**Refactoring**

Entrega de ejercicio 3 - Facturación de llamadas

**Alumnos**:

* Josué Suárez, Legajo 20474/6
* Brenda Poch, Legajo 19287/0

**Nota:**

Se adjuntan a este documento los diagramas de clase UML de los escenarios inicial y final, junto con el link del repositorio que muestra el proceso de refactoring desarrollado en detalle: <https://github.com/J-Josu/facturacion_llamadas>.

**Índice**

[Refactor 1: Declaración de atributos pública](#_gjdgxs)

[Refactor 2: Mala asignación de responsabilidades](#_1fob9te)

[Refactor 3: Declaración de atributos pública](#_3znysh7)

[Refactor 4: Reinventar la rueda](#_2et92p0)

[Refactor 5: Mysterious name](#_tyjcwt)

[Refactor 6: Speculative Generality](#_3dy6vkm)

[Refactor 7: Envidia de atributos](#_4d34og8)

[Refactor 8: Dead code](#_17dp8vu)

[Refactor 9: Switch Statements](#_3rdcrjn)

[Refactor 10: Duplicated Code](#_26in1rg)

[Refactor 11: Mysterious name](#_lnxbz9)

[Refactor 12: Reinventar la rueda](#_35nkun2)

[Refactor 13: Reinventar la rueda](#_1ksv4uv)

[Refactor 14: Dead code](#_2jxsxqh)

[Refactor 15: Envidia de atributos](#_z337ya)

[Refactor 16: Long method](#_3j2qqm3)

[Refactor 17: Switch Statements](#_1y810tw)

[Refactor 18: Dead code](#_4i7ojhp)

[Refactor 19: Mysterious name](#_2xcytpi)

[Refactor 20: Mysterious name](#_3whwml4)

[Refactor 21: Envidia de atributos](#_2bn6wsx)

[Refactor 22: Speculative Generality](#_qsh70q)

[Refactor 23: Declaración de atributos sin modificador de visibilidad](#_3as4poj)

[Refactor 24: Inappropriate Intimacy](#_1pxezwc)

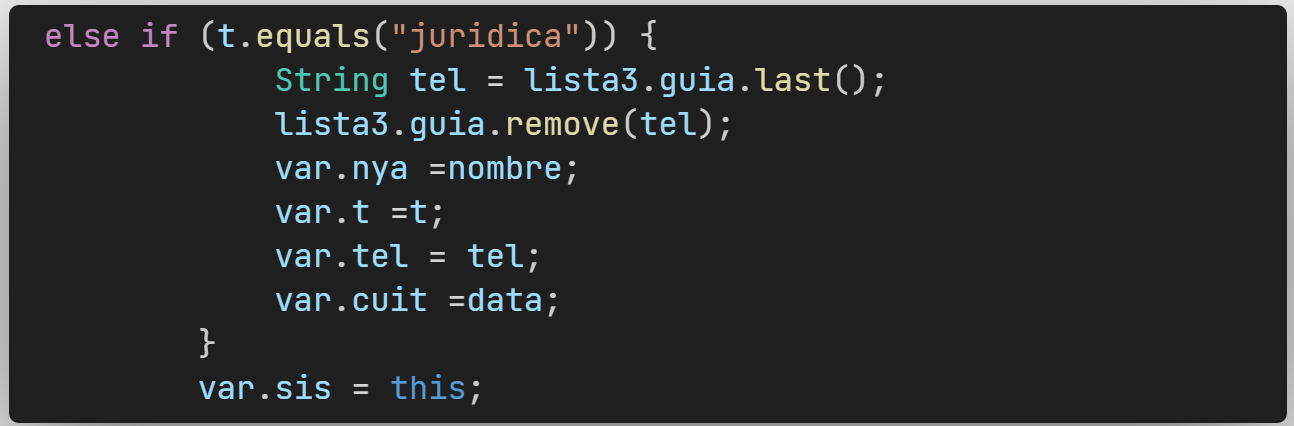
## **Refactor 1: Declaración de atributos pública**

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

Observamos declaración pública de atributos en la clase Persoona, lo cual genera ruptura del encapsulamiento.

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



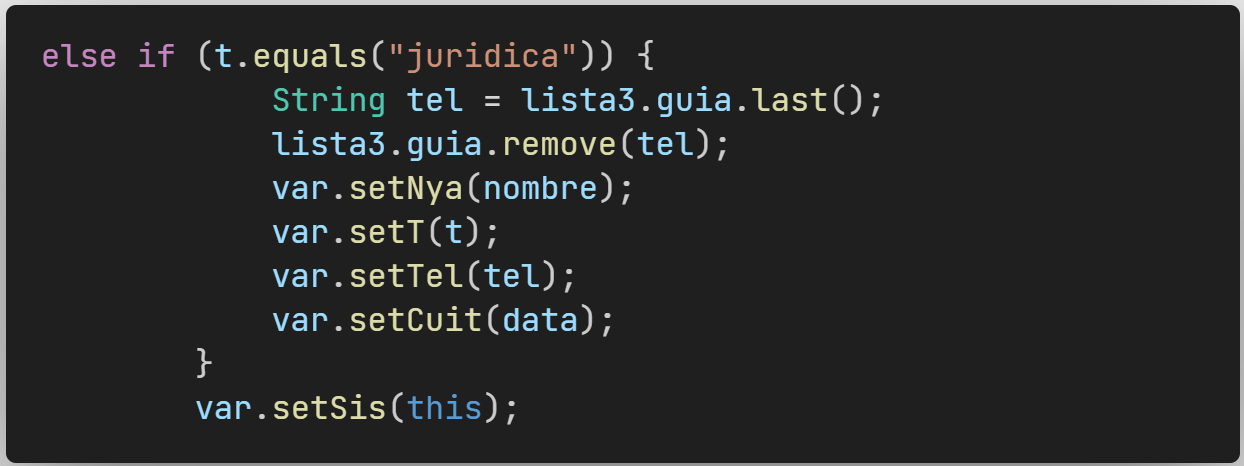


* **Refactoring a aplicar que resuelve el mal olor:**

Encapsulate field. Declaramos los atributos como privados y hacemos que se acceda a ellos sólo a través de los getters y setters.

* **Código con el refactoring aplicado:**





Ver código en paquete **refactor\_1**.

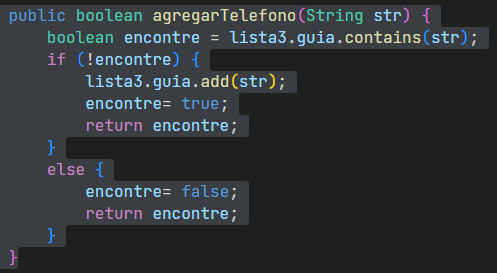
## 

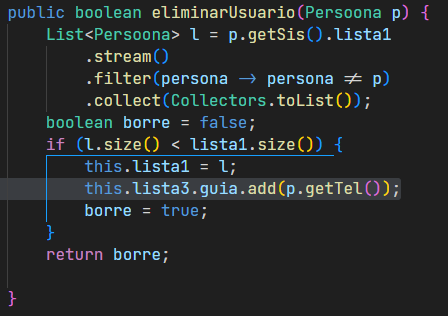
## **Refactor 2: Mala asignación de responsabilidades**

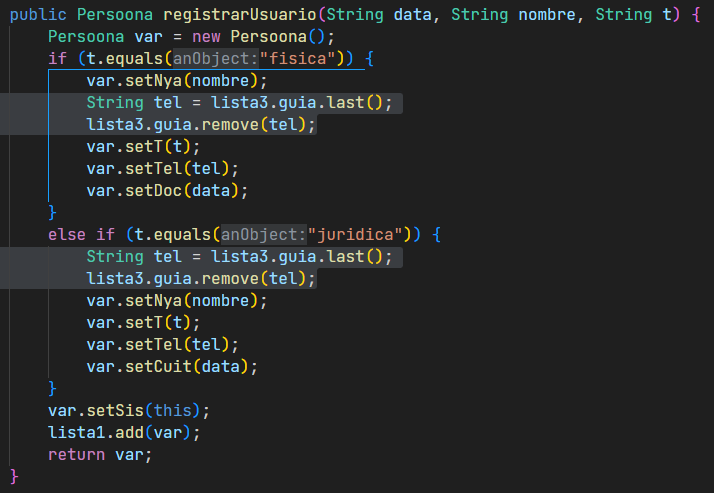
* **Mal olor detectado en el código:**

Se observa una mala asignación de responsabilidades en la clase Persoonal, asociada a una evidente envidia de atributos. Encontramos tareas en esta clase que deberían ser responsabilidad de GuiaTelefonica.

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







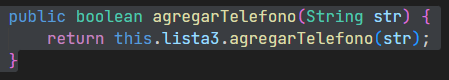
* **Refactoring a aplicar que resuelve el mal olor:**

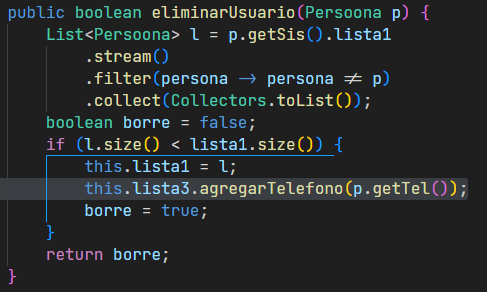
En primer lugar, realizamos un Move method del método agregarTelefono(String str) a la clase GuiaTelefonica.

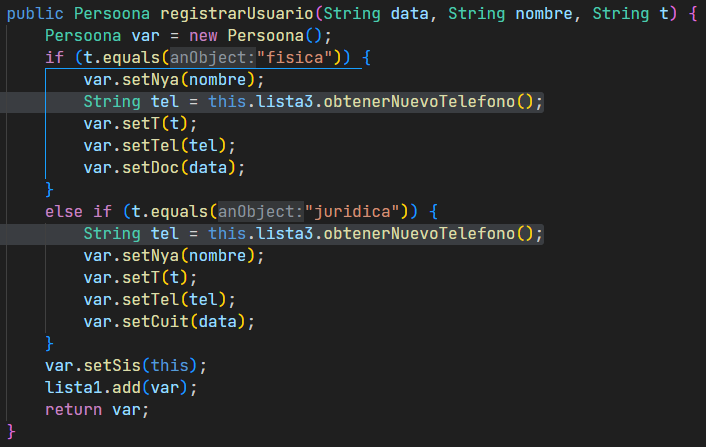
Finalmente, ejecutamos un Extract method del método obtenerNuevoTelefono() y un posterior Move method a la clase GuiaTelefonica.

* **Código con el refactoring aplicado:**









Ver paquete **refactor\_2**.

## **Refactor 3: Declaración de atributos pública**

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

Observamos declaración pública de atributos en la clase GuiaTelefonica, lo cual genera ruptura del encapsulamiento.

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



* **Refactoring a aplicar que resuelve el mal olor:**

Aplicamos el refactor encapsulate field. Declaramos el atributo como privado y eliminamos la posibilidad de que se pueda acceder a él desde otra clase.

* **Código con el refactoring aplicado:**



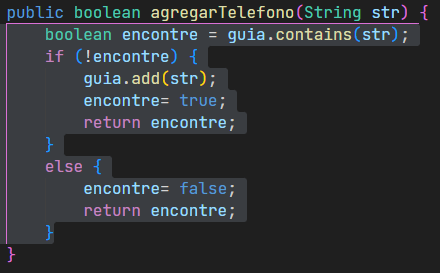
Ver código en paquete **refactor\_3**.

## **Refactor 4: Reinventar la rueda**

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

Observamos que el método agregarTelefono(String str) de GuiaTelefonica realiza la lógica de agregar el String que le llega como parámetro a la variable guía y devolver true o false de acuerdo a si el String pudo ser agregado o no. Estamos trabajando con una colección que tiene el método boolean add(E e) en su protocolo. Utilizando únicamente este método estaríamos realizando esta lógica una sola vez y de manera más resumida.

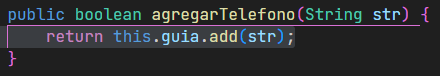
* **Extracto del código que presenta el mal olor:**



* **Refactoring a aplicar que resuelve el mal olor:**

Reemplazamos el algoritmo (Substitute Algorithm) con la implementación ya existente del método add(E e) en la estructura de datos utilizada.

* **Código con el refactoring aplicado:**



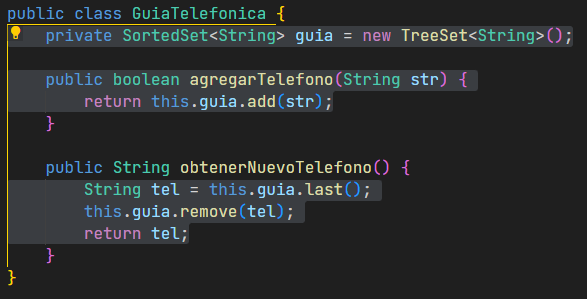
Ver código en paquete **refactor\_4**.

## **Refactor 5: Mysterious name**

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

Observamos que, dentro de la clase GuiaTelefonica, numerosos identificadores de variables de instancia y métodos no describen apropiadamente aquello a lo que hacen referencia.

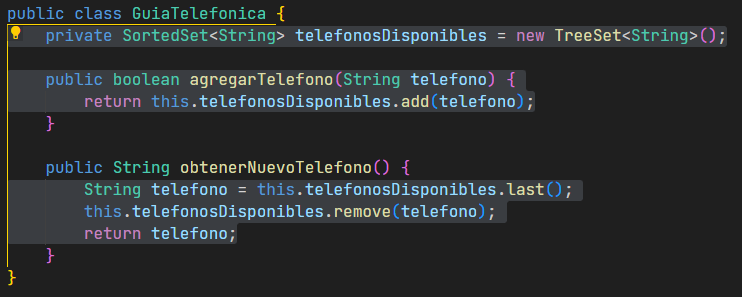
* **Extracto del código que presenta el mal olor:**



* **Refactoring a aplicar que resuelve el mal olor:**

Realizamos los siguientes renombres de los identificadores que consideramos no representativos:

* La variable de instancia guia es renombrada como telefonosDisponibles.
* El parámetro str del método agregarTelefono() es renombrado como telefono.
* La variable temporal del método obtenerNuevoTelefono() es renombrada como telefono.
* **Código con el refactoring aplicado:**

****

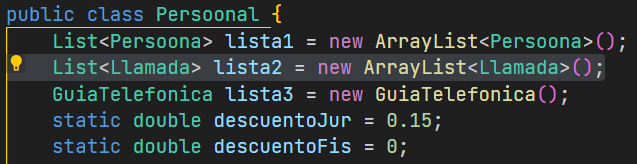
Ver código en paquete **refactor\_5**.

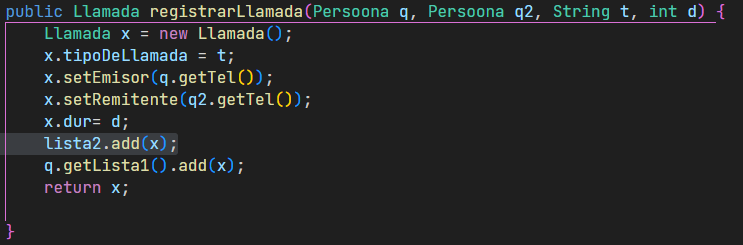
## **Refactor 6: Speculative Generality**

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

Observamos que la clase Persoonal posee la variable de instancia lista2, la cual sólo se utiliza para llevar un registro de las llamadas efectuadas por todos los clientes. Esta variable no es usada, consultada ni transformada en ningún momento. Esto da indicios de que fue incorporada para una posible utilidad futura. El llevar este registro genera un acoplamiento directo entre las clases Llamada y Persoonal que resulta innecesario en el alcance del sistema.

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

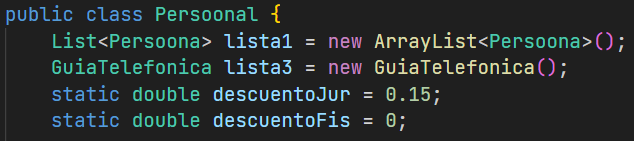


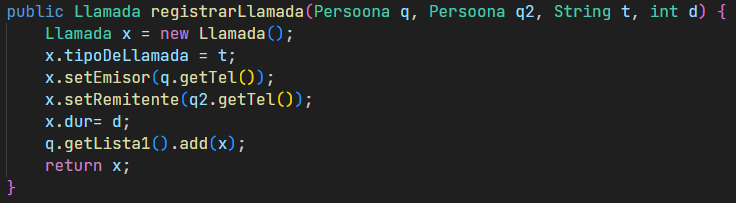


* **Refactoring a aplicar que resuelve el mal olor:**

Se decide efectuar el borrado de la variable de instancia lista2 de la clase Persoonal (Remove dead code).

* **Código con el refactoring aplicado:**

****



Ver código en paquete **refactor\_6**.

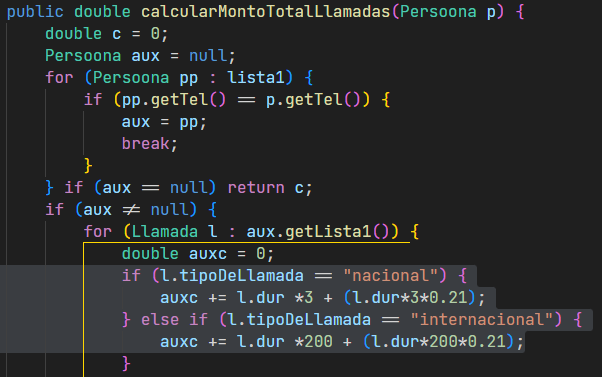
## **Refactor 7: Envidia de atributos**

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

Observamos que la clase Persoonal, en sus métodos registrarLlamada() y calcularMontoTotalLlamadas(), hace uso de las variables de instancia de la clase Llamada. Esta práctica, conocida como envidia de atributos, genera alto acoplamiento entre las clases mencionadas. Asimismo, el hecho de que Persoonal opere con las variables de instancia de Llamada en lugar de delegar las tareas a dicha clase, ocasiona que ésta última sea una DataClass, al simplemente mantener un estado interno sin mostrar comportamiento.

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

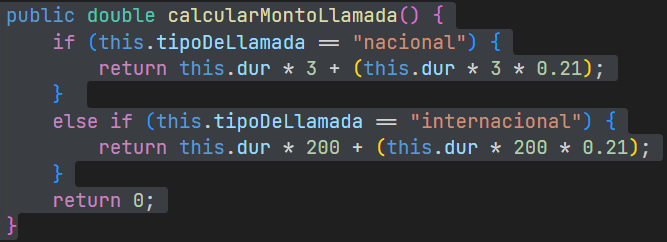


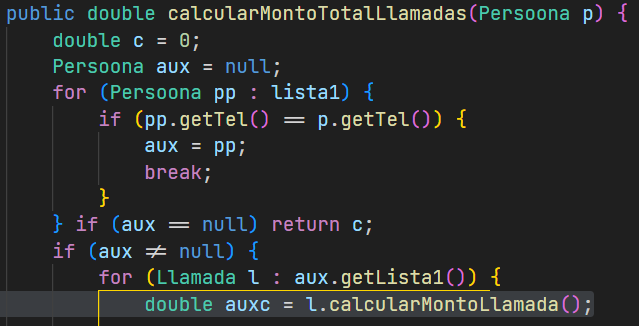


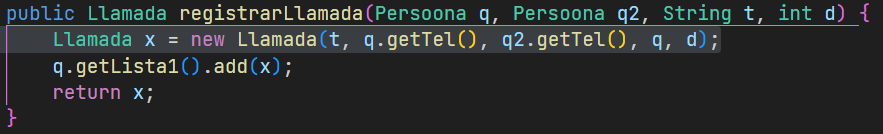
* **Refactoring a aplicar que resuelve el mal olor:**

Para resolver este mal olor realizamos una serie de refactors que se listan a continuación:

* Extract method del cálculo del monto de una llamada en sí misma, independientemente de quién la efectúa (calcularMontoLlamada()).
* Move method del método calcularMontoLlamada() a la clase Llamada.
* Uso del constructor existente en la clase Llamada en lugar de inicializar dicho objeto desde la clase Persoonal.
* **Código con el refactoring aplicado:**







Ver código en paquete **refactor\_7**.

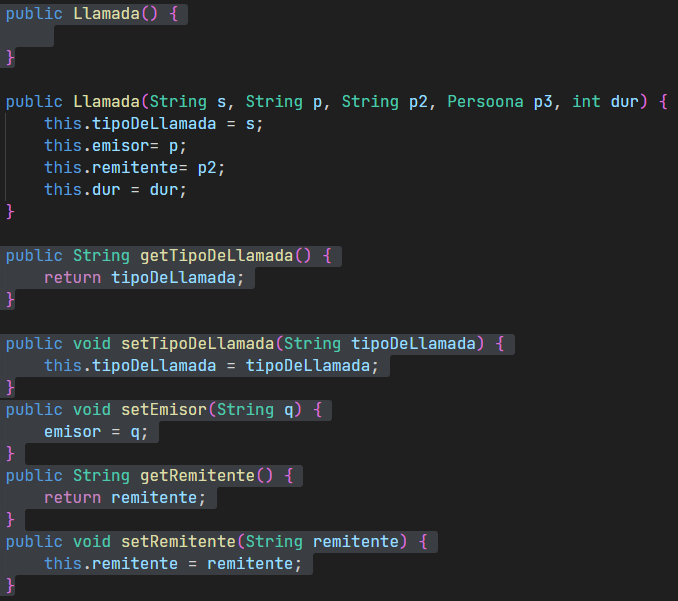
## 

## **Refactor 8: Dead code**

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

Debido a que las llamadas ya no se inicializan desde la clase Persoonal, el constructor vacío de dicha clase queda obsoleto, al igual que sus métodos setters. Asimismo, el hecho de que ahora Llamada calcule su propio monto en el método calcularMontoLlamada() hace que también queden obsoletos los métodos getters.

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



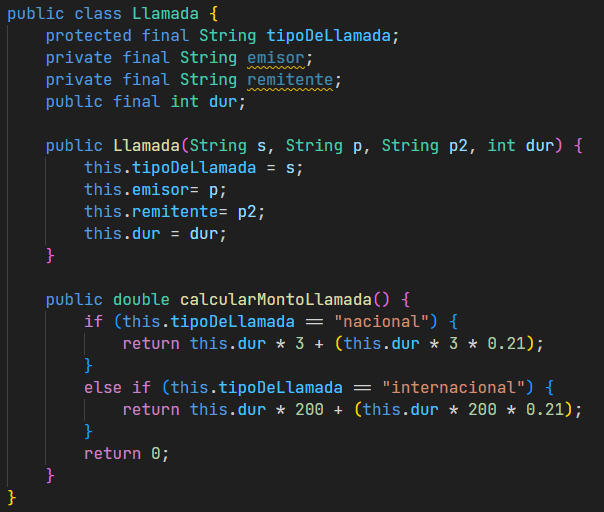
* **Refactoring a aplicar que resuelve el mal olor:**

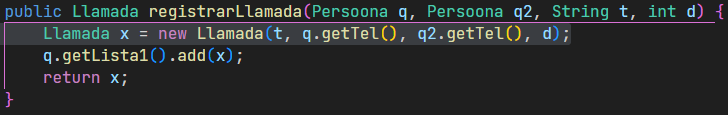
Para resolver este mal olor realizamos la eliminación de los siguientes fragmentos de código:

* Constructor vacío de la clase Llamada.
* Parámetro p3 del constructor con parámetros de la clase Llamada (Remove parameter)
* Setters no utilizados de la clase Llamada (Remove setting method)
* Getters no utilizados de la clase Llamada.

Con los cambios aplicados, los atributos de la clase Llamada son inicializados únicamente en el constructor. Al no existir la posibilidad de alterarlos con algún método setter, se les añade el modificador final.

* **Código con el refactoring aplicado:**





Ver código en paquete **refactor\_8**.

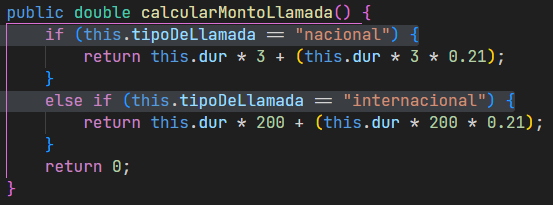
## **Refactor 9: Switch Statements**

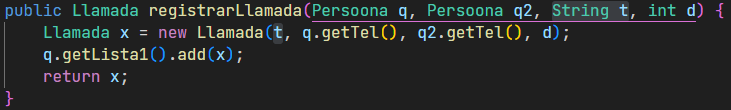
* **Mal olor detectado en el código:**

En la clase Llamada observamos que se está utilizando una variable de instancia (tipoDeLlamada) para definir el comportamiento del método calcularMontoLlamada(), dependiendo de su valor (Switch statement). Esto dificulta modificaciones y extensiones futuras en el código.

Asimismo, advertimos que los distintos tipos de llamada sólo afectan una parte del cálculo del monto de cada llamada. Esto da indicios de que, con una jerarquía donde se englobe el comportamiento general en la clase padre y se delegue el comportamiento específico en sus subclases, se podría hacer uso del polimorfismo en la decisión del comportamiento específico a tomar para cada tipo de llamada.

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



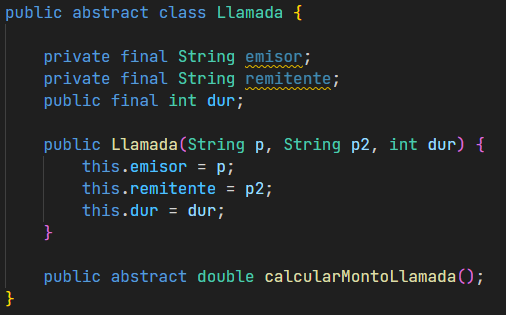


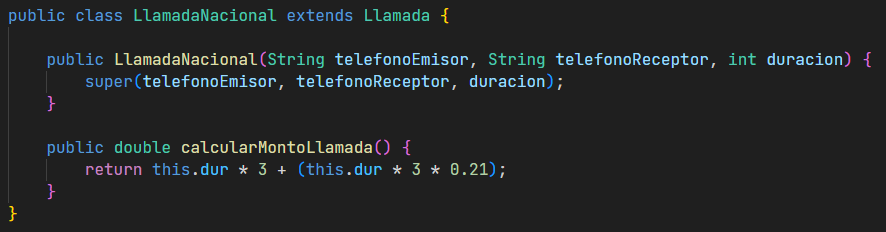
* **Refactoring a aplicar que resuelve el mal olor:**

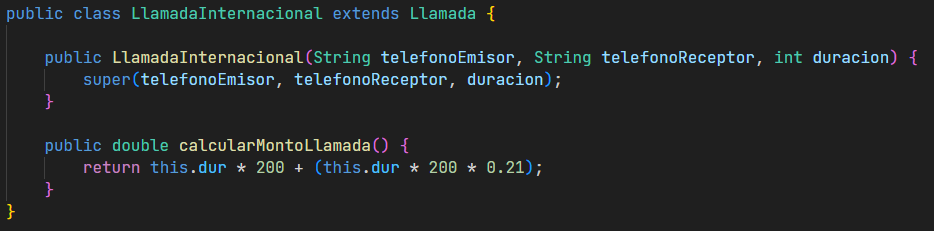
En primer lugar, aplicamos el refactor Replace type code with subclasses y adaptamos el sistema al uso de las nuevas subclases, creadas a partir de cada valor del tipo codificado: LlamadaNacional y LlamadaInternacional. Extraemos el comportamiento particular de la clase original a cada una de estas subclases y reemplazamos el código de flujo de control por polimorfismo. La clase Llamada pasa a ser abstracta porque ya no es necesario instanciarla.

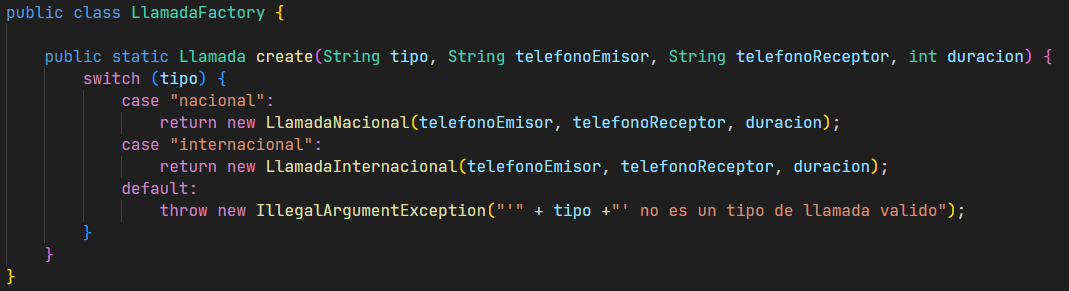
Como parte de la adaptación del sistema al uso de las nuevas subclases, y con la intención de evitar la modificación del protocolo público, se decidió crear un método factory (Factory method), delegando en él la responsabilidad de definir qué subclase instanciar dependiendo de los parámetros recibidos. A fin de reducir la cantidad de responsabilidades de la clase Llamada, se decidió extraer este método a la nueva clase LlamadaFactory, cuya única responsabilidad es la de instanciar los diferentes tipos de llamada. En el caso de necesitar incorporar nuevos tipos de llamada a futuro, sólo se necesitaría crear la nueva subclase y actualizar el método creador de la clase LlamadaFactory.

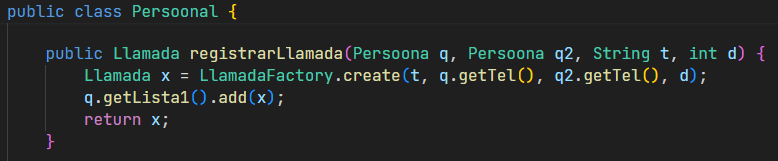
* **Código con el refactoring aplicado:**











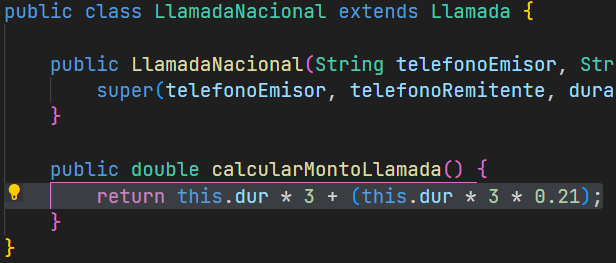
Ver código en paquete **refactor\_9**.

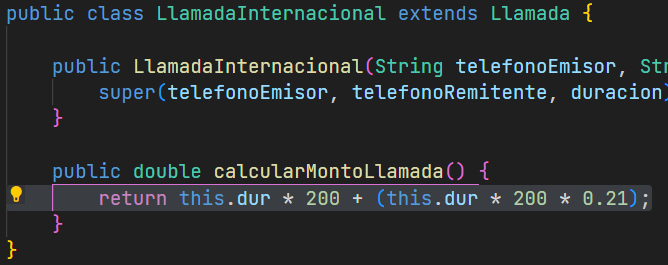
## **Refactor 10: Duplicated Code**

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

En las clases LlamadaNacional y LlamadaInternacional observamos que el método calcularMontoLlamada() posee el mismo algoritmo, pero con pequeñas variaciones en ambas clases (Duplicated code). Asimismo, notamos valores harcodeados que sería conveniente incorporar en alguna variable de clase para que cambios futuros no requieran modificar varios puntos del código.

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





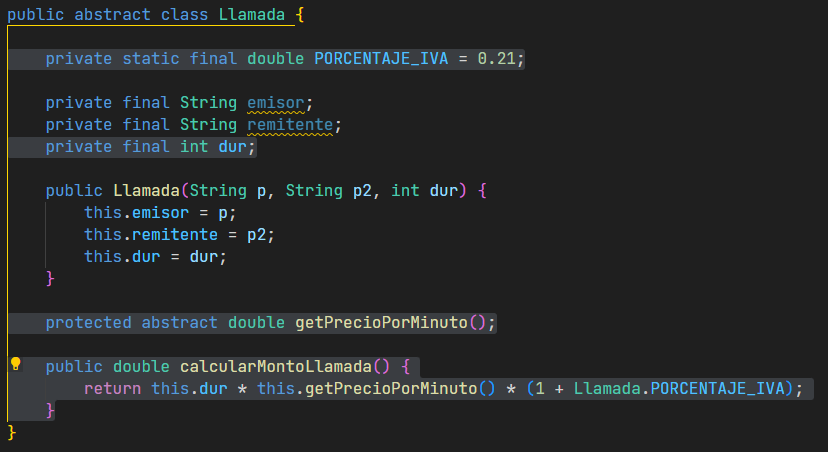
* **Refactoring a aplicar que resuelve el mal olor:**

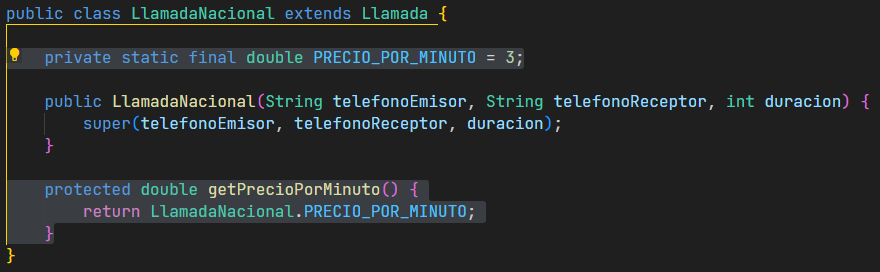
Aplicamos el refactor Form Template Method. Para ello, se exrajeron los valores 200 y 3 como variables de clase (Replace Magic Number with Symbolic Constant), accediendo a sus valores a través del método protegido getPrecioPorMinuto() y se aplicó un Pull up method del método calcularMontoLlamada().

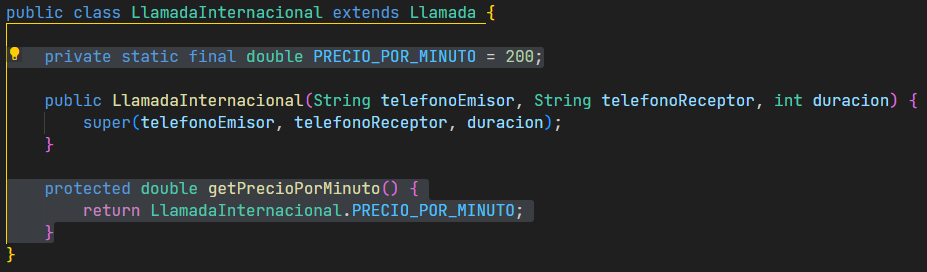
Adicionalmente, se extrajo el valor hardcodeado 0.21 a una variable de clase (Replace Magic Number with Symbolic Constant) y se aplicó el refactor Substitute algorithm para simplificar la forma en que se aplica el porcentaje de IVA.

Finalmente, se cambió el modificador de visibilidad de la variable dur a private, para corregir la ruptura de encapsulamiento producida por la declaración pública del atributo (Encapsulate field).

* **Código con el refactoring aplicado:**







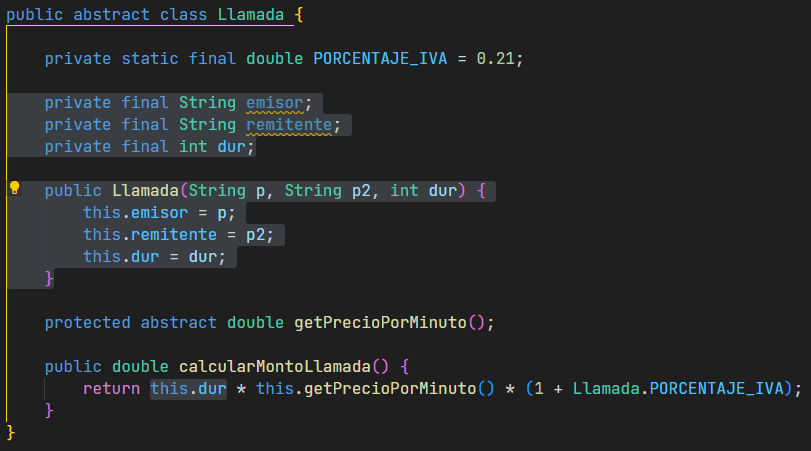
Ver código en paquete **refactor\_10**.

## **Refactor 11: Mysterious name**

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

Observamos que, dentro de la clase Llamada, numerosos identificadores de variables de instancia y parámetros no describen apropiadamente aquello a lo que hacen referencia.

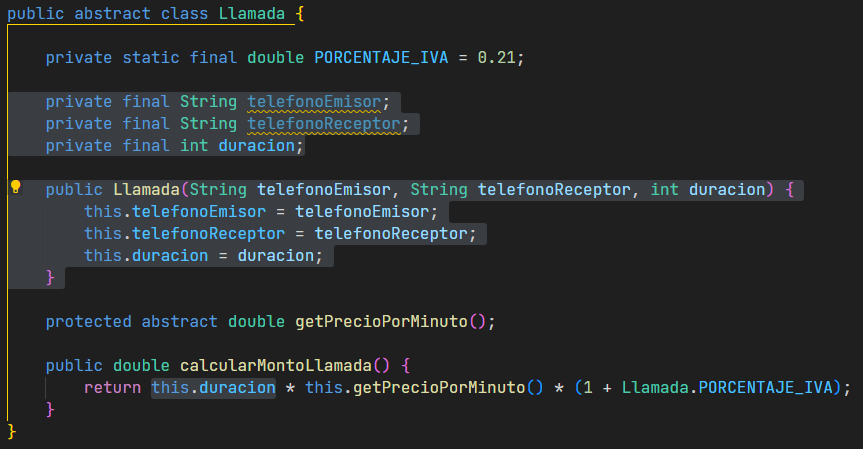
* **Extracto del código que presenta el mal olor:**



* **Refactoring a aplicar que resuelve el mal olor:**

Realizamos los siguientes renombres de los identificadores que consideramos no representativos:

* Las variables de instancia dur, emisor y remitente son renombradas como duracion, telefonoEmisor y telefonoReceptor, respectivamente.
* Los parámetros p, p2 y dur del constructor son renombrados como telefonoEmisor, telefonoReceptor y duracion, respectivamente.
* **Código con el refactoring aplicado:**



Ver código en paquete **refactor\_11**.

## **Refactor 12: Reinventar la rueda**

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

Observamos que el método eliminarUsuario() de la clase Persoonal, para lograr su propósito, está implementando una lógica que puede ser reemplazada por métodos ya existente en la clase (existeUsuario()) y en el protocolo de la colección de Persoona (remove()).

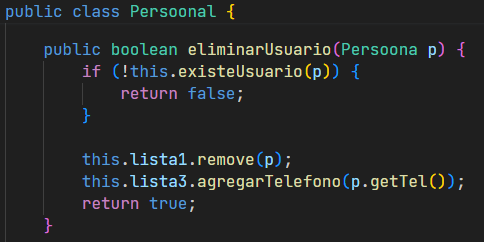
* **Extracto del código que presenta el mal olor:**



* **Refactoring a aplicar que resuelve el mal olor:**

Aplicamos el refactor Substitute Algorithm, utilizando la estrategia de Guard Clause en conjunto con el uso del método ya existente, existeUsuario(), de la clase Persoonal. Adicionalmente, empleamos el método remove(Object o) de la interfaz de la estructura de datos utilizada para evitar recrear la estructura manualmente.

* **Código con el refactoring aplicado:**



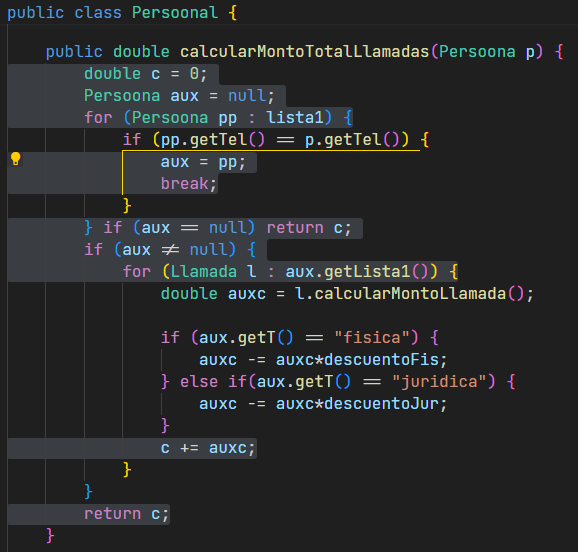
Ver código en paquete **refactor\_12**.

## **Refactor 13: Reinventar la rueda**

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

Observamos que el método calcularMontoTotalLlamadas() de la clase Persoonal utiliza bucles for para operar con estructuras de datos. En Java existe la API Stream que opera con estructuras de datos de una manera más legible y con menor posibilidad de cometer errores.

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

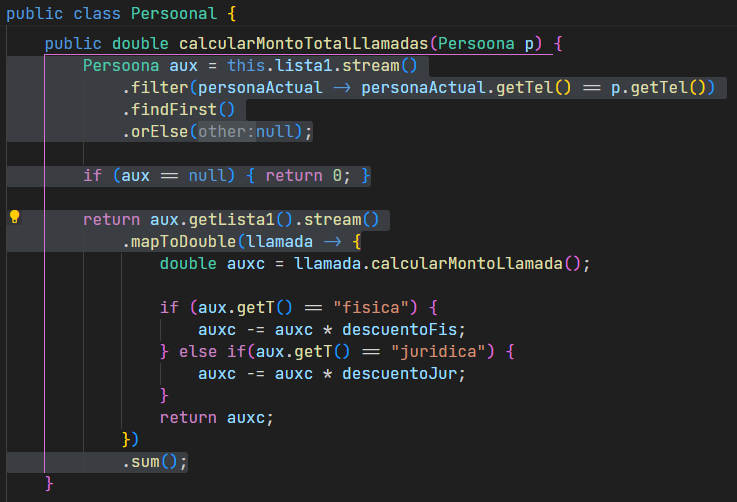


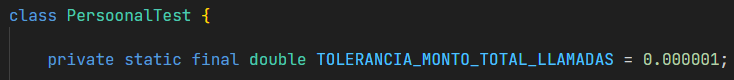
* **Refactoring a aplicar que resuelve el mal olor:**

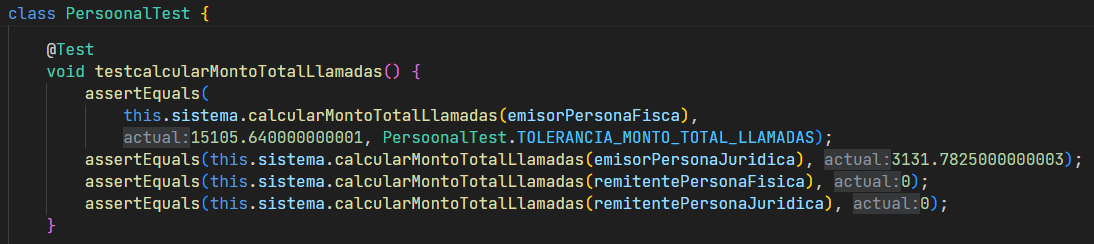
Aplicamos el refactor Substitute Algorithm para operar con estructuras de datos utilizando la API Stream en lugar de bucles for y utilizamos la estrategia de Guard Clause para mejorar la legibilidad del código.

*Aclaración: Al correr los tests, luego de la aplicación de este refactor, se advirtió que, debido errores de redondeo y representación interna ocurridos en el manejo de números de punto flotante, algunos de los assertEquals presentaban diferencias mínimas entre el valor esperado y el valor obtenido. Por esta razón, se modificaron los assertEquals() que presentaban estos errores por su equivalente con un tercer parámetro que contempla la tolerancia de la igualdad.*

* **Código con el refactoring aplicado:**







Ver código en paquete **refactor\_13**.

## 

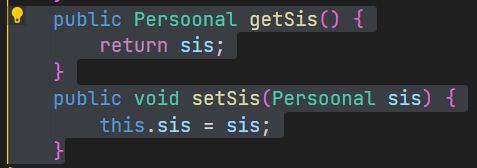
## **Refactor 14: Dead code**

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

Observamos que en el método registrarUsuario() de la clase Persoonal se setea la variable sis de Persona. Al haberse modificado la lógica del método eliminarUsuario() en el refactor\_12, ya no es necesaria la existencia de esta variable y, por lo tanto, la relación de conocimiento bidireccional entre Persoonal y Persoona, que produce un alto acoplamiento entre ambas clases.

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



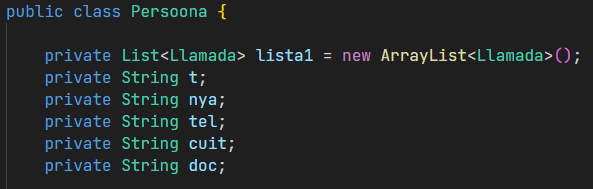


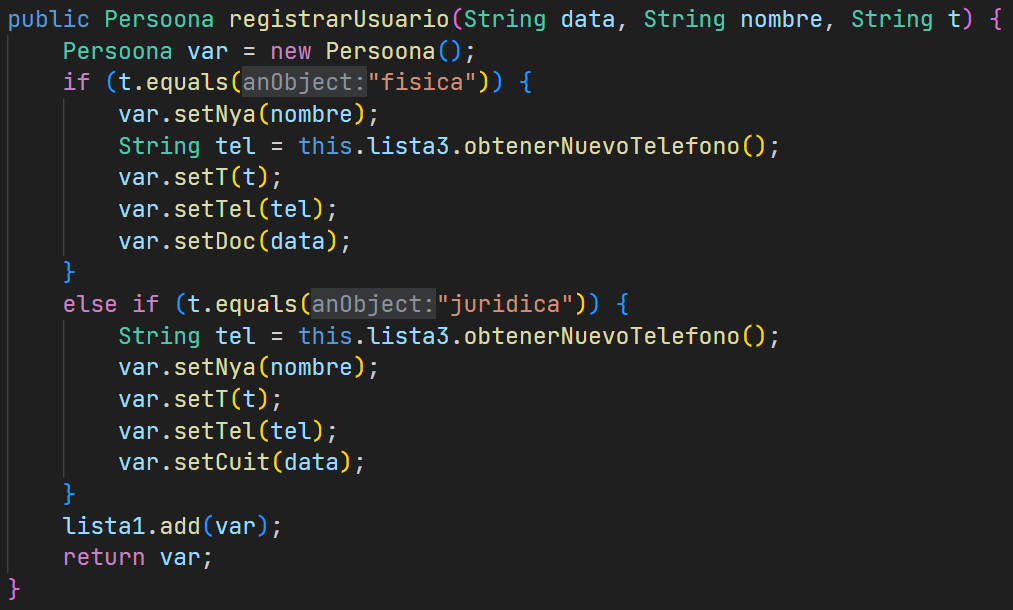


* **Refactoring a aplicar que resuelve el mal olor:**

Aplicamos el refactor Change Bidirectional Association to Unidirectional para el cual realizamos la eliminación de los siguientes fragmentos de código:

* Setteo de la variable sis en el método registrarUsuario() de la clase Persoonal.
* setSis() no utilizado de la clase Persoona (Remove setting method).
* getSis() no utilizado de la clase Persoona.
* Variable de instancia sis de la clase Persoona.
* **Código con el refactoring aplicado:**





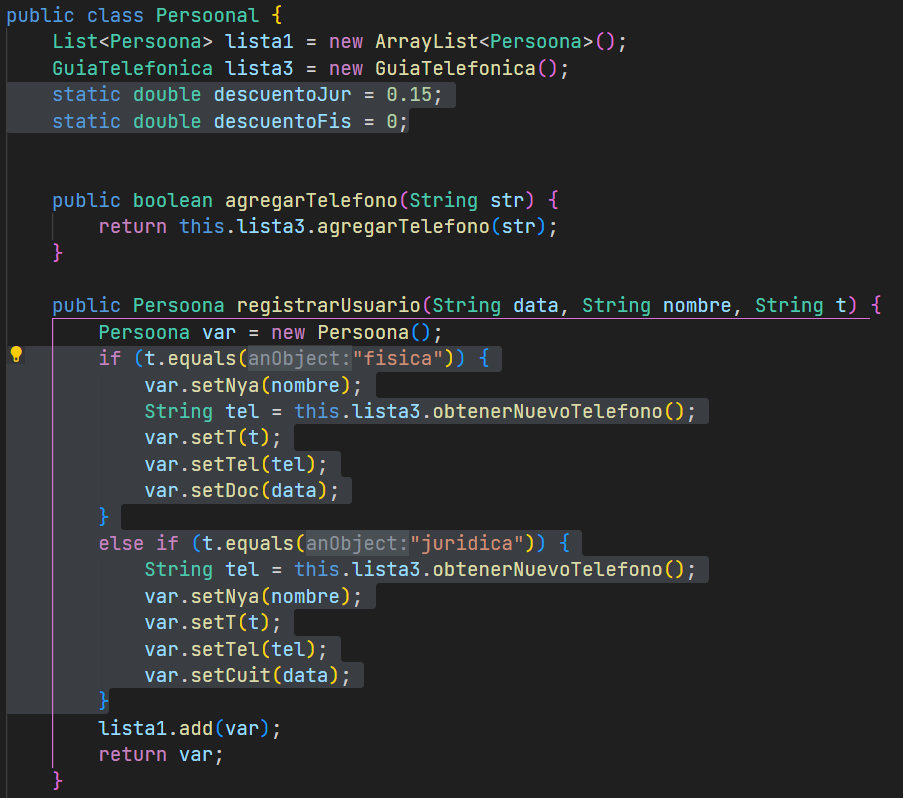
Ver código en paquete **refactor\_14**.

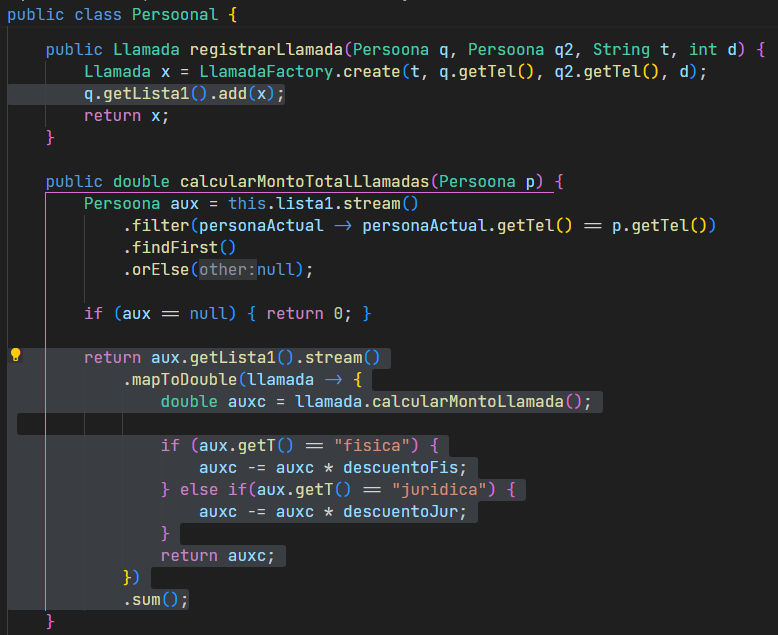
## **Refactor 15: Envidia de atributos**

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

Observamos que la clase Persoonal, en sus métodos registrarUsuario(), calcularMontoTotalLlamadas(), registrarLlamadaNacional() y registrarLlamadaInternaciona() hace uso de las variables de instancia de la clase Persoona. Esta práctica, conocida como envidia de atributos, genera alto acoplamiento entre las clases mencionadas. Asimismo, el hecho de que Persoonal opere con las variables de instancia de Persoona, en lugar de delegar las tareas a dicha clase, ocasiona que esta última sea una DataClass, al simplemente mantener un estado interno sin mostrar comportamiento.

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

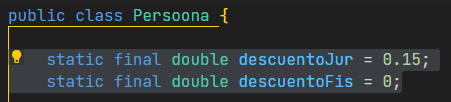


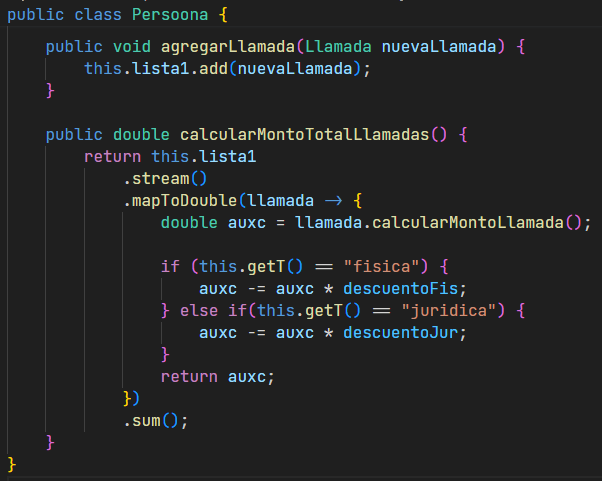


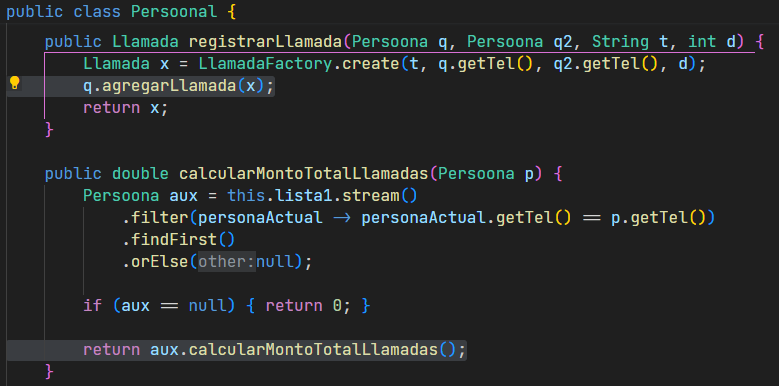
* **Refactoring a aplicar que resuelve el mal olor:**

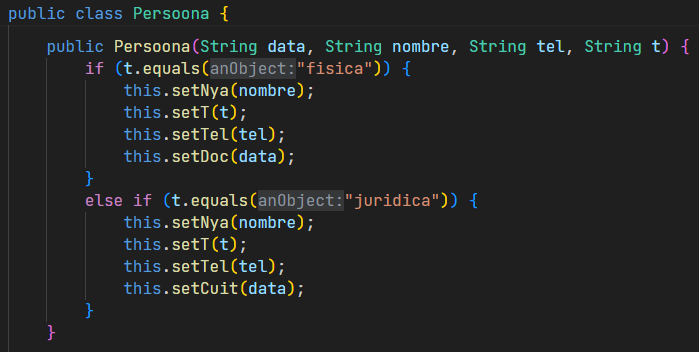
Para resolver este mal olor realizamos una serie de refactors que se listan a continuación:

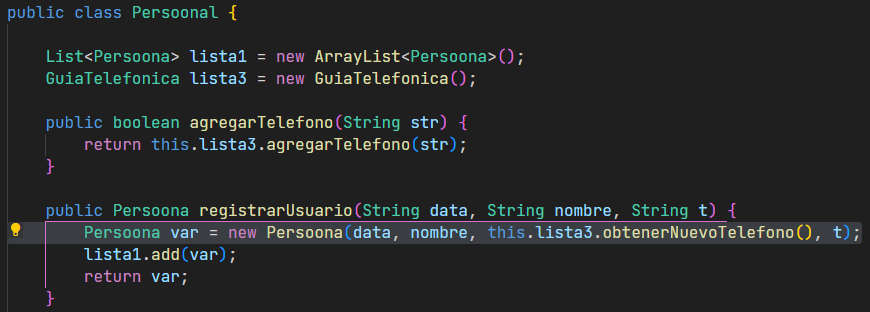
* Extract method del cálculo del monto de todas las llamadas realizadas por una persona (calcularMontoTotalLlamadas()).
* Move method del método calcularMontoTotalLlamadas() a la clase Persoona. Como consecuencia de este move method, se realizó un Move field de los campos descuentoJur y descuentoFis para mantener dichas variables en la clase en que son utilizadas.
* Extract method del agregado de la llamada registrada a la lista de llamadas del emisor (agregarLlamada()).
* Move method del método agregarLlamada() a la clase Persoona.
* Creación de un constructor con parámetros en la clase Persoona para realizar su inicialización desde esta misma clase en lugar de hacerlo desde la clase Persoonal.
* **Código con el refactoring aplicado:**











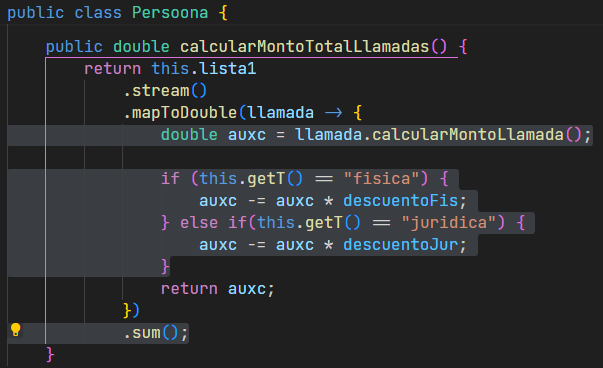
Ver código en paquete **refactor\_15**.

## **Refactor 16: Long method**

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

En la clase Persoona observamos que el método calcularMontoTotalLlamada() posee varias responsabilidades que podrían delegarse en métodos diferentes para mejorar la legibilidad del código y respetar el principio de Single responsibility.

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



* **Refactoring a aplicar que resuelve el mal olor:**

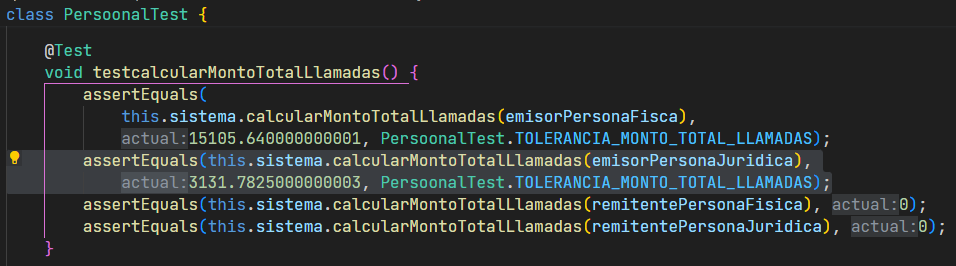
Aplicamos el refactor Extract method (getDescuento()) para delegar en este nuevo método la responsabilidad de definir el descuento a aplicar según el tipo de Persona del que se trate. Adicionalmente, realizamos una simplificación de la forma en que se realiza la aplicación del descuento (Substitute Algorithm).

Finalmente, extrajimos en un método privado la manera de calcular el monto para una llamada individual (computarMontoLlamada()).

*Aclaración: Al correr los tests, luego de la aplicación de este refactor, se advirtió que, debido errores de redondeo y representación interna ocurridos en el manejo de números de punto flotante, algunos de los assertEquals presentaban diferencias mínimas entre el valor esperado y el valor obtenido. Por esta razón, se modificaron los assertEquals() que presentaban estos errores por su equivalente con un tercer parámetro que contempla la tolerancia de la igualdad.*

* **Código con el refactoring aplicado:**





Ver código en paquete **refactor\_16**.

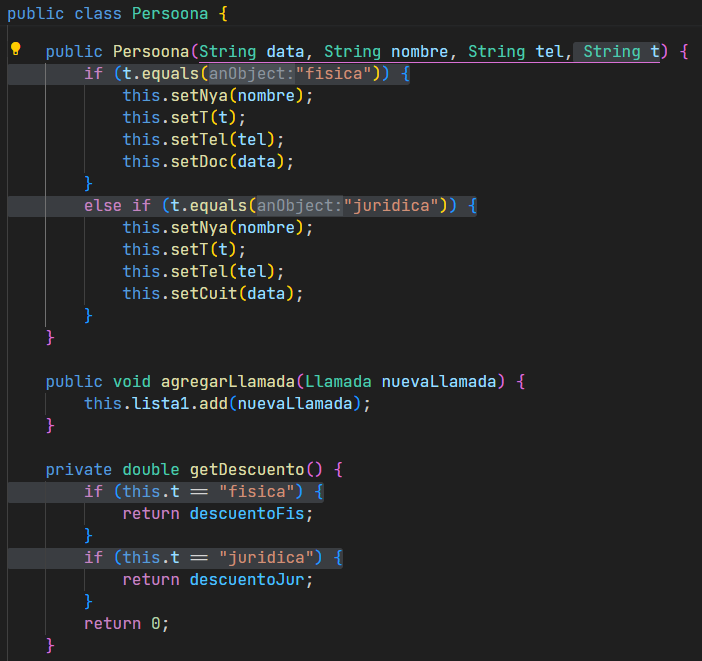
## **Refactor 17: Switch Statements**

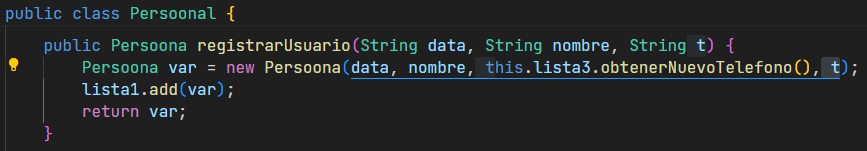
* **Mal olor detectado en el código:**

En la clase Persoona observamos que se está utilizando una variable de instancia (t) para definir el comportamiento del constructor y del método getDescuento() dependiendo de su valor (Switch statement). Esto dificulta modificaciones y extensiones futuras en el código.

Asimismo, advertimos que los distintos tipos de personas se diferencian en una única variable de instancia (doc vs. cuit) y en el comportamiento del método getDescuento(). Esto da indicios de que, con una jerarquía donde se englobe el comportamiento general en la clase padre y se delegue el comportamiento específico en sus subclases, se podría hacer uso del polimorfismo en la decisión del comportamiento específico a tomar para cada tipo de persona.

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





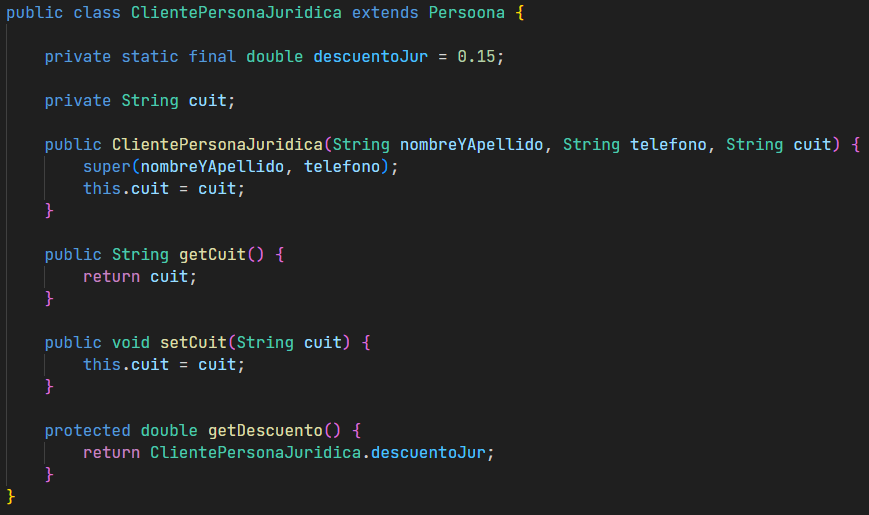
* **Refactoring a aplicar que resuelve el mal olor:**

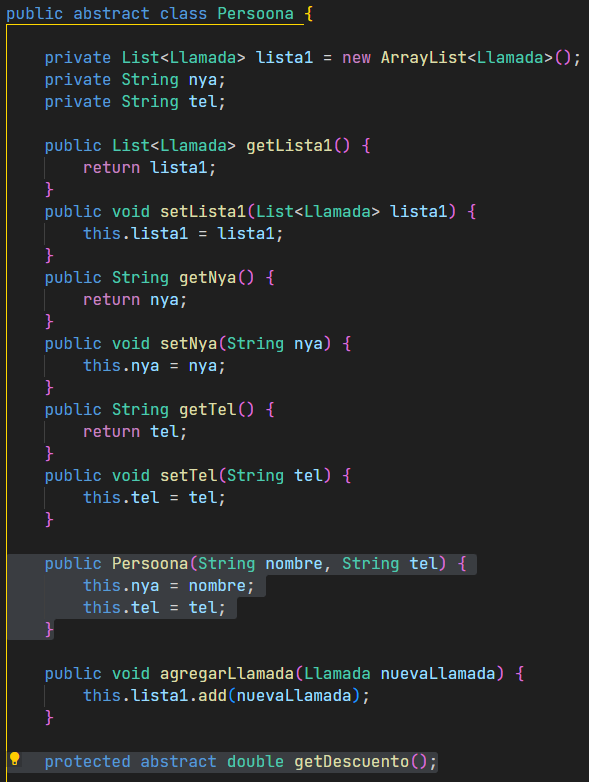
En primer lugar, aplicamos el refactor Replace type code with subclasses y adaptamos el sistema al uso de las nuevas subclases, creadas a partir de cada valor del tipo codificado: ClientePersonaFisica y ClientePersonaJuridica. Extraemos el comportamiento particular de la clase original a cada una de estas subclases, junto con las variables de clase descuentoFis y descuentoJur (Push Down Field), y reemplazamos el código de flujo de control por polimorfismo. Adaptamos los modificadores de visibilidad de las variables de clase mencionadas y del método getDescuento() para acentuar el encapsulamiento. La clase Persoona pasa a ser abstracta porque ya no es necesario instanciarla.

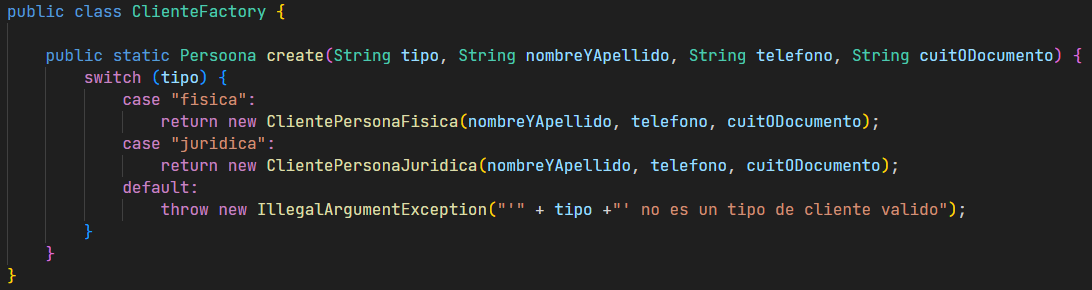
Como parte de la adaptación del sistema al uso de las nuevas subclases, y con la intención de evitar la modificación del protocolo público, se decidió crear un método factory (Factory method), delegando en él la responsabilidad de definir qué subclase instanciar dependiendo de los parámetros recibidos. A fin de reducir la cantidad de responsabilidades de la clase Persoona, se decidió extraer este método a la nueva clase ClienteFactory, cuya única responsabilidad es la de instanciar los diferentes tipos de clientes. En el caso de necesitar incorporar nuevos tipos de clientes a futuro, sólo se necesitaría crear la nueva subclase y actualizar el método creador de la clase ClienteFactory.

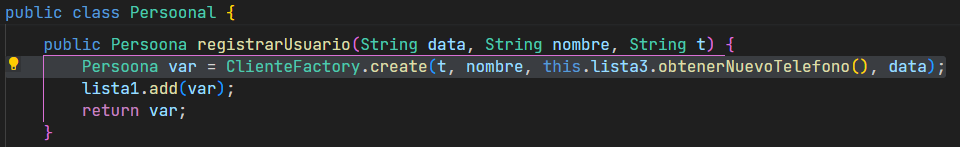
* **Código con el refactoring aplicado:**











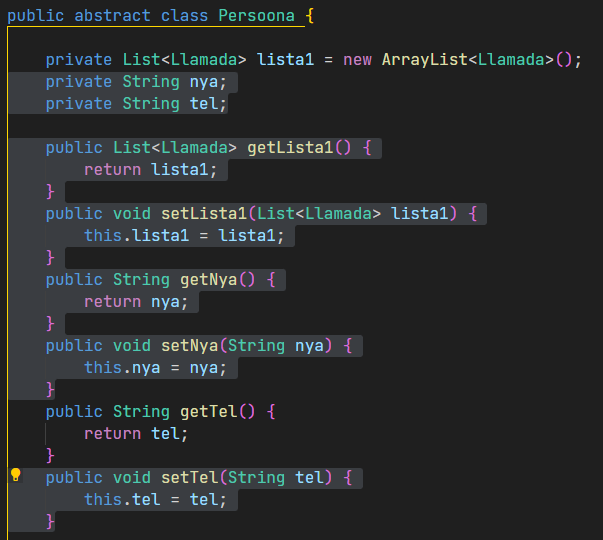
Ver código en paquete **refactor\_17**.

## **Refactor 18: Dead code**

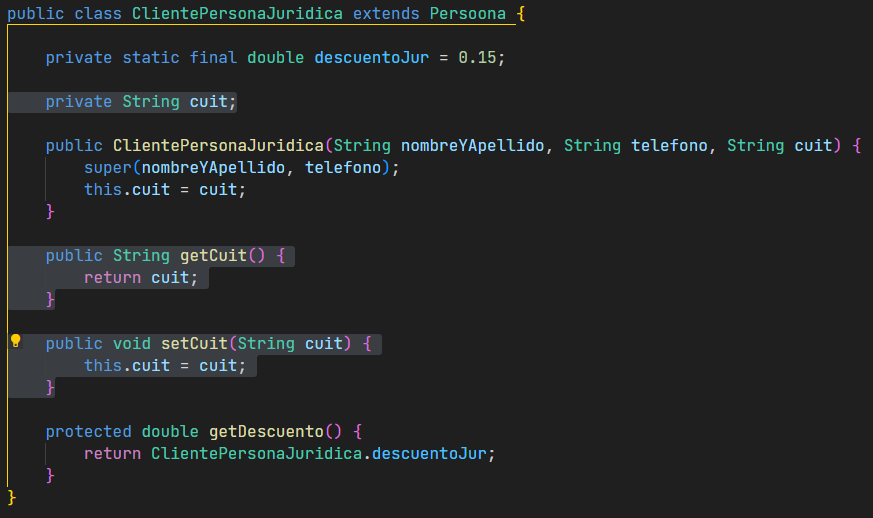
* **Mal olor detectado en el código:**

Observamos que en las clases Persoona, ClientePersonaFisica y ClientePersonaJuridica existen getters y setters que no son utilizados (Dead code), lo que opera en contra de uno de los pilares de la programación orientada a objetos: el encapsulamiento.

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







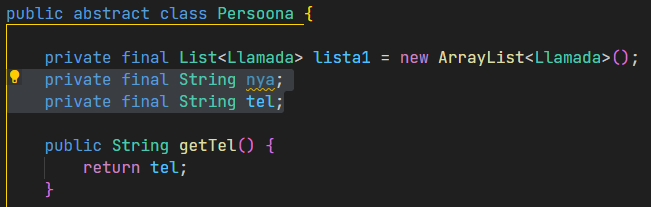
* **Refactoring a aplicar que resuelve el mal olor:**

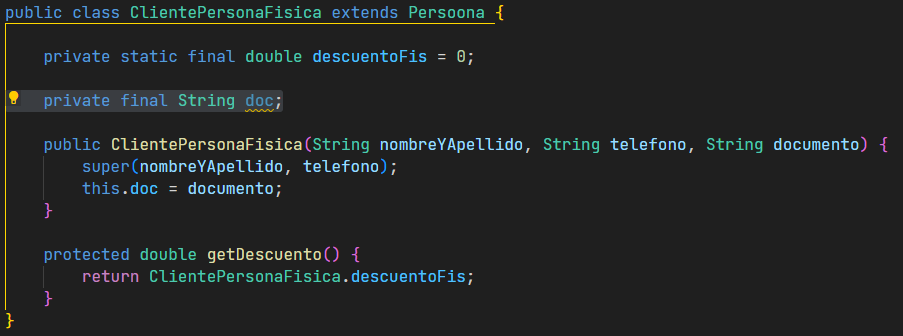
Para resolver este mal olor realizamos la eliminación de los siguientes métodos:

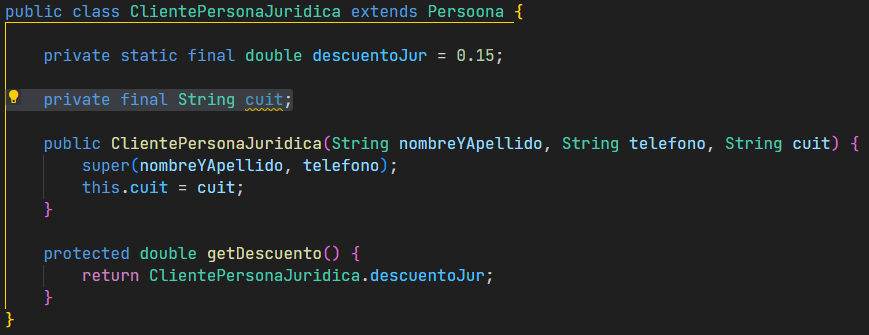
* getLista1() y getNya() de la clase Persoona.
* setLista1(), setNya() y setTel() de la clase Persoona (Remove setting method).
* getDoc() y getCuit() de las clase ClientePersonaFisica y ClientePersonaJuridica, respectivamente.
* setDoc() y setCuit() de las clase ClientePersonaFisica y ClientePersonaJuridica, respectivamente (Remove setting method).

Con los cambios aplicados, los atributos de la clase Persoona son inicializados únicamente en el constructor. Al no existir la posibilidad de alterarlos con algún método setter, se les añade el modificador final.

* **Código con el refactoring aplicado:**







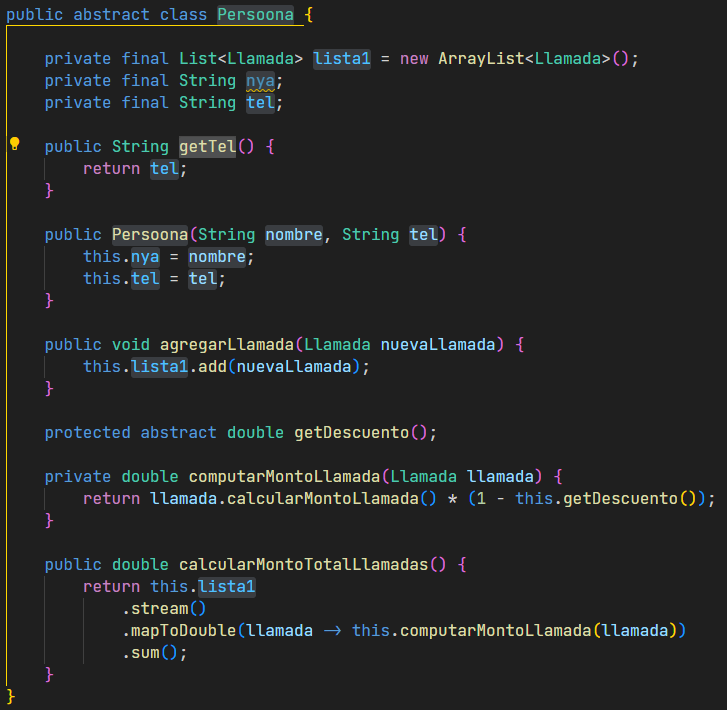
Ver código en paquete **refactor\_18**.

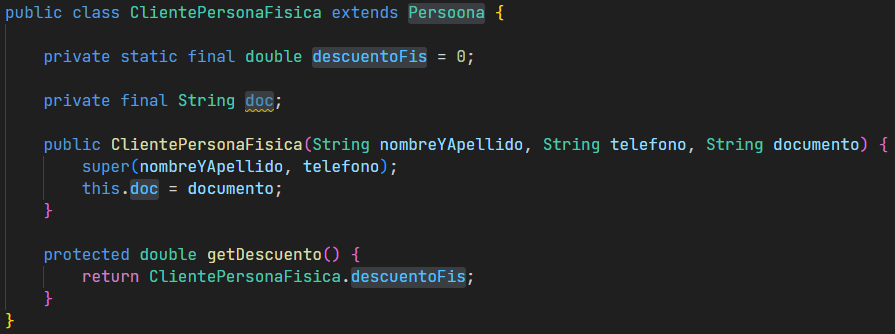
## **Refactor 19: Mysterious name**

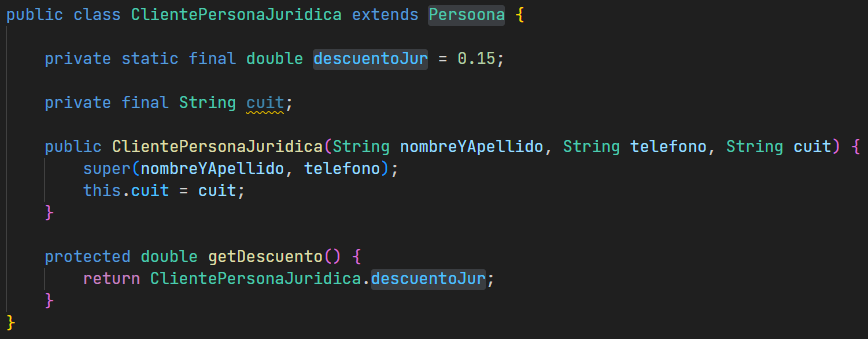
* **Mal olor detectado en el código:**

Observamos que dentro de la jerarquía de Persoona, numerosos identificadores de variables de instancia, variables de clase, parámetros y clases no describen apropiadamente aquello a lo que hacen referencia.

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





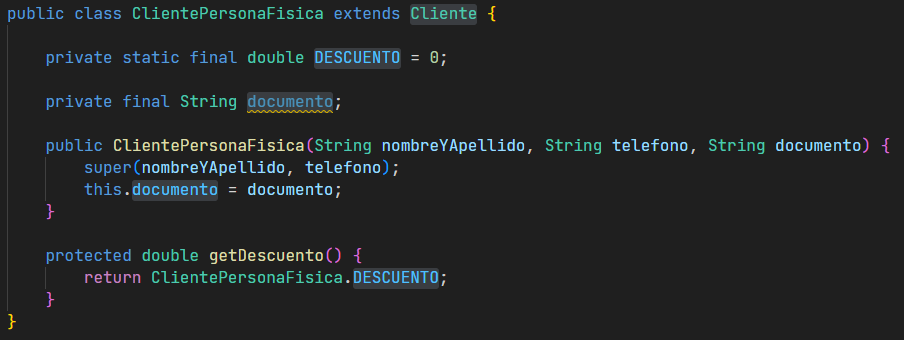


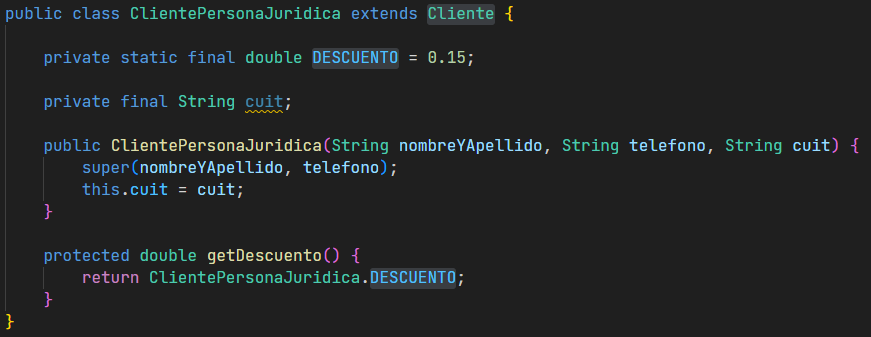
* **Refactoring a aplicar que resuelve el mal olor:**

Realizamos los siguientes renombres de los identificadores que consideramos no representativos:

* La clase Persoona es renombrada como Cliente.
* Las variables de instancia lista1, nya y tel de la clase Cliente son renombradas como llamadasEfectuadas, nombreYApellido y telefono, respectivamente.
* El método getTel() de la clase Cliente es renombrado como getTelefono().
* La variable de instancia doc de la clase PersonaFisica es renombrada como documento y la variable de clase descuentoFis como DESCUENTO.
* La variable de clase descuentoJur de la clase PersonaJuridica es renombrada como DESCUENTO.
* Los parámetros nombre y tel del constructor de Persoona son renombrados como nombreYApellido y telefono.
* Asimismo se modificaron todas las referencias a la clase Cliente y sus elementos en las clases Persoonal y PersoonalTest.
* **Código con el refactoring aplicado:**







Ver código en paquete **refactor\_19**.

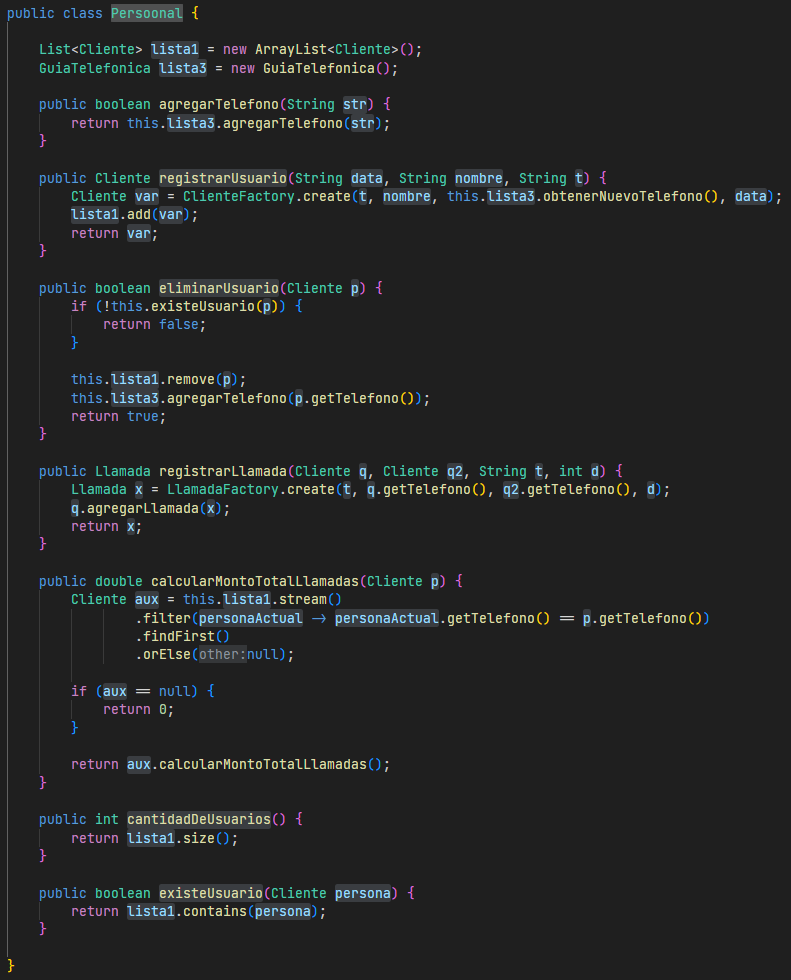
## 

## **Refactor 20: Mysterious name**

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

Observamos que dentro de la clase Persoonal, numerosos identificadores de variables de instancia, parámetros y clases no describen apropiadamente aquello a lo que hacen referencia.

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



* **Refactoring a aplicar que resuelve el mal olor:**

Realizamos los siguientes renombres de identificadores que consideramos no representativos:

* La clase Persoonal es renombrada como Sistema.
* Las variables de instancia lista1 y lista3 la clase Sistema son renombradas como clientes y telefonosDisponibles, respectivamente.
* El método registrarUsuario() de la clase Sistema es renombrado como registrarCliente(). La variable local var de este mismo método se renombra como nuevoCliente, mientras que los parámetros data, nombre y t se renombran como identificador, nombreYApellido y tipoCLiente.
* El método eliminarUsuario() de la clase Sistema es renombrado como eliminarCliente(). El parámetro p de este mismo método se renombra como cliente.
* La variable local x del método registrarLlamada() es renombrada como nuevaLlamada. Los parámetros q, q2, t y d de este mismo método se renombran como emisor, receptor, tipoLlamada y duración, respectivamente.
* La variable local aux del método calcularMontoTotalLlamadas() es renombrada como clienteEncontrado. El parámetro p de este mismo método se renombra como cliente. El parámetro personaActual de la función anónima se renombra como clienteActual.
* El método cantidadDeUsuarios() de la clase Sistema es renombrado como cantidadDeClientes().
* El método existeUsuarios() de la clase Sistema es renombrado como existeCliente(). El parámetro persona de este mismo método se renombra como cliente.
* Asimismo se modificaron todas las referencias a la clase Sistema y sus elementos en la clase PersoonalTest, que ahora pasa a llamarse SistemaTest.
* **Código con el refactoring aplicado:**



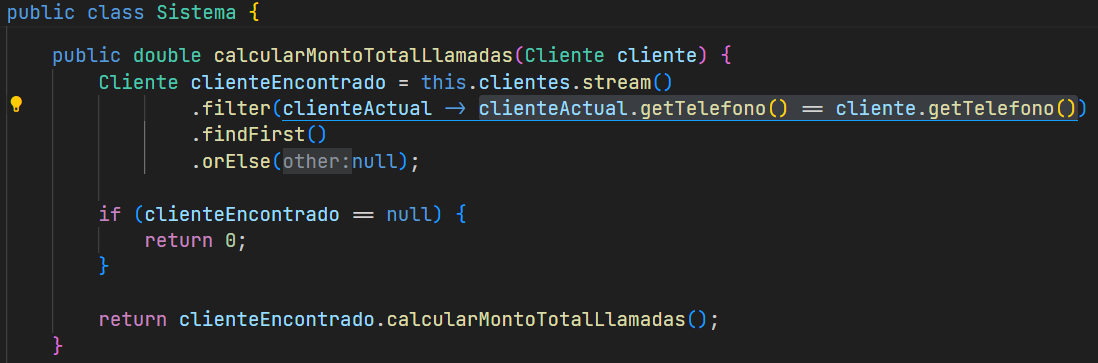
Ver código en paquete **refactor\_20**.

## **Refactor 21: Envidia de atributos**

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

Observamos que la clase Sistema, en su método calcularMontoTotalLlamadas() hace uso de variables de instancia de la clase Cliente. Esta práctica, conocida como envidia de atributos, genera alto acoplamiento entre las clases mencionadas.

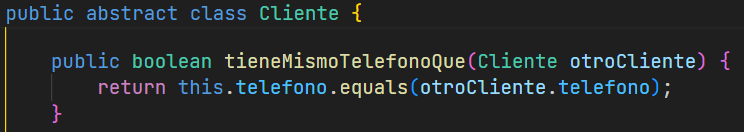
* **Extracto del código que presenta el mal olor:**

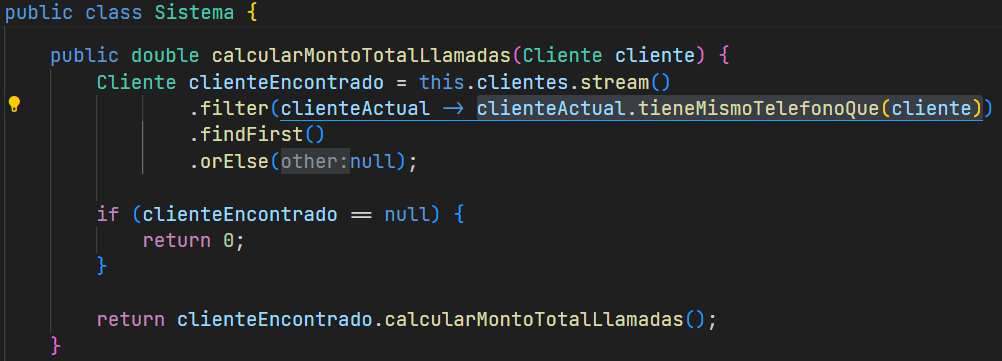


* **Refactoring a aplicar que resuelve el mal olor:**

Para resolver este mal olor realizamos una serie de refactors que se listan a continuación:

* Extract method de la comparación de teléfonos de clientes (tieneMismoTelefonoQue()).
* Move method del método tieneMismoTelefonoQue() a la clase Cliente.
* **Código con el refactoring aplicado:**

****



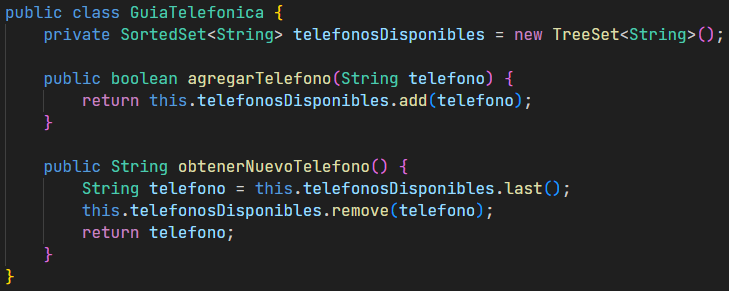
Ver código en paquete **refactor\_21**.

## **Refactor 22: Speculative Generality**

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

Reevaluamos el rol de la clase GuiaTelefonica en el sistema y llegamos a la conclusión de que, en un refactor previo (refactor\_2), especulamos que era necesario ocultar el manejo de la colección de números telefónicos utilizada. En un principio pensamos que era deseable que la colección pudiera ser alterada sin que el sistema lo notara. Posteriormente, advertimos que lo único que se hace en la clase GuiaTelefonica es ocultar el manejo de la colección. Por eso, decidimos que no tiene sentido tener una clase adicional que realice funcionalidad que ya es propia del manejo de colecciones, además de que su presencia significa tener una clase más que mantener en el sistema.

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





* **Refactoring a aplicar que resuelve el mal olor:**

El refactor que aplicamos fue el de Inline Class, de manera de implementar la colección en la clase Sistema como una variable de instancia privada más y simplificar el código. Los pasos realizados fueron:

* Move field del campo telefonosDisponibles y move method de los métodos agregarTelefono() y obtenerNuevoTelefono() de la clase GuiaTelefonica a la clase Sistema.
* Adaptación del código de la clase sistema para utilizar los nuevos métodos y campos.
* Eliminación de la clase GuiaTelefonica.
* **Código con el refactoring aplicado:**



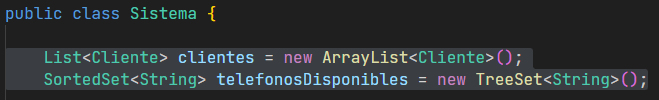
Ver código en paquete **refactor\_22**.

## **Refactor 23: Declaración de atributos sin modificador de visibilidad**

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

Observamos declaración de atributos sin modificador de visibilidad en la clase Sistema, lo cual genera ruptura del encapsulamiento.

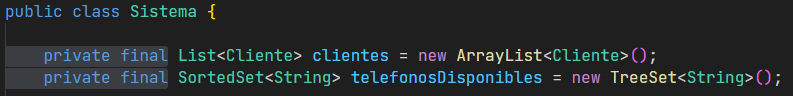
* **Extracto del código que presenta el mal olor:**



* **Refactoring a aplicar que resuelve el mal olor:**

Declaramos los atributos de la clase como privados y eliminamos la posibilidad de que se pueda acceder a ellos desde otra clase (Encapsulate field). Al no existir la posibilidad de alterar estos atributos con algún método setter, se les añade el modificador final a ambas variables de instancia.

* **Código con el refactoring aplicado:**



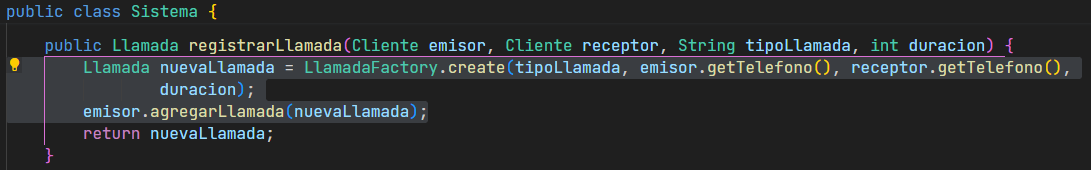
Ver código en paquete **refactor\_23**.

## **Refactor 24: Inappropriate Intimacy**

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

Observamos que la clase Sistema es la que envía el mensaje de creación a la clase LlamadaFactory para instanciar los distintos tipos de llamada. Creemos apropiado que el envío de este mensaje quede a cargo de Cliente que es quien tiene una relación de conocimiento con la clase Llamada.

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

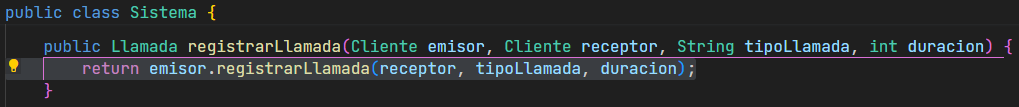


* **Refactoring a aplicar que resuelve el mal olor:**

Realizados un Extract method y un consecuente Move method del cuerpo del método registrarLlamada() desde la clase Sistema a la clase Cliente, manteniendo su nombre original. Como consecuencia de este movimiento se cambia la visibilidad del método agregarLlamada() de público a privado, ya que sólo es accedido desde la clase Cliente, mejorando así el encapsulamiento.

* **Código con el refactoring aplicado:**





Ver código en paquete **refactor\_24**.