *
        public String obtenerNumeroLibre(GestorNumerosDisponibles gestor) {
            String numero = new ArrayList<String>(gestor.getNumerosDisponibles())
                    .get(new Random().nextInt(gestor.getNumerosDisponibles().size()));
            gestor.getNumerosDisponibles().remove(numero);
            return numero;
        }
}
```

```java
public class TipoUltimo implements TipoGenerador{  1 usage  new *

    @Override  1 usage  new *
    public String obtenerNumeroLibre(GestorNumerosDisponibles gestor) {
            String numero = gestor.getNumerosDisponibles().last();
            gestor.getNumerosDisponibles().remove(numero);
            return numero;
    }
}
```

Pull Up Method de la eliminación del número seleccionado a la interfaz TipoGenerador, convirtiéndola en una clase abstracta

```java
public abstract class TipoGenerador {  6 usages  3 inheritors  new *

    public abstract String obtenerNumeroLibre(GestorNumerosDisponibles gestor);  1 usage

    public void eliminarNumeroLibre(GestorNumerosDisponibles gestor, String numero){  3
        gestor.getNumerosDisponibles().remove(numero);
    }

}
```

```java
public class TipoPrimero extends TipoGenerador{  1 usage  new *

        @Override  1 usage  new *
        public String obtenerNumeroLibre(GestorNumerosDisponibles gestor) {
            String numero = gestor.getNumerosDisponibles().first();
            super.eliminarNumeroLibre(gestor, numero);
            return numero;
        }
}
```

```java
public class TipoRandom extends TipoGenerador{ 1 usage  new *

        @Override 1 usage  new *
        public String obtenerNumeroLibre(GestorNumerosDisponibles gestor) {
            String numero = new ArrayList<String>(gestor.getNumerosDisponibles())
                    .get(new Random().nextInt(gestor.getNumerosDisponibles().size()));
            super.eliminarNumeroLibre(gestor, numero);
            return numero;
        }
}
```

```java
public class TipoUltimo extends TipoGenerador{ 1 usage  new *

    @Override 1 usage  new *
    public String obtenerNumeroLibre(GestorNumerosDisponibles gestor) {
            String numero = gestor.getNumerosDisponibles().last();
            super.eliminarNumeroLibre(gestor, numero);
            return numero;
    }
}
```

Reinventar la rueda

```java
public boolean agregarNumeroTelefono(String str) { 7 usages  Matute
    boolean encontre = getNumerosDisponibles().contains(str);
    if (!encontre) {
        getNumerosDisponibles().add(str);
        encontre= true;
        return encontre;
    }
    else {
        encontre= false;
        return encontre;
    }
}
```

Substitute Algorithm

```
public boolean agregarNumeroTelefono(String str) {  7 usages
    return getNumerosDisponibles().add(str);
}
```

Romper encapsulamiento

```
public LlamadaNacional registrarLlamadaNacional(Cliente origen, Cliente destino, int duracion) {  4 usages  Matute *
    LlamadaNacional llamada = new LlamadaNacional(origen.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
    origen.getLlamadas().add(llamada);
    return llamada;
}

public LlamadaInternacional registrarLlamadaInternacional(Cliente origen, Cliente destino, int duracion) {  4 usages  new *
    LlamadaInternacional llamada = new LlamadaInternacional(origen.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
    origen.getLlamadas().add(llamada);
    return llamada;
}
```

Encapsulate Collection: Se establece un método para agregar una llamada al cliente

```
public LlamadaNacional registrarLlamadaNacional(Cliente origen, Cliente destino, int duracion) {  4 usages  Matute *
    LlamadaNacional llamada = new LlamadaNacional(origen.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
    origen.agregarLlamada(llamada);
    return llamada;
}

public LlamadaInternacional registrarLlamadaInternacional(Cliente origen, Cliente destino, int duracion) {  4 usages  new *
    LlamadaInternacional llamada = new LlamadaInternacional(origen.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
    origen.agregarLlamada(llamada);
    return llamada;
}
```

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

Nombres no descriptivos

```
public abstract class Llamada {  6 usages  2 inheritors  Matute
    private String origen;  3 usages
    private String destino;  3 usages
    private int duracion;  3 usages
```

Rename Field

```
public abstract class Llamada {  6 usages  2 inheritors
    private String clienteOrigen;  3 usages
    private String clienteDestino;  3 usages
    private int duracion;  3 usages
```