
ORIENTACIÓN A OBJETOS 2

EJERCICIO 3: FACTURACIÓN DE LLAMADAS

Alumnos: Gonzalez, Joaquín Manuel y Felder, León.



Introducción.....	6
Code Smell: Switch Statements en clase GestorNumerosDisponibles.....	6
Refactoring: Aplicar Replace Conditional Logic with Strategy.....	7
Pasos a Seguir para llevar a cabo el Refactoring:.....	7
Resultados luego de aplicar todos los pasos.....	7
Clase Empresa.....	7
Clase GestorNumerosDisponibles.....	8
Clase EstrategiaNumeroLibre (Strategy).....	8
Clase EstrategiaPrimerNumeroLibre (Concrete Strategy).....	8
Clase EstrategiaRandomNumeroLibre (Concrete Strategy).....	9
Clase EstrategiaUltimoNumeroLibre (Concrete Strategy).....	9
Test EmpresaTest.....	9
Antes.....	9
Después.....	10
Code Smell: Feature Envy.....	10
Refactoring: Aplicar Extract Method.....	11
Pasos a Seguir para llevar a cabo el Refactoring:.....	11
Resultados luego de aplicar todos los pasos.....	11
Clase GestorNumerosDisponibles.....	11
Clase EstrategiaPrimerNumeroLibre (Concrete Strategy).....	12
Clase EstrategiaRandomNumeroLibre (Concrete Strategy).....	12
Clase EstrategiaUltimoNumeroLibre (Concrete Strategy).....	13
Code Smell: Speculative Generality.....	13
Refactoring: Aplicar Inline Temp.....	14
Pasos a Seguir para llevar a cabo el Refactoring:.....	14
Resultados luego de aplicar todos los pasos.....	14
Clase EstrategiaPrimerNumeroLibre (Concrete Strategy).....	15
Clase EstrategiaRandomNumeroLibre (Concrete Strategy).....	15
Clase EstrategiaUltimoNumeroLibre (Concrete Strategy).....	15
Code Smell: Non-Descriptive Names.....	15
Refactoring: Rename Variable.....	16
Pasos a Seguir para llevar a cabo el Refactoring:.....	16
Resultados luego de aplicar todos los pasos.....	16
Code Smell: Feature Envy.....	16
Refactoring: Move Method.....	17
Pasos a Seguir para llevar a cabo el Refactoring:.....	17

Resultados luego de aplicar todos los pasos.....	17
Clase Empresa.....	17
Clase GestorNumerosDisponibles.....	18
Code Smell: Non-Descriptive Names.....	18
Refactoring: Rename Variable.....	19
Pasos a Seguir para llevar a cabo el Refactoring:.....	19
Resultados luego de aplicar todos los pasos.....	19
Clase Empresa.....	19
Clase GestorNumerosDisponibles.....	20
Code Smell: Long Method.....	20
Refactoring: Replace Temp with Query.....	21
Pasos a Seguir para llevar a cabo el Refactoring:.....	21
Resultados luego de aplicar todos los pasos.....	21
Code Smell: Data Class.....	21
Refactoring: Encapsulate Collection y Move Method.....	23
Pasos a Seguir para llevar a cabo el Refactoring:.....	23
Resultados luego de aplicar todos los pasos.....	23
Clase Empresa.....	24
Clase Cliente.....	24
Code Smell: No Visibility Declaration.....	26
Refactoring: Encapsulate Field.....	26
Pasos a Seguir para llevar a cabo el Refactoring:.....	26
Resultados luego de aplicar todos los pasos.....	26
Code Smell: Speculative Generality.....	27
Refactoring: Remove Unused Reference.....	27
Pasos a Seguir para llevar a cabo el Refactoring:.....	27
Resultados luego de aplicar todos los pasos.....	27
Code Smell: Long Method.....	27
Refactoring: Extract Method.....	27
Pasos a Seguir para llevar a cabo el Refactoring:.....	28
Resultados luego de aplicar todos los pasos.....	28
Code Smell: Data Class.....	28
Refactoring: Move Method.....	29
Pasos a Seguir para llevar a cabo el Refactoring:.....	29
Resultados luego de aplicar todos los pasos.....	29
Clase Cliente.....	29

Clase Llamada.....	30
Code Smell: Switch Statements.....	30
Refactoring: Replace Type Code with Subclasses.....	30
Pasos a Seguir para llevar a cabo el Refactoring:.....	30
Resultados luego de aplicar todos los pasos.....	31
Clase LLamada.....	31
Clase LlamadaInternacional.....	32
Clase LlamadaNacional.....	32
Clase Empresa.....	32
Clase Cliente.....	33
Test EmpresaTest.....	33
Antes.....	33
Después.....	34
Code Smell: Duplicated Code.....	34
Refactoring: Form Template Method.....	35
Pasos a Seguir para llevar a cabo el Refactoring:.....	35
Resultados luego de aplicar todos los pasos.....	35
Clase Llamada.....	35
Clase LlamadaInternacional.....	36
Clase LlamadaNacional.....	36
Code Smell: Inconsistent Names.....	36
Refactoring: Rename Method.....	37
Pasos a Seguir para llevar a cabo el Refactoring:.....	37
Resultados luego de aplicar todos los pasos.....	37
Code Smell: Switch Statements.....	37
Refactoring: Replace Type Code with Subclasses.....	38
Pasos a Seguir para llevar a cabo el Refactoring:.....	38
Resultados luego de aplicar todos los pasos.....	39
Clase Cliente.....	40
Clase ClienteJuridico.....	41
Clase ClienteFisico.....	42
Clase Empresa.....	43
Test EmpresaTest.....	44
Antes.....	44
Después.....	45
Code Smell: Non-Descriptive Names.....	45

Refactoring: Rename Method, Rename Parameter, Rename Variable.....	46
Pasos a Seguir para llevar a cabo el Refactoring:.....	46
Resultados luego de aplicar todos los pasos.....	46
Clase Empresa.....	47
Test EmpresaTest.....	48
Antes.....	48
Después.....	49
Code Smell: Inappropriate Intimacy.....	49
Refactoring: Extract Method.....	50
Pasos a Seguir para llevar a cabo el Refactoring:.....	50
Resultados luego de aplicar todos los pasos.....	50
Clase Empresa.....	51
Clase Cliente.....	51
Clase ClienteFisico.....	51
Clase ClienteJuridico.....	52
Code Smell: Long Method.....	52
Refactoring: Extract Method.....	52
Pasos a Seguir para llevar a cabo el Refactoring:.....	52
Resultados luego de aplicar todos los pasos.....	52
Code Smell: Long Method.....	53
Refactoring: Extract Method.....	53
Pasos a Seguir para llevar a cabo el Refactoring:.....	53
Resultados luego de aplicar todos los pasos.....	53
Code Smell: Duplicated Code.....	54
Refactoring: Form Template Method.....	54
Pasos a Seguir para llevar a cabo el Refactoring:.....	55
Resultados luego de aplicar todos los pasos.....	55
Clase Cliente.....	55
Clase ClienteJuridico.....	56
Clase ClienteFisico.....	57
Code Smell: Reinventar la rueda.....	57
Refactoring: Replace Loop with Pipeline.....	58
Pasos a Seguir para llevar a cabo el Refactoring:.....	58
Resultados luego de aplicar todos los pasos.....	58
Code Smell: Non-Descriptive Names.....	58
Refactoring: Rename Method, Rename Parameter.....	59

Pasos a Seguir para llevar a cabo el Refactoring:.....	59
Resultados luego de aplicar todos los pasos.....	59
Clase Empresa.....	59
Test EmpresaTest.....	60
Antes.....	60
Después.....	60
Bibliografía Utilizada:.....	60

Introducción

En este trabajo, se desarrollará la resolución de un ejercicio de Refactoring con el objetivo de transformar un código existente en uno que cumpla con mejores prácticas de programación. Se partirá de un código inicial que, aunque funcional, presenta áreas de mejora que intentaré desarrollar.

Code Smell: Switch Statements

Los malos olores generados por Switch Statements normalmente llevan a la duplicación de código. Una buena solución hacia estos problemas es utilizar la noción de Polimorfismo que nos ofrece la Programación Orientada a Objetos. Buscaremos solucionar el *switch statement* presentado en la clase GestorNumerosDisponibles.

```
● ● ●  
1  public String obtenerNumeroLibre() {  
2      String linea;  
3      switch (tipoGenerador) {  
4          case "ultimo":  
5              linea = lineas.last();  
6              lineas.remove(linea);  
7              return linea;  
8          case "primero":  
9              linea = lineas.first();  
10             lineas.remove(linea);  
11             return linea;  
12         case "random":  
13             linea = new ArrayList<String>(lineas)  
14                 .get(new Random().nextInt(lineas.size()));  
15             lineas.remove(linea);  
16             return linea;  
17     }  
18     return null;  
19 }
```

Refactoring: Aplicar Replace Conditional Logic with Strategy

Pasos a Seguir para llevar a cabo el Refactoring:

- Crear una nueva clase para poder aplicar Strategy.
- Definir una variable de instancia en el contexto para referenciar a la nueva Estrategia, aplicar *Extract Parameter* en el contexto para permitir a los clientes setear la estrategia dentro del constructor del contexto.

-
- Aplicar *Move Method* para mover el método con los switch statements al Strategy y arreglar los parámetros necesarios.
 - Dejar un método en el contexto que delegue con sus parámetros necesarios.
 - Aplicar *Replace Conditional with Polymorphism* en el método del Strategy.
 - Cambiar referencias, parámetros y tipos para que sea posible aplicar los cambios utilizando el Strategy.
 - Modificar *obtenerNumeroLibre()* de EmpresaTest para que utilice objetos correspondientes al Strategy y no Strings.

Resultados luego de aplicar todos los pasos

Clase Empresa

```
● ● ●
1 private GestorNumerosDisponibles guia = new GestorNumerosDisponibles(new EstrategiaUltimoNumeroLibre());
```

Clase GestorNumerosDisponibles

```
● ● ●
1 public class GestorNumerosDisponibles {
2     private SortedSet<String> lineas = new TreeSet<String>();
3     private EstrategiaNumeroLibre tipoGenerador;
4
5     public GestorNumerosDisponibles(EstrategiaNumeroLibre tipoGenerador) {
6         this.tipoGenerador = tipoGenerador;
7     }
8
9     public SortedSet<String> getLineas() {
10         return lineas;
11     }
12
13     public String obtenerNumeroLibre() {
14         return this.tipoGenerador.obtenerNumeroLibre(this.lineas);
15     }
16
17     public void cambiarTipoGenerador(EstrategiaNumeroLibre valor) {
18         this.tipoGenerador = valor;
19     }
20 }
```

Clase EstrategiaNumeroLibre (Strategy)



```
1 public abstract class EstrategiaNumeroLibre {  
2  
3     public abstract String obtenerNumeroLibre(SortedSet<String> lineas);  
4  
5 }
```

Clase EstrategiaPrimerNumeroLibre (Concrete Strategy)



```
1 public class EstrategiaPrimerNumeroLibre extends EstrategiaNumeroLibre {  
2  
3     @Override  
4     public String obtenerNumeroLibre(SortedSet<String> lineas) {  
5         String linea = lineas.first();  
6         lineas.remove(linea);  
7         return linea;  
8     }  
9  
10 }
```

Clase EstrategiaRandomNumeroLibre (Concrete Strategy)



```
1 public class EstrategiaRandomNumeroLibre extends EstrategiaNumeroLibre {  
2  
3     @Override  
4     public String obtenerNumeroLibre(SortedSet<String> lineas) {  
5         String linea = new ArrayList<String>(lineas)  
6             .get(new Random().nextInt(lineas.size()));  
7         lineas.remove(linea);  
8         return linea;  
9     }  
10  
11 }
```

Clase EstrategiaUltimoNumeroLibre (Concrete Strategy)

```
● ● ●  
1 public class EstrategiaUltimoNumeroLibre extends EstrategiaNumeroLibre {  
2  
3     @Override  
4     public String obtenerNumeroLibre(SortedSet<String> lineas) {  
5         String linea = lineas.last();  
6         lineas.remove(linea);  
7         return linea;  
8     }  
9  
10 }
```

Test EmpresaTest

Antes

```
● ● ●  
1 @Test  
2     void obtenerNumeroLibre() {  
3         // por defecto es el ultimo  
4         assertEquals("2214444559", this.sistema.obtenerNumeroLibre());  
5  
6         this.sistema.getGestorNumeros().cambiarTipoGenerador("primero");  
7         assertEquals("2214444554", this.sistema.obtenerNumeroLibre());  
8  
9         this.sistema.getGestorNumeros().cambiarTipoGenerador("random");  
10        assertNotNull(this.sistema.obtenerNumeroLibre());  
11    }
```

Después

```
● ● ●  
1 @Test  
2     void obtenerNumeroLibre() {  
3         // por defecto es el ultimo  
4         assertEquals("2214444559", this.sistema.obtenerNumeroLibre());  
5  
6         this.sistema.getGestorNumeros().cambiarTipoGenerador(new EstrategiaPrimerNumeroLibre());  
7         assertEquals("2214444554", this.sistema.obtenerNumeroLibre());  
8  
9         this.sistema.getGestorNumeros().cambiarTipoGenerador(new EstrategiaRandomNumeroLibre());  
10        assertNotNull(this.sistema.obtenerNumeroLibre());  
11    }
```

Code Smell: Feature Envy

Este Code Smell normalmente se genera cuando se realizan movimientos de código de una clase a otra, en este caso se genera esta situación dentro de las Estrategias concretas (`EstrategiaPrimerNumeroDisponible`, `EstrategiaRandomNumeroDisponible` y `EstrategiaUltimoNumeroDisponible`) ya que ahora se presenta una duplicación de código en dentro de su método `obtenerNumeroLibre(SortedSet<String> lineas)`. Debemos poder eliminar esta duplicación para poder tener un código más simple, corto, legible, fácil de mantener y más organizado.

```
● ● ●
1 public class EstrategiaPrimerNumeroLibre extends EstrategiaNumeroLibre {
2
3     @Override
4     public String obtenerNumeroLibre(SortedSet<String> lineas) {
5         String linea = lineas.first();
6         lineas.remove(linea);
7         return linea;
8     }
9
10 }
```

```
● ● ●
1 public class EstrategiaRandomNumeroLibre extends EstrategiaNumeroLibre {
2
3     @Override
4     public String obtenerNumeroLibre(SortedSet<String> lineas) {
5         String linea = new ArrayList<String>(lineas)
6             .get(new Random().nextInt(lineas.size()));
7         lineas.remove(linea);
8         return linea;
9     }
10
11 }
```



```
1 public class EstrategiaUltimoNumeroLibre extends EstrategiaNumeroLibre {  
2  
3     @Override  
4     public String obtenerNumeroLibre(SortedSet<String> lineas) {  
5         String linea = lineas.last();  
6         lineas.remove(linea);  
7         return linea;  
8     }  
9  
10 }
```

Refactoring: Aplicar Extract Method

Pasos a Seguir para llevar a cabo el Refactoring:

- Aplicar *Extract Method* dentro de las 3 Estrategias Concretas para mover la sentencia `lineas.remove(linea)` a la Clase GestorNumerosDisponibles.

Resultados luego de aplicar todos los pasos

Clase GestorNumerosDisponibles



```
1 public String obtenerNumeroLibre() {  
2     String linea = this.tipoGenerador.obtenerNumeroLibre(this.lineas);  
3     this.lineas.remove(linea);  
4     return linea;  
5 }
```

Clase EstrategiaPrimerNumeroLibre (Concrete Strategy)



```
1 public class EstrategiaPrimerNumeroLibre extends EstrategiaNumeroLibre {  
2  
3     @Override  
4     public String obtenerNumeroLibre(SortedSet<String> lineas) {  
5         String linea = lineas.first();  
6         return linea;  
7     }  
8  
9 }
```

Clase EstrategiaRandomNumeroLibre (Concrete Strategy)



```
1 public class EstrategiaRandomNumeroLibre extends EstrategiaNumeroLibre {  
2  
3     @Override  
4     public String obtenerNumeroLibre(SortedSet<String> lineas) {  
5         String linea = new ArrayList<String>(lineas)  
6             .get(new Random().nextInt(lineas.size()));  
7         return linea;  
8     }  
9  
10 }
```

Clase EstrategiaUltimoNumeroLibre (Concrete Strategy)



```
1 public class EstrategiaUltimoNumeroLibre extends EstrategiaNumeroLibre {  
2  
3     @Override  
4     public String obtenerNumeroLibre(SortedSet<String> lineas) {  
5         String linea = lineas.last();  
6         return linea;  
7     }  
8  
9 }
```

Code Smell: Speculative Generality

Muchas veces mantenemos métodos, variables, atributos, etc. por si en algún futuro le tengamos que dar algún uso, pero esto no es necesario, estos elementos que mantenemos solo hacen que nuestro código sea más extenso y menos legible, por lo tanto se debe solucionar estas situaciones. Esto nos pasa con la variable *linea* que almacena el retorno de un método dentro de las 3 Estrategias Concretas, es una variable que solo se usa para eso y se mantiene en el código de manera ineficiente.

```
● ● ●  
1 public class EstrategiaPrimerNumeroLibre extends EstrategiaNumeroLibre {  
2  
3     @Override  
4     public String obtenerNumeroLibre(SortedSet<String> lineas) {  
5         String linea = lineas.first();  
6         return linea;  
7     }  
8  
9 }
```

```
● ● ●  
1 public class EstrategiaRandomNumeroLibre extends EstrategiaNumeroLibre {  
2  
3     @Override  
4     public String obtenerNumeroLibre(SortedSet<String> lineas) {  
5         String linea = new ArrayList<String>(lineas)  
6             .get(new Random().nextInt(lineas.size()));  
7         return linea;  
8     }  
9  
10 }
```

```
● ● ●  
1 public class EstrategiaUltimoNumeroLibre extends EstrategiaNumeroLibre {  
2  
3     @Override  
4     public String obtenerNumeroLibre(SortedSet<String> lineas) {  
5         String linea = lineas.last();  
6         return linea;  
7     }  
8  
9 }
```

Refactoring: Aplicar Inline Temp

Pasos a Seguir para llevar a cabo el Refactoring:

- Aplicar *Inline Temp* para eliminar la variable *linea* que únicamente almacena el resultado de una llamada a un método.

Resultados luego de aplicar todos los pasos

Clase EstrategiaPrimerNumeroLibre (Concrete Strategy)

```
● ● ●  
1 public class EstrategiaPrimerNumeroLibre extends EstrategiaNumeroLibre {  
2  
3     @Override  
4     public String obtenerNumeroLibre(SortedSet<String> lineas) {  
5         return lineas.first();  
6     }  
7  
8 }
```

Clase EstrategiaRandomNumeroLibre (Concrete Strategy)

```
● ● ●  
1 public class EstrategiaRandomNumeroLibre extends EstrategiaNumeroLibre {  
2  
3     @Override  
4     public String obtenerNumeroLibre(SortedSet<String> lineas) {  
5         return new ArrayList<String>(lineas)  
6             .get(new Random().nextInt(lineas.size()));  
7     }  
8  
9 }  
10 }
```

Clase EstrategiaUltimoNumeroLibre (Concrete Strategy)

```
● ● ●  
1 public class EstrategiaUltimoNumeroLibre extends EstrategiaNumeroLibre {  
2  
3     @Override  
4     public String obtenerNumeroLibre(SortedSet<String> lineas) {  
5         return lineas.last();  
6     }  
7  
8 }
```

Code Smell: Non-Descriptive Names

Es una mala práctica utilizar nombres poco descriptivos para métodos, variables, clases, etc. Necesitamos que el código que escribimos sea legible para las próximas personas que lo vayan a leer, por lo tanto, tenemos que hacer uso de nombres descriptivos para diversos elementos de nuestro código. En este caso dentro de la clase GestorNumerosDisponibles el método *cambiarTipoGenerador* hace uso de un parámetro con nombre *valor* esto lo hace poco descriptivo.



```
1 public void cambiarTipoGenerador(EstrategiaNumeroLibre valor) {  
2     this.tipoGenerador = valor;  
3 }
```

Refactoring: Rename Variable

Pasos a Seguir para llevar a cabo el Refactoring:

- Aplicar *Rename Variable* para modificar el nombre poco descriptivo por uno más adecuado.

Resultados luego de aplicar todos los pasos



```
1 public void cambiarTipoGenerador(EstrategiaNumeroLibre tipoGenerador) {  
2     this.tipoGenerador = tipoGenerador;  
3 }
```

Code Smell: Feature Envy

Este Code Smell también se puede dar cuando un método de una clase accede más a los datos de otro objeto que a los propios. Esto ocurre con el método *agregarNumeroTelefono* de la clase Empresa, ya que dentro de la clase Empresa, se accede a la colección *lineas* y se manipula la misma en un entorno el cuál no es el de la clase que en verdad contiene la colección (clase GestorNumerosDisponibles). Esto genera un acoplamiento de clases que hay que evitar.



```
1 public boolean agregarNumeroTelefono(String str) {  
2     boolean encontre = guia.getLineas().contains(str);  
3     if (!encontre) {  
4         guia.getLineas().add(str);  
5         encontre= true;  
6         return encontre;  
7     }  
8     else {  
9         encontre= false;  
10        return encontre;  
11    }  
12}
```

Refactoring: Move Method

Pasos a Seguir para llevar a cabo el Refactoring:

- Aplicar *Move Method* para mover el método a la clase GestorNumerosDisponibles y en la clase Empresa únicamente dejar un método que se encargue de delegar la operación a la clase correspondiente.

Resultados luego de aplicar todos los pasos

Clase Empresa



```
1 public boolean agregarNumeroTelefono(String str) {  
2     return this.getGestorNumeros().agregarNumeroTelefono(str);  
3 }
```

Clase GestorNumerosDisponibles



```
1 public boolean agregarNumeroTelefono(String str) {
2     boolean encontre = this.getLineas().contains(str);
3     if (!encontre) {
4         this.getLineas().add(str);
5         encontre= true;
6         return encontre;
7     }
8     else {
9         encontre= false;
10    return encontre;
11 }
12 }
```

Code Smell: Non-Descriptive Names

En este caso dentro de la clase Empresa y GestorNumerosDisponibles el método *agregarNumeroTelefono* hace uso de un parámetro con nombre *str* esto lo hace poco descriptivo y por lo tanto, debe ser modificado.



```
1 public boolean agregarNumeroTelefono(String str) {
2     return this.getGestorNumeros().agregarNumeroTelefono(str);
3 }
```



```
1 public boolean agregarNumeroTelefono(String str) {  
2     boolean encontre = this.getLineas().contains(str);  
3     if (!encontre) {  
4         this.getLineas().add(str);  
5         encontre= true;  
6         return encontre;  
7     }  
8     else {  
9         encontre= false;  
10        return encontre;  
11    }  
12}
```

Refactoring: Rename Variable

Pasos a Seguir para llevar a cabo el Refactoring:

- Aplicar *Rename Variable* para modificar el nombre poco descriptivo por uno más adecuado.

Resultados luego de aplicar todos los pasos

Clase Empresa



```
1 public boolean agregarNumeroTelefono(String linea) {  
2     return this.getGestorNumeros().agregarNumeroTelefono(linea);  
3 }
```

Clase GestorNumerosDisponibles

```
1 public boolean agregarNumeroTelefono(String linea) {  
2     boolean encontre = this.getLineas().contains(linea);  
3     if (!encontre) {  
4         this.getLineas().add(linea);  
5         encontre= true;  
6         return encontre;  
7     }  
8     else {  
9         encontre= false;  
10        return encontre;  
11    }  
12}
```

Code Smell: Long Method

Es esencial poder mantener métodos chicos y concisos, que no hagan uso excesivo de parámetros o de variables temporales. Esto se puede ver con el método *agregarNumeroTelefono* en la clase GestorNumerosDisponibles con el uso excesivo e innecesario de la variable temporal *encontre*.

```
1 public boolean agregarNumeroTelefono(String linea) {  
2     boolean encontre = this.getLineas().contains(linea);  
3     if (!encontre) {  
4         this.getLineas().add(linea);  
5         encontre= true;  
6         return encontre;  
7     }  
8     else {  
9         encontre= false;  
10        return encontre;  
11    }  
12}
```

Refactoring: Replace Temp with Query

Pasos a Seguir para llevar a cabo el Refactoring:

- Aplicar *Replace Temp With Query* para extraer el uso de la variable *encontre* que únicamente se encarga de contener el valor de una expresión, convirtiéndose esta última en un método privado *encontreNumeroTelefono*.

Resultados luego de aplicar todos los pasos



```
1 private boolean encuentreNumeroTelefono(String linea) {  
2     return this.getLineas().contains(linea);  
3 }  
4  
5     public boolean agregarNumeroTelefono(String linea) {  
6         if (!this.encontreNumeroTelefono(linea)) {  
7             this.getLineas().add(linea);  
8             return this.encontreNumeroTelefono(linea);  
9         }  
10        else {  
11            return this.encontreNumeroTelefono(linea);  
12        }  
13    }
```

Code Smell: Data Class

Las *Data Class* son clases que únicamente contienen atributos, getters y setters, lo que las vuelve poseedoras de datos, siendo casi seguro que otra clase está actuando como manipuladora minuciosa de la clase *Data Class*. Esto nos ocurre con la clase Cliente y la clase Empresa, Cliente es una *Data Class* y Empresa es la clase que está manipulando a la *Data Class*.



```
1 public class Cliente {
2     public List<Llamada> llamadas = new ArrayList<Llamada>();
3     private String tipo;
4     private String nombre;
5     private String numeroTelefono;
6     private String cuit;
7     private String dni;
8
9     public String getTipo() {
10         return tipo;
11     }
12     public void setTipo(String tipo) {
13         this.tipo = tipo;
14     }
15     public String getNombre() {
16         return nombre;
17     }
18     public void setNombre(String nombre) {
19         this.nombre = nombre;
20     }
21     public String getNumeroTelefono() {
22         return numeroTelefono;
23     }
24     public void setNumeroTelefono(String numeroTelefono) {
25         this.numeroTelefono = numeroTelefono;
26     }
27     public String getCuit() {
28         return cuit;
29     }
30     public void setCuit(String cuit) {
31         this.cuit = cuit;
32     }
33     public String getDNI() {
34         return dni;
35     }
36     public void setDNI(String dni) {
37         this.dni = dni;
38     }
39 }
```

```
1 public Llamada registrarLlamada(Cliente origen, Cliente destino, String t, int duracion) {
2     Llamada llamada = new Llamada(t, origen.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
3     llamadas.add(llamada);
4     origen.llamadas.add(llamada);
5     return llamada;
6 }
7
8 public double calcularMontoTotalLlamadas(Cliente cliente) {
9     double c = 0;
10    for (Llamada l : cliente.llamadas) {
11        double auxc = 0;
12        if (l.getTipoDeLlamada() == "nacional") {
13            // el precio es de 3 pesos por segundo más IVA sin adicional por establecer la llamada
14            auxc += l.getDuracion() * 3 + (l.getDuracion() * 3 * 0.21);
15        } else if (l.getTipoDeLlamada() == "internacional") {
16            // el precio es de 150 pesos por segundo más IVA más 50 pesos por establecer la llamada
17            auxc += l.getDuracion() * 150 + (l.getDuracion() * 150 * 0.21) + 50;
18        }
19
20        if (cliente.getTipo() == "fisica") {
21            auxc -= auxc*descuentoFis;
22        } else if(cliente.getTipo() == "juridica") {
23            auxc -= auxc*descuentoJur;
24        }
25        c += auxc;
26    }
27    return c;
28 }
```

Refactoring: Encapsulate Collection y Move Method

Pasos a Seguir para llevar a cabo el Refactoring:

- Aplicar *Encapsulate Collection* para que la colección *llamadas* de la clase Cliente no sea pública para cualquier otro objeto.
- Aplicar *Move Method* en los métodos *registrarLlamada* y en *calcularMontoTotalLlamadas* de la clase Empresa para asignar correctamente la responsabilidad al Cliente de registrar sus propias llamadas y de calcular el monto total de las mismas. Se deja en Empresa un método para cada uno de estos que delega la operación a la clase correspondiente y retorna los resultados generados. Como las variables *descuentoFis* y *descuentoJur* solo son usadas por el método de *calcularMontoTotalLlamadas*, movemos las mismas también a la clase Cliente.

Resultados luego de aplicar todos los pasos

Clase Empresa



```
1  public Llamada registrarLlamada(Cliente origen, Cliente destino, String t, int duracion) {  
2      return origen.registrarLlamada(destino, t, duracion);  
3  }  
4  
5  public double calcularMontoTotalLlamadas(Cliente cliente) {  
6      return cliente.calcularMontoTotalLlamadas();  
7  }
```

Clase Cliente



```
1  public class Cliente {
2      private List<Llamada> llamadas = new ArrayList<Llamada>();
3      private String tipo;
4      private String nombre;
5      private String numeroTelefono;
6      private String cuit;
7      private String dni;
8      static double descuentoJur = 0.15;
9      static double descuentoFis = 0;
10
11     public String getTipo() {
12         return tipo;
13     }
14     public void setTipo(String tipo) {
15         this.tipo = tipo;
16     }
17     public String getNombre() {
18         return nombre;
19     }
20     public void setNombre(String nombre) {
21         this.nombre = nombre;
22     }
23     public String getNumeroTelefono() {
24         return numeroTelefono;
25     }
26     public void setNumeroTelefono(String numeroTelefono) {
27         this.numeroTelefono = numeroTelefono;
28     }
29     public String getCuit() {
30         return cuit;
31     }
32     public void setCuit(String cuit) {
33         this.cuit = cuit;
34     }
35     public String getDNI() {
36         return dni;
37     }
38     public void setDNI(String dni) {
39         this.dni = dni;
40     }
41
42     public Llamada registrarLlamada(Cliente destino, String t, int duracion) {
43         Llamada llamada = new Llamada(t, this.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
44         this.llamadas.add(llamada);
45         return llamada;
46     }
47
48     public double calcularMontoTotalLlamadas() {
49         double c = 0;
50         for (Llamada l : this.llamadas) {
51             double auxc = 0;
52             if (l.getTipoDeLlamada() == "nacional") {
53                 // el precio es de 3 pesos por segundo más IVA sin adicional por establecer la llamada
54                 auxc += l.getDuracion() * 3 + (l.getDuracion() * 3 * 0.21);
55             } else if (l.getTipoDeLlamada() == "internacional") {
56                 // el precio es de 150 pesos por segundo más IVA más 50 pesos por establecer la llamada
57                 auxc += l.getDuracion() * 150 + (l.getDuracion() * 150 * 0.21) + 50;
58             }
59
60             if (this.getTipo() == "fisica") {
61                 auxc -= auxc*descuentoFis;
62             } else if(this.getTipo() == "juridica") {
63                 auxc -= auxc*descuentoJur;
64             }
65             c += auxc;
66         }
67         return c;
68     }
69 }
70 }
```

Code Smell: No Visibility Declaration

Es necesario indicar la visibilidad de los atributos de una clase, esto hace que el código que escribamos sea más legible y fácil de entender por otras personas. En este caso, la clase Cliente posee dos variables estáticas cuyas visibilidades no están especificadas.



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

Refactoring: Encapsulate Field

Pasos a Seguir para llevar a cabo el Refactoring:

- Aplicar *Encapsulate Field* para proteger a las variables *descuentoJur* y *descuentoFis* de ser accedidas indebidamente. Agregamos getters y setters en el caso de que requieran modificarse. (Suponemos que los descuentos que aplica la empresa a cada tipo de cliente puede variar con el tiempo).

Resultados luego de aplicar todos los pasos



```
1 private static double descuentoJur = 0.15;
2 private static double descuentoFis = 0;
3
4 public double getDescuentoJur() {
5     return Cliente.descuentoJur;
6 }
7 public void setDescuentoJur(double descuentoJur) {
8     Cliente.descuentoJur = descuentoJur;
9 }
10 public double getDescuentoFis() {
11     return Cliente.descuentoFis;
12 }
13 public void setDescuentoFis(double descuentoFis) {
14     Cliente.descuentoFis = descuentoFis;
15 }
```

Code Smell: Speculative Generality

Dentro de la clase Empresa, se está almacenando la colección de llamadas de la misma, pero la clase en cuestión no hace uso de esta colección, por lo tanto no haría falta mantener este atributo ya que actualmente no se necesita.

```
● ● ●  
1 public class Empresa {  
2     private List<Cliente> clientes = new ArrayList<Cliente>();  
3     private List<Llamada> llamadas = new ArrayList<Llamada>();  
4     private GestorNumerosDisponibles guia = new GestorNumerosDisponibles(new EstrategiaUltimoNumeroLibre());
```

Refactoring: Remove Unused Reference

Pasos a Seguir para llevar a cabo el Refactoring:

- Aplicamos *Remove unused reference* para eliminar esta referencia que no tiene uso actualmente.

Resultados luego de aplicar todos los pasos

```
● ● ●  
1 public class Empresa {  
2     private List<Cliente> clientes = new ArrayList<Cliente>();  
3     private GestorNumerosDisponibles guia = new GestorNumerosDisponibles(new EstrategiaUltimoNumeroLibre());
```

Code Smell: Long Method

Dentro de la clase Cliente se puede ver el método *calcularMontoTotalLlamadas* el cual calcula el monto individual de cada llamada y además aplica un descuento, estas operaciones hacen que el método sea largo y no específico, por lo tanto, se debe solucionar esta situación.

```
● ● ●  
1 if (l.getTipoDeLlamada() == "nacional") {  
2     // el precio es de 3 pesos por segundo más IVA sin adicional por establecer la llamada  
3     auxc += l.getDuracion() * 3 + (l.getDuracion() * 3 * 0.21);  
4 } else if (l.getTipoDeLlamada() == "internacional") {  
5     // el precio es de 150 pesos por segundo más IVA más 50 pesos por establecer la llamada  
6     auxc += l.getDuracion() * 150 + (l.getDuracion() * 150 * 0.21) + 50;  
7 }  
8  
9     if (this.getTipo() == "fisica") {  
10         auxc -= auxc*descuentoFis;  
11     } else if(this.getTipo() == "juridica") {  
12         auxc -= auxc*descuentoJur;  
13     }
```

Refactoring: Extract Method

Pasos a Seguir para llevar a cabo el Refactoring:

- Aplicar *Extract Method* en el método *calcularMontoTotalLlamadas* para crear un nuevo método *calcularMontoLlamada* que calcula el monto de una llamada y reemplazamos en el método original por una llamada a este método.
- Aplicar *Extract Method* en el método *calcularMontoTotalLlamadas* para crear un nuevo método *aplicarDescuento* que aplica el descuento correspondiente al valor de la llamada y reemplazamos en el método original por una llamada a este método.

Resultados luego de aplicar todos los pasos

```
● ● ●
1  public double calcularMontoTotalLlamadas() {
2      double c = 0;
3      for (Llamada l : this.llamadas) {
4          double auxc = 0;
5          auxc += this.calcularMontoLlamada(l);
6          auxc -= this.aplicarDescuento(auxc);
7          c += auxc;
8      }
9      return c;
10 }
11
12 public double calcularMontoLlamada(Llamada llamada) {
13     if (llamada.getTipoDeLlamada() == "nacional") {
14         // el precio es de 3 pesos por segundo más IVA sin adicional por establecer la llamada
15         return llamada.getDuracion() * 3 + (llamada.getDuracion() * 3 * 0.21);
16     } else if (llamada.getTipoDeLlamada() == "internacional") {
17         // el precio es de 150 pesos por segundo más IVA más 50 pesos por establecer la llamada
18         return llamada.getDuracion() * 150 + (llamada.getDuracion() * 150 * 0.21) + 50;
19     }
20     return 0;
21 }
22
23 public double aplicarDescuento(double monto) {
24     if (this.getTipo() == "fisica") {
25         return monto*descuentoFis;
26     } else if(this.getTipo() == "juridica") {
27         return monto*descuentoJur;
28     }
29     return 0;
30 }
```

Code Smell: Data Class

La clase Llamada está actuando como una *Data Class*, es decir, no posee ninguna funcionalidad propia y ahora es la clase Cliente la que se está encargando de manipularla excesivamente, realizando el cálculo del valor de una llamada por ella.



```
1 public double calcularMontoLlamada(Llamada llamada) {  
2     if (llamada.getTipoDeLlamada() == "nacional") {  
3         // el precio es de 3 pesos por segundo más IVA sin adicional por establecer la llamada  
4         return llamada.getDuracion() * 3 + (llamada.getDuracion() * 3 * 0.21);  
5     } else if (llamada.getTipoDeLlamada() == "internacional") {  
6         // el precio es de 150 pesos por segundo más IVA más 50 pesos por establecer la llamada  
7         return llamada.getDuracion() * 150 + (llamada.getDuracion() * 150 * 0.21) + 50;  
8     }  
9     return 0;  
10 }
```

Refactoring: Move Method

Pasos a Seguir para llevar a cabo el Refactoring:

- Aplicar *Move Method* para mover el método *calcularMontoLlamada* de la clase Cliente a la clase Llamada, donde la responsabilidad se encuentra correctamente asignada. En Cliente, se modifica el método *calcularMontoTotalLlamadas* para que realice el llamado del método *calcularMontoLlamada* a la clase Llamada en vez de a la clase Cliente.

Resultados luego de aplicar todos los pasos

Clase Cliente



```
1 public double calcularMontoTotalLlamadas() {  
2     double c = 0;  
3     for (Llamada l : this.llamadas) {  
4         double auxc = 0;  
5         auxc += l.calcularMontoLlamada();  
6         auxc -= this.aplicarDescuento(auxc);  
7         c += auxc;  
8     }  
9     return c;  
10 }
```

Clase Llamada



```
1 public double calcularMontoLlamada() {
2     if (this.getTipoDeLlamada() == "nacional") {
3         // el precio es de 3 pesos por segundo más IVA sin adicional por establecer la llamada
4         return this.getDuracion() * 3 + (this.getDuracion() * 3 * 0.21);
5     } else if (this.getTipoDeLlamada() == "internacional") {
6         // el precio es de 150 pesos por segundo más IVA más 50 pesos por establecer la llamada
7         return this.getDuracion() * 150 + (this.getDuracion() * 150 * 0.21) + 50;
8     }
9     return 0;
10 }
```

Code Smell: Switch Statements

La clase Llamada presenta *Switch Statements* que dan a entender que existen tipos de llamadas distintos, por lo tanto, la representación de estos tipos se debe realizar mediante el uso del *polimorfismo* para hacer un buen uso de las políticas de la Programación Orientada a Objetos.



```
1 public double calcularMontoLlamada() {
2     if (this.getTipoDeLlamada() == "nacional") {
3         // el precio es de 3 pesos por segundo más IVA sin adicional por establecer la llamada
4         return this.getDuracion() * 3 + (this.getDuracion() * 3 * 0.21);
5     } else if (this.getTipoDeLlamada() == "internacional") {
6         // el precio es de 150 pesos por segundo más IVA más 50 pesos por establecer la llamada
7         return this.getDuracion() * 150 + (this.getDuracion() * 150 * 0.21) + 50;
8     }
9     return 0;
10 }
```

Refactoring: Replace Type Code with Subclasses

Pasos a Seguir para llevar a cabo el Refactoring:

- Aplicar *Replace Type Code with Subclasses* para crear dos nuevas subclases de Llamada acorde con los tipos que dice. La clase Llamada pasa a ser abstracta ya que una llamada debe conocer su tipo.
- Cambiar todas las referencias al tipo de la llamada para que se hagan referencias a objetos de las nuevas clases creadas.
- Añadir métodos necesarios en las clases Empresa y Cliente para poder registrar llamadas de los tipos correspondientes.
- Aplicar *Replace Conditional with Polymorphism*, usando primero *Extract Method* para extraer el funcionamiento de cada cálculo en un método con el mismo nombre y luego *Push Down Method* para bajarlos a las subclases correspondientes. El método original queda como abstracto.

-
- Se modifican los test para que en vez de enviar de qué tipo es la Llamada como parámetro, se utilice el método correspondiente para registrar ese tipo de Llamada.

Resultados luego de aplicar todos los pasos

Clase LLamada

```
● ● ●

1  public abstract class Llamada {
2      private String origen;
3      private String destino;
4      private int duracion;
5
6      public Llamada(String origen, String destino, int duracion) {
7          this.origen= origen;
8          this.destino= destino;
9          this.duracion = duracion;
10     }
11
12     public String getRemitente() {
13         return destino;
14     }
15
16     public int getDuracion() {
17         return this.duracion;
18     }
19
20     public String getOrigen() {
21         return origen;
22     }
23
24     public abstract double calcularMontoLlamada();
25 }
```

Clase LlamadaInternacional

```
● ● ●  
1 public class LlamadaInternacional extends Llamada {  
2  
3     public LlamadaInternacional(String origen, String destino, int duracion) {  
4         super(origen, destino, duracion);  
5     }  
6  
7     @Override  
8     public double calcularMontoLlamada() {  
9         // el precio es de 150 pesos por segundo más IVA más 50 pesos por establecer la llamada  
10        return this.getDuracion() * 150 + (this.getDuracion() * 150 * 0.21) + 50;  
11    }  
12 }  
13 }
```

Clase LlamadaNacional

```
● ● ●  
1 public class LlamadaNacional extends Llamada {  
2  
3     public LlamadaNacional(String origen, String destino, int duracion) {  
4         super(origen, destino, duracion);  
5     }  
6  
7     @Override  
8     public double calcularMontoLlamada() {  
9         // el precio es de 3 pesos por segundo más IVA sin adicional por establecer la llamada  
10        return this.getDuracion() * 3 + (this.getDuracion() * 3 * 0.21);  
11    }  
12 }  
13 }
```

Clase Empresa

```
● ● ●  
1 public Llamada registrarLlamadaInternacional(Cliente origen, Cliente destino, int duracion) {  
2     return origen.registrarLlamadaInternacional(destino, duracion);  
3 }  
4  
5 public Llamada registrarLlamadaNacional(Cliente origen, Cliente destino, int duracion) {  
6     return origen.registrarLlamadaNacional(destino, duracion);  
7 }
```

Clase Cliente

```
● ● ●  
1 public Llamada registrarLlamadaInternacional(Cliente destino, int duracion) {  
2     Llamada llamada = new LlamadaInternacional(this.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);  
3     this.llamadas.add(llamada);  
4     return llamada;  
5 }  
6  
7 public Llamada registrarLlamadaNacional(Cliente destino, int duracion) {  
8     Llamada llamada = new LlamadaNacional(this.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);  
9     this.llamadas.add(llamada);  
10    return llamada;  
11 }
```

Test EmpresaTest

Antes

```
● ● ●  
1 @Test  
2 void testcalcularMontoTotalLlamadas() {  
3     Cliente emisorPersonaFisca = sistema.registrarUsuario("11555666", "Brendan Eich", "fisica");  
4     Cliente remitentePersonaFisica = sistema.registrarUsuario("00000001", "Doug Lea", "fisica");  
5     Cliente emisorPersonaJuridica = sistema.registrarUsuario("17555222", "Nvidia Corp", "juridica");  
6     Cliente remitentePersonaJuridica = sistema.registrarUsuario("25765432", "Sun Microsystems", "juridica");  
7  
8     this.sistema.registrarLlamada(emisorPersonaJuridica, remitentePersonaFisica, "nacional", 10);  
9     this.sistema.registrarLlamada(emisorPersonaJuridica, remitentePersonaFisica, "internacional", 8);  
10    this.sistema.registrarLlamada(emisorPersonaJuridica, remitentePersonaJuridica, "nacional", 5);  
11    this.sistema.registrarLlamada(emisorPersonaJuridica, remitentePersonaJuridica, "internacional", 7);  
12    this.sistema.registrarLlamada(emisorPersonaFisca, remitentePersonaFisica, "nacional", 15);  
13    this.sistema.registrarLlamada(emisorPersonaFisca, remitentePersonaFisica, "internacional", 45);  
14    this.sistema.registrarLlamada(emisorPersonaFisca, remitentePersonaJuridica, "nacional", 13);  
15    this.sistema.registrarLlamada(emisorPersonaFisca, remitentePersonaJuridica, "internacional", 17);  
16  
17    assertEquals(11454.64, this.sistema.calcularMontoTotalLlamadas(emisorPersonaFisca), 0.01);  
18    assertEquals(2445.40, this.sistema.calcularMontoTotalLlamadas(emisorPersonaJuridica), 0.01);  
19    assertEquals(0, this.sistema.calcularMontoTotalLlamadas(remitentePersonaFisica));  
20    assertEquals(0, this.sistema.calcularMontoTotalLlamadas(remitentePersonaJuridica));  
21 }
```

Después

```
● ● ●  
1  @Test  
2      void testcalcularMontoTotalLlamadas() {  
3          Cliente emisorPersonaFisca = sistema.registrarUsuario("11555666", "Brendan Eich", "fisica");  
4          Cliente remitentePersonaFisica = sistema.registrarUsuario("00000001", "Doug Lea", "fisica");  
5          Cliente emisorPersonaJuridica = sistema.registrarUsuario("17555222", "Nvidia Corp", "juridica");  
6          Cliente remitentePersonaJuridica = sistema.registrarUsuario("25765432", "Sun Microsystems", "juridica");  
7  
8          this.sistema.registrarLlamadaNacional(emisorPersonaJuridica, remitentePersonaFisica, 10);  
9          this.sistema.registrarLlamadaInternacional(emisorPersonaJuridica, remitentePersonaFisica, 8);  
10         this.sistema.registrarLlamadaNacional(emisorPersonaJuridica, remitentePersonaJuridica, 5);  
11         this.sistema.registrarLlamadaInternacional(emisorPersonaJuridica, remitentePersonaJuridica, 7);  
12         this.sistema.registrarLlamadaNacional(emisorPersonaFisca, remitentePersonaFisica, 15);  
13         this.sistema.registrarLlamadaInternacional(emisorPersonaFisca, remitentePersonaFisica, 45);  
14         this.sistema.registrarLlamadaNacional(emisorPersonaFisca, remitentePersonaJuridica, 13);  
15         this.sistema.registrarLlamadaInternacional(emisorPersonaFisca, remitentePersonaJuridica, 17);  
16  
17         assertEquals(11454.64, this.sistema.calcularMontoTotalLlamadas(emisorPersonaFisca), 0.01);  
18         assertEquals(2445.40, this.sistema.calcularMontoTotalLlamadas(emisorPersonaJuridica), 0.01);  
19         assertEquals(0, this.sistema.calcularMontoTotalLlamadas(remitentePersonaFisica));  
20         assertEquals(0, this.sistema.calcularMontoTotalLlamadas(remitentePersonaJuridica));  
21     }  
}
```

Code Smell: Duplicated Code

Las clases LlamadaInternacional y LlamadaNacional mantienen un comportamiento casi idéntico a la hora de calcular el monto de una llamada, esto genera una duplicación de código entre las dos clases que mediante una mejor organización puede desaparecer.

```
● ● ●  
1  public class LlamadaInternacional extends Llamada {  
2  
3      public LlamadaInternacional(String origen, String destino, int duracion) {  
4          super(origen, destino, duracion);  
5      }  
6  
7      @Override  
8      public double calcularMontoLlamada() {  
9          // el precio es de 150 pesos por segundo más IVA más 50 pesos por establecer la llamada  
10         return this.getDuracion() * 150 + (this.getDuracion() * 150 * 0.21) + 50;  
11     }  
12 }  
13 }
```

```
● ● ●  
1 public class LlamadaNacional extends Llamada {  
2  
3     public LlamadaNacional(String origen, String destino, int duracion) {  
4         super(origen, destino, duracion);  
5     }  
6  
7     @Override  
8     public double calcularMontoLlamada() {  
9         // el precio es de 3 pesos por segundo más IVA sin adicional por establecer la llamada  
10        return this.getDuracion() * 3 + (this.getDuracion() * 3 * 0.21);  
11    }  
12 }  
13 }
```

Refactoring: Form Template Method

Pasos a Seguir para llevar a cabo el Refactoring:

- Aplicar *Extract Method* en los cálculos de las distintas partes que conforman el cálculo del monto total de la llamada, este se divide en tres métodos: *obtenerPrecioLlamada*, *obtenerPrecioIVA* y *aplicarAdicional*. Se eliminan los comentarios ya que los nombres de los métodos explican lo que hacen.
- Aplicar *Pull Up Method* para el método *obtenerPrecioIVA* que es idéntico en ambas subclases.
- Aplicar *Pull Up Method* sobre los métodos *calcularMontoLlamada* de las subclases para que quede únicamente en la superclase. (Se elimina la declaración abstracta del método que existía en la superclase)
- Definir métodos abstractos en la superclase por cada método único de las subclases (*obtenerPrecioLlamada* y *aplicarAdicional*).

Resultados luego de aplicar todos los pasos

Clase Llamada

```
● ● ●  
1 public abstract double obtenerPrecioLlamada();  
2  
3     public abstract double aplicarAdicional();  
4  
5     public double obtenerIVA() {  
6         return this.obtenerPrecioLlamada() * 0.21;  
7     }  
8  
9     public double calcularMontoLlamada() {  
10        return this.obtenerPrecioLlamada() + this.obtenerIVA() + this.aplicarAdicional();  
11    }  
12 }
```

Clase LlamadaInternacional



```
1  public class LlamadaInternacional extends Llamada {  
2  
3      public LlamadaInternacional(String origen, String destino, int duracion) {  
4          super(origen, destino, duracion);  
5      }  
6  
7      @Override  
8      public double obtenerPrecioLlamada() {  
9          return this.getDuracion() * 150;  
10     }  
11  
12     @Override  
13     public double aplicarAdicional() {  
14         return 0;  
15     }  
16  
17 }
```

Clase LlamadaNacional



```
1  public class LlamadaNacional extends Llamada {  
2  
3      public LlamadaNacional(String origen, String destino, int duracion) {  
4          super(origen, destino, duracion);  
5      }  
6  
7      @Override  
8      public double obtenerPrecioLlamada() {  
9          return this.getDuracion() * 3;  
10     }  
11  
12     @Override  
13     public double aplicarAdicional() {  
14         return 50;  
15     }  
16  
17 }
```

Code Smell: Inconsistent Names

En este caso la clase Llamada presenta un atributo *destino* cuyo getter tiene un nombre inconsistente el cual es *getRemitente* esto se debe corregir renombrando el método para que haga referencia correctamente al nombre del atributo.



```
1 public String getRemitente() {  
2     return destino;  
3 }
```

Refactoring: Rename Method

Pasos a Seguir para llevar a cabo el Refactoring:

- Aplicar *Rename Method* para utilizar un nombre acorde al atributo *destino* de la clase en cuestión.

Resultados luego de aplicar todos los pasos



```
1 public String getDestino() {  
2     return destino;  
3 }
```

switch

Code Smell: Switch Statements

La clase Cliente en el método *aplicarDescuento* y la clase Empresa en el método *registrarUsuario* presentan *Switch Statements* que dan a entender que existen distintos tipos de clientes, por lo tanto es mejor crear subclases para cada tipo de cliente, para así poder aprovechar el polimorfismo y seguir los delineamientos de la programación orientada a objetos.



```
1 public Cliente registrarUsuario(String data, String nombre, String tipo) {  
2     Cliente var = new Cliente();  
3     if (tipo.equals("fisica")) {  
4         var.setNombre(nombre);  
5         String tel = this.obtenerNumeroLibre();  
6         var.setTipo(tipo);  
7         var.setNumeroTelefono(tel);  
8         var.setDNI(data);  
9     }  
10    else if (tipo.equals("juridica")) {  
11        String tel = this.obtenerNumeroLibre();  
12        var.setNombre(nombre);  
13        var.setTipo(tipo);  
14        var.setNumeroTelefono(tel);  
15        var.setCuit(data);  
16    }  
17    clientes.add(var);  
18    return var;  
19}
```



```
1 public double aplicarDescuento(double monto) {  
2     if (this.getTipo() == "fisica") {  
3         return monto*descuentoFis;  
4     } else if(this.getTipo() == "juridica") {  
5         return monto*descuentoJur;  
6     }  
7     return 0;  
8 }
```

Refactoring: Replace Type Code with Subclasses

Pasos a Seguir para llevar a cabo el Refactoring:

- Aplicar *Replace Type Code with Subclasses* para crear dos nuevas subclases de Cliente acorde con los tipos que dice, ClienteFisico y ClienteJuridico. La clase Cliente pasa a ser abstracta.

-
- Cambiar todas las referencias al tipo del cliente para que se hagan referencias a objetos de las nuevas clases creadas.
 - Añadir métodos necesarios en las clases Empresa para poder registrar clientes de los tipos correspondientes. Para esto dividimos el método RegistrarUsuario en RegistrarUsuarioFisico y RegistrarUsuarioJuridico.
 - Se elimina el atributo de tipo en Cliente. Se realiza *Move Field* para mover los atributos de *dni* y *descuentoFis* al ClienteFisico y mover el atributo *cuit* y *descuentoJur* al ClienteJuridico, junto a esto se realiza *Move Method* para mover los getters y setters correspondientes a los mismos.
 - Aplicar *Replace Conditional with Polymorphism*, usando primero *Extract Method* para extraer el funcionamiento de cada cálculo en un método con el mismo nombre, *aplicarDescuento*, y luego *Push Down Method* para bajarlos a las subclases correspondientes. El método original queda como abstracto.
 - Se modifican los test para que en vez de enviar de qué tipo es cada cliente como parámetro, se utilice el método correspondiente para registrar ese tipo de Cliente.

Resultados luego de aplicar todos los pasos

Clase Cliente

```
● ● ●
1 public abstract class Cliente {
2
3     private List<Llamada> llamadas = new ArrayList<Llamada>();
4     private String nombre;
5     private String numeroTelefono;
6
7     public String getNombre() {
8         return nombre;
9     }
10
11    public void setNombre(String nombre) {
12        this.nombre = nombre;
13    }
14
15    public String getNumeroTelefono() {
16        return numeroTelefono;
17    }
18
19    public void setNumeroTelefono(String numeroTelefono) {
20        this.numeroTelefono = numeroTelefono;
21    }
22
23    public Llamada registrarLlamadaInternacional(Cliente destino, int duracion) {
24        Llamada llamada = new LlamadaInternacional(this.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
25        this.llamadas.add(llamada);
26        return llamada;
27    }
28
29    public Llamada registrarLlamadaNacional(Cliente destino, int duracion) {
30        Llamada llamada = new LlamadaNacional(this.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
31        this.llamadas.add(llamada);
32        return llamada;
33    }
34
35    public double calcularMontoTotalLlamadas() {
36        double c = 0;
37        for (Llamada l : this.llamadas) {
38            double auxc = 0;
39            auxc += l.calcularMontoLlamada();
40            auxc -= this.aplicarDescuento(auxc);
41            c += auxc;
42        }
43        return c;
44    }
45
46    public abstract double aplicarDescuento(double monto);
47
48 }
```

Clase ClienteJuridico

```
● ● ●  
1 public class ClienteJuridico extends Cliente {  
2  
3     private String cuit;  
4     private static double descuentoJur = 0.15;  
5  
6     public double getDescuentoJur() {  
7         return ClienteJuridico.descuentoJur;  
8     }  
9  
10    public void setDescuentoJur(double descuentoJur) {  
11        ClienteJuridico.descuentoJur = descuentoJur;  
12    }  
13  
14    public String getCuit() {  
15        return cuit;  
16    }  
17  
18    public void setCuit(String cuit) {  
19        this.cuit = cuit;  
20    }  
21  
22    @Override  
23    public double aplicarDescuento(double monto) {  
24        return monto*descuentoJur;  
25    }  
26  
27 }
```

Clase ClienteFisico

```
● ● ●  
1 public class ClienteFisico extends Cliente {  
2  
3     private String dni;  
4     private static double descuentoFis = 0;  
5  
6     public double getDescuentoFis() {  
7         return ClienteFisico.descuentoFis;  
8     }  
9  
10    public void setDescuentoFis(double descuentoJur) {  
11        ClienteFisico.descuentoFis = descuentoJur;  
12    }  
13  
14    public String getDNI() {  
15        return dni;  
16    }  
17  
18    public void setDNI(String dni) {  
19        this.dni = dni;  
20    }  
21  
22    @Override  
23    public double aplicarDescuento(double monto) {  
24        return monto*descuentoFis;  
25    }  
26  
27 }
```

Clase Empresa



```
1 public Cliente registrarUsuarioFisico(String data, String nombre) {
2     ClienteFisico var = new ClienteFisico();
3     var.setNombre(nombre);
4     String tel = this.obtenerNumeroLibre();
5     var.setNumeroTelefono(tel);
6     var.setDNI(data);
7     clientes.add(var);
8     return var;
9 }
10
11 public Cliente registrarUsuarioJuridico(String data, String nombre) {
12     ClienteJuridico var = new ClienteJuridico();
13     var.setNombre(nombre);
14     String tel = this.obtenerNumeroLibre();
15     var.setNumeroTelefono(tel);
16     var.setCuit(data);
17     clientes.add(var);
18     return var;
19 }
```

Test EmpresaTest

Antes

```
● ● ●

1  @Test
2      void testcalcularMontoTotalLlamadas() {
3          Cliente emisorPersonaFisca = sistema.registrarUsuario("11555666", "Brendan Eich", "fisica");
4          Cliente remitentePersonaFisica = sistema.registrarUsuario("00000001", "Doug Lea", "fisica");
5          Cliente emisorPersonaJuridica = sistema.registrarUsuario("17555222", "Nvidia Corp", "juridica");
6          Cliente remitentePersonaJuridica = sistema.registrarUsuario("25765432", "Sun Microsystems", "juridica");
7
8          this.sistema.registrarLlamadaNacional(emisorPersonaJuridica, remitentePersonaFisica, 10);
9          this.sistema.registrarLlamadaInternacional(emisorPersonaJuridica, remitentePersonaFisica, 8);
10         this.sistema.registrarLlamadaNacional(emisorPersonaJuridica, remitentePersonaJuridica, 5);
11         this.sistema.registrarLlamadaInternacional(emisorPersonaJuridica, remitentePersonaJuridica, 7);
12         this.sistema.registrarLlamadaNacional(emisorPersonaFisca, remitentePersonaFisica, 15);
13         this.sistema.registrarLlamadaInternacional(emisorPersonaFisca, remitentePersonaFisica, 45);
14         this.sistema.registrarLlamadaNacional(emisorPersonaFisca, remitentePersonaJuridica, 13);
15         this.sistema.registrarLlamadaInternacional(emisorPersonaFisca, remitentePersonaJuridica, 17);
16
17         assertEquals(11454.64, this.sistema.calcularMontoTotalLlamadas(emisorPersonaFisca), 0.01);
18         assertEquals(2445.40, this.sistema.calcularMontoTotalLlamadas(emisorPersonaJuridica), 0.01);
19         assertEquals(0, this.sistema.calcularMontoTotalLlamadas(remitentePersonaFisica));
20         assertEquals(0, this.sistema.calcularMontoTotalLlamadas(remitentePersonaJuridica));
21     }
22
23     @Test
24     void testAgregarUsuario() {
25         assertEquals(this.sistema.cantidadDeUsuarios(), 0);
26         this.sistema.agregarNumeroTelefono("2214444558");
27         Cliente nuevaPersona = this.sistema.registrarUsuario("2444555", "Alan Turing", "fisica");
28
29         assertEquals(1, this.sistema.cantidadDeUsuarios());
30         assertTrue(this.sistema.existeUsuario(nuevaPersona));
31     }
}
```

Después

```
● ● ●  
1  @Test  
2      void testcalcularMontoTotalLlamadas() {  
3          Cliente emisorPersonaFisica = sistema.registrarUsuarioFisico("11555666", "Brendan Eich");  
4          Cliente remitentePersonaFisica = sistema.registrarUsuarioFisico("00000001", "Doug Lea");  
5          Cliente emisorPersonaJuridica = sistema.registrarUsuarioJuridico("17555222", "Nvidia Corp");  
6          Cliente remitentePersonaJuridica = sistema.registrarUsuarioJuridico("25765432", "Sun Microsystems");  
7  
8          this.sistema.registrarLlamadaNacional(emisorPersonaJuridica, remitentePersonaFisica, 10);  
9          this.sistema.registrarLlamadaInternacional(emisorPersonaJuridica, remitentePersonaFisica, 8);  
10         this.sistema.registrarLlamadaNacional(emisorPersonaJuridica, remitentePersonaJuridica, 5);  
11         this.sistema.registrarLlamadaInternacional(emisorPersonaJuridica, remitentePersonaJuridica, 7);  
12         this.sistema.registrarLlamadaNacional(emisorPersonaFisica, remitentePersonaFisica, 15);  
13         this.sistema.registrarLlamadaInternacional(emisorPersonaFisica, remitentePersonaFisica, 45);  
14         this.sistema.registrarLlamadaNacional(emisorPersonaFisica, remitentePersonaJuridica, 13);  
15         this.sistema.registrarLlamadaInternacional(emisorPersonaFisica, remitentePersonaJuridica, 17);  
16  
17         assertEquals(11454.64, this.sistema.calcularMontoTotalLlamadas(emisorPersonaFisica), 0.01);  
18         assertEquals(2445.40, this.sistema.calcularMontoTotalLlamadas(emisorPersonaJuridica), 0.01);  
19         assertEquals(0, this.sistema.calcularMontoTotalLlamadas(remitentePersonaFisica));  
20         assertEquals(0, this.sistema.calcularMontoTotalLlamadas(remitentePersonaJuridica));  
21     }  
22  
23     @Test  
24     void testAgregarUsuario() {  
25         assertEquals(this.sistema.cantidadDeUsuarios(), 0);  
26         this.sistema.agregarNumeroTelefono("2214444558");  
27         Cliente nuevaPersona = this.sistema.registrarUsuarioFisico("2444555", "Alan Turing");  
28  
29         assertEquals(1, this.sistema.cantidadDeUsuarios());  
30         assertTrue(this.sistema.existeUsuario(nuevaPersona));  
31     }  
}
```

Code Smell: Non-Descriptive Names

En este caso dentro de la clase Empresa los métodos registrarUsuarioJuridico y registrarUsuarioFisico tienen nombres que no coinciden con los nombres de las clases ClienteJuridico y ClienteFisico, por lo que puede resultar confuso. También ciertos parámetros y variables resultan poco explicativos y sería mejor renombrarlos.



```
1 public Cliente registrarUsuarioFisico(String data, String nombre) {
2     ClienteFisico var = new ClienteFisico();
3     var.setNombre(nombre);
4     String tel = this.obtenerNumeroLibre();
5     var.setNumeroTelefono(tel);
6     var.setDNI(data);
7     clientes.add(var);
8     return var;
9 }
10
11 public Cliente registrarUsuarioJuridico(String data, String nombre) {
12     ClienteJuridico var = new ClienteJuridico();
13     var.setNombre(nombre);
14     String tel = this.obtenerNumeroLibre();
15     var.setNumeroTelefono(tel);
16     var.setCuit(data);
17     clientes.add(var);
18     return var;
19 }
```

Refactoring: Rename Method, Rename Parameter, Rename Variable

Pasos a Seguir para llevar a cabo el Refactoring:

- Aplicar *Rename Method* para renombrar los métodos *registrarUsuarioFisico* que pasa a ser *registrarClienteFisico* y *registrarUsuarioJuridico* que pasa a ser *registrarClienteJuridico*.
- Aplicar *Rename Parameter* para renombrar los parámetros *data* de ambos métodos para que pasen a llamarse *dni* en el caso del *registrarClienteFisico* y *cuit* en el caso de *registrarClienteJuridico*.
- Aplicar *Rename Variable* para modificar el nombre de *var* por *cliente* y el nombre de *tel* por *numeroTelefono*.
- Se modifican los test por el cambio de nombre de los métodos.

Resultados luego de aplicar todos los pasos

Clase Empresa



```
1 public Cliente registrarClienteFisico(String dni, String nombre) {
2     ClienteFisico cliente = new ClienteFisico();
3     cliente.setNombre(nombre);
4     String numeroTelefono = this.obtenerNumeroLibre();
5     cliente.setNumeroTelefono(numeroTelefono);
6     cliente.setDNI(dni);
7     clientes.add(cliente);
8     return cliente;
9 }
10
11 public Cliente registrarClienteJuridico(String cuit, String nombre) {
12     ClienteJuridico cliente = new ClienteJuridico();
13     cliente.setNombre(nombre);
14     String numeroTelefono = this.obtenerNumeroLibre();
15     cliente.setNumeroTelefono(numeroTelefono);
16     cliente.setCuit(cuit);
17     clientes.add(cliente);
18     return cliente;
19 }
```

Test EmpresaTest

Antes

```
● ○ ●
1  @Test
2  void testcalcularMontoTotalLlamadas() {
3      Cliente emisorPersonaFisica = sistema.registrarUsuarioFisico("11555666", "Brendan Eich");
4      Cliente remitentePersonaFisica = sistema.registrarUsuarioFisico("00000001", "Doug Lea");
5      Cliente emisorPersonaJuridica = sistema.registrarUsuarioJuridico("17555222", "Nvidia Corp");
6      Cliente remitentePersonaJuridica = sistema.registrarUsuarioJuridico("25765432", "Sun Microsystems");
7
8      this.sistema.registrarLlamadaNacional(emisorPersonaJuridica, remitentePersonaFisica, 10);
9      this.sistema.registrarLlamadaInternacional(emisorPersonaJuridica, remitentePersonaFisica, 8);
10     this.sistema.registrarLlamadaNacional(emisorPersonaJuridica, remitentePersonaJuridica, 5);
11     this.sistema.registrarLlamadaInternacional(emisorPersonaJuridica, remitentePersonaJuridica, 7);
12     this.sistema.registrarLlamadaNacional(emisorPersonaFisica, remitentePersonaFisica, 15);
13     this.sistema.registrarLlamadaInternacional(emisorPersonaFisica, remitentePersonaFisica, 45);
14     this.sistema.registrarLlamadaNacional(emisorPersonaFisica, remitentePersonaJuridica, 13);
15     this.sistema.registrarLlamadaInternacional(emisorPersonaFisica, remitentePersonaJuridica, 17);
16
17     assertEquals(11454.64, this.sistema.calcularMontoTotalLlamadas(emisorPersonaFisica), 0.01);
18     assertEquals(2445.40, this.sistema.calcularMontoTotalLlamadas(emisorPersonaJuridica), 0.01);
19     assertEquals(0, this.sistema.calcularMontoTotalLlamadas(remitentePersonaFisica));
20     assertEquals(0, this.sistema.calcularMontoTotalLlamadas(remitentePersonaJuridica));
21 }
22
23 @Test
24 void testAgregarUsuario() {
25     assertEquals(this.sistema.cantidadDeUsuarios(), 0);
26     this.sistema.agregarNumeroTelefono("2214444558");
27     Cliente nuevaPersona = this.sistema.registrarUsuarioFisico("2444555", "Alan Turing");
28
29     assertEquals(1, this.sistema.cantidadDeUsuarios());
30     assertTrue(this.sistema.existeUsuario(nuevaPersona));
31 }
```

Después

```
● ○ ●
1  @Test
2      void testcalcularMontoTotalLlamadas() {
3          Cliente emisorPersonaFisica = sistema.registrarClienteFisico("11555666", "Brendan Eich");
4          Cliente remitentePersonaFisica = sistema.registrarClienteFisico("00000001", "Doug Lea");
5          Cliente emisorPersonaJuridica = sistema.registrarClienteJuridico("17555222", "Nvidia Corp");
6          Cliente remitentePersonaJuridica = sistema.registrarClienteJuridico("25765432", "Sun Microsystems");
7
8          this.sistema.registrarLlamadaNacional(emisorPersonaJuridica, remitentePersonaFisica, 10);
9          this.sistema.registrarLlamadaInternacional(emisorPersonaJuridica, remitentePersonaFisica, 8);
10         this.sistema.registrarLlamadaNacional(emisorPersonaJuridica, remitentePersonaJuridica, 5);
11         this.sistema.registrarLlamadaInternacional(emisorPersonaJuridica, remitentePersonaJuridica, 7);
12         this.sistema.registrarLlamadaNacional(emisorPersonaFisica, remitentePersonaFisica, 15);
13         this.sistema.registrarLlamadaInternacional(emisorPersonaFisica, remitentePersonaFisica, 45);
14         this.sistema.registrarLlamadaNacional(emisorPersonaFisica, remitentePersonaJuridica, 13);
15         this.sistema.registrarLlamadaInternacional(emisorPersonaFisica, remitentePersonaJuridica, 17);
16
17         assertEquals(11454.64, this.sistema.calcularMontoTotalLlamadas(emisorPersonaFisica), 0.01);
18         assertEquals(2445.40, this.sistema.calcularMontoTotalLlamadas(emisorPersonaJuridica), 0.01);
19         assertEquals(0, this.sistema.calcularMontoTotalLlamadas(remitentePersonaFisica));
20         assertEquals(0, this.sistema.calcularMontoTotalLlamadas(remitentePersonaJuridica));
21     }
22
23     @Test
24     void testAgregarUsuario() {
25         assertEquals(this.sistema.cantidadDeUsuarios(), 0);
26         this.sistema.agregarNumeroTelefono("2214444558");
27         Cliente nuevaPersona = this.sistema.registrarClienteFisico("2444555", "Alan Turing");
28
29         assertEquals(1, this.sistema.cantidadDeUsuarios());
30         assertTrue(this.sistema.existeUsuario(nuevaPersona));
31     }
}
```

Code Smell: Inappropriate Intimacy

Dentro de la clase Empresa, específicamente en los métodos *registrarClienteFisico* y *registrarClienteJuridico* se crea una instancia de un objeto ClienteJuridico o ClienteFisico y se setea los valores del mismo a partir del uso de setters ejecutados en un contexto que no es el del objeto Cliente creado, esto nos lleva a pensar que la clase Empresa tiene una intimidad inapropiada a la hora de crear un cliente ya sea Jurídico o Físico.



```
1 public Cliente registrarClienteFisico(String dni, String nombre) {
2     ClienteFisico cliente = new ClienteFisico();
3     cliente.setNombre(nombre);
4     String numeroTelefono = this.obtenerNumeroLibre();
5     cliente.setNumeroTelefono(numeroTelefono);
6     cliente.setDNI(dni);
7     clientes.add(cliente);
8     return cliente;
9 }
10
11 public Cliente registrarClienteJuridico(String cuit, String nombre) {
12     ClienteJuridico cliente = new ClienteJuridico();
13     cliente.setNombre(nombre);
14     String numeroTelefono = this.obtenerNumeroLibre();
15     cliente.setNumeroTelefono(numeroTelefono);
16     cliente.setCuit(cuit);
17     clientes.add(cliente);
18     return cliente;
19 }
```

Refactoring: Extract Method

Pasos a Seguir para llevar a cabo el Refactoring:

- Aplicar *Extract Method* para extraer el comportamiento que viola la intimidad de los objetos clientes, depositando estas responsabilidades de creación en las clases que las deben tener usando constructores.
- Dejar en la clase empresa una delegación de creación a los constructores de las clases que desea instanciar.

Resultados luego de aplicar todos los pasos

Clase Empresa

```
● ● ●  
1 public Cliente registrarClienteFisico(String dni, String nombre) {  
2     String numeroTelefono = this.obtenerNumeroLibre();  
3     Cliente cliente = new ClienteFisico(nombre, numeroTelefono, dni);  
4     clientes.add(cliente);  
5     return cliente;  
6 }  
7  
8 public Cliente registrarClienteJuridico(String cuit, String nombre) {  
9     String numeroTelefono = this.obtenerNumeroLibre();  
10    Cliente cliente = new ClienteJuridico(nombre, numeroTelefono, cuit);  
11    clientes.add(cliente);  
12    return cliente;  
13 }
```

Clase Cliente

```
● ● ●  
1 public Cliente(String nombre, String numeroTelefono) {  
2     this.nombre = nombre;  
3     this.numeroTelefono = numeroTelefono;  
4 }
```

Clase ClienteFisico

```
● ● ●  
1 public ClienteFisico(String nombre, String numeroTelefono, String dni) {  
2     super(nombre, numeroTelefono);  
3     this.dni = dni;  
4 }
```

Clase ClienteJuridico



```
1 public ClienteJuridico(String nombre, String numeroTelefono, String cuit) {  
2     super(nombre, numeroTelefono);  
3     this.cuit = cuit;  
4 }
```

Code Smell: Long Method

La clase Empresa presenta 2 *Long Methods* *registrarClienteJuridico* y *registrarClienteFisico*. Esto se puede solucionar separando la creación de un Cliente del proceso para agregarlo al mismo a la colección de Clientes que almacena la clase Empresa.



```
1 public Cliente registrarClienteFisico(String dni, String nombre) {  
2     String numeroTelefono = this.obtenerNumeroLibre();  
3     Cliente cliente = new ClienteFisico(nombre, numeroTelefono, dni);  
4     clientes.add(cliente);  
5     return cliente;  
6 }  
7  
8 public Cliente registrarClienteJuridico(String cuit, String nombre) {  
9     String numeroTelefono = this.obtenerNumeroLibre();  
10    Cliente cliente = new ClienteJuridico(nombre, numeroTelefono, cuit);  
11    clientes.add(cliente);  
12    return cliente;  
13 }
```

Refactoring: Extract Method

Pasos a Seguir para llevar a cabo el Refactoring:

- Aplicar *Extract Method* para separar la creación de un Cliente del proceso de agregar al mismo a la colección de Clientes de la clase Empresa, dejando una delegación de esta última operación en el método *agregarCliente*, que lo dejamos como privado porque solo debe ser accedido desde la misma clase.

Resultados luego de aplicar todos los pasos



```
1 private void agregarCliente(Cliente cliente) {
2     this.clientes.add(cliente);
3 }
4
5 public Cliente registrarClienteFisico(String dni, String nombre) {
6     String numeroTelefono = this.obtenerNumeroLibre();
7     Cliente cliente = new ClienteFisico(nombre, numeroTelefono, dni);
8     this.agregarCliente(cliente);
9     return cliente;
10 }
11
12 public Cliente registrarClienteJuridico(String cuit, String nombre) {
13     String numeroTelefono = this.obtenerNumeroLibre();
14     Cliente cliente = new ClienteJuridico(nombre, numeroTelefono, cuit);
15     this.agregarCliente(cliente);
16     return cliente;
17 }
```

Code Smell: Long Method

La clase Cliente presenta 2 *Long Methods* *registrarLlamadaNacional* y *registrarLlamadaInternacional*. Esto se soluciona de la misma forma que en el caso anterior.



```
1 public Llamada registrarLlamadaInternacional(Cliente destino, int duracion) {
2     Llamada llamada = new LlamadaInternacional(this.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
3     this.llamadas.add(llamada);
4     return llamada;
5 }
6
7 public Llamada registrarLlamadaNacional(Cliente destino, int duracion) {
8     Llamada llamada = new LlamadaNacional(this.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
9     this.llamadas.add(llamada);
10    return llamada;
11 }
```

Refactoring: Extract Method

Pasos a Seguir para llevar a cabo el Refactoring:

- Aplicar *Extract Method* para separar la creación de un Llamada del proceso de agregar a la misma a la colección de Llamadas de la clase Cliente, dejando una delegación de esta última operación en el método *agregaLlamada*, que lo dejamos como privado porque solo debe ser accedido desde la misma clase.

Resultados luego de aplicar todos los pasos



```
1 private void agregarLlamada(Llamada llamada) {  
2     this.llamadas.add(llamada);  
3 }  
4  
5 public Llamada registrarLlamadaInternacional(Cliente destino, int duracion) {  
6     Llamada llamada = new LlamadaInternacional(this.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);  
7     this.agregarLlamada(llamada);  
8     return llamada;  
9 }  
10  
11 public Llamada registrarLlamadaNacional(Cliente destino, int duracion) {  
12     Llamada llamada = new LlamadaNacional(this.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);  
13     this.agregarLlamada(llamada);  
14     return llamada;  
15 }
```

Code Smell: Duplicated Code

Las clases ClienteJuridico y ClienteFisico mantienen un comportamiento casi idéntico a la hora de realizar el método *aplicarDescuento*, esto genera una duplicación de código entre las dos clases que mediante una mejor organización puede desaparecer.



```
1 @Override  
2     public double aplicarDescuento(double monto) {  
3         return monto*descuentoFis;  
4     }
```



```
1 @Override  
2     public double aplicarDescuento(double monto) {  
3         return monto*descuentoJur;  
4     }
```

Refactoring: Form Template Method

Pasos a Seguir para llevar a cabo el Refactoring:

- No usamos *Extract Method* porque ya existen los métodos *getDescuentoJur* y *getDescuentoFis*. Cambiamos los usos directos de las variables por sus getters correspondientes
- Aplicamos *Rename Method* para cada método específico de cada clase para que tengan el mismo nombre, unificandolos en *getDescuento*. Para mantener la coherencia también se modifican los setters.
- Aplicar *Pull Up Method* sobre los métodos *aplicarDescuento* de las subclases para que quede únicamente en la superclase. (Se elimina la declaración abstracta del método que existía en la superclase)
- Definir métodos abstractos en la superclase por cada método único de las subclases (*getDescuento* y *setDescuento*).

Resultados luego de aplicar todos los pasos

Clase Cliente

```
1  public abstract double getDescuento();
2
3      public abstract void setDescuento(double descuento);
4
5      public double aplicarDescuento(double monto) {
6          return monto * this.getDescuento();
7      }
```

Clase ClienteJuridico



```
1 public class ClienteJuridico extends Cliente {
2
3     private String cuit;
4     private static double descuentoJur = 0.15;
5
6     public ClienteJuridico(String nombre, String numeroTelefono, String cuit) {
7         super(nombre, numeroTelefono);
8         this.cuit = cuit;
9     }
10
11    @Override
12    public double getDescuento() {
13        return ClienteJuridico.descuentoJur;
14    }
15
16    @Override
17    public void setDescuento(double descuento) {
18        ClienteJuridico.descuentoJur = descuento;
19    }
20
21    public String getCuit() {
22        return cuit;
23    }
24
25    public void setCuit(String cuit) {
26        this.cuit = cuit;
27    }
28
29 }
```

Clase ClienteFisico

```
● ● ●  
1 public class ClienteFisico extends Cliente {  
2  
3     private String dni;  
4     private static double descuentoFis = 0;  
5  
6     public ClienteFisico(String nombre, String numeroTelefono, String dni) {  
7         super(nombre, numeroTelefono);  
8         this.dni = dni;  
9     }  
10  
11    @Override  
12    public double getDescuento() {  
13        return ClienteFisico.descuentoFis;  
14    }  
15  
16    @Override  
17    public void setDescuento(double descuento) {  
18        ClienteFisico.descuentoFis = descuento;  
19    }  
20  
21    public String getDNI() {  
22        return dni;  
23    }  
24  
25    public void setDNI(String dni) {  
26        this.dni = dni;  
27    }  
28  
29 }
```

Code Smell: Reinventar la rueda

El método *calcularMontoLlamada* en la clase Cliente realiza un cálculo que puede fácilmente ser reemplazado por un stream, es decir, que hay código ya existente que realiza la misma funcionalidad de una manera segura y testeada. Además, nos deshacemos de variables temporales que son innecesarias y poseían nombres poco explicativos.



```
1 public double calcularMontoTotalLlamadas() {  
2     double c = 0;  
3     for (Llamada l : this.llamadas) {  
4         double auxc = 0;  
5         auxc += l.calcularMontoLlamada();  
6         auxc -= this.aplicarDescuento(auxc);  
7         c += auxc;  
8     }  
9     return c;  
10 }
```

Refactoring: Replace Loop with Pipeline

Pasos a Seguir para llevar a cabo el Refactoring:

- Aplicamos *Replace Loop with Pipeline* para reemplazar el cálculo del monto de todas las llamadas realizadas por el cliente en el método *calcularMontoLlamadas* por un stream que realiza la misma funcionalidad resumidamente. Eliminamos las variables temporales que usaba el método anteriormente ya que son innecesarias cuando realizas un stream.

Resultados luego de aplicar todos los pasos



```
1 public double calcularMontoTotalLlamadas() {  
2     return this.llamadas.stream()  
3         .mapToDouble(  
4             llamada → llamada.calcularMontoLlamada()  
5             -  
6             this.aplicarDescuento(llamada.calcularMontoLlamada())  
7         )  
8         .sum();  
9 }
```

Code Smell: Non-Descriptive Names

En este caso dentro de la clase Empresa el método *existeUsuario* tiene un nombre que no coincide con lo que almacena la clase Empresa, que son clientes, por lo que puede resultar confuso. También el parámetro *persona* resulta poco explicativo y sería mejor renombrarlo.



```
1 public boolean existeUsuario(Cliente persona) {  
2     return clientes.contains(persona);  
3 }
```

Refactoring: Rename Method, Rename Parameter

Pasos a Seguir para llevar a cabo el Refactoring:

- Aplicar *Rename Method* para renombrar el método *existeUsuario* que pasa a ser *existeCliente*.
- Aplicar *Rename Parameter* para renombrar el parámetro *persona* para que pase a llamarse *cliente*.
- Se modifican los test por el cambio de nombre del método.

Resultados luego de aplicar todos los pasos

Clase Empresa



```
1 public boolean existeCliente(Cliente cliente) {  
2     return clientes.contains(cliente);  
3 }
```

Test EmpresaTest

Antes

```
● ● ●  
1 @Test  
2     void testAgregarUsuario() {  
3         assertEquals(this.sistema.cantidadDeUsuarios(), 0);  
4         this.sistema.agregarNumeroTelefono("2214444558");  
5         Cliente nuevaPersona = this.sistema.registrarClienteFisico("2444555","Alan Turing");  
6  
7         assertEquals(1, this.sistema.cantidadDeUsuarios());  
8         assertTrue(this.sistema.existeUsuario(nuevaPersona));  
9     }
```

Después

```
● ● ●  
1 @Test  
2     void testAgregarUsuario() {  
3         assertEquals(this.sistema.cantidadDeUsuarios(), 0);  
4         this.sistema.agregarNumeroTelefono("2214444558");  
5         Cliente nuevaPersona = this.sistema.registrarClienteFisico("2444555","Alan Turing");  
6  
7         assertEquals(1, this.sistema.cantidadDeUsuarios());  
8         assertTrue(this.sistema.existeCliente(nuevaPersona));  
9     }
```

Bibliografía Utilizada:

- Fowler, M. (2004). *Refactoring: Improving the design of existing code* (1st ed.). Addison-Wesley.
- Kerievsky, J. (2004). *Refactoring to patterns (Draft)*. Addison-Wesley.
- SourceMaking. (n.d.). *Code smells*. Retrieved from <https://sourcemaking.com/refactoring/smells>
- Fowler, M. (n.d.). *Refactoring catalog*. Retrieved from <https://refactoring.com/catalog/>