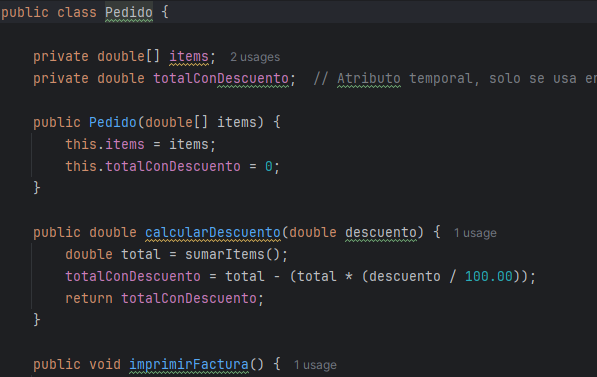
Bad Smells

1. **Bad Smell: Temporary field (atributo temporal)**

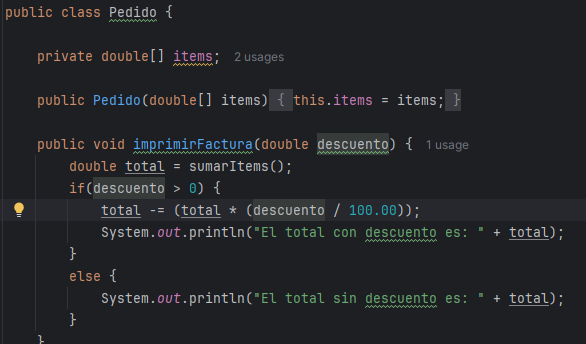
Ocurre cuando una clase tiene atributos que se usan solo en algunos métodos específicos y no son esenciales para la clase en general. Estos atributos pueden hacer que la clase sea más difícil de entender y mantener, ya que no son verdaderamente parte del estado permanente del objeto.

Ejemplo: El atributo totalConDescuento es temporal y solo se usa en algunos métodos

Sin refactorizar:



Refactorización:

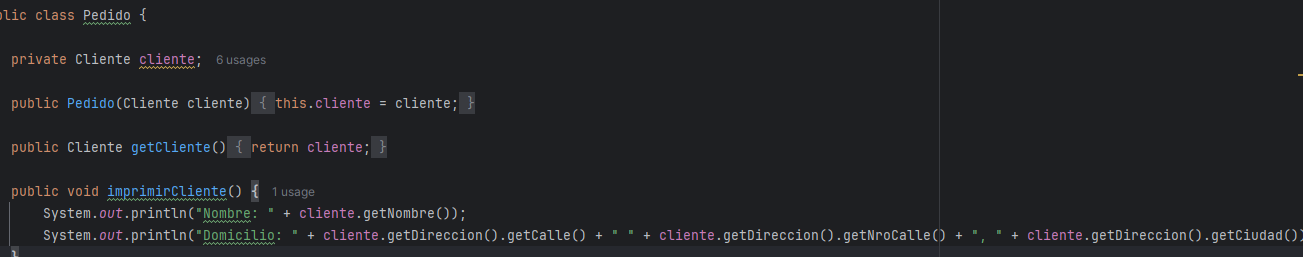


1. **Bad Smell: Message chains (cadena de mensajes)**

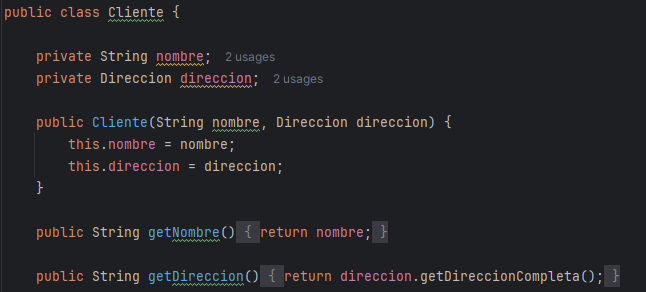
Ocurre cuando se tiene múltiples niveles de invocaciones de métodos a través de objetos, generando una cadena larga de llamadas como objetoA.getObjetoB().getObjetoC().hacerAlgo().

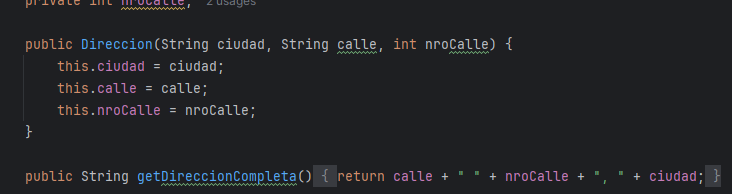
Ejemplo: El método imprimirCliente() en la clase Pedidoestá accediendo al domicilio del cliente mediante una cadena de mensajes como ser cliente.getDireccion().getCiudad().

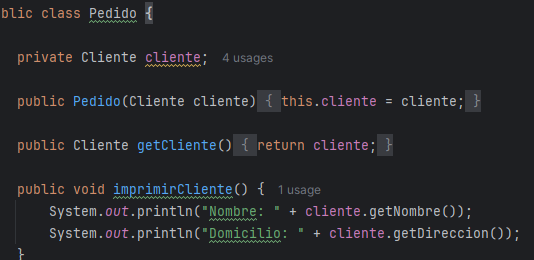
Sin refactorizar:



Refactorizado:







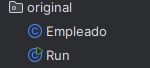
1. **Bad Smell: Divergent change (cambio divergente)**

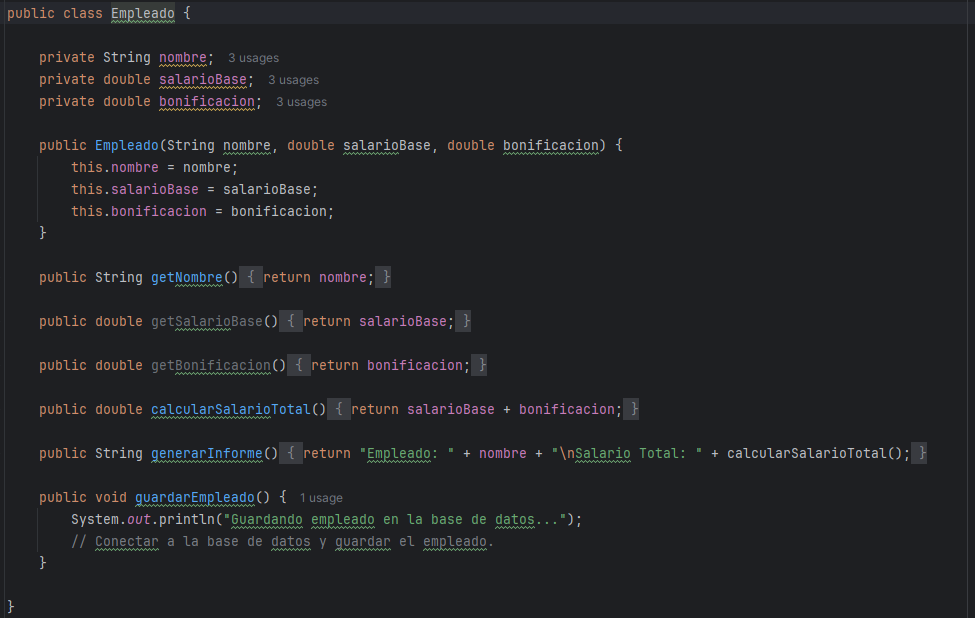
Ocurre cuando una clase se modifica frecuentemente por diferentes razones o para diferentes propósitos, violando el principio de responsabilidad única. Esto significa que la clase tiene demasiadas responsabilidades y, como consecuencia, se ve afectada por múltiples tipos de cambios.

Ejemplo: La clase Empleado tendría que cambiar por diferentes motivos:

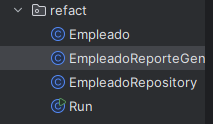
* Si cambia la lógica de negocio para calcular el salario.
* Si cambia la forma en que se debe formatear la salida para el informe.
* Si cambian los detalles de persistencia.

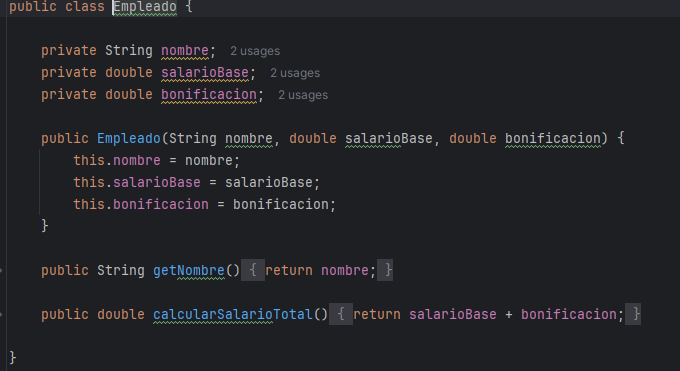
Sin refactorizar:

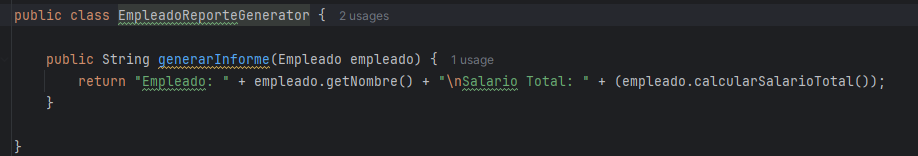


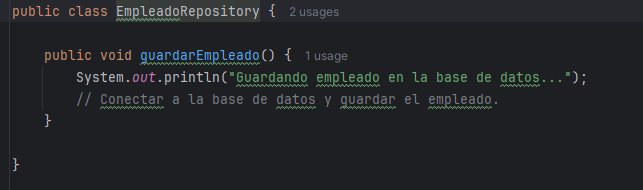


Refactorizado:







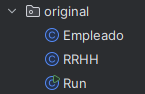


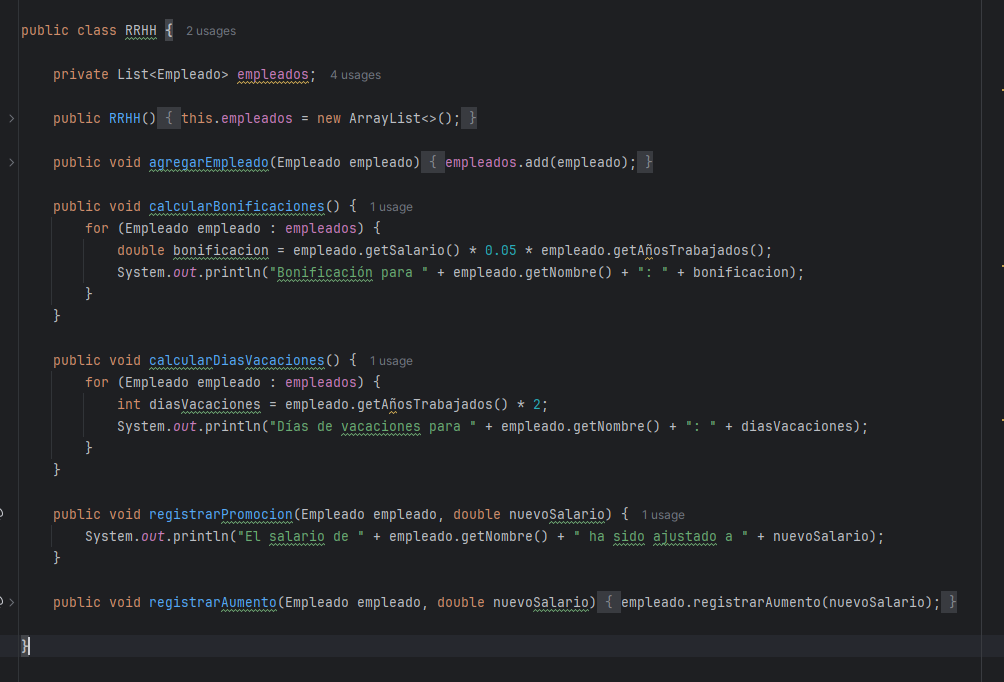
1. **Bad Smell: Shotgun surgery (cambio en cadena)**

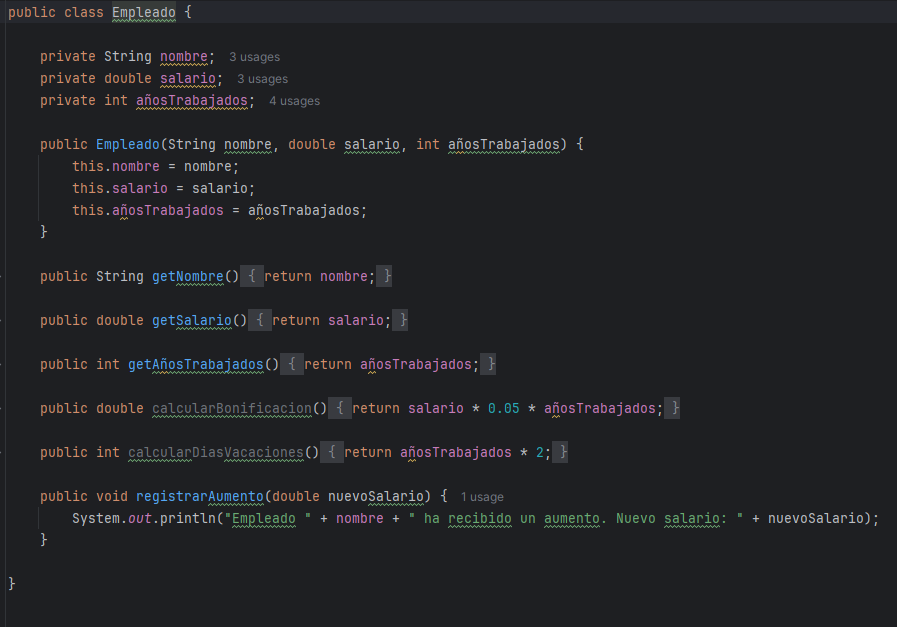
Ocurre cuando hacer un pequeño cambio en una clase provoca la necesidad de hacer cambios en muchas otras clases.

Ejemplo:

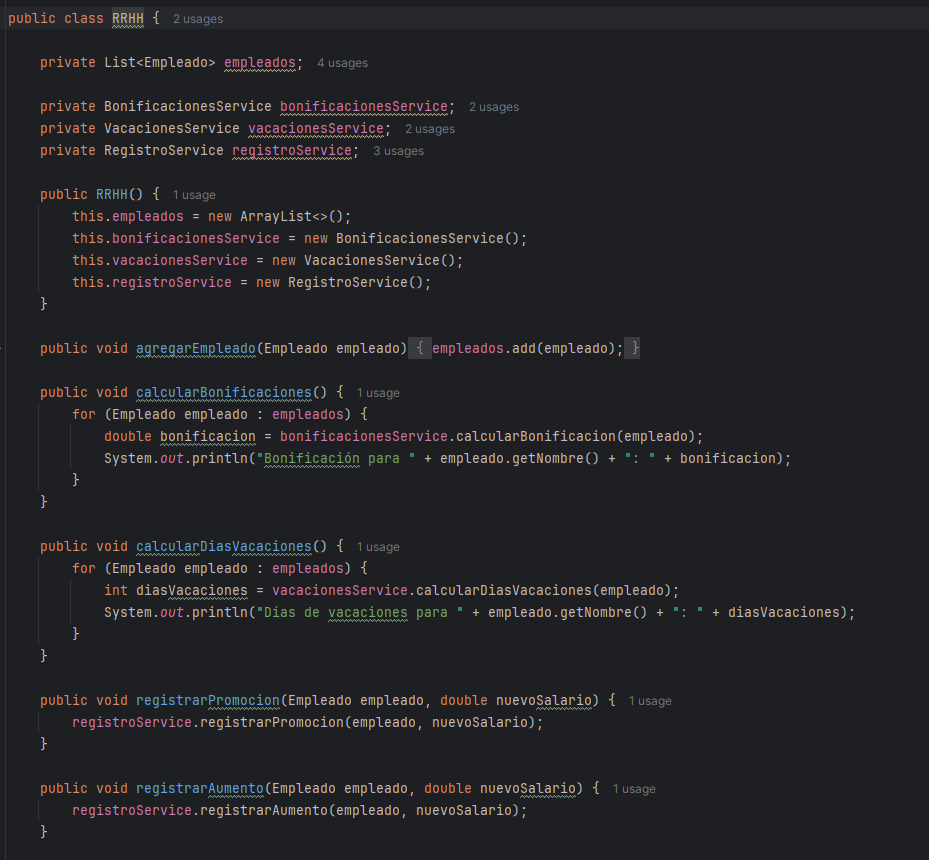
* Tanto en la clase Empleado como en RRHH, se calculan las bonificaciones y los días de vacaciones de la misma manera. Si cambia la política de bonificaciones o vacaciones, tendríamos que modificar ambas clases.
* La lógica para registrar promociones o aumentos está en ambas clases. Cualquier cambio en cómo se registran estas acciones requiere modificaciones en ambas clases.

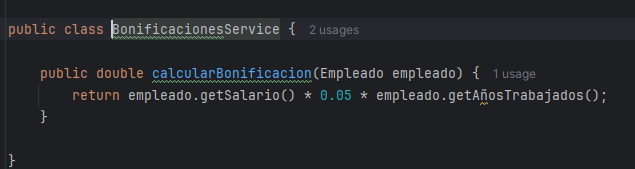
Sin refactorizar:   


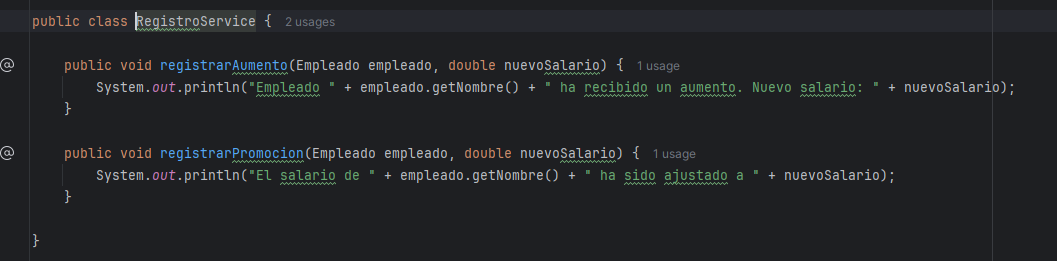


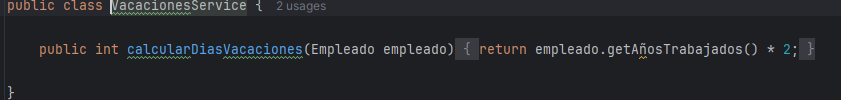


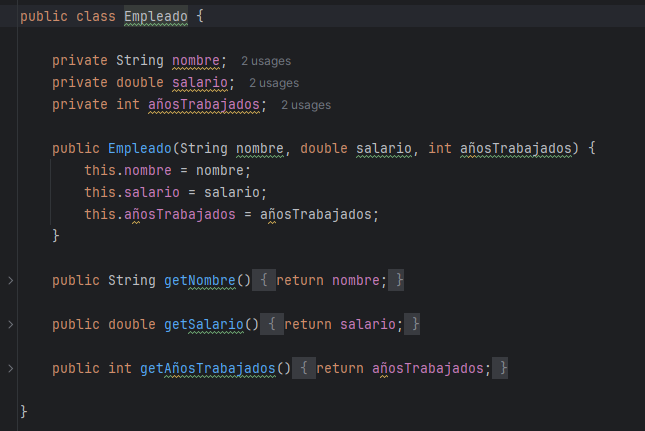
Refactorizado:







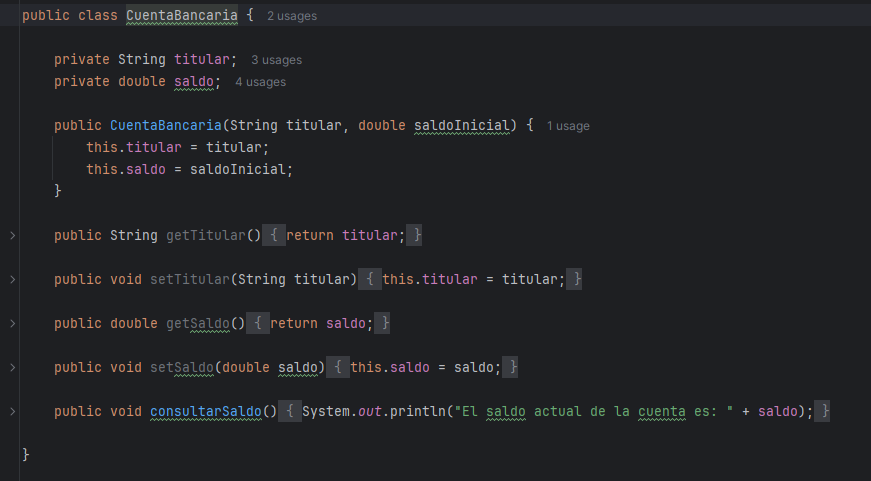
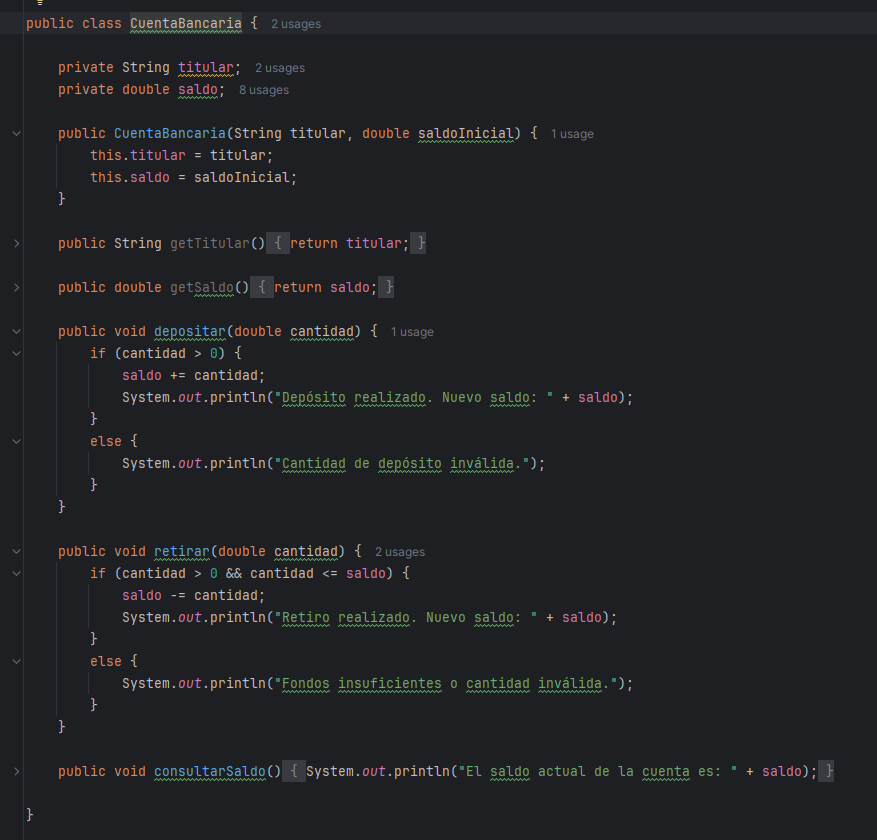




1. **Bad Smell: Data class (clase de datos)**

Aparece cuando una clase solo tiene atributos y métodos de acceso (getters y setters) y no tiene ninguna lógica o comportamiento significativo. Estas clases actúan simplemente como contenedores de datos y, a menudo, son un síntoma de falta de diseño, ya que no encapsulan adecuadamente el comportamiento asociado a los datos.

Ejemplo: La clase CuentaBancaria solo tiene atributos (titular y saldo) y métodos de acceso (get y set), pero no encapsula ningún comportamiento relevante para una cuenta bancaria (como depositar, retirar dinero, etc.). Si necesitamos realizar operaciones bancarias (por ejemplo, depositar o retirar dinero), esa lógica tendría que estar dispersa en otras clases que interactúan con CuentaBancaria, en lugar de estar encapsulada en la propia clase.

Sin refactorizar:   
Refactorizado:   


1. **Dead code (código muerto)**

Aparece cuando variables, atributos, métodos o clases ya no se usan, ya sea porque cambiaron los requerimientos del proyecto o por correcciones posteriores que hicieron que ya no fueran necesarios.

1. **Large class (clase grande)**

Aparece cuando una clase tiene demasiados atributos, métodos y líneas de código, lo que la convierte en una clase que intenta manejar demasiadas responsabilidades.

1. **Lazy class (clase perezosa)**

Una “Lazy Class” es una clase que no aporta valor significativo al programa y está subutilizada.   
Si una clase no tiene suficientes responsabilidades o su funcionalidad es tan pequeña que no justifica su existencia, debe ser eliminada o fusionada con otra clase más apropiada.

1. **Comments (comentarios)**

Los comentarios son útiles en el código, pero cuando son excesivos o innecesarios, pueden indicar que algo está mal en el diseño.   
Si un bloque de código necesita comentarios extensos para explicarlo, probablemente el código no es lo suficientemente claro o no está bien estructurado.

1. **Duplicated code (código duplicado)**

Se refiere a situaciones en las que el mismo bloque de código aparece en más de un lugar dentro de un proyecto, o cuando varias partes del código realizan operaciones similares, pero no idénticas

Tipos de duplicación:

* **Duplicación directa:** El mismo código aparece en varios lugares.
* **Duplicación sutil:** Fragmentos de código que son diferentes en apariencia, pero en realidad realizan la misma tarea o tienen la misma lógica subyacente.

1. **Feature envy (envidia de características)**

Ocurre cuando un método de una clase accede más frecuentemente a los atributos o métodos de otra clase que a los de su propia clase.   
Este comportamiento indica que el método probablemente esté en la clase incorrecta y debería ser movido a la clase de la que más depende.

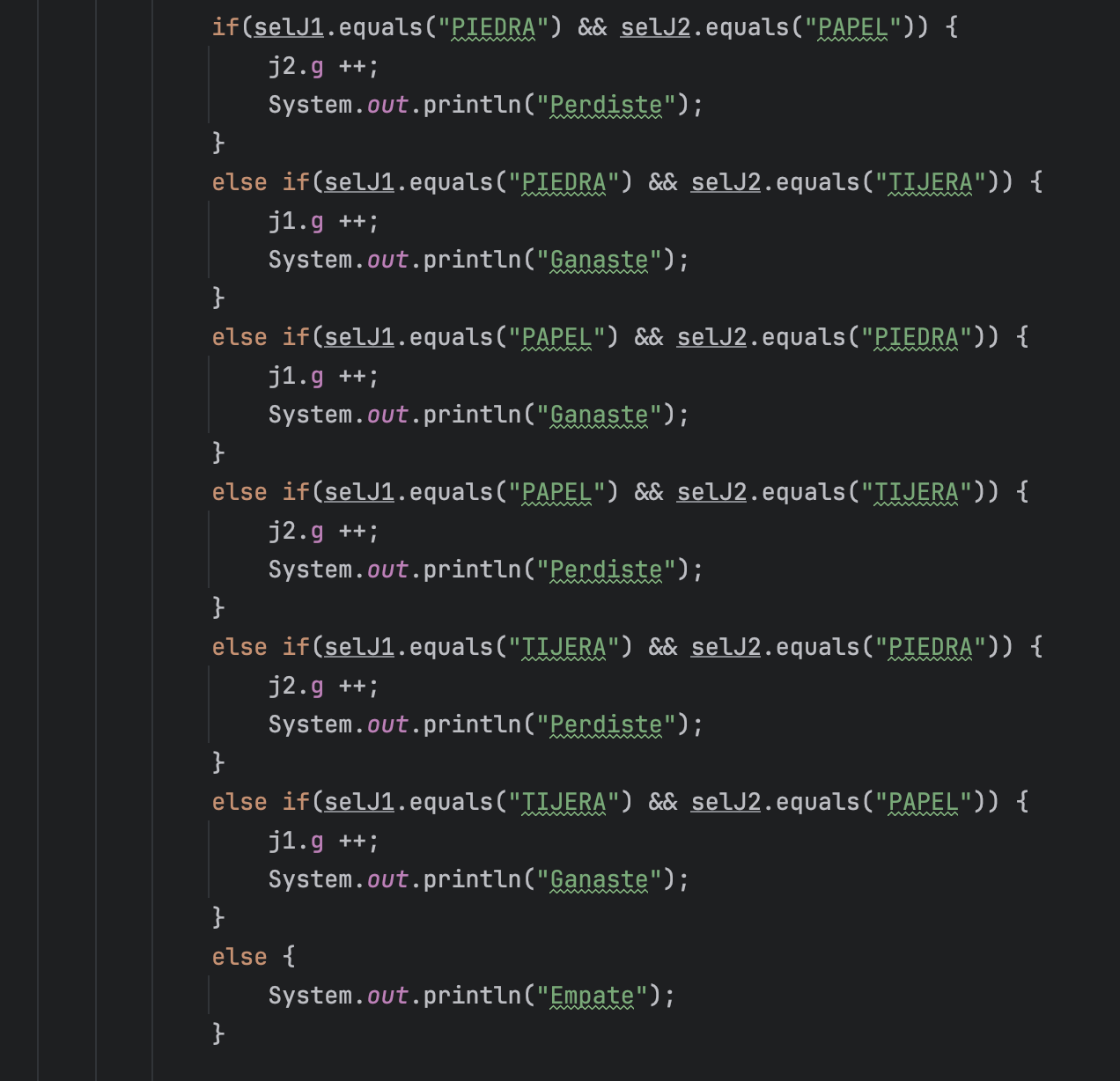
Ejemplo: El método calcularDescuento() de la clase Pedido accede frecuentemente a los datos de la clase Cliente que a sus propios atributos. Esto refleja una baja cohesión ya que depende más de la información que reside en la otra clase.

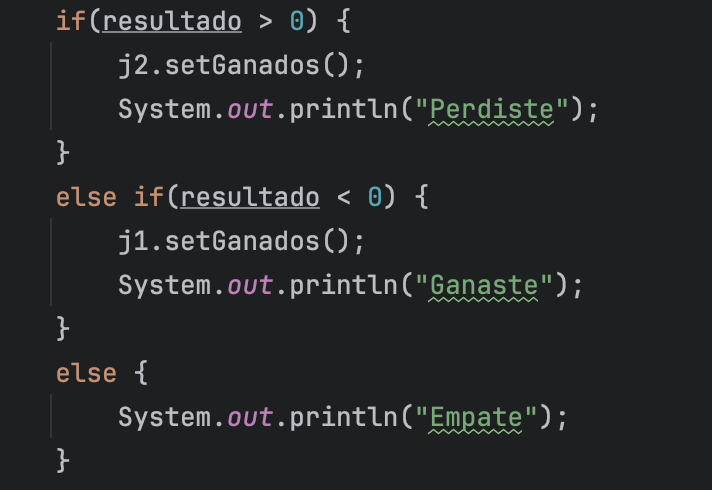
1. **Switch statement (estructura de agrupación condicional)**

Aparece cuando en un programa se encuentran sentencias switch o múltiples if-else anidados en diferentes partes del código para manejar decisiones basadas en el tipo de un objeto.

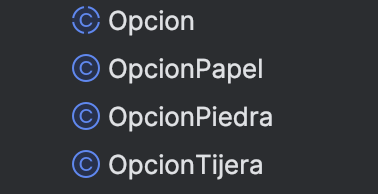
Por ejemplo:

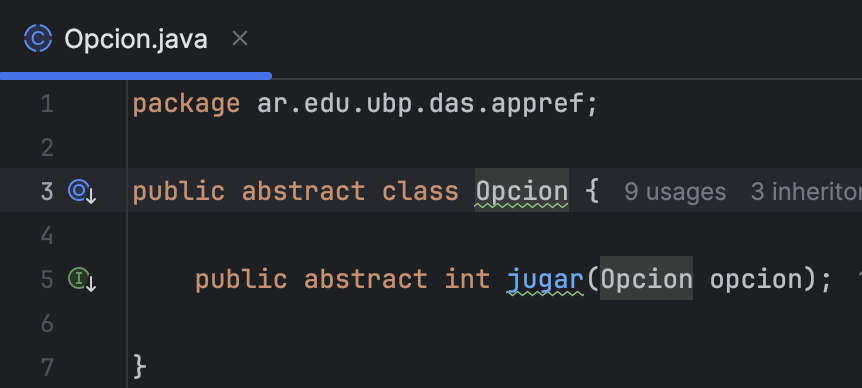
Sin refactorizar:

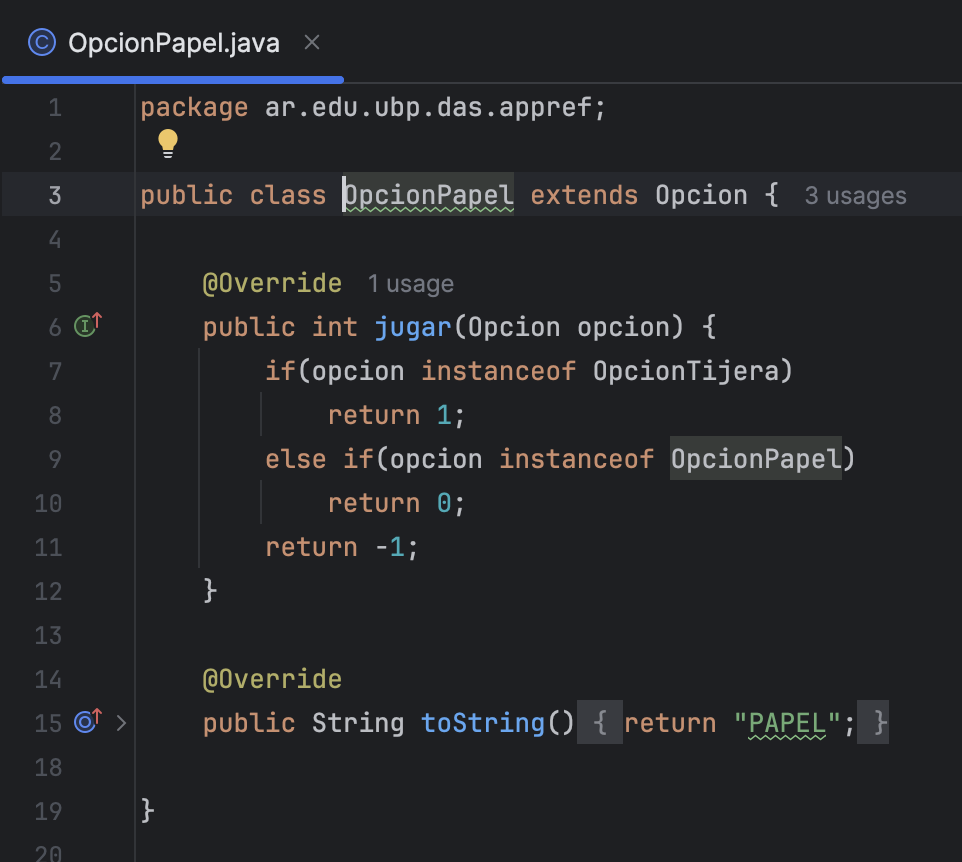


Refactorizado:   


Se creo una clase abstracta Opcion y una clase para cada una de las opciones posibles que heredan de la clase Opcion:







**13. Bad Smell: Speculative generality (generalidad especulativa)**

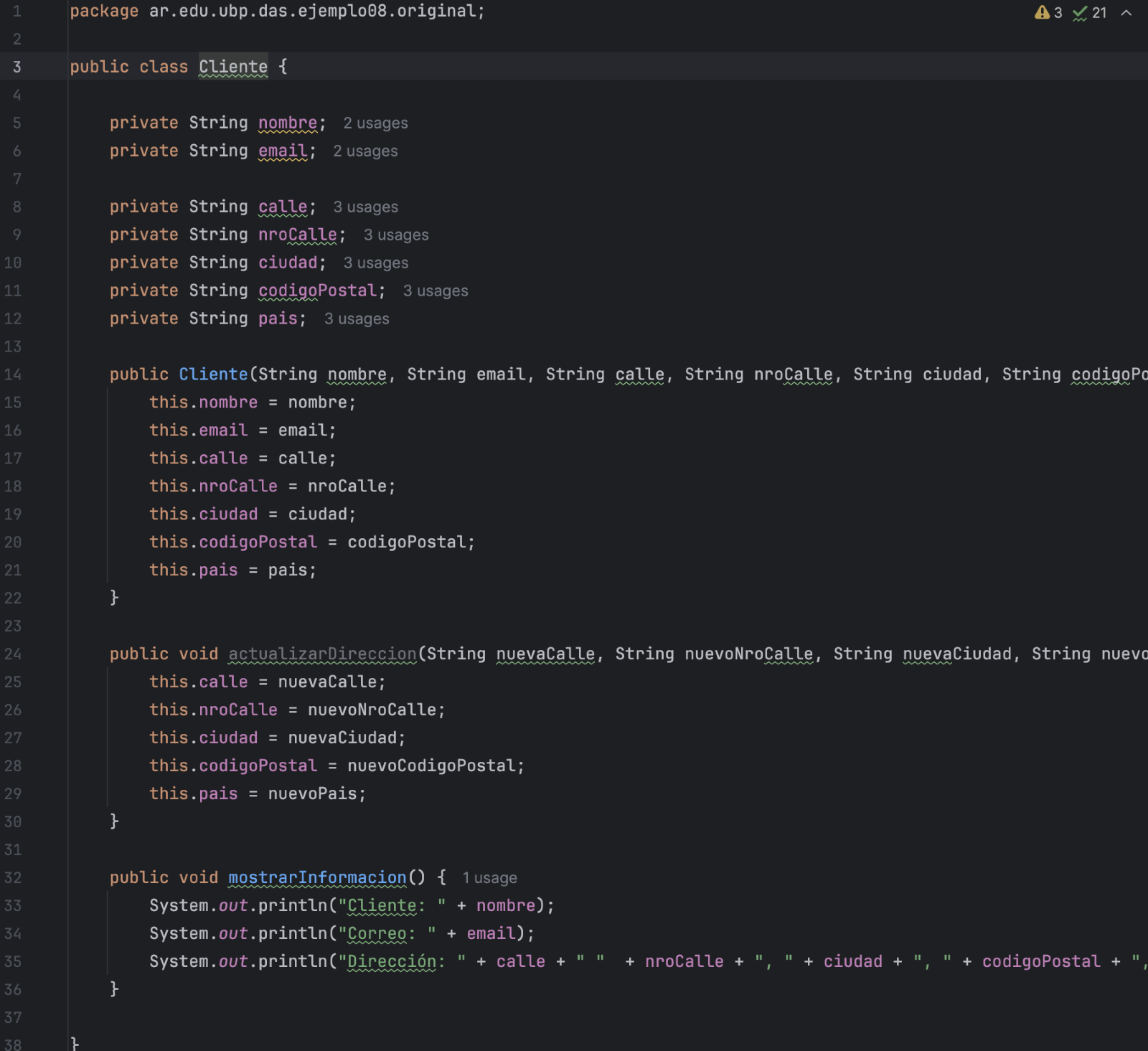
Ocurre cuando se agregan funcionalidades al código “por si acaso” que podrían ser necesaria en el futuro, pero que actualmente no tiene uso.

1. **Bad Smell: Data clumps (grupo de datos)**

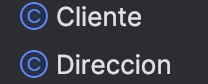
Ocurre cuando ciertos elementos de datos (por ejemplo, atributos o parámetros) siempre aparecen juntos en varios lugares del código, ya sea como parámetros en múltiples métodos o como atributos en varias clases.

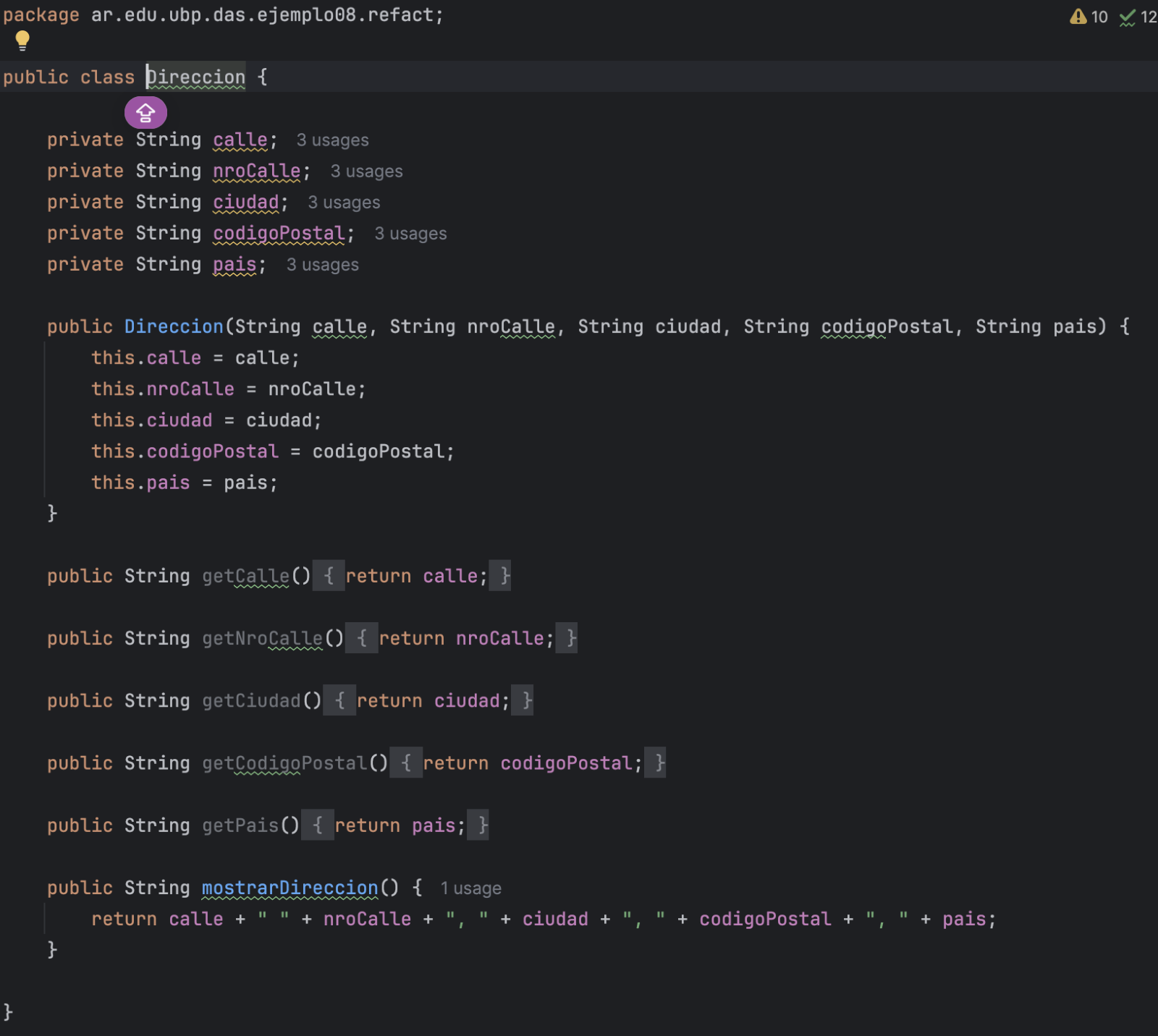
Por ejemplo: En la clase Cliente, los datos relacionados con la dirección (calle, nroCalle, ciudad, codigoPostal, pais) aparecen repetidos como parámetros en varios métodos, y estos mismos datos podrían aparecer en otras partes del sistema, lo que genera duplicación.

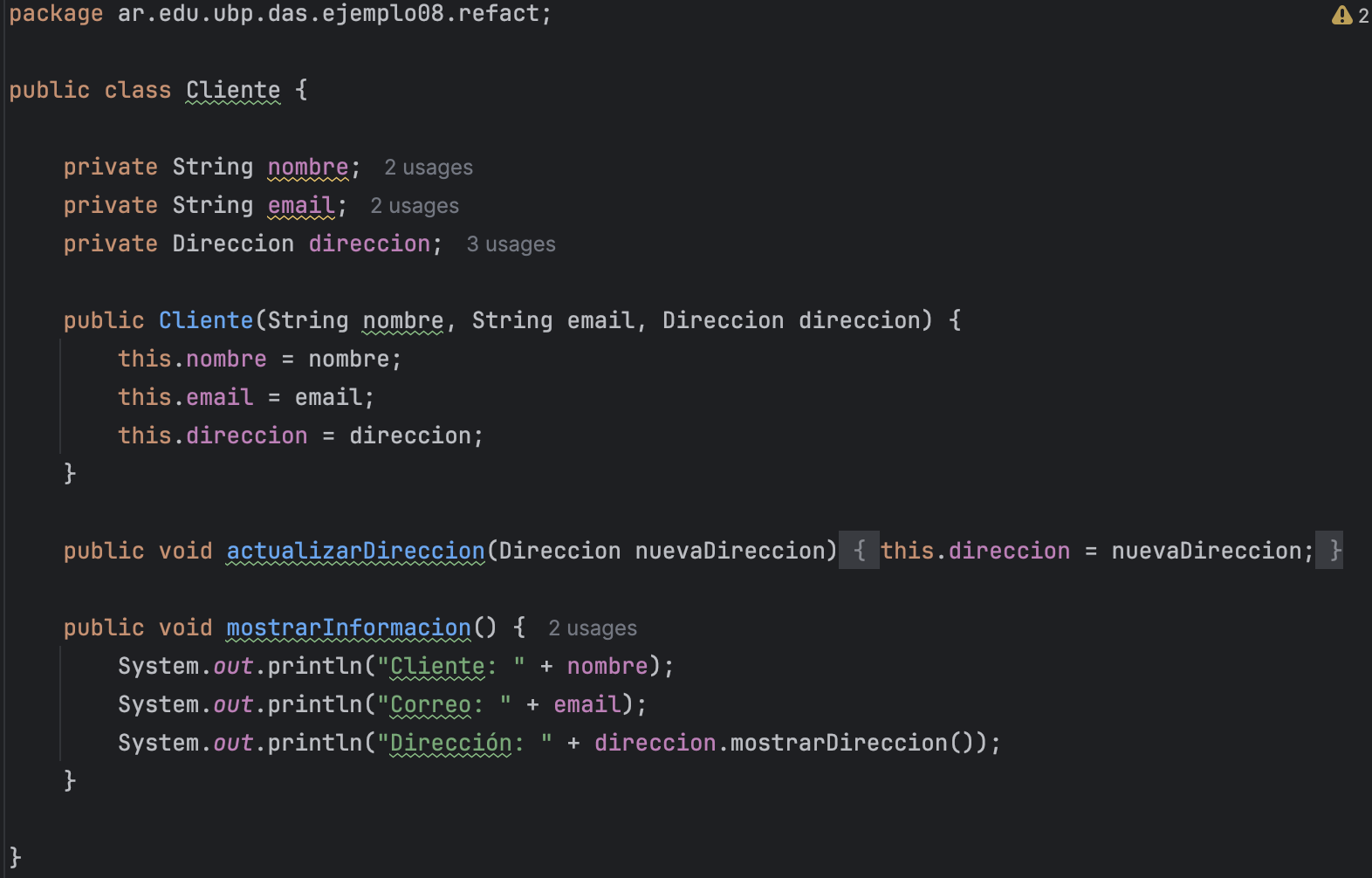
Sin refactorizacion:



Refactorizado:





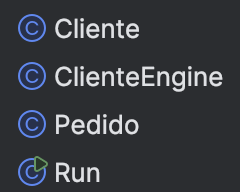


1. **Bad Smell: Middleman (intermediario)**

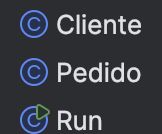
Ocurre cuando una clase no hace mucho por sí misma y delega la mayor parte de sus responsabilidades o comportamientos a otra clase. Esto rompe con el principio de encapsulación, ya que la clase intermediaria no tiene un propósito claro y simplemente actúa como un puente innecesario.

Puede aparecer como resultado de haber aplicado demasiada delegación para evitar otros bad smells como “MessageChains” (cadenas de mensajes), pero termina generando una clase que añade poca o ninguna funcionalidad.

Sin refactorizar:



Refactorizado:



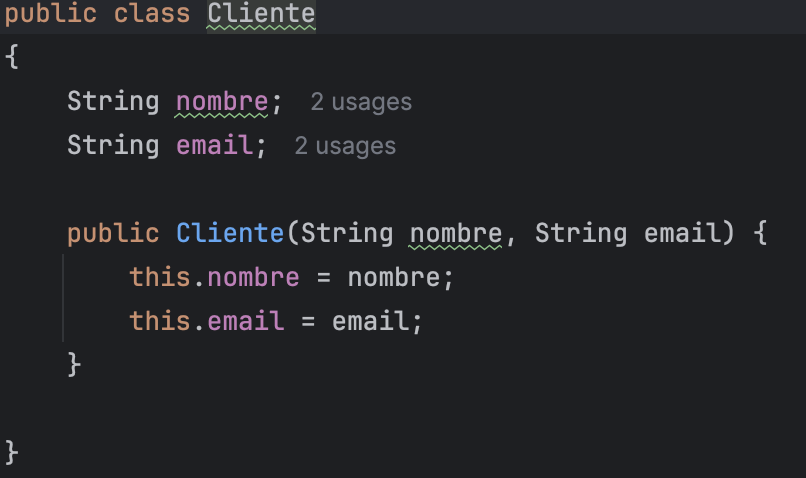
Basicamente lo que se hizo fue eliminar la clase ClienteEngine ya que no tenia sentido porque le delegaba todas las solicitudes entre cliente y pedido, cuando directamente podrian interactuar entre ellas.

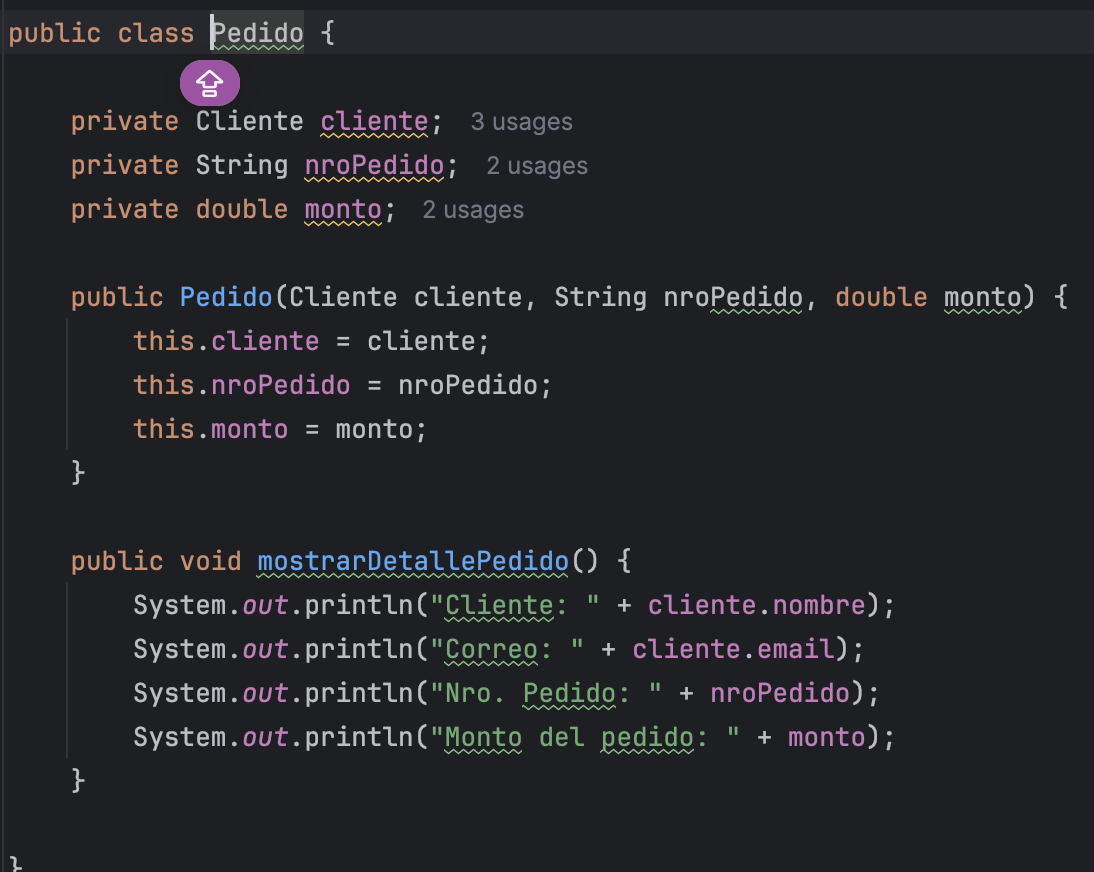
1. **Bad Smell: Inappropiate intimacy (intimidad inadecuada)**

Se presenta cuando una clase accede directamente a los atributos o métodos privados o protegidos de otra clase, violando el principio de encapsulación.

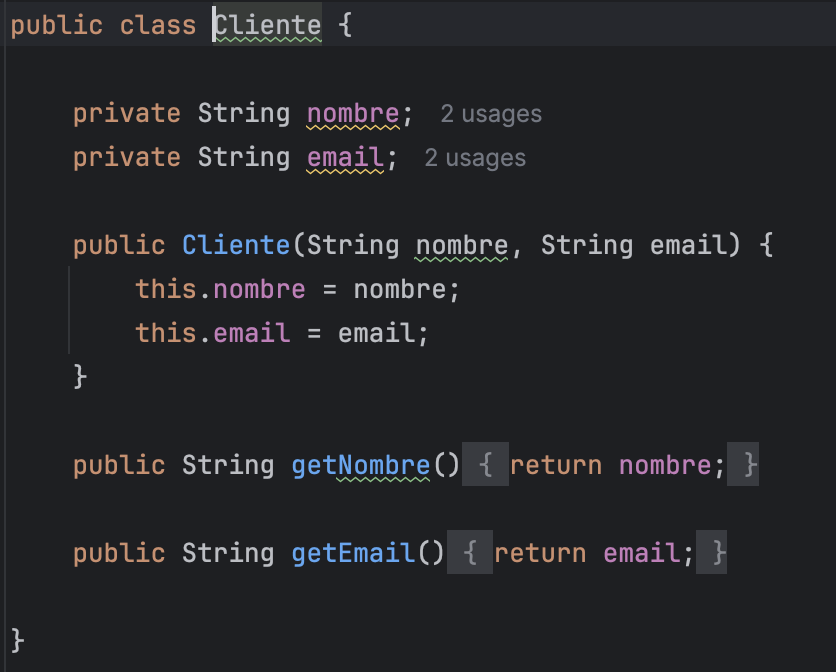
Por ejemplo: La clase Pedido accede directamente a los atributos nombre y email de la clase Cliente en el método mostrarDetallePedido().

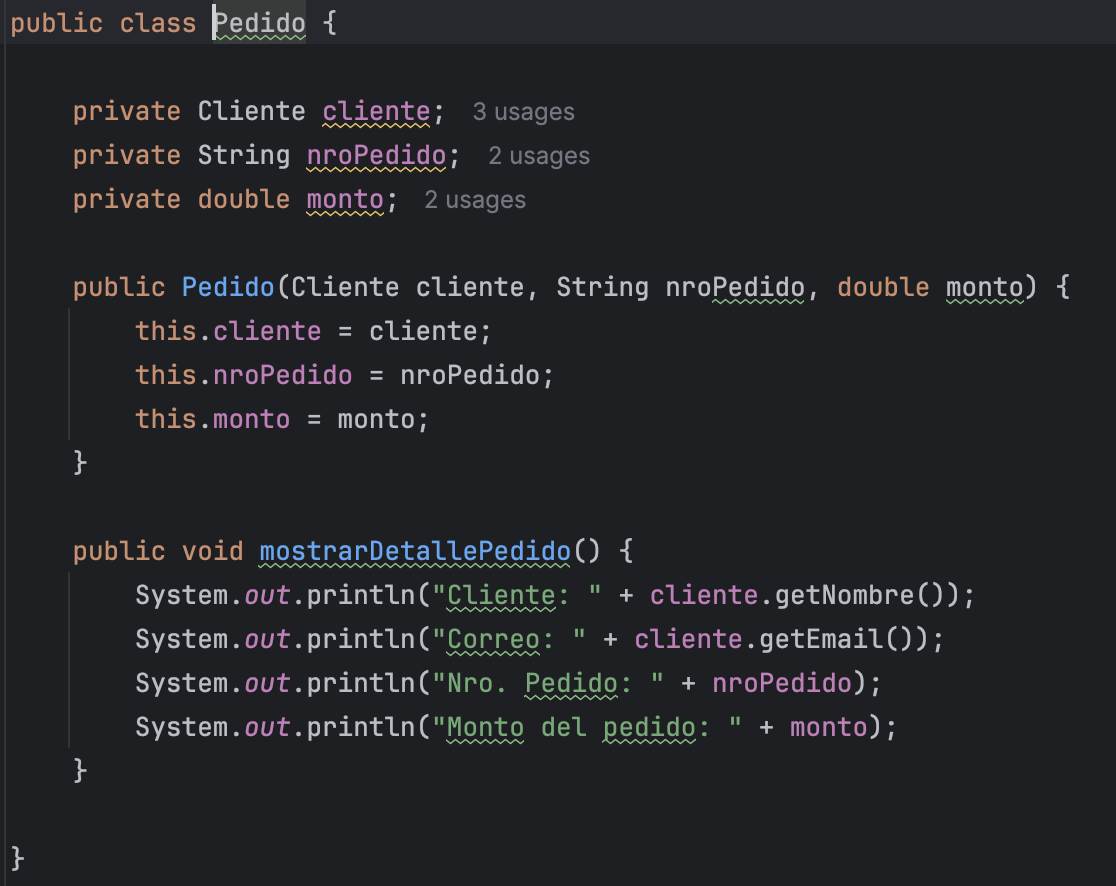
Sin refactorizar:





Refactorizado:





Lo que se hizo fue crear los metodos accesores y acceder a traves de ellos y agregarle en la clase Cliente los modificadores de acceso a las variables.

1. **Bad Smell: Parallel inheritance hierarchies (jerarquías paralelas)**

Ocurre cuando dos jerarquías de clases crecen en paralelo. Esto significa que cada vez que se crea una subclase en una jerarquía, también se debe crear una subclase correspondiente en la otra jerarquía. Este tipo de diseño causa problemas de duplicación de código y hace que el sistema sea más difícil de mantener, ya que cualquier cambio o adición en una jerarquía requiere cambios en la otra.

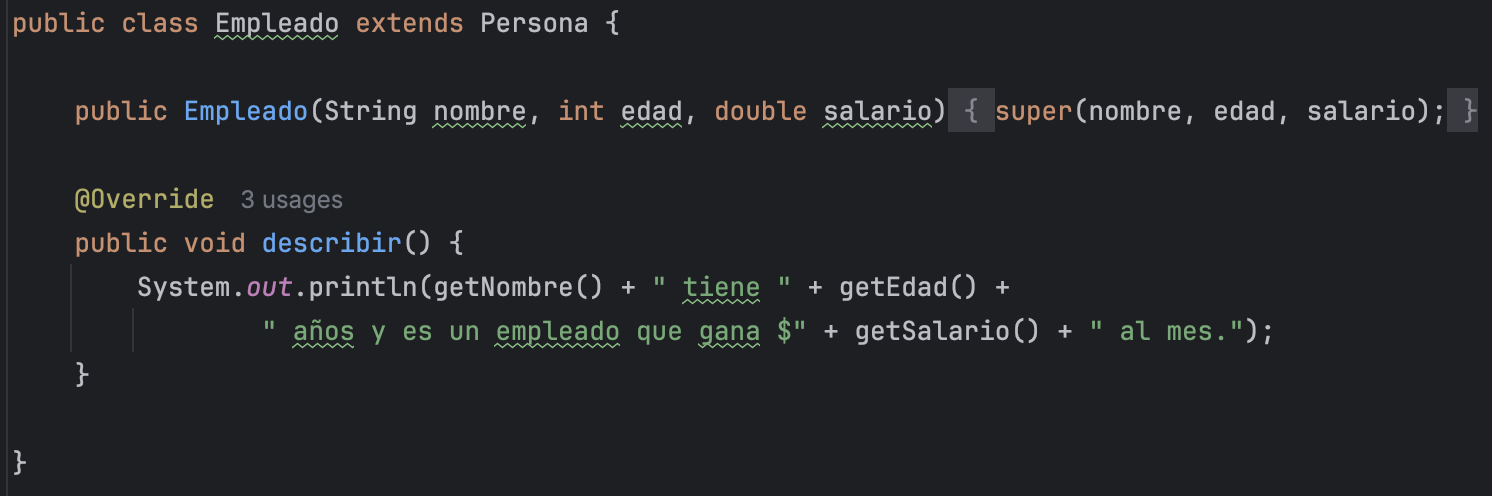
Por ejemplo: : Cada vez que se crea una subclase de Empleado, también se debe crear una subclase correspondiente en la jerarquía de EstrategiaPago. Esto genera duplicación de código y es difícil de mantener, ya que cualquier cambio en una jerarquía requiere cambios paralelos en la otra.

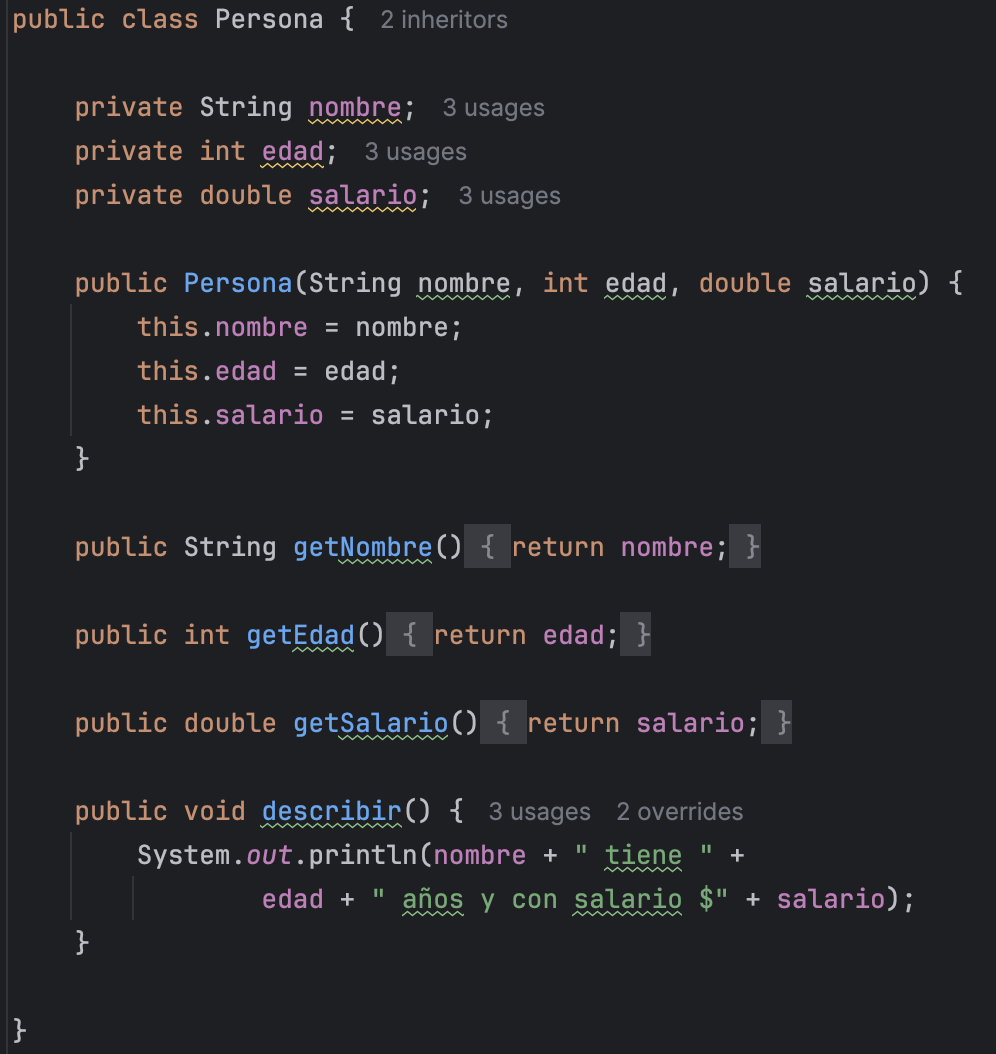
1. **Bad Smell: Refused bequest (legado rechazado)**

Ocurre cuando una subclase hereda métodos o atributos de una clase base que no necesita o no utiliza. Esto puede suceder porque la subclase no está completamente alineada con el propósito de la clase base, lo que conduce a una relación de herencia inadecuada. En estos casos, la subclase “rechaza” parte del comportamiento heredado, lo que indica que quizás la herencia no es la solución adecuada.

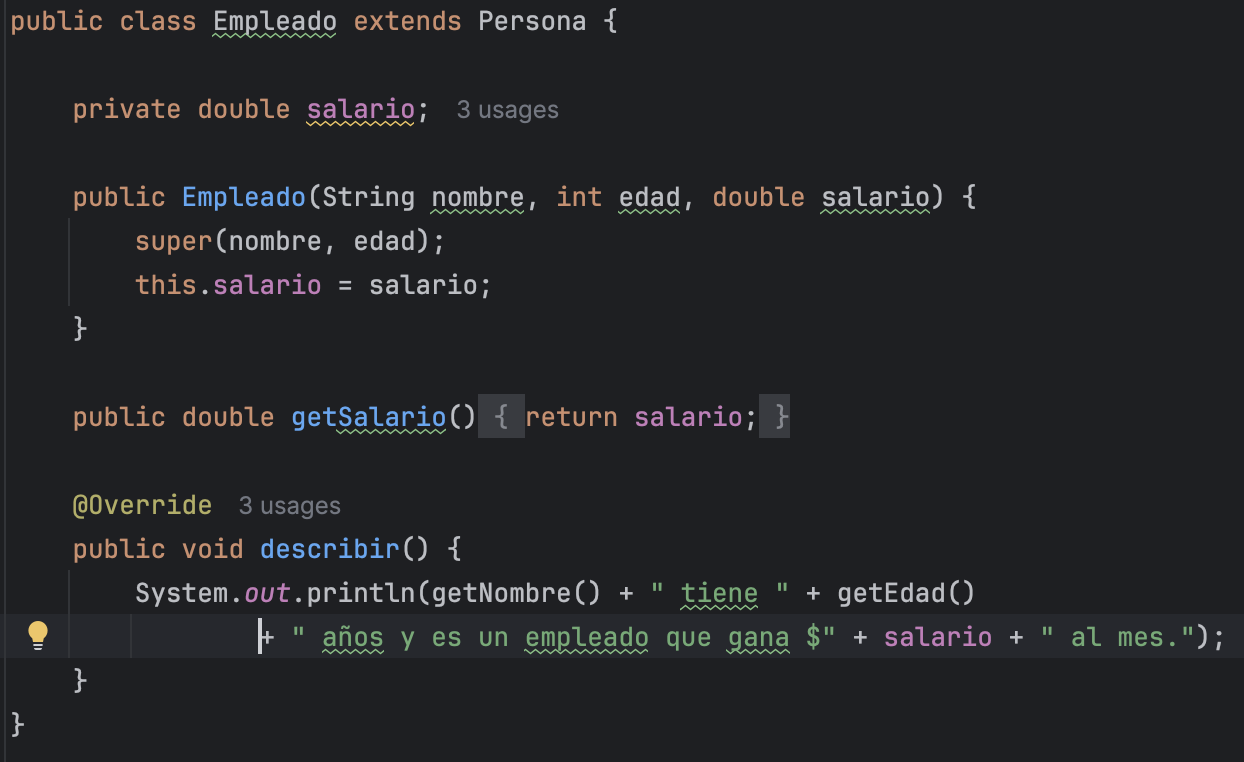
Cuando esto sucede, puede ser mejor refactorizar el código utilizando composición en lugar de herencia, o crear una jerarquía de clases más apropiada.

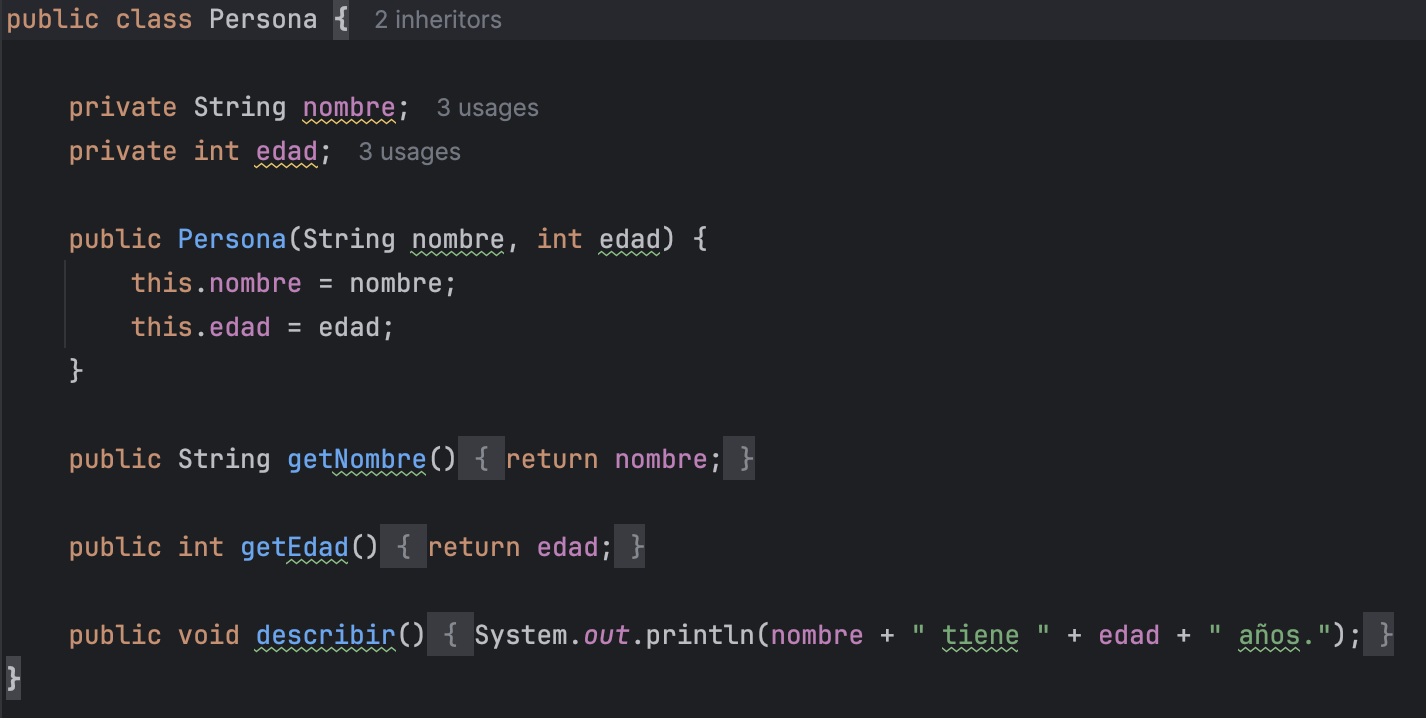
Sin refactorizar





Refactorizado:



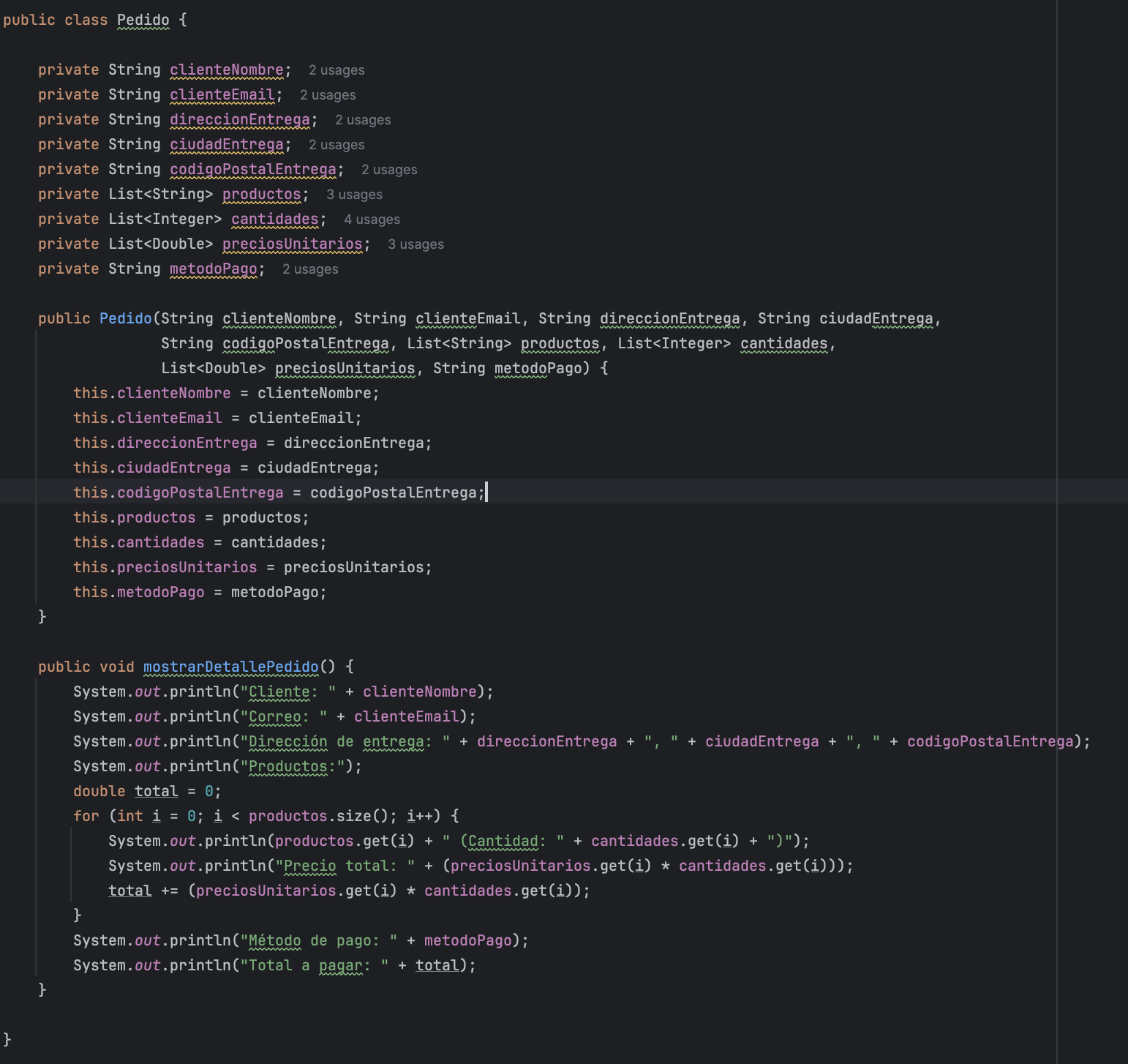


Lo que se hizo fue, en la clase Empleado se agregó un atributo ”salario” con su respectivo metodo getter, eliminando este atributo que estaba en la clase Persona, cuando en realidad el salario no tendria que corresponder a la persona sino al empleado.

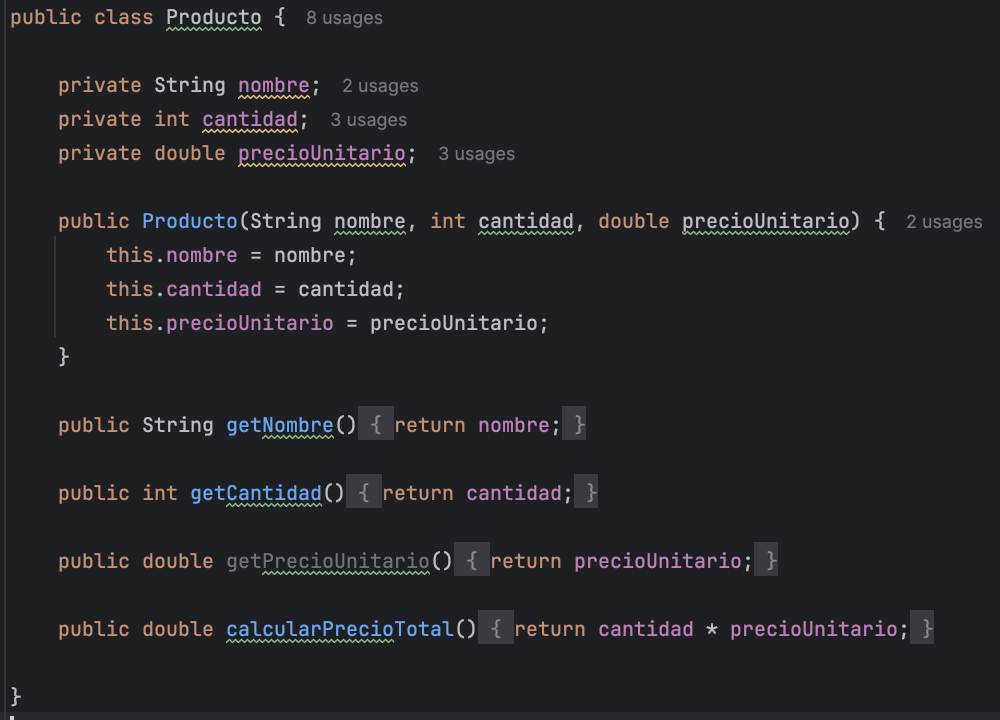
1. **Bad Smell: Long parameter list (lista de parámetros larga)**

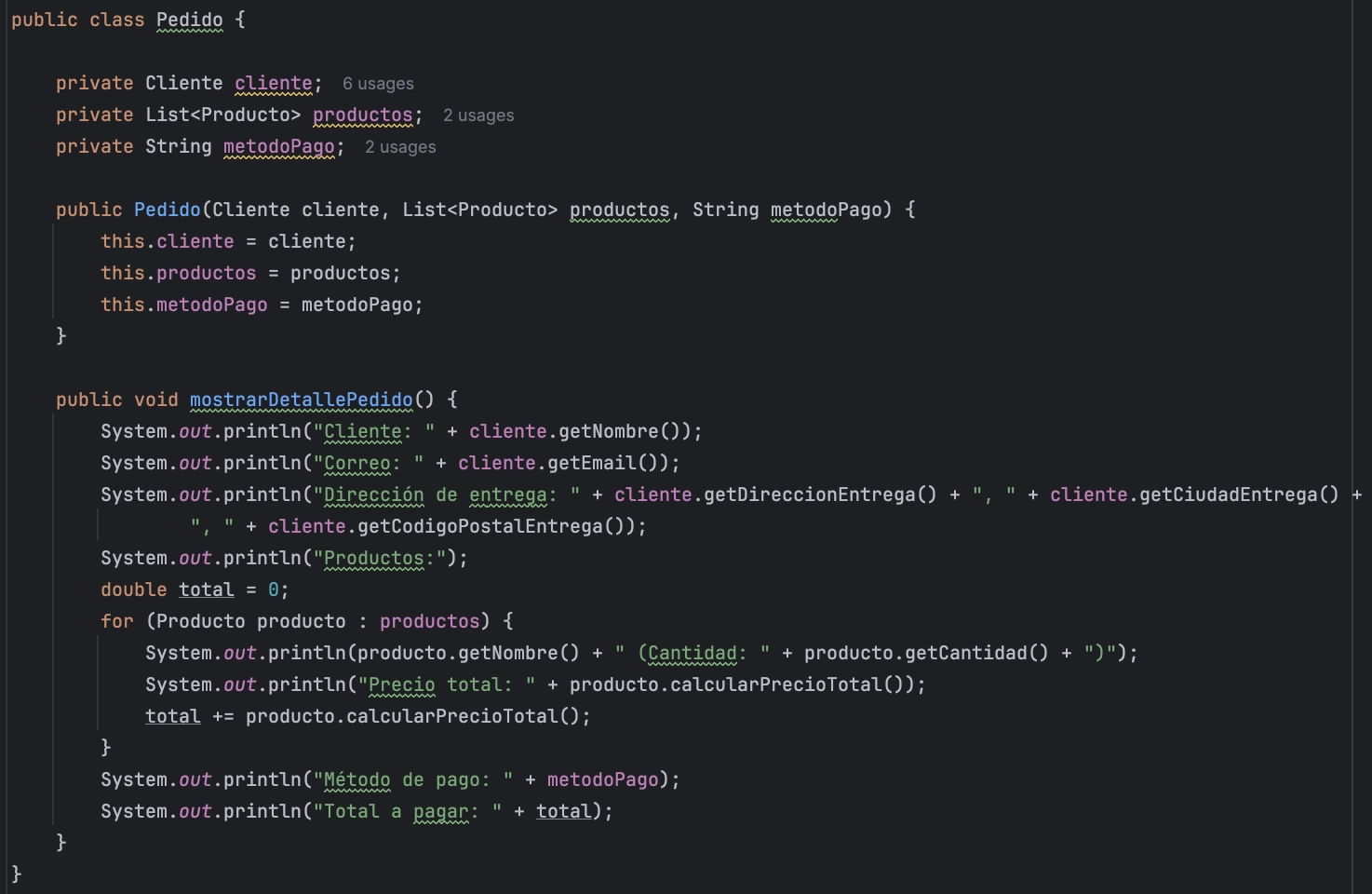
Ocurre cuando un método o constructor recibe demasiados parámetros. Esto no solo aumenta la complejidad de la comprensión del código, sino que también incrementa el acoplamiento entre clases, ya que los métodos que dependen de muchos parámetros son difíciles de mantener y cambiar.

Sin refactorizar



Refactorizado: se crearon 2 clases, una pedido y otra producto para separar la logica.





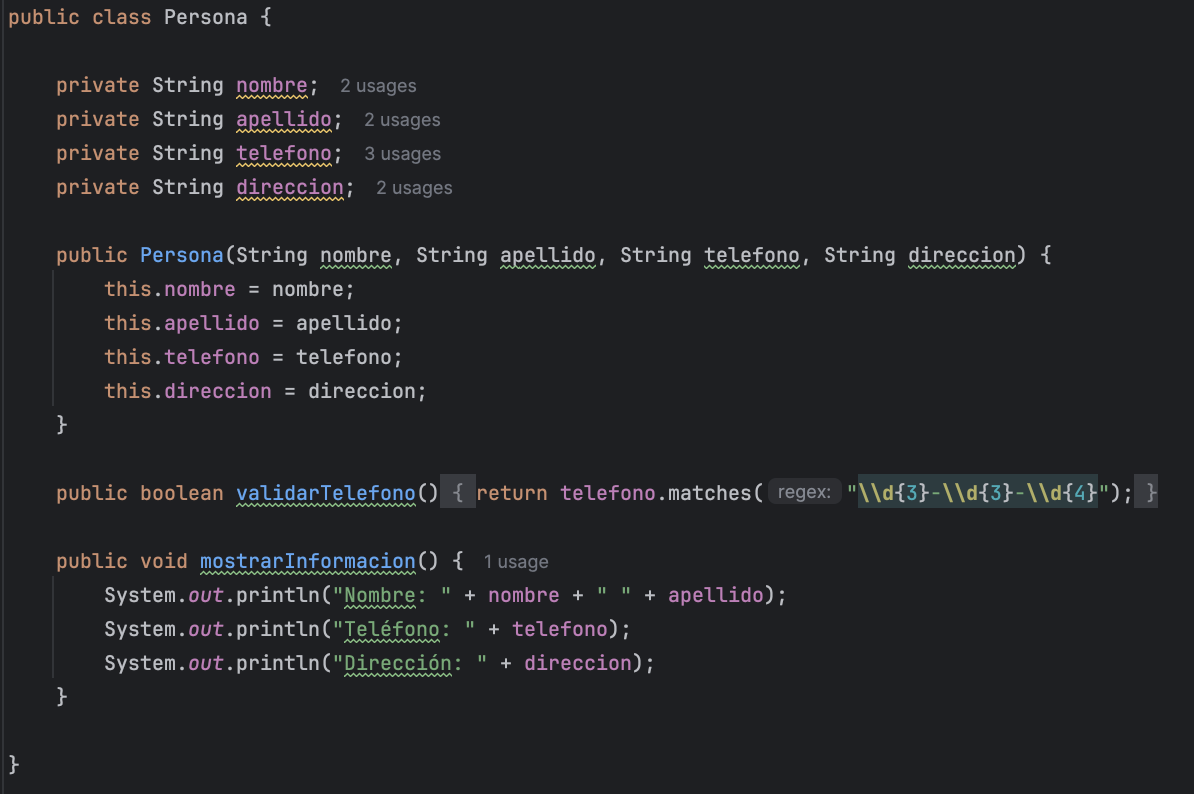
1. **Smell: Long method (método largo)**

Ocurre cuando un método tiene demasiadas líneas de código, lo que lo hace difícil de entender, mantener y reutilizar.

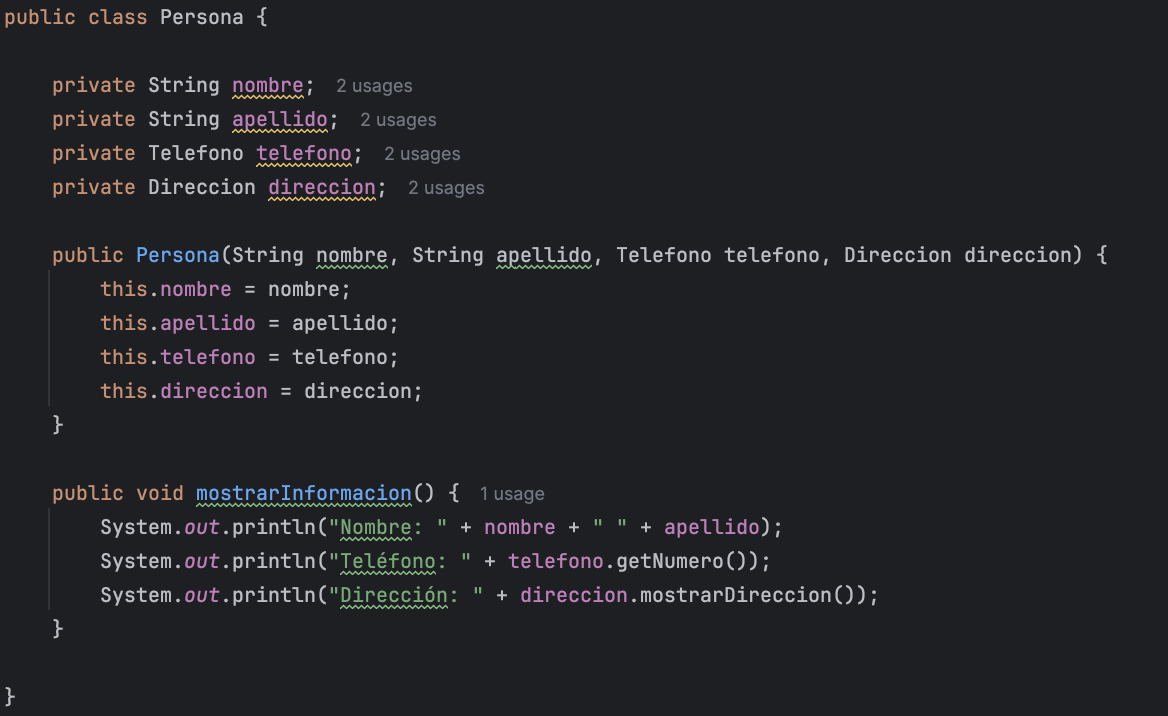
1. **Bad Smell: Primitive obsession (obsesión primitiva)**

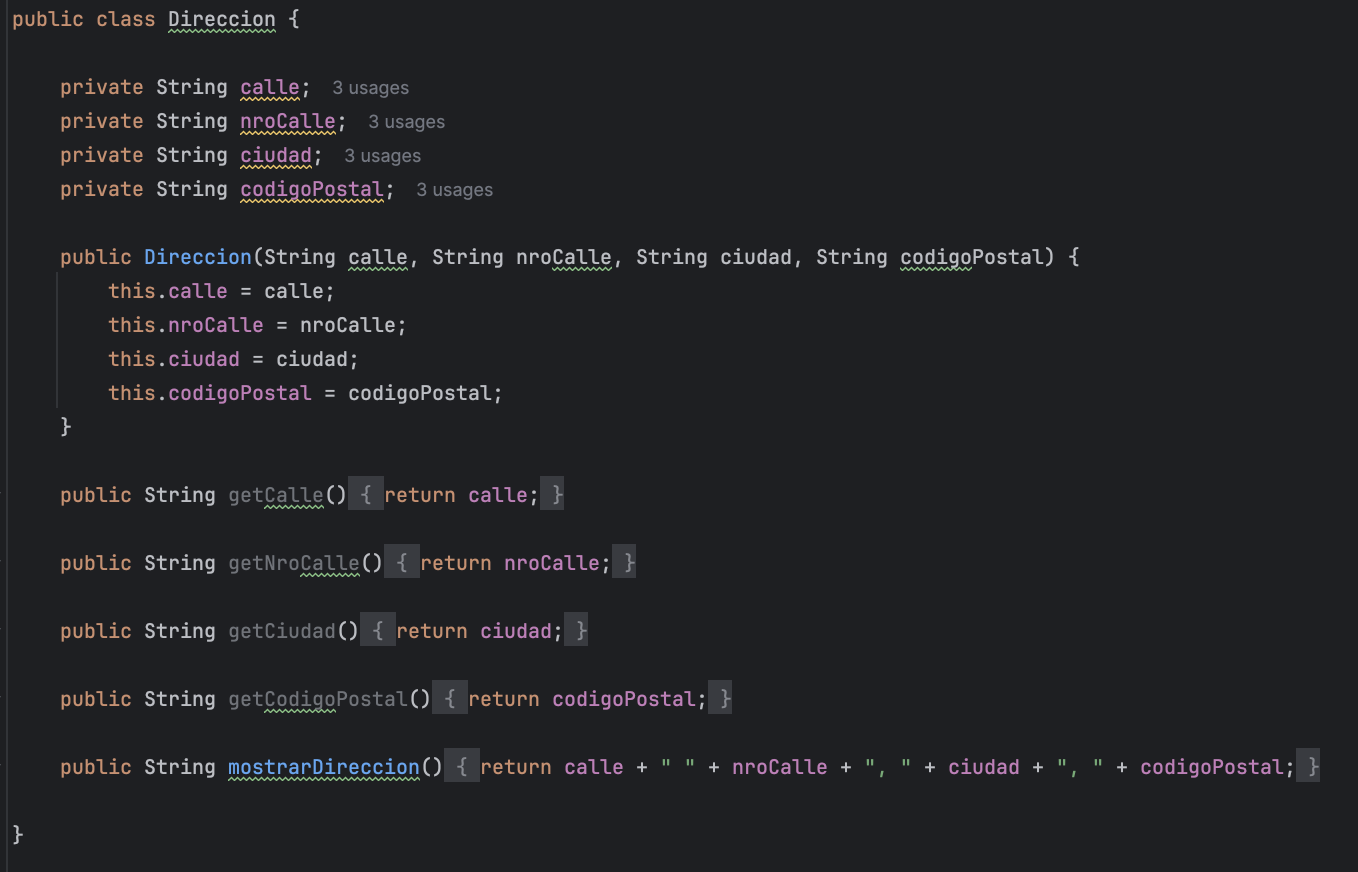
Se manifiesta cuando se usan tipos primitivos para representar entidades que deberían tener un comportamiento específico o cuando se agrupan varios tipos primitivos relacionados que podrían encapsularse en objetos.

Sin refactorizar



Refactorizado





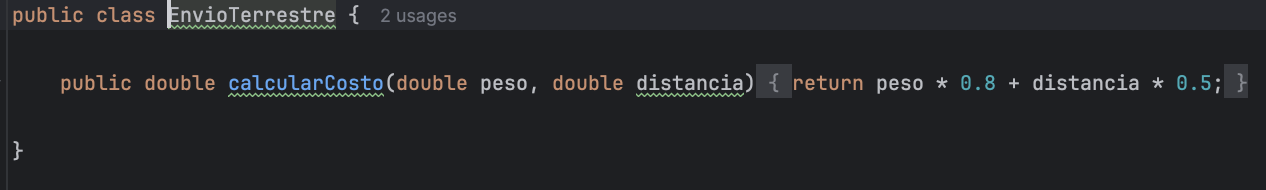


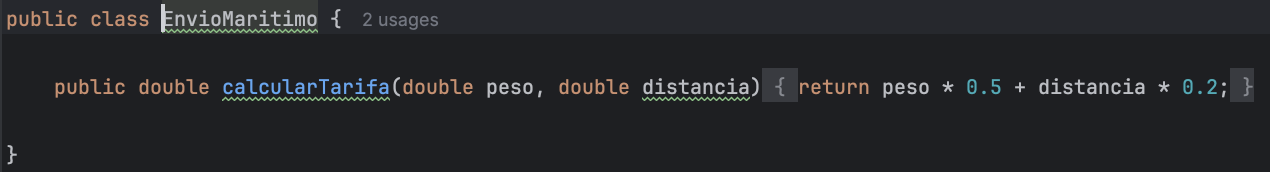
1. **Bad Smell: Alternative Classes with Different Interfaces (clases alternativas con diferentes interfaces)**

Ocurre cuando dos o más clases realizan esencialmente la misma función, pero exponen métodos con diferentes nombres o firmas (parámetros). Esto genera inconsistencia en el código, lo que puede dificultar su uso y mantenimiento, ya que los desarrolladores deben recordar diferentes formas de llamar métodos que hacen lo mismo.

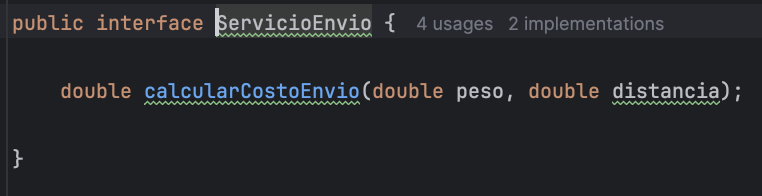
Este bad smell también puede ocurrir cuando diferentes clases están diseñadas para el mismo propósito, pero tienen interfaces incompatibles, lo que complica la integración de estas clases.

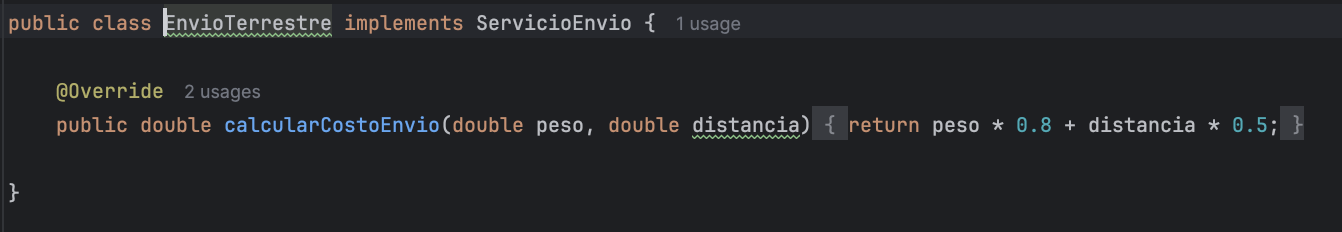
Sin refactorizar

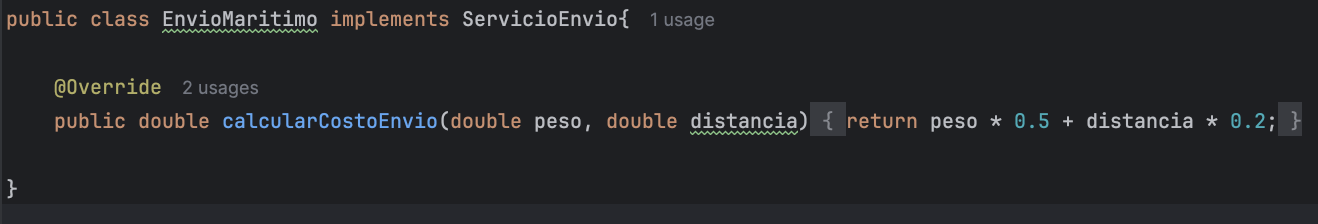




Refactorizado







**23. Bad Smell: Incomplete Library Class (biblioteca de clases incompleta)**

Se presenta cuando usamos una biblioteca de clases de terceros o del propio sistema que no ofrece toda la funcionalidad que necesitamos, pero no podemos modificar directamente el código de la biblioteca.