Refactoring: Ejercicio 3

Integrantes: Ivo Milicchio y Mateo Carusotti

Mal olor: Switch Statements

En el método `obtenerNumeroLibre()` de la clase `GestorNumerosDisponibles` tenemos una estructura de control switch para distintos valores que podría tomar la variable de instancia `tipoGenerador`. Esto genera como consecuencia que el método sea largo, poco legible y complejo para mantener.

Notar que también encontramos el mal olor **Primitive Obsession** el cual es generado por la declaración de `tipoGenerador` como una variable de instancia de tipo `String` en lugar de utilizar clases que encapsulen el comportamiento de cada tipo de generador

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

Refactoring aplicado: Replace Type Code with Strategy

Aplicamos el patrón strategy a la clase `GeneradorNumerosDisponibles`.Para esto creamos una nueva interfaz llamado `Generador` con la firma `obtenerNumeroLibre()` y tres nuevas clases concretas: `GeneradorPrimero`, `GeneradorUltimo` y `GeneradorRandom` las cuales implementarán la interfaz `Generador` que habíamos creado anteriormente. Con estos cambios aplicados ahora debemos cambiar el tipo de la variable `generador` (antes llamada `tipoGenerador`) a `Generador` en lugar de `String`. Del mismo modo debemos modificar el método `cambiarTipoGenerador()` el cual pasará a recibir como parámetro un objeto de tipo `Generador` en lugar de un `String`.

Nota: aplicamos **rename field** cambiando el nombre de la variable de instancia `tipoGenerador` por `generador` ya que lo consideramos más adecuado

```
public class GestorNumerosDisponibles {
    ...
    private Generador generador= new GeneradorUltimo();
    ...
    public GestorNumerosDisponibles() {

        this.generador = new GeneradorUltimo();

    }

    public String obtenerNumeroLibre() {
        String linea = this.generador.obtenerNumeroLibre(this.lineas);
        this.lineas.remove(linea);
        return linea;
```

```
    }
    public void cambiarTipoGenerador(Generador generador) {
        this.generador = generador;
    }
}
```

```
public interface Generador {


    public String obtenerNumeroLibre(SortedSet<String> lineas);


}
```

```
public class GeneradorUltimo implements Generador{


    public GeneradorUltimo() {};

    public String obtenerNumeroLibre(SortedSet<String> lineas) {
        return lineas.last();

    }
}
```

```
public class GeneradorPrimero implements Generador{


    public GeneradorPrimero() {}

    public String obtenerNumeroLibre(SortedSet<String> lineas) {
        return lineas.first();
    }
}
```

```
public class GeneradorRandom implements Generador{


    public GeneradorRandom() {}

    public String obtenerNumeroLibre(SortedSet<String> lineas) {
        return new ArrayList<String>(lineas).get(new Random().nextInt(lineas.size()));
    }

}
```

Test luego del refactoring:

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

        Generador primero = new GeneradorPrimero();

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

        Generador random = new GeneradorRandom();
```

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

Mal olor: Declaración de Atributo Público

La clase `Cliente` tiene una variable de instancia `llamadas` la cual está declarada como pública. Las variables públicas constituyen un mal olor ya que rompen con el encapsulamiento de las clases que las contienen

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

Refactoring aplicado: Encapsulate collection

El refactoring mencionado es sencillo de aplicar, únicamente tenemos que cambiar el tipo de la variable `llamadas` a privado. Además, dado que la clase `Empresa` utiliza la variable de instancia `llamadas` de la clase `Cliente`, debemos agregar los métodos `getLlamadas()` y `addLlamada()` a la clase mencionada para que la clase `Empresa` pueda seguir realizando las mismas funciones que realizaba antes con la variable de instancia `llamadas`

```
public class Cliente {
    private List<Llamada> llamadas = new ArrayList<Llamada>();
    ...
    public List<Llamada> getLlamadas(){
        return this.llamadas;
    }

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

Test luego del refactoring:

No se hizo ningún cambio en los tests

Mal olor: Feature Envy

La clase `Empresa` tiene declarado el método `agregarNumeroTelefono()` el cual invoca dos veces al método `getLineas()` perteneciente a la clase `GestorNumerosDisponibles`. Es decir, la clase `Empresa` utiliza más datos de la clase `GestorNumerosDisponibles` que datos propios en la implementación de este método. Esto nos da un indicio de que tenemos una **responsabilidad mal asignada**

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

Refactoring aplicado: Move Method

El refactoring que aplicamos unicamente consiste en mover el método `agregarNumeroTelefono()` de la clase `Empresa` a la clase `GestorNumerosDisponibles`

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

Test luego del refactoring:

```
@BeforeEach
    public void setUp() {
        this.sistema = new Empresa();
        this.sistema.getGestorNumeros().agregarNumeroTelefono("2214444554");
        this.sistema.getGestorNumeros().agregarNumeroTelefono("2214444555");
        this.sistema.getGestorNumeros().agregarNumeroTelefono("2214444556");
        this.sistema.getGestorNumeros().agregarNumeroTelefono("2214444557");
        this.sistema.getGestorNumeros().agregarNumeroTelefono("2214444558");
        this.sistema.getGestorNumeros().agregarNumeroTelefono("2214444559");
    }
...

@Test
    void testAgregarUsuario() {
        ...
        this.sistema.getGestorNumeros().agregarNumeroTelefono("2214444558");
        ...
    }
```

Mal olor: Reinventar la Rueda

La clase `GestorNumerosDisponibles` tiene declarado el método `agregarNumeroTelefono()` el cual recibe como parámetro un `String`, verifica que el mismo no se encuentre en un `Set` y si es así, lo agrega a esa misma colección. El problema acá es que en una colección de tipo `Set` no pueden haber elementos duplicados, por lo tanto, nos podríamos abstraer de realizar esa verificación (ya que ya la hace el set internamente). Además, el método `add()` de la clase `Set` nos devuelve un valor booleano que nos indica si el elemento ya se encontraba en la colección y fue añadido (`true`), o no (`false`) por lo que también podríamos eliminar a la variable `encontre` utilizada como control flag

```
public class GestorNumerosDisponibles {
...
    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 aplicado: **Remove Control Flag**

Aplicando este refactoring directamente devolvemos el valor booleano que nos retorna el método `add()` del set y eliminamos la control flag `encontre`

```
public boolean agregarNumeroTelefono(String str) {

        return getLineas().add(str);
}
```

Test luego del refactoring:

No se hizo ningún cambio en los test

Mal olor: Switch Statements

En la clase `Empresa` aun tenemos a los métodos `calcularMontoTotalLlamadas()` y `registrarUsuario()` con switch statements (no son switch statements particularmente, pero si el día de mañana se desean agregar otros tipos de llamadas y/o otros tipos de clientes si lo serían). Esto es producido en gran parte por la ausencia de polimorfismo en la clase `Cliente`. Es decir, en lugar de utilizar clases para cada tipo de cliente que encapsulen su comportamiento, se utiliza una variable de tipo `String` que almacena su tipo (**Primitive Obsession**)

```
public class Empresa {

    ...
    static double descuentoJur = 0.15;
    static double descuentoFis = 0;
    ...
    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 double calcularMontoTotalLlamadas(Cliente cliente) {
        double c = 0;
        for (Llamada l : cliente.getLlamadas()) {
            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;
    }
}
```

```
public class Cliente {
    ...
    private String tipo;
    ...
    private String cuit;
    private String dni;

    public String getTipo() {
        return tipo;
    }
    public void setTipo(String tipo) {
        this.tipo = tipo;
    }
    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;
    }
}
```

Refactoring aplicado: Replace Conditional with Polymorphism

Aplicamos este refactoring aplicando herencia en la clase `Cliente` (la cual ahora pasa a ser una clase abstracta) haciendo que de esta hereden `ClienteFisico` y `ClienteJuridico` . La variable de instancia `dni` y `cuit` de `Cliente` ya no están más. Ahora `ClienteFisico` posee `dni` y `ClienteJuridico` `cuit`, ambos con sus respectivos setters y getters. Los descuentos para cada tipo de cliente ahora ya no son variables estáticas de la clase `Empresa` , sino que son valores que retornan los respectivos `getDescuento()` de cada tipo de cliente.

Dado que ahora no podemos instanciar un objeto de la clase `Cliente` (ya que es abstracta) debemos tener dos métodos para el registro de usuarios pertenecientes a las clases concretas `ClienteJuridico` y `ClienteFisico` : `registrarUsuarioFisico()` y `registrarUsuarioJuridico()`

```
public class Empresa {

    ...


    public ClienteFisico registrarUsuarioFisico(String dni, String nombre) {
        ClienteFisico cliente = new ClienteFisico();
        cliente.setNombre(nombre);
        String tel = this.obtenerNumeroLibre();
        cliente.setNumeroTelefono(tel);
        cliente.setDNI(dni);
        clientes.add(cliente);
        return cliente;

    }

    public ClienteJuridico registrarUsuarioJuridico(String cuit, String nombre) {
        ClienteJuridico cliente = new ClienteJuridico();
        cliente.setNombre(nombre);
        String tel = this.obtenerNumeroLibre();
        cliente.setNumeroTelefono(tel);
```

```
        cliente.setCuit(cuit);
        clientes.add(cliente);
        return cliente;

    }

    ...

    public double calcularMontoTotalLlamadas(Cliente cliente) {
        double c = 0;
        for (Llamada l : cliente.getLlamadas()) {
            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 -= auxc*cliente.getDescuento();

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

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


}
```

```
public class ClienteJuridico extends Cliente{
    private String cuit;


    public ClienteJuridico(){

    }

    public String getCuit() {
        return cuit;
    }
```

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

    public double getDescuento() {

        return 0.15;
    }
}
```

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

    public ClienteFisico(){

    }

    public String getDNI() {
        return dni;
    }

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

    public double getDescuento() {

        return 0;
    }
}
```

Test luego del refactoring:

```
@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() {
        ...
        Cliente nuevaPersona = this.sistema.registrarUsuarioFisico("2444555","Alan Turing");
        ...
    }
```

Mal olor: Switch Statements

Nuevamente seguimos teniendo switch statements en el método `clacularMontoTotalLlamadas()` de la clase `Empresa`. En este caso es en gran parte porque la clase `Llamada` no aplica polimorfismo para sus distintos tipos. Es decir, en lugar de utilizar clases para cada tipo de llamada que encapsulen su comportamiento, se utiliza una variable de tipo `String` que almacena su tipo (**Primitive Obsession**). Además, podemos ver que el método hace uso de **magic numbers** (valores hardcodeados), lo que hace al código menos legible y más complejo de mantener

```
public class Empresa {

    public double calcularMontoTotalLlamadas(Cliente cliente) {
        double c = 0;
```

```
        for (Llamada l : cliente.getLlamadas()) {
            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 -= auxc*cliente.getDescuento();

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

Refactoring aplicado: Replace Conditional with Polymorphism y Replace Magic Number with Method

Para aplicar este primer refactoring modificamos la clase `Llamada` para que ahora pase a ser abstracta y creamos dos clases concretas que extiendan de esta: `LlamadaNacional` y `LlamadaInternacional`, cada una con su constructor la cual hace uso del constructor de `Llamada`. Además, con la aplicación del segundo refctoring eliminamos los magic numbers que contenía el método `calcularMontoTotalLlamadas()` agregando en la clase Llamada los métodos `getPrecio()` (abstracto), `getAdicional()` (abstracto) y `getIva()` (concreto)

Nota: el refactoring Replace Magic Number with Method no existe como tal

```
public class Empresa {
    ...

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

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

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

            auxc += l.getDuracion() * l.getPrecio() +
        (l.getDuracion() * l.getPrecio() l.getIva()) + l.getAdicional();

            auxc -= auxc*cliente.getDescuento();

            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 getPrecio();

    public abstract double getAdicional();

    public double getIva() {
        return 0.21;
    }

    ...
}
```

```
public class LlamadaNacional extends Llamada{


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

    public double getPrecio() {
        return 3;
    }

    public double getAdicional() {
        return 0;
    }

}
```

```
public class LlamadaInternacional extends Llamada{


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


    public double getPrecio() {
        return 150;
    }

    public double getAdicional() {
        return 50;
    }

}
```

Test luego del refactoring:

```
@Test
    void testcalcularMontoTotalLlamadas() {
        ...
```

```
        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);

        ...
    }
```

Mal olor: Feature Envy

El método `calcularMontoTotalLlamadas()` de la clase `Empresa` utiliza varios atributos de las clases `Cliente` y `Llamada` , esto nos da un indicio de que la responsabilidad de este método está mal asignada. Dado que es la clase `Cliente` la que contiene a sus llamadas, resulta más lógico y natural que sea él quien calcule el monto total de sus llamadas

```
public class Empresa {

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

            auxc += l.getDuracion() * l.getPrecio() +
            (l.getDuracion() * l.getPrecio() * l.getIva()) + l.getAdicional();


            auxc -= auxc*cliente.getDescuento();

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

Refactoring aplicado: Extract Method

Se movió una porción del código del método `calcularMontoTotalLlamadas()` de `Empresa` hacía la clase `Cliente` creando un método en esta clase llamado `calcularLlamadas()` , al código solo hizo falta cambiarle la referencia de `cliente` por `this`

Nota: la aplicación de este Extract Method también implica la aplicación de un Move Method ya que se extrae una porción del método `calcularMontoTotalLlamadas()` pero además esa porción se mueve a la clase `Cliente`

```
public class Empresa {
    ...

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

    ...
}
```

```
public abstract class Cliente {
    ...


    public double calcularLlamadas() {
        double c = 0;
        for (Llamada l : this.getLlamadas()) {
```

```
            double auxc = 0;

            auxc += l.getDuracion() * l.getPrecio()
            + (l.getDuracion() * l.getPrecio() * l.getIva()) + l.getAdicional();


            auxc -= auxc*this.getDescuento();

            c += auxc;
        }
        return c;
    }

    ...
}
```

Test luego del refactoring:

No se hizo ningún cambio en los tests

Mal olor: Feature Envy

El método `calcularLlamadas()` de la clase `Cliente` utiliza muchos atributos de la clase `Llamada`. Nuevamente volvemos a tener una clase que realiza una funcionalidad que no le corresponde. En este caso lo más lógico sería que la clase `Llamada` calcule su costo en lugar de que lo haga la clase `Cliente`

```
public class Cliente {
    ...
    public double calcularLlamadas() {
        double c = 0;
        for (Llamada l : this.getLlamadas()) {
            double auxc = 0;

            auxc += l.getDuracion() * l.getPrecio()
            + (l.getDuracion() * l.getPrecio() * l.getIva()) + l.getAdicional();


            auxc -= auxc*this.getDescuento();

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

Refactoring Aplicado: Extract Method

La línea dentro del método `calcularLlamadas()` que sumaba el costo de una llamada se extrajo y se movió a la clase `Llamada` dentro de un nuevo método `calcularCosto()`.

Por consecuencia, al crear el método `calcularCosto()` en la clase `Llamada` aplicamos un refactoring del tipo **Form Template Method** debido a que aunque solo cambien en el valor a retornar, los métodos `getPrecio()` y `getAdicional()` de las subclases de `Llamada` son diferentes.

```
public abstract class Cliente {

    ...

    public double calcularLlamadas() {
        double c = 0;
        for (Llamada l : this.getLlamadas()) {
            double auxc = 0;
```

```
            auxc += l.calcularCosto();


            auxc -= auxc*this.getDescuento();

            c += auxc;
        }
        return c;
    }

    ...
}
```

```
public abstract class Llamada {
    ...



    public double calcularCosto() {
        return this.getDuracion() * this.getPrecio()
        + (this.getDuracion() * this.getPrecio() * this.getIva()) + this.getAdicional();
    }

    ...


}
```

Test luego del refactoring:

No se hizo ningún cambio en los tests

Mal olor: Código Duplicado

En la clase `Llamada` el método `calcularCosto()` que habíamos definido anteriormente tiene código duplicado (se realiza `this.getDuracion() * this.getPrecio()` dos veces). Además, el hecho de tener tantas operaciones en un solo método hace al código menos legible y largo

Refactoring aplicado: Form Template Method

Hacemos que el método `calcularCosto()` llame a 2 nuevos métodos privados que definimos, `calcularMontoNeto()` y `calcularMontoIva()`. De esta forma se generalizan los pasos del método `calcularCosto()` , pero luego cada clase concreta implementa los pasos a su manera, aunque sea solo se diferencien por su valor de retorno. Como resultado nos queda un código más legible y sin duplicación

```
public abstract class Llamada {

    ...

    public double calcularCosto() {
        return this.calcularMontoNeto() + this.calcularMontoIva() + this.getAdicional();
    }

    private double calcularMontoNeto() {
        return this.getDuracion() * this.getPrecio();
    }

    private double calcularMontoIva() {
        return this.calcularMontoNeto() * this.getIva();
    }

    ...
```

```
}
```

Test luego del refactoring:

No se hizo ningún cambio en los tests

Mal olor: Long Method

Si bien `calcularLlamadas()` no es un método muy largo, dentro del mismo podemos ver que se realizan dos tareas: por un lado se calcula la sumatoria de los costos de todas las llamadas y por otro se calcula el descuento para cada llamada. Podríamos extraer la porción de código que calcula el descuento de la llamada (valor - descuento) de manera que `calcularLlamadas()` se abstraiga de hacer ese cálculo

```
public class Cliente {
...
    public double calcularLlamadas() {
        double c = 0;
        for (Llamada l : this.getLlamadas()) {
            double auxc = 0;

            auxc += l.calcularCosto();


            auxc -= auxc*this.getDescuento();

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

Refactoring aplicado: Extract Method

Definimos un nuevo método `calcularDescuento(Double costo)` que nos retornará el precio de la llamada con su descuento aplicado.

Nota: aplicar el descuento para el costo de cada llamada y luego sumarlos es lo mismo que aplicar el descuento sobre la sumatoria de los costos de todas las llamadas. Cambiamos esto ya que de esta manera el método `calcularDescuento()` se invocaría una única vez dentro del método `calcularLlamadas()`

```
public abstract class Cliente {


    ...

    public double calcularLlamadas() {
        double c = 0;
        for (Llamada l : this.getLlamadas()) {
            double auxc = 0;

            auxc += l.calcularCosto();
            c += auxc;
        }
        return this.calcularDescuento(c);
    }

    private double calcularDescuento(Double costo) {
        return costo - (costo * this.getDescuento());
    }

    ...

}
```

Test luego del refactoring:

No se hizo ningún cambio en los tests

Mal olor: Reinventar la Rueda

En el método `calcularLlamadas()` de la clase `Cliente` se utiliza un bucle for en el cual se realiza la sumatoria de los costos correspondientes a todas las llamadas del cliente. Nos podríamos abstraer de implementar esto y hacer uso de los pipelines que nos brinda Java

Refactoring aplicado: Replace Loop with Pipeline

En lugar de utilizar una estructura de control for utilizamos `stream()` con el cual calculamos el costo de todas las llamadas de un cliente en particular, luego se le aplica el descuento a este resultado.

```
public class Cliente {
    ...

    public double calcularLlamadas() {
        double costo = this.llamadas.stream().mapToDouble(l -> l.calcularCosto()).sum();
        return this.calcularDescuento(costo);
    }



    private double calcularDescuento(Double costo) {
        return costo - (costo * this.getDescuento());
    }
    ...
}
```

Test luego del refactoring:

No se hizo ningún cambio en los tests

Aspectos de diseño

Uso de constructores

Si bien no hay ningún refactoring que su implementación implique el uso de constructores, ofrecerlos es una buena práctica de la Programación Orientada a Objetos. Además, al tener un constructor en la clase `Cliente` (y en sus respectivas subclases), los métodos `registrarUsuarioFisico()` y `registrarUsuarioJuridico()` de la clase `Empresa` se vuelven más cortos y simples al no tener que utilizar los setters de las clases correspondientes a los objetos que se quieren instanciar

Eliminación de relación redundante

Resulta redundante que la clase `Empresa` conozca a la clase `Llamada` siendo que `Empresa` ya conoce a `Cliente` (el cual conoce a `Llamada`). Por lo tanto, decidimos eliminar esta relación para una simplificación del diseño

Luego de aplicar los cambios de diseño:

La clase `Empresa` ya no poseé la variable de instancia `lineas` (es decir, no conoce más a la clase `Llamada` de forma directa) , este conocimiento ahora solo está en la clase `Cliente`. También se redujeron las lineas de los métodos encargados de registrar un `Usuario` y una `Llamada` debido al uso de constructores, el uso de estos permite también preservar el encapsulamiento dejando de lado la utilización de setters.

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


 ...

    public ClienteFisico registrarUsuarioFisico(
        String dni, String nombre) {
        ClienteFisico cliente = new ClienteFisico(
        nombre,this.obtenerNumeroLibre(),dni);
```

```
        clientes.add(cliente);
        return cliente;

    }

    public ClienteJuridico registrarUsuarioJuridico(
        String cuit, String nombre) {
        ClienteJuridico cliente = new ClienteJuridico(
        nombre,this.obtenerNumeroLibre(),cuit);
        clientes.add(cliente);
        return cliente;

    }

    public LlamadaNacional registrarLlamadaNacional(
        Cliente origen, Cliente destino, int duracion) {
        LlamadaNacional llamada = new LlamadaNacional(
        origen.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
        origen.addLlamada(llamada);
        return llamada;
    }

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

    ...
}
```

```
public abstract class Cliente {
    ...

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

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

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

    ...
}
```

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

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

```
    ...
}
```

Test luego de los cambios:

No se hizo ningún cambio en los tests

Mal olor: Dead Code

En las clases `Cliente` y `Llamada` encontramos métodos que no se utilizan (getters y setters). El hecho de tener código que no se utiliza complejiza a la clase que lo contiene y genera confusión entre los desarrolladores

```
public class Llamada {
    ...
    public String getRemitente() {
        return destino;
    }
    ...
    public String getOrigen() {
        return origen;
    }
    ...
}
```

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

Refactoring aplicado: Remove Dead Code

El refactoring aplicado únicamente consiste en la eliminación de los métodos que no se utilizaban. Más precisamente se eliminaron: `getRemitente()` y `getOrigen()` de la clase `Llamada` y `getNombre()`, `setNombre()` y `setNumeroTelefono()` de la clase `Cliente`

```
public class Cliente {

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

    public double calcularLlamadas() {
        double costo = this.llamadas.stream().mapToDouble(l -> l.calcularCosto()).sum();
        return this.calcularDescuento(costo);
    }

    private double calcularDescuento(Double costo) {
        return costo - (costo * this.getDescuento());
    }

    public String getNumeroTelefono() {
        return numeroTelefono;
    }

    public abstract double getDescuento();
```

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

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

```
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 double calcularCosto() {
        return this.calcularMontoNeto() + this.calcularMontoIva() + this.getAdicional();
    }

    private double calcularMontoNeto() {
        return this.getDuracion() * this.getPrecio();
    }

    private double calcularMontoIva() {
        return this.calcularMontoNeto() * this.getIva();
    }

    public abstract double getPrecio();

    public abstract double getAdicional();

    public double getIva() {
        return 0.21;
    }

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

Test luego del refactoring:

No se hizo ningún cambio en los tests

Aspectos de Diseño:

Consideramos como una buena práctica cambiar nombres de variables o parámetros los cuales sean poco descriptivos. En este caso en particular la clase `GestorNumerosDisponibles` tiene al método `agregarNumeroTelefono()` que recibe como parámetro un variable de tipo `String` llamada `str`. El nombre es poco descriptivo por lo que consideramos más adecuado cambiarlo por `numero`

```
public class GestorNumerosDisponibles {

    ...

    public boolean agregarNumeroTelefono(String str) {

        return getLineas().add(str);
```

```
    }
}
```

Código luego del cambio:

```
public class GestorNumerosDisponibles {

    ...

    public boolean agregarNumeroTelefono(String numero) {

        return getLineas().add(numero);
    }
}
```