UCES

Programación 3

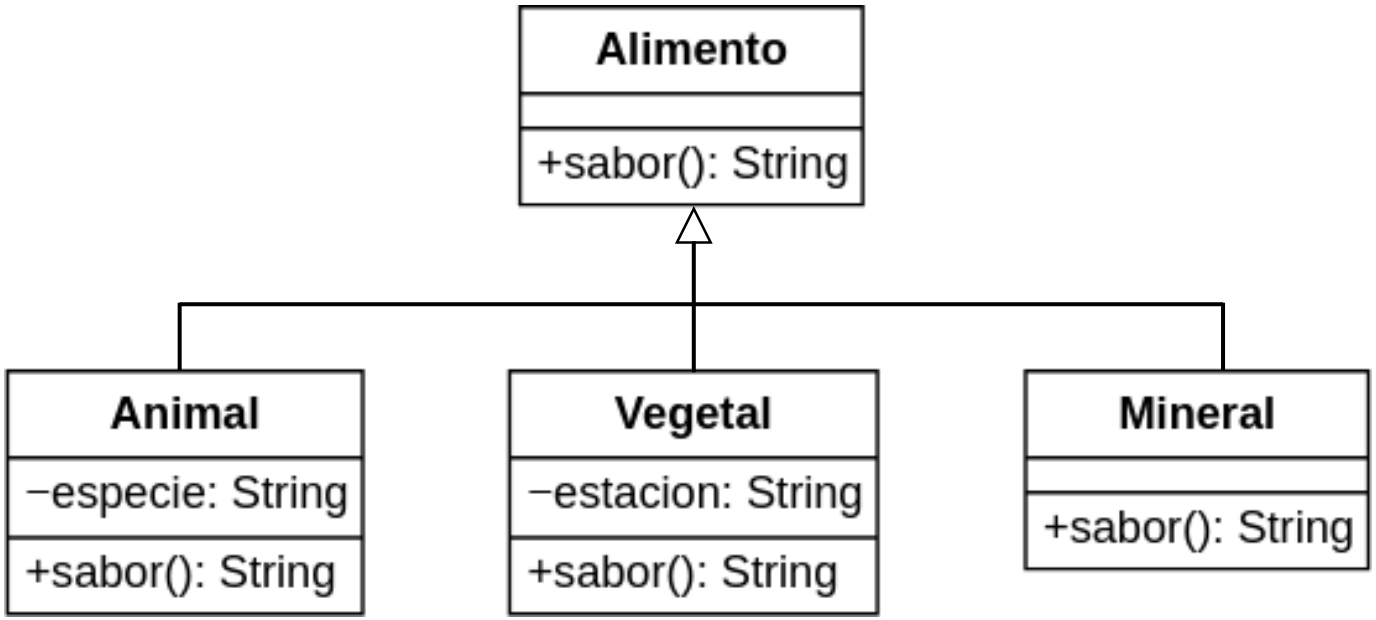
Clases abstractas, interfaces y composición

Arranquemos con un ejemplo claro de herencia...

Se sabe que los alimentos pueden ser clasificados según su origen, en animal, vegetal y mineral

Siendo así, un bife caería en la categoría de animal, un tomate en la de los vegetales y la sal la clasificaríamos con un mineral

Poniéndolo en términos de clases y objetos, en la siguiente jerarquía:



Objeto	Instancia de la clase
bife	Animal
tomate	Vegetal
sal	Mineral

En este ejemplo todos son alimentos, pero cada uno es de un tipo distinto. Y es más, su tipo nunca va a variar: un bife no se va a convertir en un vegetal, ni de repente la sal en un animal

La herencia representa de manera correcta la situación, subclasificando alimentos, sabiendo que su tipo no va a variar durante todo el ciclo de vida de cada uno de ellos

Ahora bien, teniendo los alimentos subclasificados, *¿tiene sentido poder instanciar objetos directamente de la clase* `Alimento`*?*

La respuesta es: probablemente no.

¿Y por qué? Porque si instanciamos un objeto de tipo `Alimento` estaríamos instanciando un objeto *"a medias"*, dado que el concepto `Alimento` es en sí mismo incompleto: sólo se completa de la mano de sus subclases más específicas: `Animal`, `Vegetal` y `Mineral`

Cuando estamos ante un escenario como éste, donde deberíamos limitar, por diseño, la instanciación de alguna clase *"incompleta"*, podemos recurrir a definir dicha clase como *abstracta*

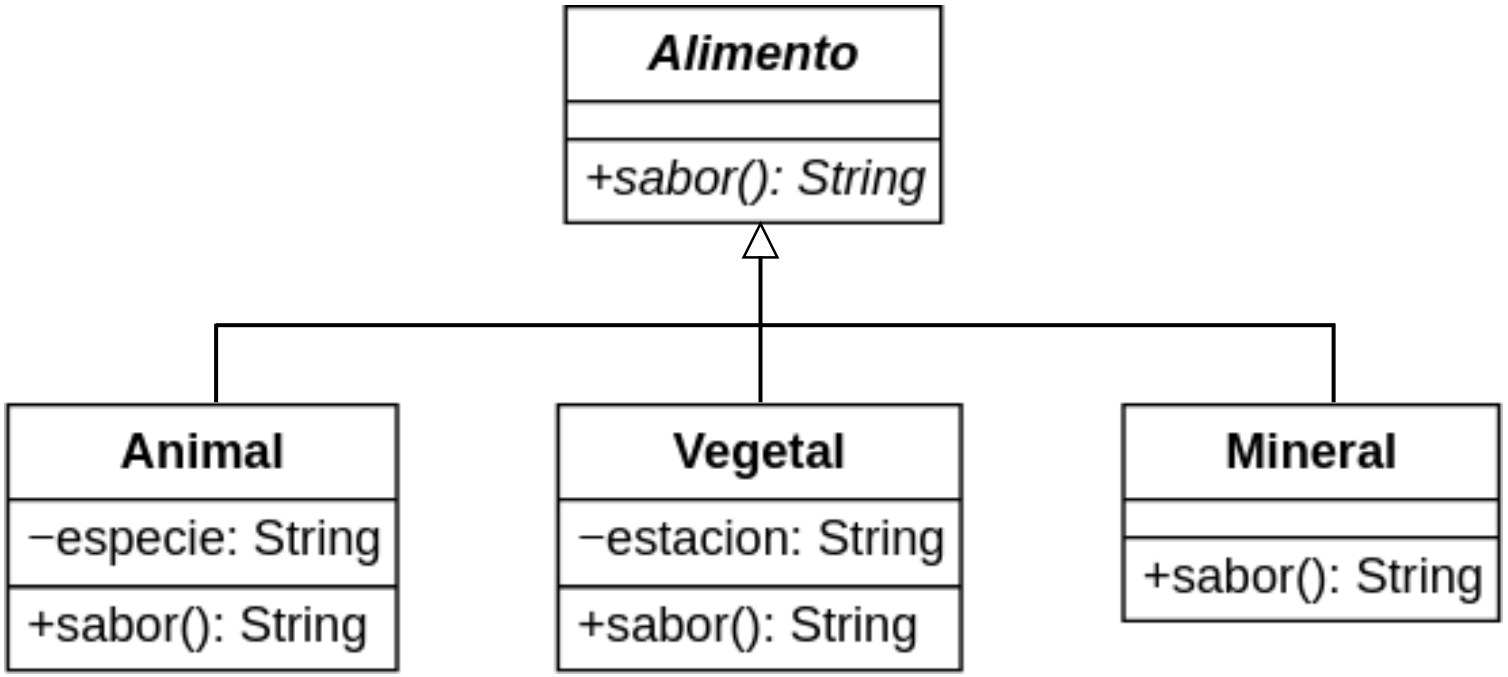
Una clase *concreta* es aquella que puede ser instanciada, creándose objetos a partir de ella

Una clase *abstracta* es aquella que no puede ser instanciada. Su razón de ser es la especialización, es decir, ser heredada o extendida por subclases más específicas

Una clase abstracta, usualmente, no tiene todas sus operaciones implementadas, sino que sólo especifica la firma de las mismas

Es como si la clase dijera: *"tengo estos comportamientos, pero dejo su implementación concreta a mis subclases"*

Teniendo todo esto en cuenta, nuestro ejemplo quedaría de la siguiente manera:



Donde la clase `Alimento` es abstracta por contar con una operación abstracta, `sabor():String`

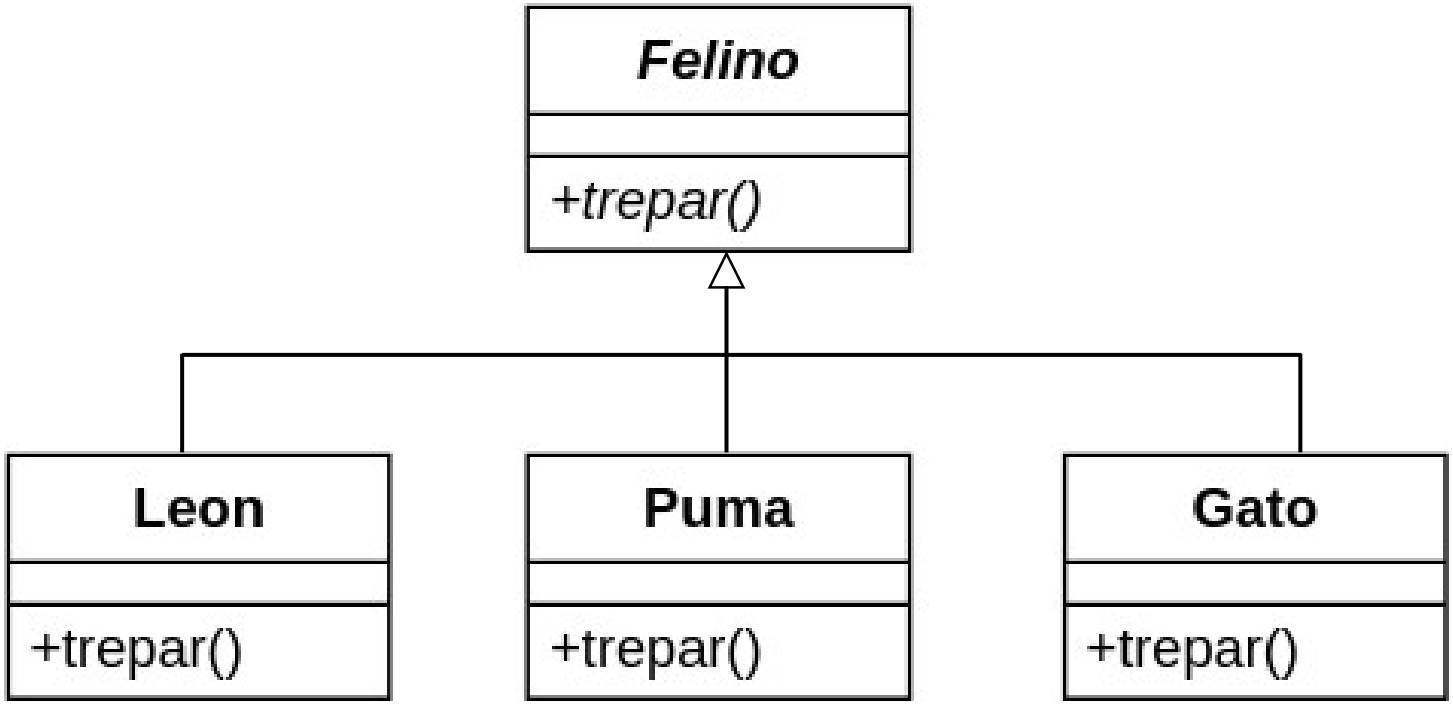
Ventajas de clases abstractas...

- Son muy útiles para definir jerarquías complejas, dando libertad a las subclases a implementar sus comportamientos, sin necesidad de que éstos estén definidos *"artificialmente"* en una superclase en pos de la uniformidad que otorga el polimorfismo
- Pueden definir operaciones concretas además de las abstractas

La otra cara de la moneda...

Muchos lenguajes de programación sólo permiten la herencia simple, limitando a veces el diseño

Pensemos en el siguiente ejemplo...



Aquí las clases `Leon`, `Puma` y `Gato` *heredan* características y comportamiento de la clase abstracta `Felino`, y están obligadas a darle implementación a sus operaciones abstractas

¿Qué pasaría si, por ejemplo, quisiésemos expresar que un `Gato` además de *ser un* `Felino` a su vez *es una* `Mascota`?

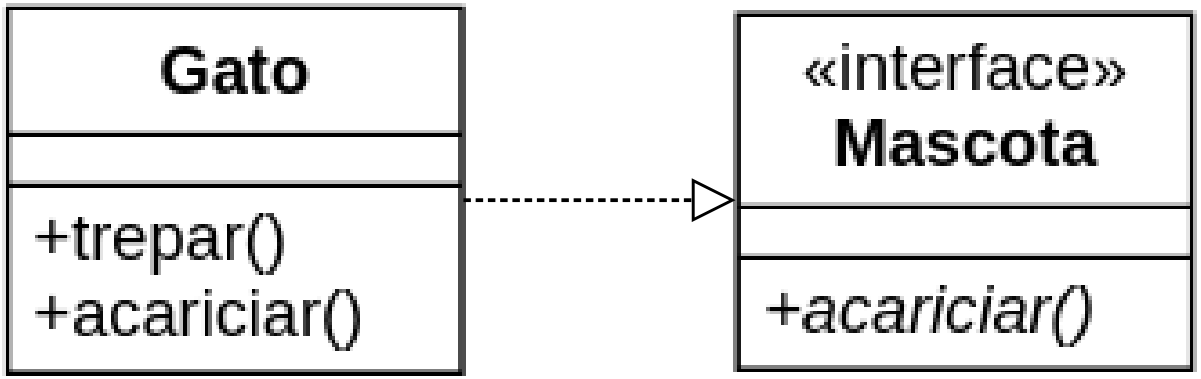
Aquí es cuando aparece el concepto de *interfaz*

Las *interfaces* definen un conjunto de operaciones abstractas para ser implementadas por las clases que así lo deseen

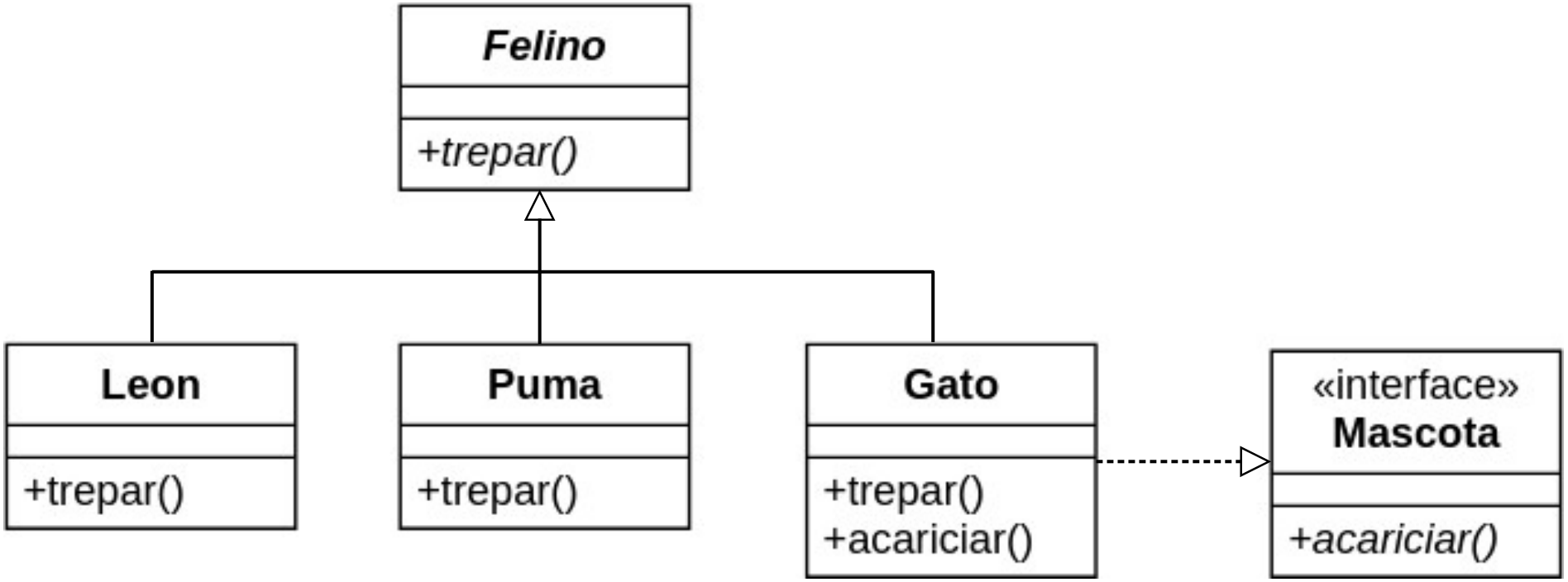
Son utilizadas cuando uno necesita declarar las operaciones que las clases concretas deben implementar, pero no se lo quiere (o puede) hacer a través de una clase abstracta

Una interfaz se puede pensar como un contrato que declara: *"éstas son las operaciones que deben ser implementadas por las clases que deseen cumplir con este contrato"*

En nuestro ejemplo, el *contrato* establecería qué debe implementar una clase que quiere comportarse como `Mascota`, en este caso la operación `acariciar()`:



El diseño entonces quedaría de la siguiente manera...



De esta manera, un gato *hereda* características y comportamiento de la clase `Felino`, y además *implementa* los comportamientos dados por la interfaz `Mascota`

A tener en cuenta...

- una interfaz no puede tener operaciones concretas, pero sí atributos, aunque estos suelen ser constantes y definidos a nivel de clase
- cuando una clase implementa una interfaz, se dice que la realiza, indicándose con una línea punteada terminada en un triángulo vacío que apunta a la interfaz
- cuando definimos un atributo, su tipo podrá ser tanto del tipo específico de la clase como de cualquiera de las interfaces que ésta implementa

```
Felino gato
Gato gato
Mascota gato
```

Los tres atributos definidos podrían contener instancias de la clase `Gato`

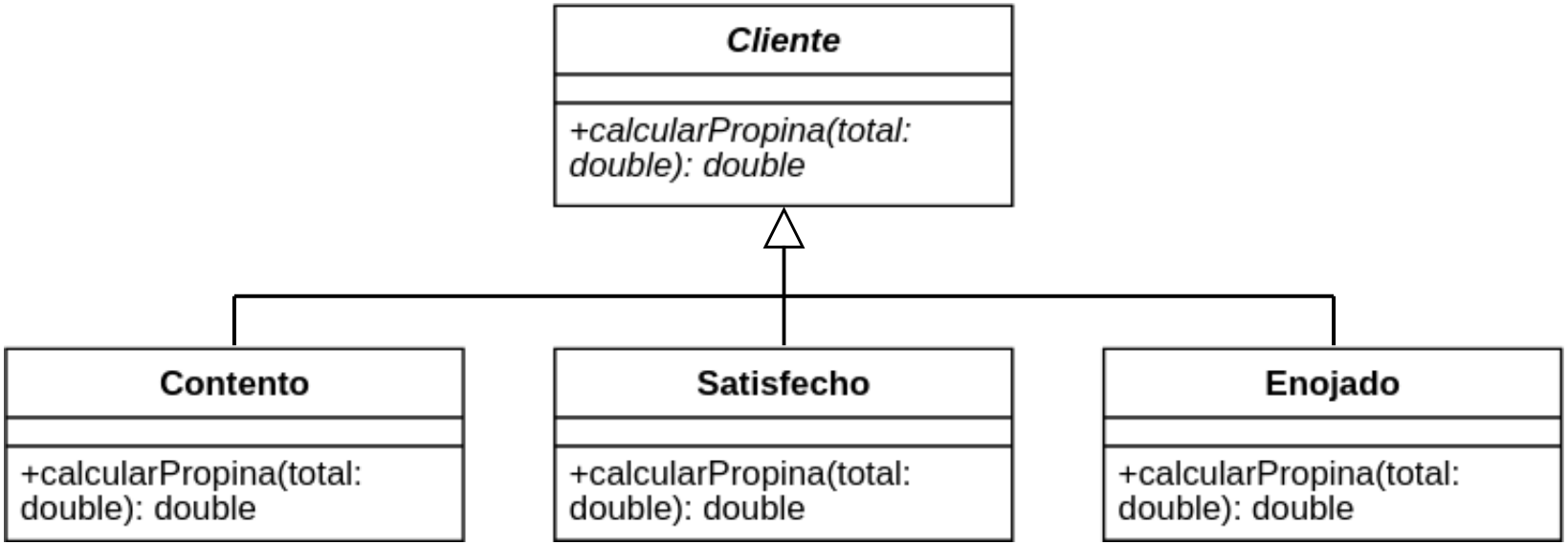
Herencia vs Composición...

Vamos con otro ejemplo...

Pensemos en un cliente que va a un restaurant y al finalizar su comida debe elegir cuánta propina dejarle al mozo. Su elección estará dada por su humor:

- si está *contento* dejará un 15% de propina
- si está *satisfecho* dejará un 10% de propina
- si está *enojado*, no dejará nada

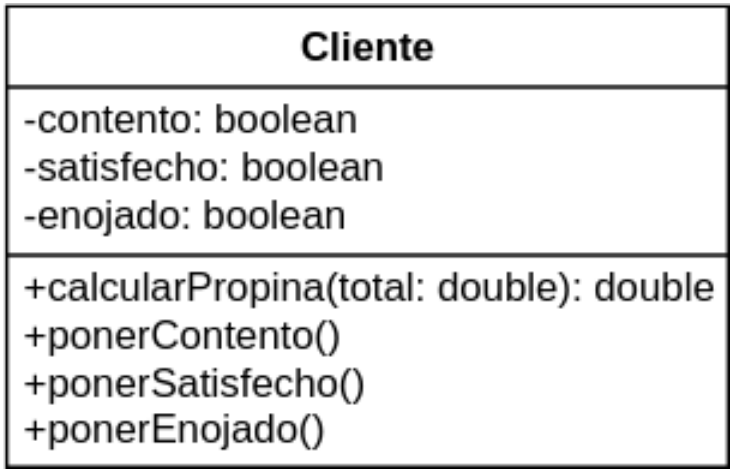
Primera aproximación: jerarquía de clientes



Problema

La herencia es una relación rígida => un cliente que se crea a partir de la clase `Contento` morirá contento, y uno que sea instancia de `Enojado` será un limonagrio por el resto de la eternidad (o hasta que termine el programa)

Segunda aproximación: una única clase `Cliente` con booleanos



Problema $\rightarrow$ *Poca mantenibilidad*

Segunda aproximación: una única clase `Cliente` con booleanos

```
ponerContento() {
  contento = true;
  if (satisfecho) satisfecho = false;
  if (enojado) enojado = false;
}
```

Problema → Poca mantenibilidad

- es necesario que los booleanos correspondientes al humor estén *sincronizados*

Segunda aproximación: una única clase `Cliente` con booleanos

```
ponerContento() {
  contento = true;
  if (satisfecho) sa
  if (enojado) enoja
}
```

```
if (contento) {
  return monto * 0.15;
} else if (satisfecho) {
  return monto * 0.10;
} else if (enojado) {
  return 0;
}
```

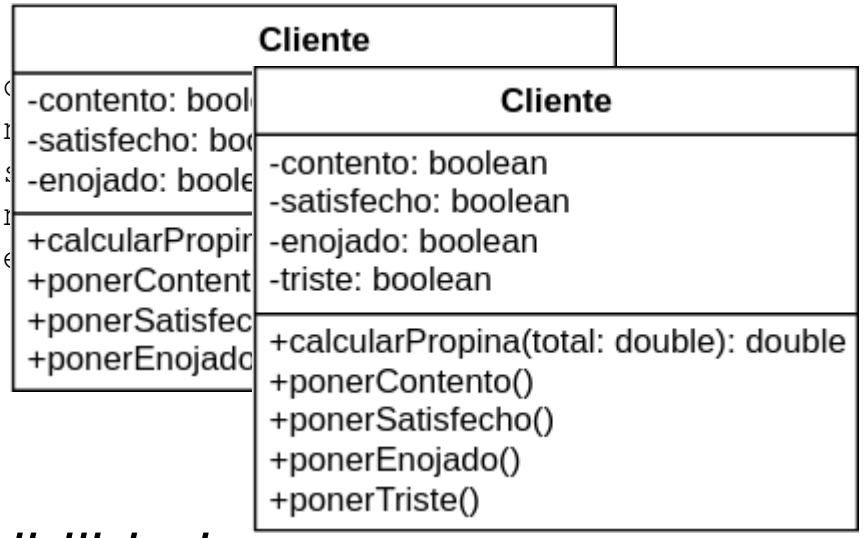
Problema → Poca mantenibilidad

- es necesario que los booleanos correspondientes al humor estén *sincronizados*
- los condicionales a la hora de preguntarse cuánta propina debe dejar pueden tornarse complejos

Segunda aproximación: una única clase `Cliente` con booleanos

```
ponerContento() {
  contento = true;
  if (satisfecho) sa
  if (enojado) enoja
}
```

```
if (content
  return mor
} else if (s
  return mor
} else if (e
  return 0;
}
```



Problema → Poca mantenibilidad

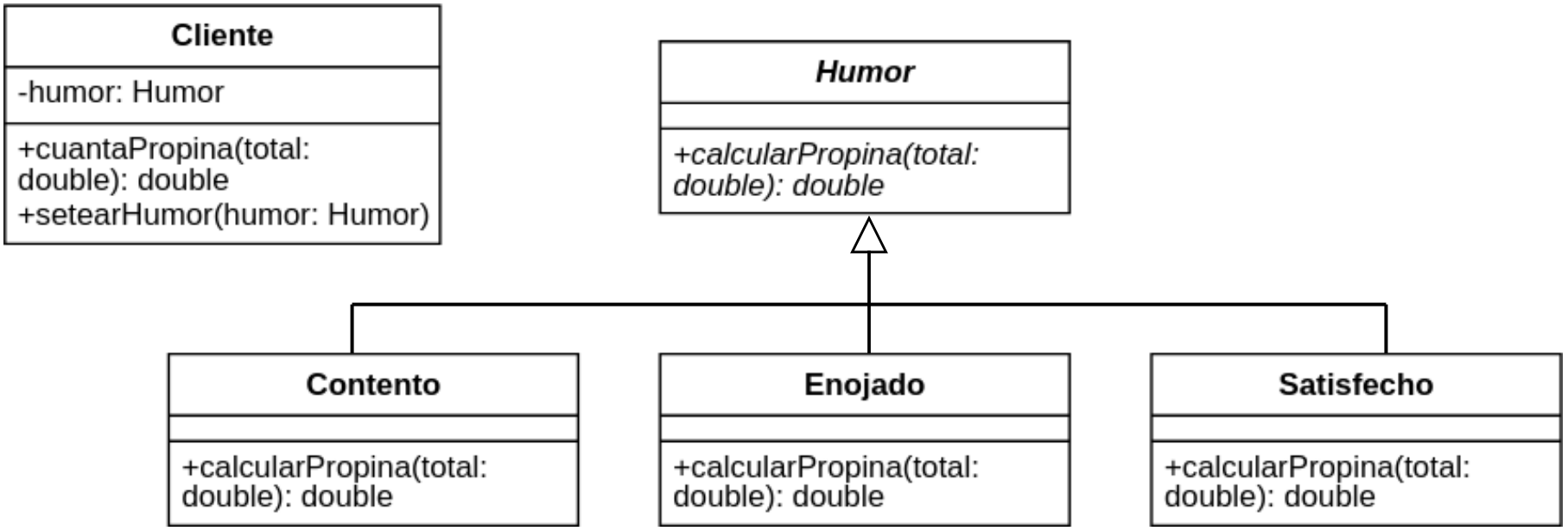
- es necesario que los booleanos correspondientes al humor estén *sincronizados*
- los condicionales a la hora de preguntarse cuánta propina debe dejar pueden tornarse complejos
- cada vez que querramos introducir un nuevo humor, deberemos modificar la clase y los condicionales correspondientes

¿Y si la tercera es la vencida?

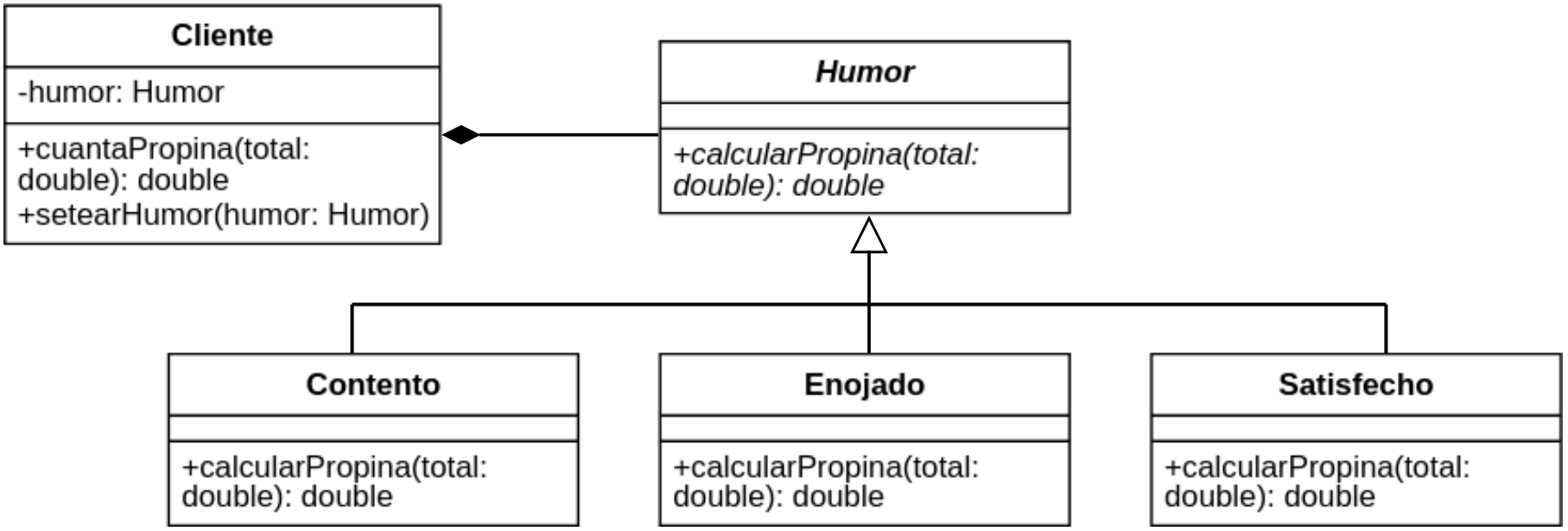
Si analizamos un poco más la situación, podríamos:

- definir al *humor* del cliente como una *característica* propia del cliente
- definir distintos *tipos de humor*: contento, satisfecho, enojado

Si representamos esto, tendríamos las siguientes piezas:



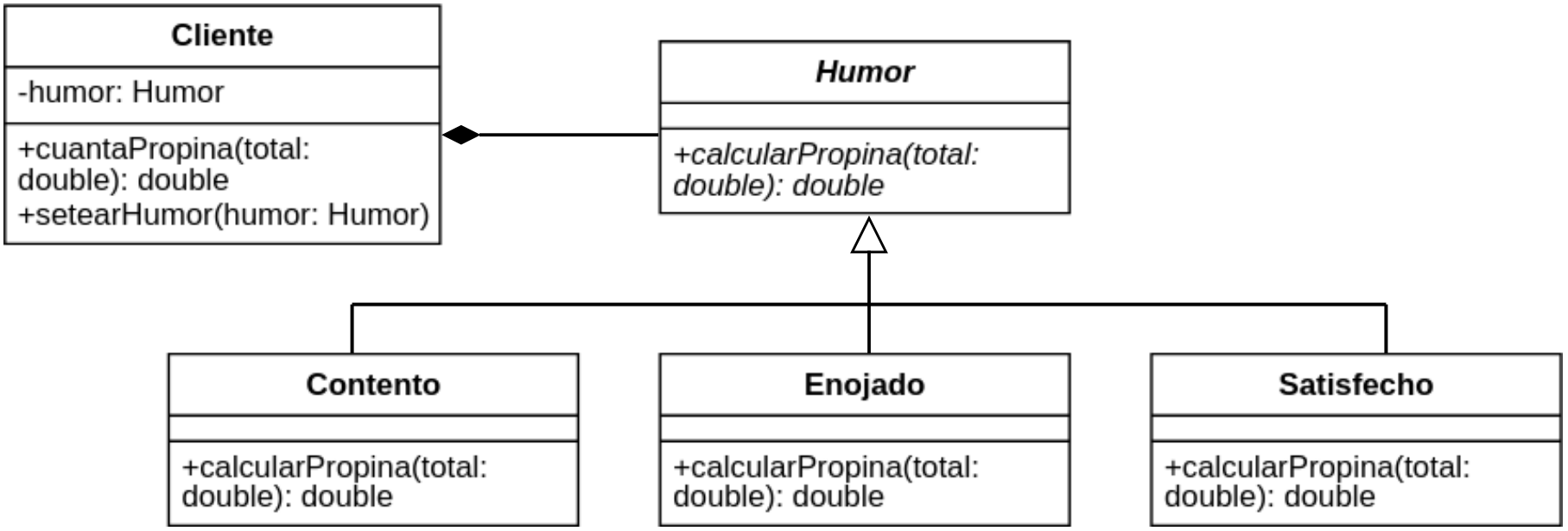
Si representamos esto, tendríamos las siguientes piezas:



¿Y cómo las unimos?

Con una asociación, pero no cualquier asociación, sino con una **composición**

Si representamos esto, tendríamos las siguientes piezas:



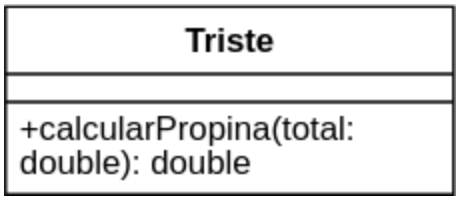
Al realizar este planteo, se puede ver que el *cliente* no puede *funcionar* sin su *humor*, ya que es el *humor* el que termina decidiendo la cantidad de propina. Es entonces una relación fuerte, donde el humor es parte del cliente.

¿Cuáles son las ventajas de esta solución?

- desacoplamos el humor del cliente, pero hacemos que ambos trabajen en conjunto
- la lógica referida a la cantidad de propina está contenida en cada clase de humor, en lugar de estar embebida en un conjunto de condicionales
- agregar un humor es tan sencillo como crear una subclase de `Humor`, que implemente el método `calcularPropina():double`
- agregar un nuevo humor no afecta al cliente

Si necesitamos agregar un nuevo humor...

Lo *modelamos*, incluyendo toda su lógica asociada...



Lo *relacionamos* con nuestra jerarquía...