# Ejercicio 3 - Facturación de llamadas | Leandro Lopez

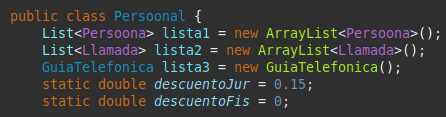
## Refactoring 1

**Mal olor detectado en el código:** Múltiples atributos que no son privados en las clases Personal, Persona, Llamada y GuiaTelefonica, lo cual rompen el encapsulamiento

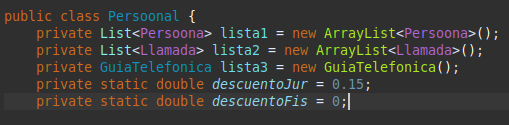
**Método de Refactoring**: Encapsulate Field

### Caso 1

**Extracto del código que presenta el mal olor:**

****

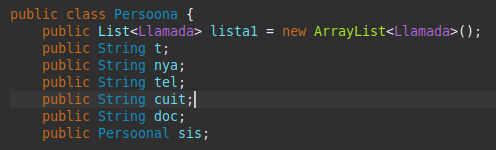
Se ve que tiene múltiples atributos que no tienen modificador de acceso por lo cual es package private por defecto, lo que significa que son privados al paquete, pero puede tener múltiples clases el paquete y pueden modificarlo, lo cual rompe la encapsulación. Aplicó ENCAPSULATE FIELD, poniendo los atributos en privado y creando métodos de acceso si es que no existen ya



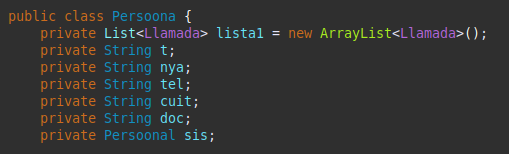


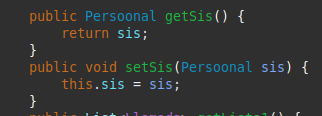
### Caso 2

**Extracto del código que presenta el mal olor:**

****

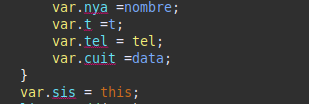
Esta clase presenta múltiples atributos en público, por lo cual los privatizo y le creo metodos de acceso, dado a que ya posee varios, le creo el único que le falta



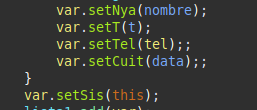


Y actualizo las referencias donde se utilizaba el atributo directamente

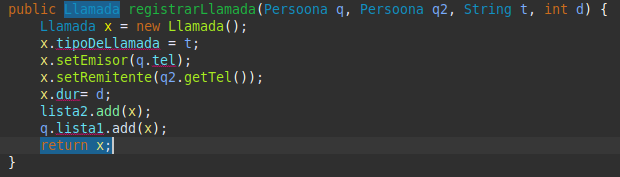
Método registrarUsuario de la clase Persoonal



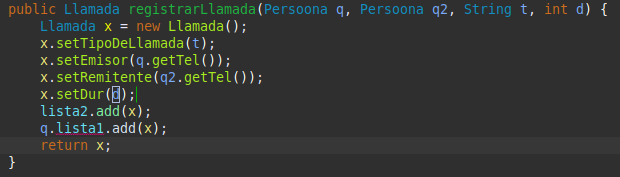
Quedando así



Método registrarLlamada de la clase Persoonal



Quedando así



Método calcularMontoTotalLlamadas de la clase Persoonal



Quedando así



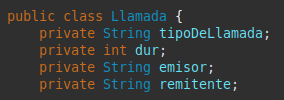
Y algunas referencias a lista1.add y lista1.remove qué se reemplazaron por getLista1().add y getLista1.remove()

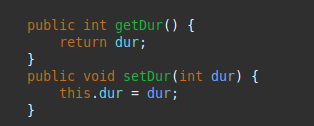
### Caso 3

**Extracto del código que presenta el mal olor:**

****

Privatizó los atributos y le creo metodos de acceso si es que no tiene





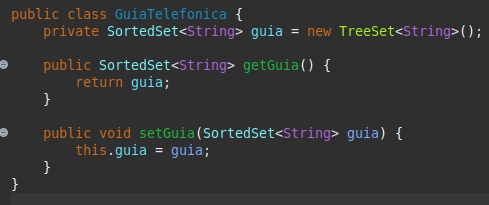


### Caso 4

**Extracto del código que presenta el mal olor:**

****

Privatizó el atributo y le creo metodos de acceso



Y unas referencias a donde en la clase Personal se hacía guia.add y guia.remove se reemplazaron por getGuia().add y getGuia().remove

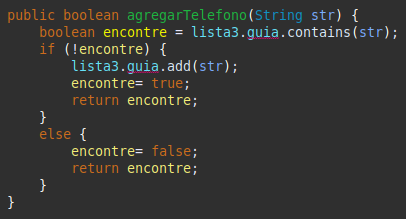
## Refactoring 2

**Mal olor detectado en el código:** Envidia de atributos, se está utilizando múltiples veces un atributo que es perteneciente a otra clase

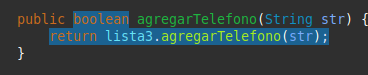
**Método de Refactoring**: Move Method - Consolidate Duplicate Conditional Fragments

### Caso 1

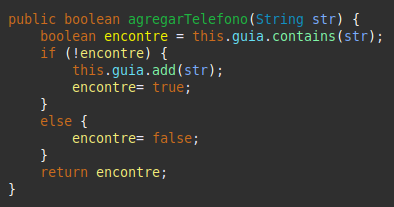
**Extracto del código que presenta el mal olor:**

****

Se está utilizando múltiples veces lista3 que es una instancia de GuiaTelefonica y un atributo y métodos que deberían realizarse desde la clase, la responsabilidad está mal asignada, por lo cual creo en la clase GuiaTelefonica un método para agregarTelefono y que haga toda la logica que contiene actualmente el método, y al método actual solo le quedaria una llamada al método en la otra clase  
  
En persoonal

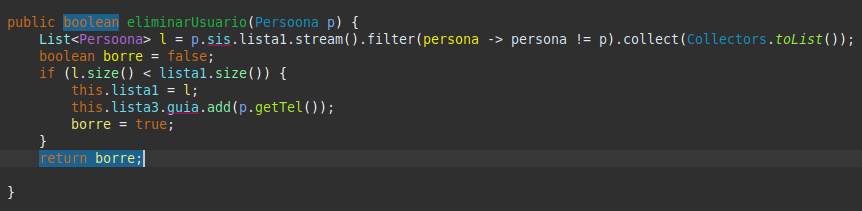


En GuiaTelefonica

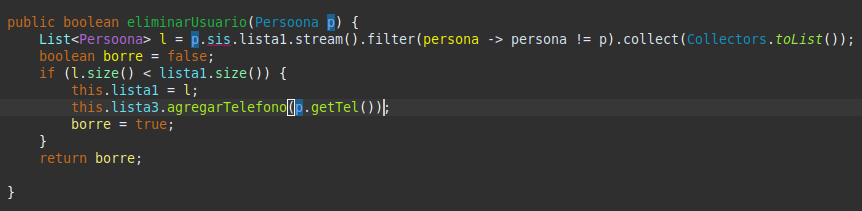
  
Aplique de paso refactoring **Consolidate Duplicate Conditional Fragments** quitando el return encontré afuera dado a que ambos condicionales hacían lo mismo

Y de paso modificar las referencias donde se agregaban teléfonos a la guia mediante el atributo, pero ahora agregando la referencia al método

En Personal



Reemplazo el guia.add



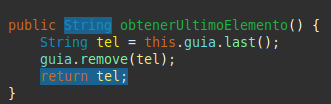
### Caso 2

**Extracto del código que presenta el mal olor:**

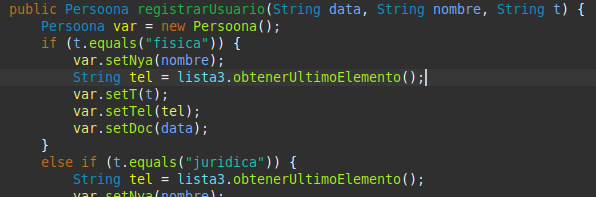
****

Esas dos lineas de código se repiten en el IF y ELSE del método registrarUsuario en la clase Personal, por lo cual es duplicación de codigo ademas de que es envidia de atributos dado a que es una responsabilidad mal asignada, es algo que la clase Guia Telefonica debería poder resolver

Creo el método en la clase Guia Telefonica



Y reemplazo las referencias



## Refactoring 3

**Mal olor detectado en el código:** Data class, revisando bien, la clase Guia Telefonica unicamente tiene una lista, getter y setter, por lo cual es innecesaria tener una clase para esto, podria tenerlo directamente en la clase que la utiliza junto a sus métodos

**Método de Refactoring**: Inline Class

### Caso 1

**Extracto del código que presenta el mal olor:**



Muevo todo esto a la clase Personal y elimino el atributo lista3 y el metodo agregarTelefono, getLista3 y setLista3 que esta ya tenía. Y por último actualizar las referencias para referirse directamente al atributo guia

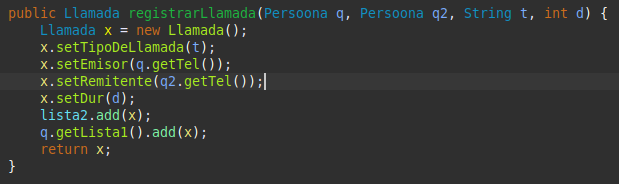
## Refactoring 4

**Mal olor detectado en el código:** No se utiliza el constructor de la clase Llamada y existen métodos setters innecesarios

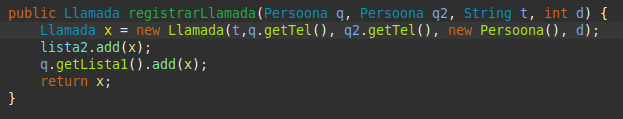
**Método de Refactoring**: Remove setting method

### Caso 1

**Extracto del código que presenta el mal olor:**



La clase Llamada posee un constructor que puede abarcar el tipo de llamada, emisor, remitente, una persona (parámetro tipo Persoona que no se utiliza), duración, por lo cual es innecesario todos esos setter para setear los valores teniendo el constructor, además de que solo se setean una vez los valores, y esto es en la creación de la instancia de Llamada, por lo cual se podría remover todos estos setters y quedarse únicamente con el constructor



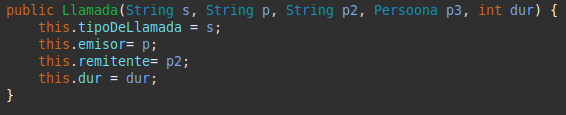
## Refactoring 5

**Mal olor detectado en el código:** Parámetro no utilizado

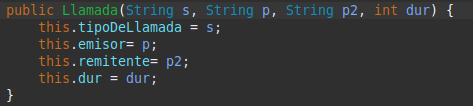
**Método de Refactoring**: Remove parameter

### Caso 1

**Extracto del código que presenta el mal olor:**

****

El constructor de la clase Llamada posee un argumento de tipo Persoona p3, pero este no es utilizado en el método, por lo cual se lo puede eliminar, y quitar el argumento innecesario en donde se haga llamar a este constructor



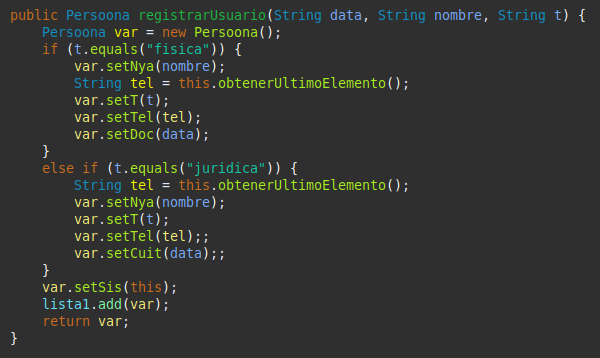
## Refactoring 6

**Mal olor detectado en el código:** Método largo y condicionales

**Método de Refactoring**: Extract Method - Extract Superclass - Extract Subclass - Remove setting method

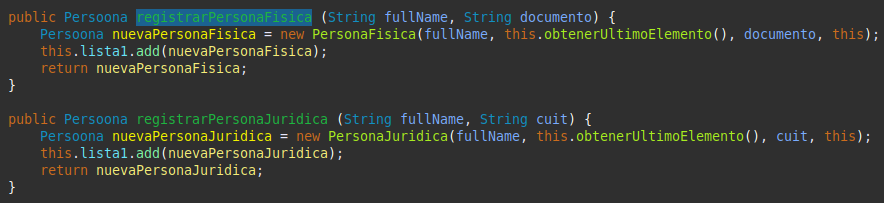
### Caso 1

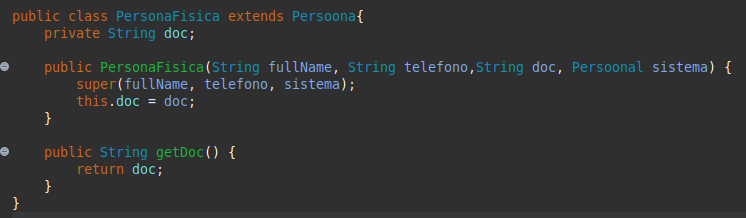
**Extracto del código que presenta el mal olor:**

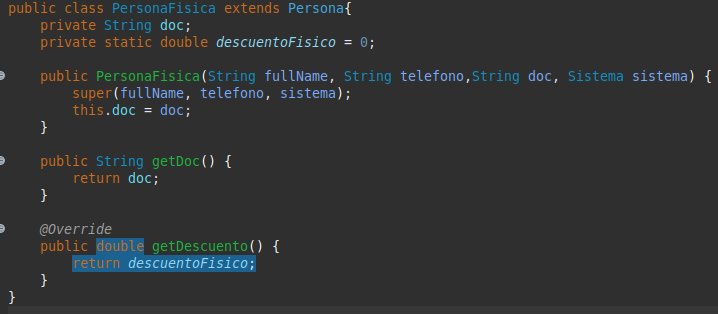
****

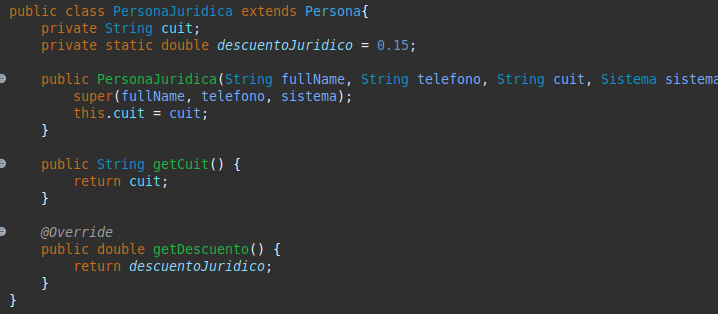
El método es largo y tiene dos condicionales que en base a si la persona es fisica o juridica, completa ciertos datos, por lo cual es algo que podría hacer en subclases, creando dos subclases que hereden de Persoona, PersonaFisica y PersonaJuridica, los atributos nya, tel son compartido por ambos tipos de persona, por lo cual puede quedarse en la superclase, lo que varía es Cuit y Doc. Además ambos tienen un descuento en base a lo que son, pero se calcula distinto

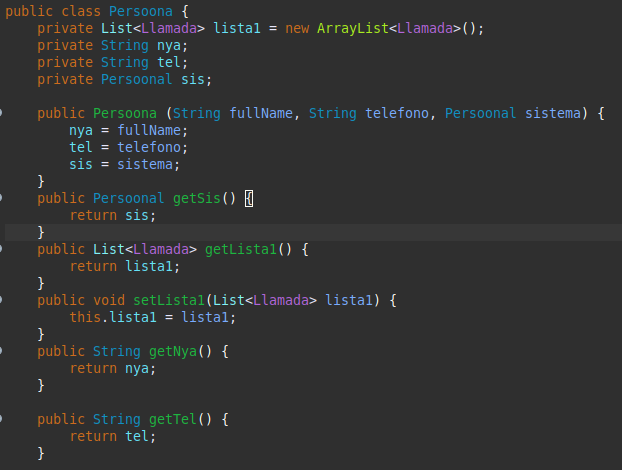
Dado a que **instanciación** de la clase Persoona, solo requiere setear los atributos en su creación, pero no requiere que más adelante se modifique los atributos, por lo cual puedo aplicar el refactoring de **Remove setting method**, los setter son innecesarios y con setearlos desde el constructor ya basta, por lo cual borró los setter de la clase Persoona y creo un constructor con los atributos necesarios en cada subclase

Y como el método es largo podria separar en dos métodos la creacion de una persona jurídica y una persona física dado a que se conoce si la persona será fisica o jurídica ya que se settea manualmente eso y puedo reemplazar registrarUsuario(“fisico”) con registrarPersonaFisica y lo mismo con juridica y elimino el método registrarUsuario de la clase Persoonal  
  










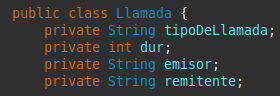
## Refactoring 7

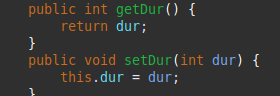
**Mal olor detectado en el código:** Nombre de atributos/métodos/clases pocos descriptivos

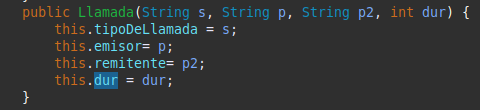
**Método de Refactoring**: Rename Method - Rename Variable - Rename Class

### Caso 1

**Extracto del código que presenta el mal olor:**

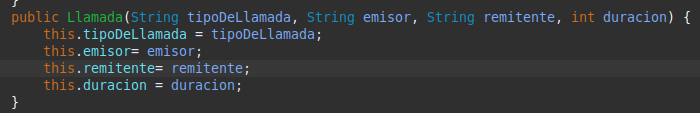
****

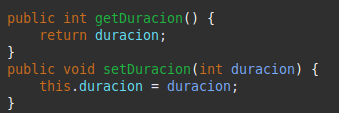
****

****

El atributo dur y sus métodos de acceso, y los parámetros del constructor, son poco descriptivos por lo cual los renombro a algo más descriptivo y actualizar las referencias a los métodos

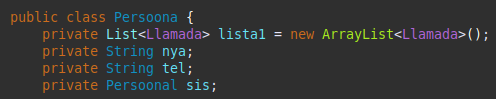


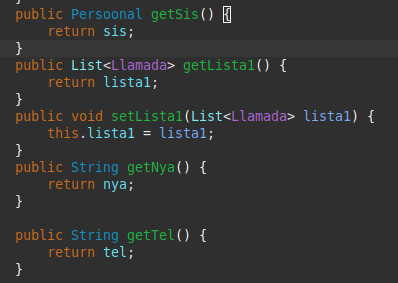




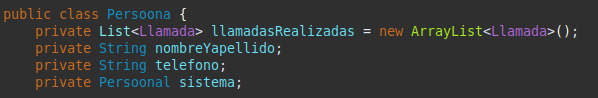
### Caso 2

**Extracto del código que presenta el mal olor:**

****

****

Los atributos y nombre de métodos son poco descriptivos por lo cual los renombro para que sean entendibles y actualizar las referencias a los métodos





### Caso 3

**Extracto del código que presenta el mal olor:**

****

****

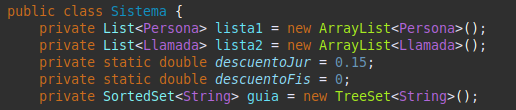
Los nombres de las clases son poco descriptivos o confusos, por lo cual las renombro, la clase Persoona que contiene información de una persona a Persona y Persoonal que tiene cosas más relevantes del sistema, a Sistema



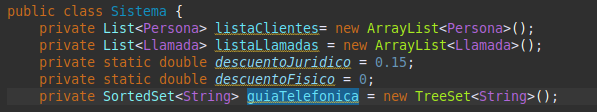


### Caso 4

**Extracto del código que presenta el mal olor:**



Los nombre de los atributos son poco descriptivos, por lo cual los renombro a algo más descriptivo y actualizar referencias



## Refactoring 8

**Mal olor detectado en el código:** Método largo

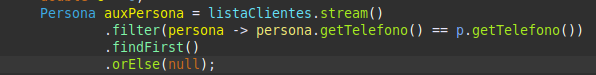
**Método de Refactoring**: Uso streams - Move Method

### Caso 1

**Extracto del código que presenta el mal olor:**



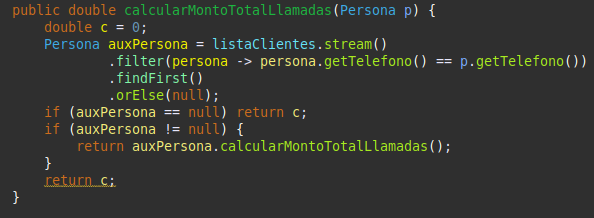
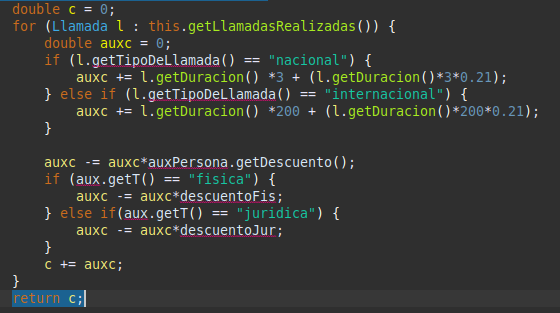
Reemplazo el primer for por un stream uniendolo a la declaración de la primer variable Persona



### Caso 2

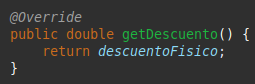
**Extracto del código que presenta el mal olor:**



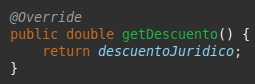
El FOR ademas de ser bastante largo, esta utilizando información de la clase Persona con el getLlamadasRealizadas, por lo cual aplicó **move method** a la clase Persona y dentro del if auxPersona != null hago la invocación al método.  
  
Metodo calcularMontoTotalLlamadas en la clase Persona  
  
Pero además se puede apreciar dos estructuras de control con condicional en el cual puedo aplicar **Replace Conditional with Polymorphism.** La primera creando dos clases que hereden de Llamada que sean LlamadaNacional y LlamadaInternacional ya que son tipos de llamada que hacen algo en base al tipo que tienen. La segunda aplicando polimorfismo con las subclases de Persona ya creadas con anterioridad, pero ahora se le debe aplicar métodos para obtener el descuento fisico y jurídico, entonces aplicaría un **extract superclass** para subir a la superclase Persona un método getDescuento dado a que ambos lo van a tener pero lo aplican de forma distinta por distintos valores

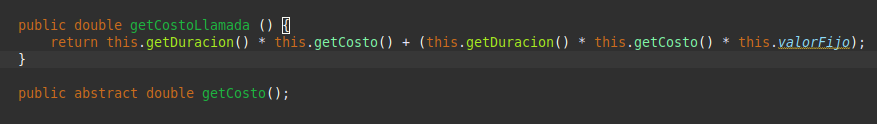
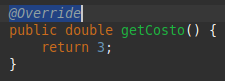
En la clase Persona  


En la clase persona fisica

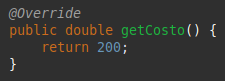


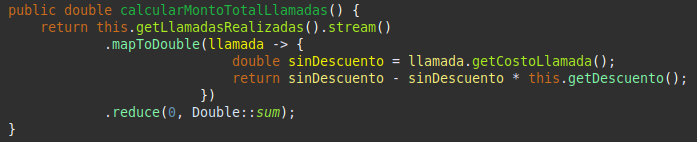
En la clase persona jurídica

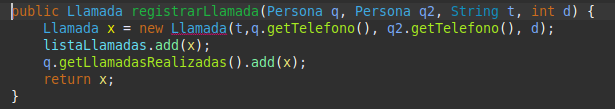


Y para la parte de Llamada nacional o internacional, se crea dos subclases de Llamada la cual lo que cambia en cada una, es el costo, ambas clases tienen la misma fórmula para calcular el costo pero cambia el valor según sea internacional o nacional, por lo cual aplique **Form Template Method** y queda así  
  
En la clase Llamada, valorFijo seria el valor de 0.21 de la formula que seria como un recargo, ambos lo comparten  
  
  
En la clase LlamadaNacional, en la fórmula, el costo de llamada nacional es de 3  


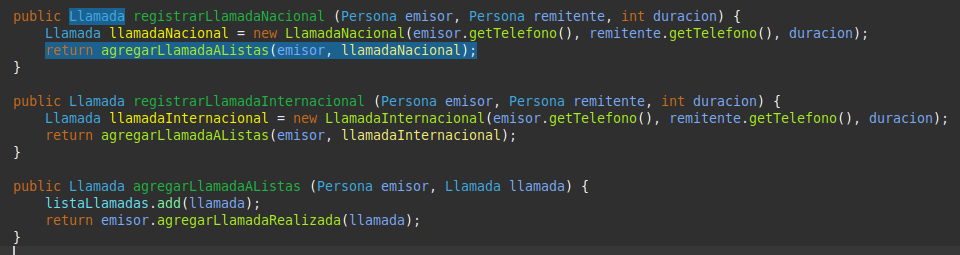
En la clase LlamadaInternacional, en la fórmula, el costo de llamada internacional es de 200



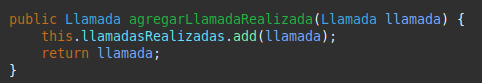
Entonces con todo esto, ahora puedo aplicar **Replace Conditional with Polymorphism** El condicional de llamada nacional o internacional, y el condicional de persona jurídica o física, y todo esto en un stream, simplificando el método a lo que queda a continuación  


Y como para registrar una llamada se utilizaba este método  


Aplicó **Extract Method** y **Move Method**, Extract para mover la instanciación a dos nuevos métodos, uno para la clase fisica y otro para la juridica y el comportamiento idéntico de ambos (agregar la instanciación a la lista de llamadas y agregar a la lista de llamada realizadas por el emisor) lo hago un método solo y le paso como parámetro la llamada



Move method para el q.getLlamadasRealizadas().add(x); dado a que es una responsabilidad mal asignada y deberá resolverlo la clase Persona el agregar a alguien a su lista



## Refactoring 9

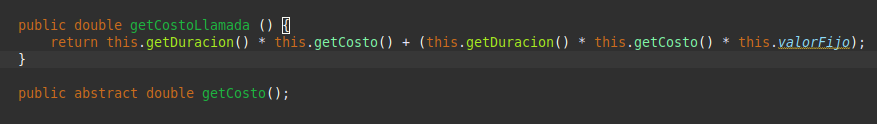
**Mal olor detectado en el código:** Métodos que no se utilizan o se utilizan solo dentro de su clase

**Método de Refactoring**: Borrar codigo innecesario

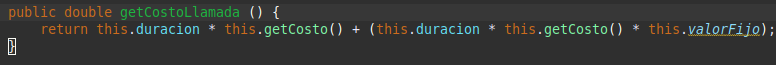
### Caso 1

**Extracto del código que presenta el mal olor:**

Clase Llamada



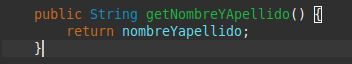
Un ejemplo seria este de arriba, el getDuracion solo se utiliza dentro de su propia clase y no lo utiliza en ninguna otra clase, por lo cual para que quede el código más limpio puedo directamente utilizar el atributo y eliminar getters que no se utilizan como getDuracion y quedando así



### Caso 2

**Extracto del código que presenta el mal olor:**

En la clase Persona



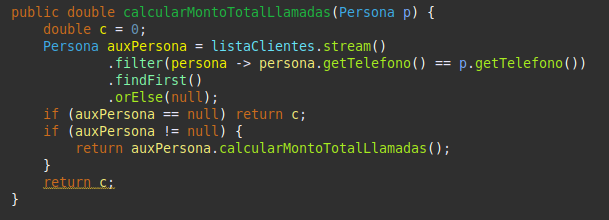
Este método nunca se utiliza, por lo cual lo puedo eliminar

### Caso 3

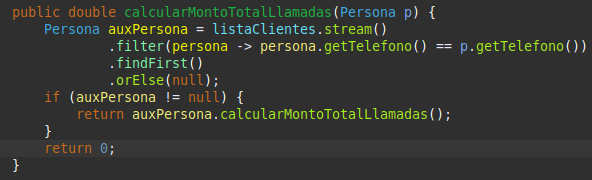
En la clase Sistema, los getters de las listas nunca se utilizan tampoco, por lo cual los elimino

### Caso 4

**Extracto del código que presenta el mal olor:**



La declaración de c es totalmente innecesaria, porque solo guarda 0 si auxPersona es null, si no guardara el retorno del método calcularMontoTotalLlamadas por lo que puedo simplificar el método



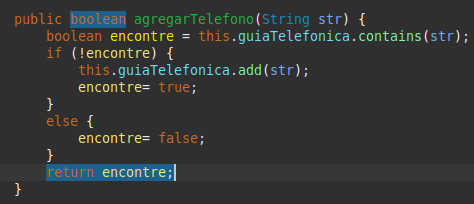
## Refactoring 10

**Mal olor detectado en el código:** Resultado de una expresión para procesar un condicional

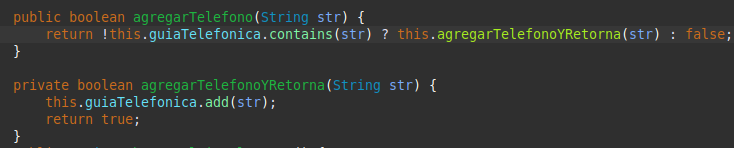
**Método de Refactoring**: Replace Temp with Query

### Caso 1

**Extracto del código que presenta el mal olor:**



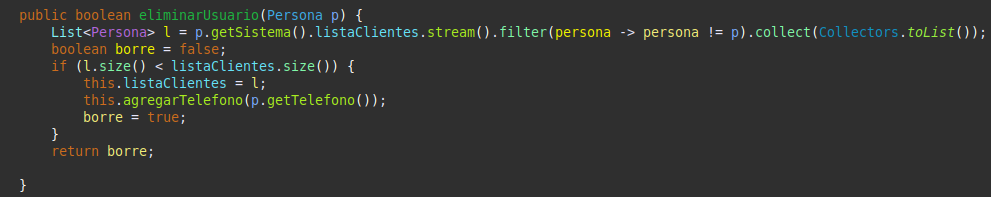
El resultado de guiaTelefonica.contains se guarda en encontré y con eso se resuelve en un if y else, pero es innecesario pudiendo utilizar un operador ternario



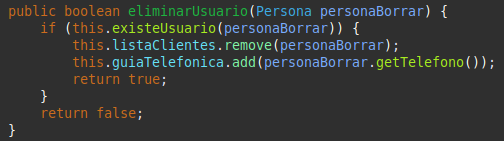
## Refactorización 11

**Mal olor detectado en el código:** Código dificil de entender

**Metodo de Refactoring**: Simplificar código  
  
**Extracto del código que presenta el mal olor:**



El sistema es uno solo, por lo cual es innecesario que cada Persona tenga asignado un sistema, por lo cual puedo borrar ese atributo y en lugar de pedirle a la Persona, su sistema y que este me de la lista de clientes, directamente desde la clase Sistema puedo pedir la lista de Clientes y si el que quiero eliminar existe en la lista o no, si existe lo borra y añade al teléfono a la lista de disponibles, y si no, retorna falso

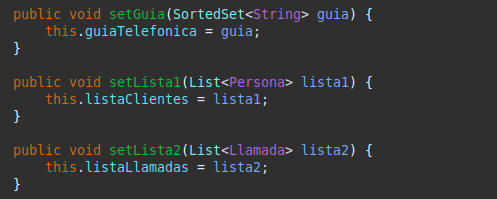


## Refactorización 12

**Mal olor detectado en el código:** Métodos setters innecesarios que no se usan

**Método de Refactoring**: Remove Setting Method

**Extracto del código que presenta el mal olor:**



Nunca se utilizan por lo cual los elimino