Ejercicio 5

NOTA:

Tengo entendido que tenemos que aplicar refactorings pero SIN MODIFICAR LOS TESTS PROVISTOS.

Me estuve rompiendo la cabeza para hacer este ejercicio, igualmente hay implementaciones que no terminan de ser del todo prolijas, como por ejemplo recibir en el test por parametro un String del tipo de clase.

Literalmente hice lo que podia hacer SIN MODIFICAR LOS TESTS, por lo menos asi lo entendi yo.

Aplicando Refactorings:

(1) Refactoring:

Mal olor: Se esta preguntando en condicionales por "tipos"

```
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 que lo corrige: Replace conditional with Polymorphism
Refactoring aplicado:

```
public class Llamada_Nacional extends Llamada {

}

public class Llamada_Internacional extends Llamada {

}
```

```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;
}

}
```

(2) Refactoring:

Mal olor: Envidia de atributos. Calcular el precio de una llamada en especifico es responsabilidad de la clase abstracta Llamada.

```java
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 que lo corrige: Extract Method:

Refactoring aplicado:

```
public double calcularMontoTotalLlamadas(Cliente cliente) {
double c = 0;
for (Llamada l : cliente.llamadas) {
double auxc = 0;
auxc = l.CalcularPrecio();

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

```
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 abstract double CalcularPrecio();
public String getRemitente() {
return destino;
}

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

public String getOrigen() {
return origen;
}

}
```

```java
public class Llamada_Nacional extends Llamada {
public Llamada_Nacional(String origen, String destino, int duracion) {
super(origen, destino, duracion);
}

@Override
public double CalcularPrecio() {
// 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);
}

}
```

```java
public class Llamada_Internacional extends Llamada {
public Llamada_Internacional(String origen, String destino, int duracion) {
super(origen, destino, duracion);
// TODO Auto-generated constructor stub
}
@Override
public double CalcularPrecio() {
// 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;
}
}
```

(3) Refactoring

Mal olor: Se esta preguntando en condicionales por "tipos"

```java
public double calcularMontoTotalLlamadas(Cliente cliente) {
    double c = 0;
    for (Llamada l : cliente.llamadas) {
        double auxc = 0;

        auxc = l.CalcularPrecio();

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

Refactoring que lo corrige: Replace Conditional With Polymorphism
Refactoring aplicado:

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

    private String nombre;
    private String numeroTelefono;
    private String cuit;
    private String dni;

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

```
public class ClienteFisico extends Cliente {
```

```
public class ClienteJuridico extends Cliente {
```

(4) Refactoring:
Mal olor: Inappropriate Intimacy, Empresa esta accediendo directamente a los descuentos en vez de delegar esa responsabilidad. La clase empresa no tiene por que almacenar los descuentos, solamente deberia llamar a un metodo get donde cada subclase de cliente implemente el descuento.

```
public class Empresa extends Cliente{
    private List<Cliente> clientes = new ArrayList<Cliente>();
    private List<Llamada> llamadas = new ArrayList<Llamada>();
    private GestorNumerosDisponibles guia = new GestorNumerosDisponibles();

    static double descuentoJur = 0.15;
    static double descuentoFis = 0;
```

Refactoring a aplicar: Replace Data value with Method

Refactoring aplicado:

Clase Cliente:

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

    public abstract double getPorcentajeDescuento();
}
```

Clase empresa:

```
  public double calcularMontoTotalLlamadas(Cliente cliente) {
      double c = 0;
      for (Llamada l : cliente.llamadas) {
          double auxc = 0;

          auxc = l.CalcularPrecio();
          auxc -= auxc* cliente.getPorcentajeDescuento();
          c += auxc;
      }
      return c;
  }
```

Clase cliente Fisico:

```
@Override
public double getPorcentajeDescuento() {
    return 0;
}
```

Clase cliente juridico:

```
public class ClienteJuridico extends Cliente {
    @Override
    public double getPorcentajeDescuento() {
        return 0.15;
    }
}
```

(5) Refactoring:

Mal olor: El metodo Registrar Llamada esta decidiendo que tipo de llamada crear en base a un parametro String t.

Refactoring: Introduce Builder
Creamos un llamadaBuilder, el cual recibe un string y decide la instancia concreta (Llamada nacional o llamda internacional)

Refactoring aplicado:

```
public class LlamadaBuilder {

    public static Llamada crearLlamada(String tipo, Cliente origen, Cliente destino, int duracion) {
        switch (tipo.toLowerCase()) {
            case "nacional":
                return new Llamada_Nacional(origen.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
            case "internacional":
                return new Llamada_Internacional(origen.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
            default:
                throw new IllegalArgumentException("Tipo de llamada no soportado: " + tipo);
        }
    }
}
```

```
public Llamada registrarLlamada(Cliente origen, Cliente destino, String t, int duracion) {
    Llamada llamada =  LlamadaBuilder.crearLlamada(t, origen, destino, duracion);
    llamadas.add(llamada);
    origen.llamadas.add(llamada);
    return llamada;
}
```

(6) Refactoring:
Mal olor: El metodo Registrar Usuario esta decidiendo que tipo de Usuario crear en base a un parametro String tipo.

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

Refactoring a aplicar: Introduce Builder
Creamos un usuarioBuilder el cual recibe un string y decide la instancia concreta de cliente (ClienteFisico o ClienteJuridico)

```java
public class usuarioBuilder {

    public static Cliente crearUsuario (String data, String nombre, String tipo)
    {
        switch (tipo.toLowerCase()) {
        case "juridica":
            return new ClienteJuridico(data,nombre);
        case "fisica":
            return new ClienteFisico(data,nombre);
        default:
            throw new IllegalArgumentException("Tipo de llamada no soportado: " + tipo);
    }
    }
}
```

```java
  public Cliente registrarUsuario(String data, String nombre, String tipo) {
      Cliente var = usuarioBuilder.crearUsuario(data, nombre, tipo);
      clientes.add(var);
      return var;
  }
```

(7) Refactoring

Mal olor: Condicional para seleccionar un comportamiento especifico.

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

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

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

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

Refactoring a aplicar: Replace condicional with strategy pattern.

Refactoring aplicado:

NOTA: Aca me saco de encima el condicional para el comportamiento, pero voy a tener otro condicional para ver de que strategy se crea la instancia (se podria plantear de otra manera pero los test me obligan a implementarlo asi, a no ser que cree un builder de un strategy que ya me parece una banda....)

```java
import java.util.TreeSet;

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

    public GestorNumerosDisponibles ()
    {
        this.strategyGenerador = new GeneradorUltimo();
    }

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

    public String obtenerNumeroLibre() {
        return this.strategyGenerador.obtenerNumero(lineas);
    }

    public void cambiarTipoGenerador(String valor) {
        if (valor.equals("primero"))
        {
            this.strategyGenerador = new GeneradorPrimero();
        }
        else if (valor.equals("ultimo"))
        {
            this.strategyGenerador = new GeneradorUltimo();
        }
        else if (valor.equals("random"))
        {
            this.strategyGenerador = new GeneradorRandom();
        }

    }
}
```

```java
// Strategy
public interface GeneradorLinea {
    String obtenerNumero(SortedSet<String> lineas);

}
```

```java
public class GeneradorPrimero implements GeneradorLinea{

    @Override
    public String obtenerNumero(SortedSet<String> lineas) {
            String linea = lineas.first();
            lineas.remove(linea);
            return linea;
    }

}
```

```java
public class GeneradorRandom implements GeneradorLinea {

    @Override
    public String obtenerNumero(SortedSet<String> lineas) {
        String linea = new ArrayList<>(lineas).get(new Random().nextInt(lineas.size()));
        lineas.remove(linea);
        return linea;
    }

}
```

```java
public class GeneradorUltimo implements GeneradorLinea {

    @Override
    public String obtenerNumero(SortedSet<String> lineas) {
            String linea = lineas.last();
             lineas.remove(linea);
             return linea;
    }

}
```

(8) Refactoring

Mal olor: No reinventar la rueda

```java
public double calcularMontoTotalLlamadas(Cliente cliente) {
    double c = 0;
    for (Llamada l : cliente.llamadas) {
        double auxc = 0;

        auxc = l.CalcularPrecio();
        auxc -= auxc* cliente.getPorcentajeDescuento();
        c += auxc;
    }
    return c;
}
```

Refactoring a aplicar: Replace Loop with Stream

Refactoring aplicado:

```
public double calcularMontoTotalLlamadas(Cliente cliente) {
    return cliente.llamadas.stream()
            .mapToDouble(llamada -> llamada.CalcularPrecio() * cliente.getPorcentajeDescuento())
            .sum();
}
```