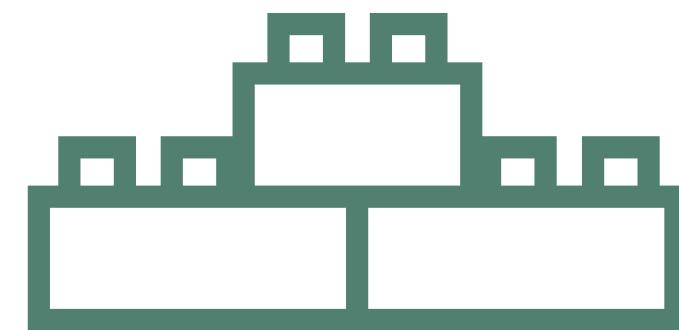


Tópicos Especiales de Programación

Algunos Patrones de Diseño

Recordando un poco la POO

¿Qué son los patrones de diseño?



Singleton

Patrón creacional que nos permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.

Problema

Garantizar que una clase tenga una única instancia

Controlar el acceso a algún recurso compartido, por ejemplo, una base de datos o un archivo.

Creamos un objeto y al cabo de un tiempo queremos crear otro nuevo. En lugar de recibir un objeto nuevo, obtendremos el que ya habíamos creado.

Proporcionar un punto de acceso global a dicha instancia

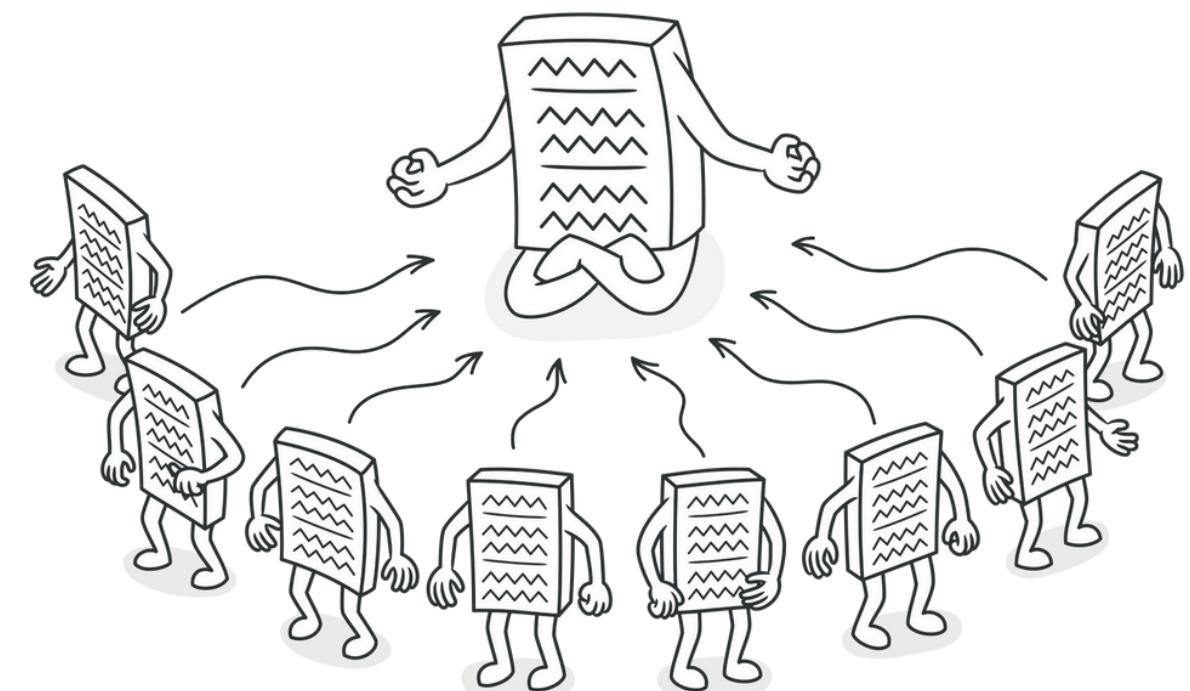
Las variables globales suelen ser útiles, pero también son poco seguras, ya que cualquier código podría sobrescribir el contenido de esas variables y descomponer la aplicación.

Al igual que una variable global, el patrón Singleton nos permite acceder a un objeto desde cualquier parte del programa. No obstante, también evita que otro código sobreesciba esa instancia.

Solución

- Hacer privado el constructor por defecto para evitar que otros objetos utilicen el operador **new** con la clase Singleton.
- Crear un método de creación estático que actúe como constructor. Este método invoca al constructor privado para crear un objeto y lo guarda en un campo estático. Las siguientes llamadas a este método devuelven el objeto almacenado en “caché”.

Si tu código tiene acceso a la clase Singleton, podrá invocar su método estático. De esta manera, cada vez que se invoque este método, siempre se devolverá el mismo objeto.



Analogía en el mundo real

La torre de control de un aeropuerto

En cada aeropuerto hay una única torre de control que coordina despegues y aterrizajes. Y si existieran dos torres independientes, podrían dar órdenes contradictorias y provocar accidentes.

Por lo tanto:

- Única instancia (**Singleton**): una sola torre por aeropuerto.
- Punto de acceso global (**getInstance**): todos los aviones usan la misma frecuencia para comunicarse con esa torre.
- Estado compartido: la torre mantiene un estado único (qué pista está libre, clima, prioridades).
- Constructor privado: no “levantas” otra torre a mitad de la operación.

¿Qué pasa si hubiera dos “instancias”?

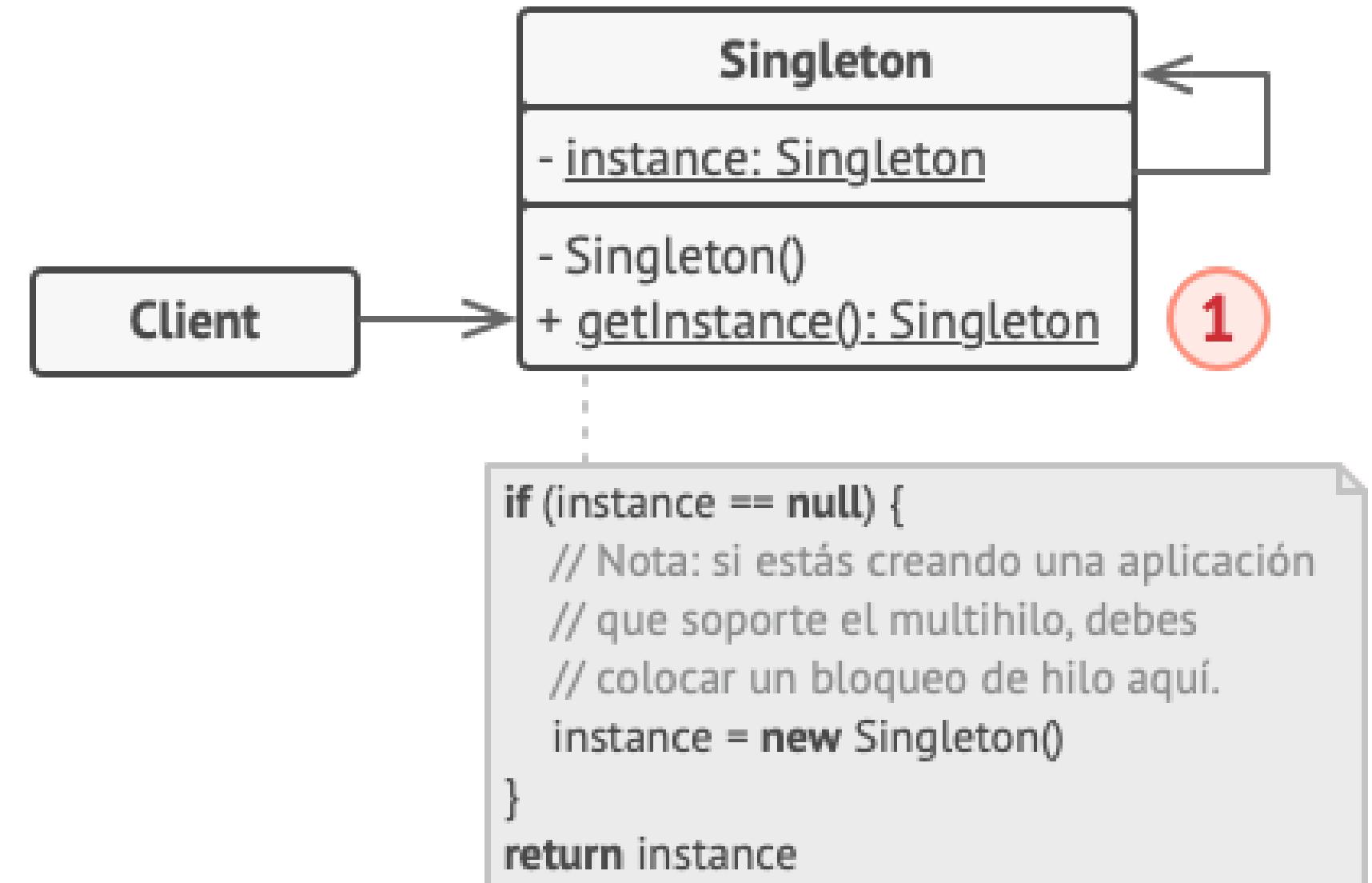
Un avión recibe “autorizado a aterrizar” de una torre y “espere en el aire” de la otra, teniendo como resultado:



Estructura

La clase **Singleton** declara el método estático `getInstance()` que devuelve la misma instancia de su propia clase.

El constructor del **Singleton** debe ocultarse del código cliente. La llamada al método `getInstance()` debe ser la única manera de obtener el objeto de **Singleton**.



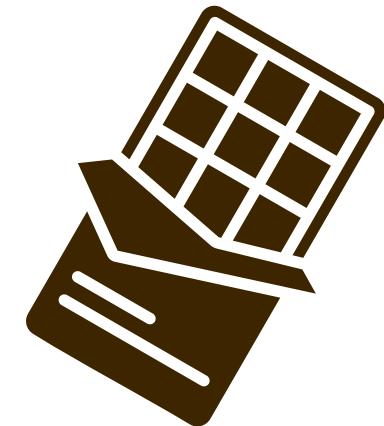
Ejemplo

Chocolate Boiler

Todo el mundo sabe que las fábricas modernas de chocolate tienen calderas controladas por computadora. El trabajo de la caldera es recibir chocolate y leche, llevarlos a ebullición y luego enviarlos a la siguiente fase de fabricación de tabletas.

La fábrica chocolatera ha hecho un trabajo decente para garantizar que no ocurran cosas malas, ¿no crees?

Pero claro, probablemente sospechas que si llegan a existir dos instancias de caldera, pueden ocurrir consecuencias muy graves.



¿Qué podría salir mal si se crea más de una instancia de caldera en una aplicación?

Planteamiento

```
export class ChocolateBoiler {  
    private empty: boolean;  
    private boiled: boolean;  
  
    constructor() {  
        this.empty = true;  
        this.boiled = false;  
    }  
  
    fill(): void {  
        if (this.isEmpty()) {  
            this.empty = false;  
            this.boiled = false;  
            // llenar con mezcla de leche/chocolate  
        }  
    }  
  
    drain(): void {  
        if (!this.isEmpty() && this.isBoiled()) {  
            // drenar mezcla hervida  
            this.empty = true;  
        }  
    }  
  
    boil(): void {  
        if (!this.isEmpty() && !this.isBoiled()) {  
            // hervir contenido  
            this.boiled = true;  
        }  
    }  
  
    isEmpty(): boolean {  
        return this.empty;  
    }  
  
    isBoiled(): boolean {  
        return this.boiled;  
    }  
}
```

Resuelto

La idea es “asegurar una sola caldera” en toda la app. Dos instancias permitirían estados contradictorios (una cree que está vacía mientras la otra hierva), así que centralizamos el estado en un único objeto global controlado.

```
class ChocolateBoiler {  
    private static instance: ChocolateBoiler | null = null;  
  
    private empty: boolean;  
    private boiled: boolean;  
  
    // Constructor privado: evita nuevas instancias desde fuera  
    private constructor() {  
        this.empty = true;  
        this.boiled = false;  
    }  
  
    // Punto de acceso global a la única instancia  
    static getInstance(): ChocolateBoiler {  
        if (!this.instance) {  
            this.instance = new ChocolateBoiler();  
        }  
        return this.instance;  
    }  
  
    fill(): void {  
        if (this.isEmpty()) {  
            this.empty = false;  
            this.boiled = false;  
            // llenar con mezcla de leche/chocolate  
        }  
    }  
    // resto del código...  
}
```

Factory

Patrón creacional que proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán.

Problema

Imagina que trabajas en el mostrador de una pizzería. Cuando un cliente pide una pizza de pepperoni, tú mismo vas a la cocina, tomas la masa, le pones salsa, queso y pepperoni. Si luego piden una de champiñones, repites el proceso con los ingredientes correctos.

El problema es que tú, en el mostrador, necesitas saber la receta exacta de cada pizza. Si el dueño decide añadir una pizza "Hawaiana", tienes que aprenderte la nueva receta. Tu trabajo se vuelve más complicado con cada cambio en el menú.

En términos de código, esto significa que la parte de tu programa que recibe las peticiones (el "**cliente**") está fuertemente **acoplada** a la lógica de creación (**new PepperoniPizza()**, **new MushroomPizza()**). Cualquier cambio en los "productos" te obliga a modificar al "cliente".

Solución

El patrón Factory propone contratar a un **Chef de Pizzas (una Fábrica)** que se queda en la cocina.

Ahora, cuando un cliente pide una de pepperoni, tú ya no vas a la cocina. Ahora se la pides a él, porque conoce la receta y te entrega la pizza lista.

La gran ventaja es que tu única responsabilidad es tomar el pedido y pasárselo al chef. Si mañana se añade la pizza hawaiana, el único que necesita aprender la nueva receta es el chef. Tu trabajo en el mostrador no cambia en absoluto.

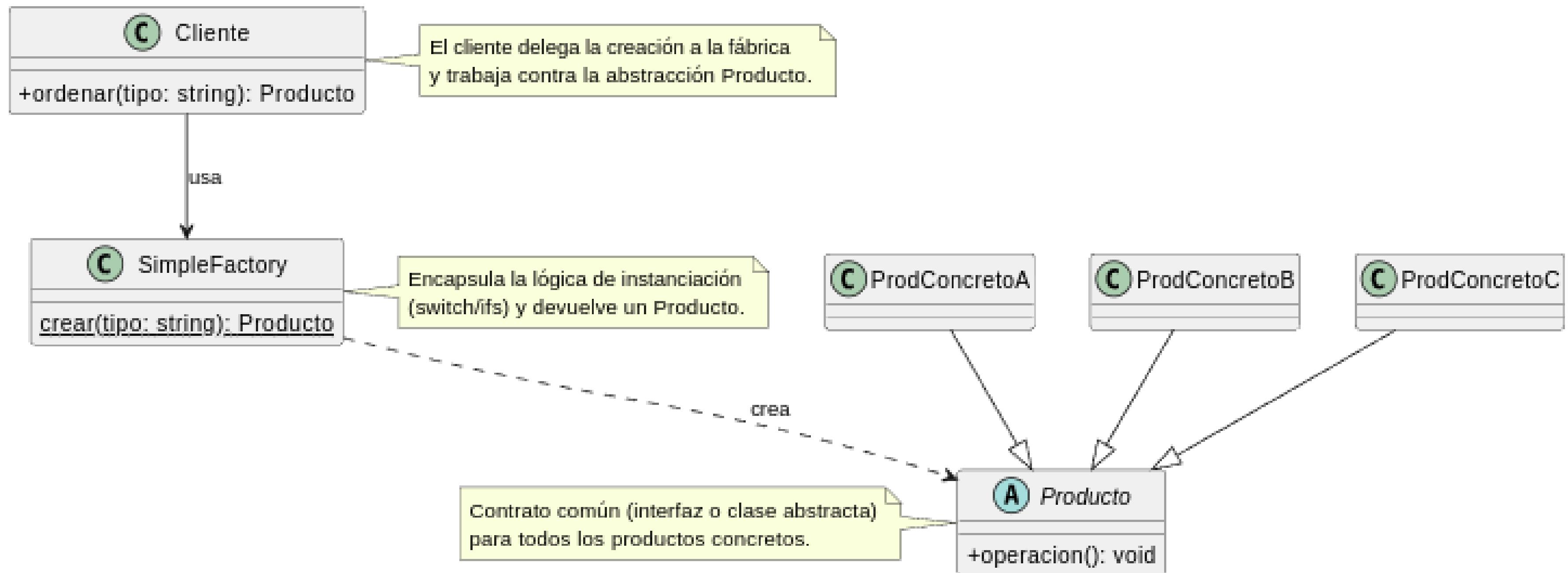
- Cliente: Eres tú en el mostrador, tomando el pedido.
- Fábrica (**SimplePizzaFactory**): Es el chef. Es el único que conoce los detalles de creación. Tiene un método como **crearPizza(tipo)**.
- Producto: Es la interfaz o clase base **Pizza**. Tú en el mostrador solo sabes que vas a recibir "una pizza", no te importa el detalle interno de sus ingredientes.
- Productos Concretos: Son las clases **PepperoniPizza**, **MushroomPizza**, etc. El chef es quien las instancia (**new PepperoniPizza()**).

El objetivo principal es **encapsular la lógica de creación de objetos**.

El cliente se desacopla de la instanciación de las clases concretas, delegando esa responsabilidad a la fábrica. Facilitamos el mantenimiento y la extensibilidad.

Estructura

Simple Factory — Diagrama General



Planteamiento

Paso 1:

La función **orderPizza**
crea una Pizza concreta
y la procesa.

Pero...

*Ahora queremos más
tipos de pizza...*

```
class Pizza {  
    prepare() { console.log("Preparing dough, sauce, toppings..."); }  
    bake() { console.log("Baking at 350°F for 25 minutes"); }  
    cut() { console.log("Cutting the pizza into slices"); }  
    box() { console.log("Boxing the pizza"); }  
}  
  
function orderPizza(): Pizza {  
    const pizza = new Pizza();  
  
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
    return pizza;  
}  
  
// Ejemplo  
orderPizza();
```

Paso 2:

Ahora pizza es una interfaz, añadimos tipos concretos y lógica condicional para instanciarlos.

Pero...

Ahora cambian las pizzas del menu...

El cliente (`orderPizza`) queda acoplado a las clases concretas.

```
interface Pizza {  
    prepare(): void;  
    bake(): void;  
    cut(): void;  
    box(): void;  
}  
  
class CheesePizza implements Pizza { /* Implementaciones */ }  
class GreekPizza implements Pizza { /* Implementaciones */ }  
class PepperoniPizza implements Pizza { /* Implementaciones */ }  
  
function orderPizza(type: String): Pizza {  
    let pizza: Pizza;  
  
    if (type === "cheese") {  
        pizza = new CheesePizza();  
    } else if (type === "greek") {  
        pizza = new GreekPizza();  
    } else if (type === "pepperoni") {  
        pizza = new PepperoniPizza();  
    } else {  
        throw new Error("Unknown pizza type");  
    }  
  
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
    return pizza;  
}
```

Paso 3:

Crecen las variantes (**clam**, **veggie**) y se elimina **greek**.
if-else escala mal: cada cambio de menú obliga a editar **orderPizza**.

Tener que lidiar con qué clase concreta instanciamos (qué pizza) está haciendo que **orderPizza** sufra muchos cambios y ya no está "*Cerrada para Modificación*".

Pero sabemos qué pieza del código es la culpable de estas modificaciones, y debemos encapsular!

```
// Nuevas clases de pizzas...

function orderPizza(type: String): Pizza {
    let pizza: Pizza;

    if (type === "cheese") {
        pizza = new CheesePizza();
    } else if (type === "pepperoni") {
        pizza = new PepperoniPizza();
    } else if (type === "clam") {
        pizza = new ClamPizza();
    } else if (type === "veggie") {
        pizza = new VeggiePizza();
    } else {
        throw new Error("Unknown pizza type");
    }

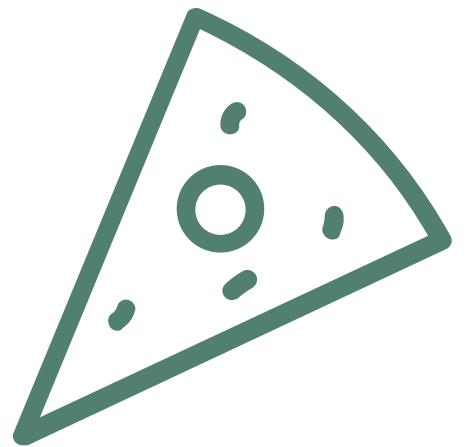
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
}

// Ejemplo
orderPizza("veggie");
```

Resuelto

Encapsulamos al culpable de estas modificaciones mediante una **Fabrica**.

La función **orderPizza()** debe ser un método de una clase **PizzaStore**



```
// clases de pizzas...

class SimplePizzaFactory {
    createPizza(type: String): Pizza {
        switch (type) {
            case "cheese": return new CheesePizza();
            case "pepperoni": return new PepperoniPizza();
            case "clam": return new ClamPizza();
            case "veggie": return new VeggiePizza();
            default:
                throw new Error(`Unknown pizza type: ${type}`);
        }
    }
}

class PizzaStore {
    let factory: SimplePizzaFactory

    constructor(SimplePizzaFactory factory) {
        this.factory = factory
    }

    function orderPizza(type: String): Pizza {
        let pizza: Pizza;

        pizza = factory.createPizza(type)

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
}
```

Strategy

Patrón de comportamiento que te permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer sus objetos intercambiables.

Strategy deja que sus algoritmos varíen independientemente de los clientes que los usen.

Abordaremos el patrón Strategy de una manera un tanto distinta.

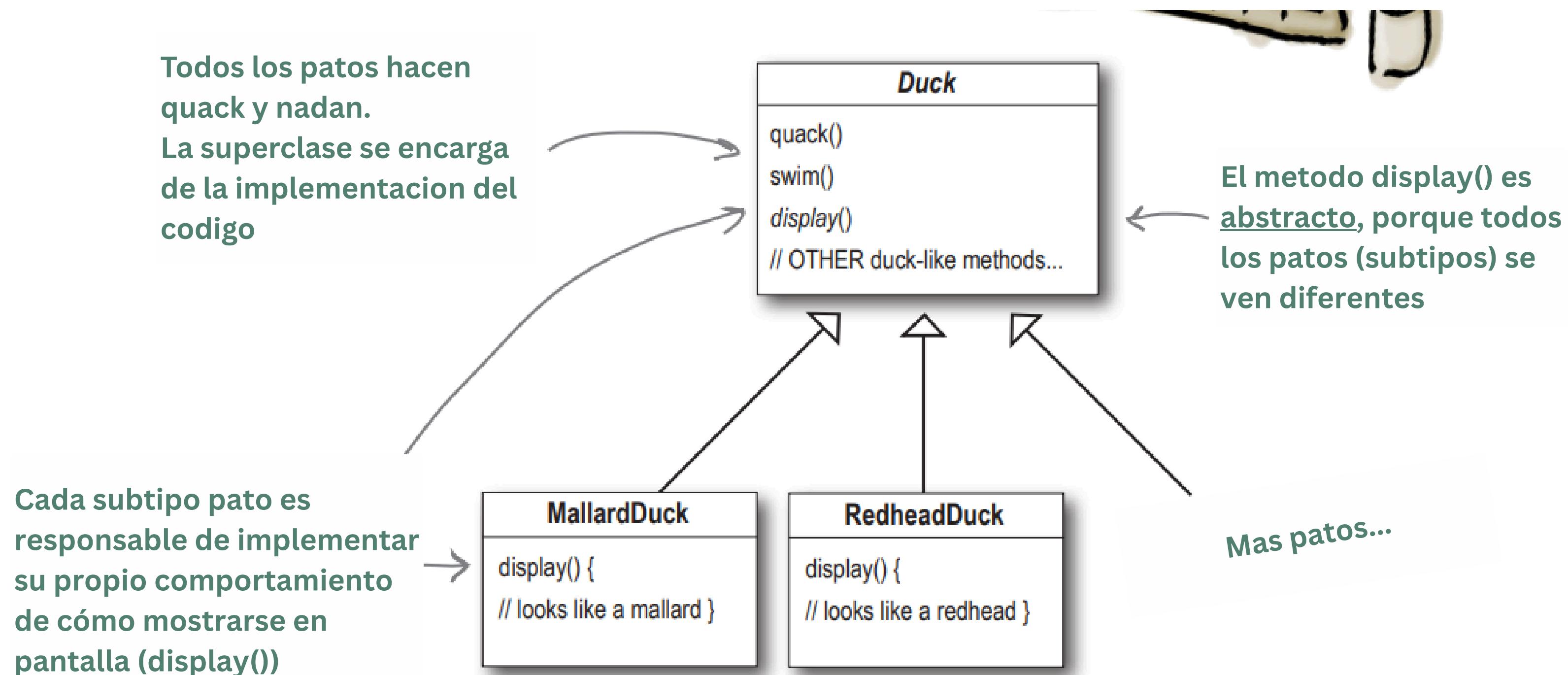
Ahora estamos trabajando en un **juego** de simulación sobre **patos**, en el juego hay una gran variedad de especies, y cada una de ellos pueden **nadar** y **hacer quack**.

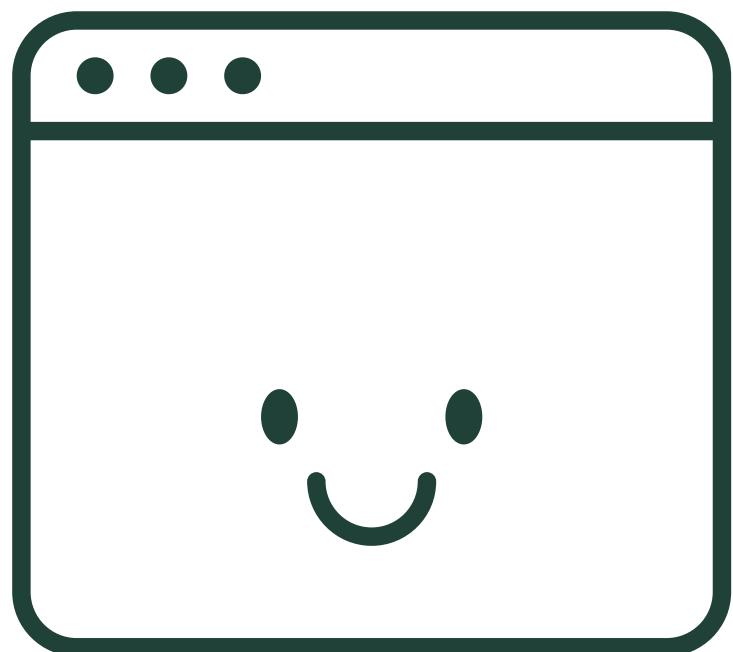


Sencillo, ¿Verdad?

¿Como lo resloverian ustedes?

Nuestro planteamiento inicial es el siguiente:





¡Resolvimos todo de una manera rápida y sencilla!
Nuestro trabajo aquí ha terminado, podemos
descansar y dejar que la POO siga con su magia.

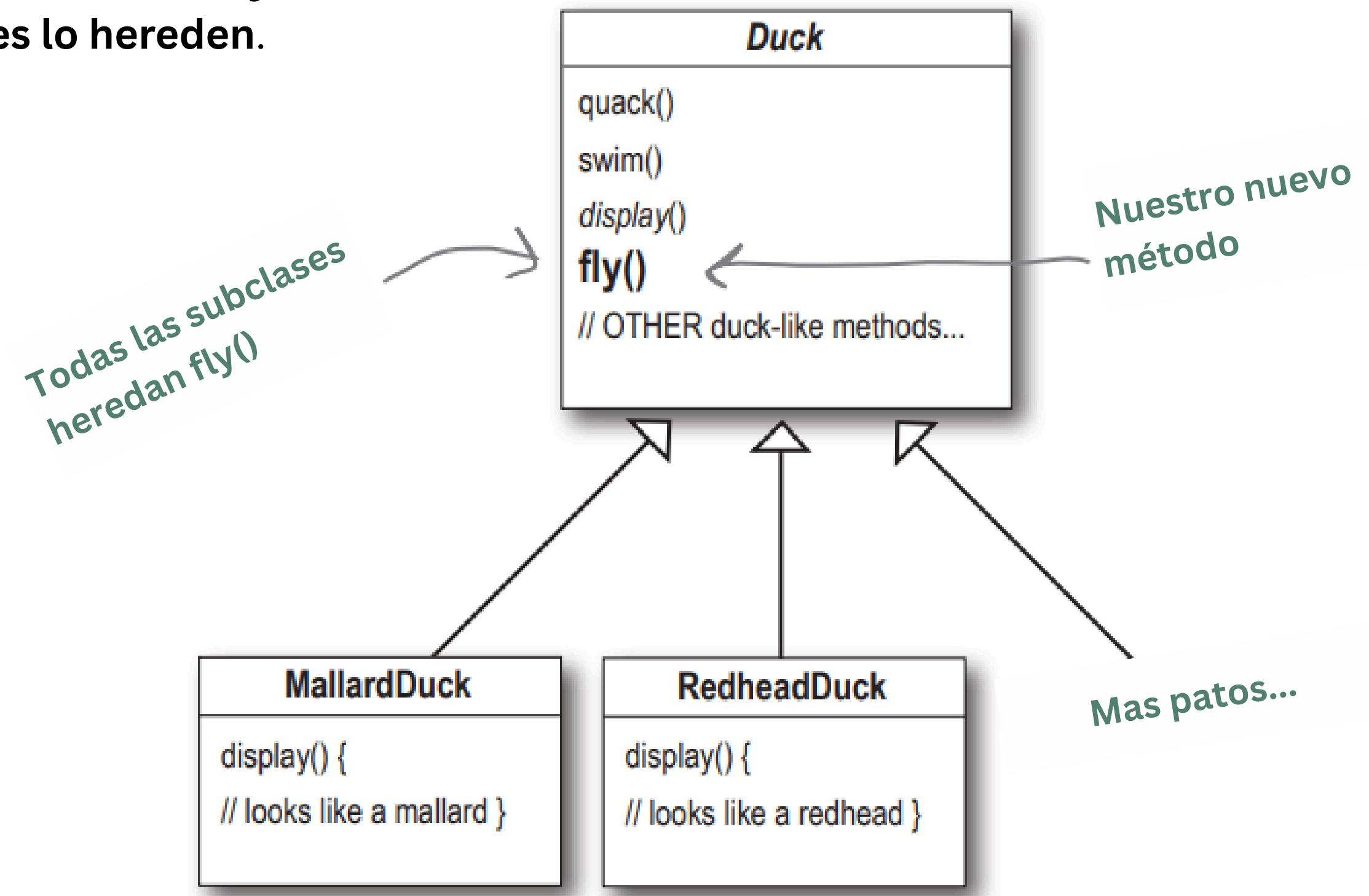
Fin



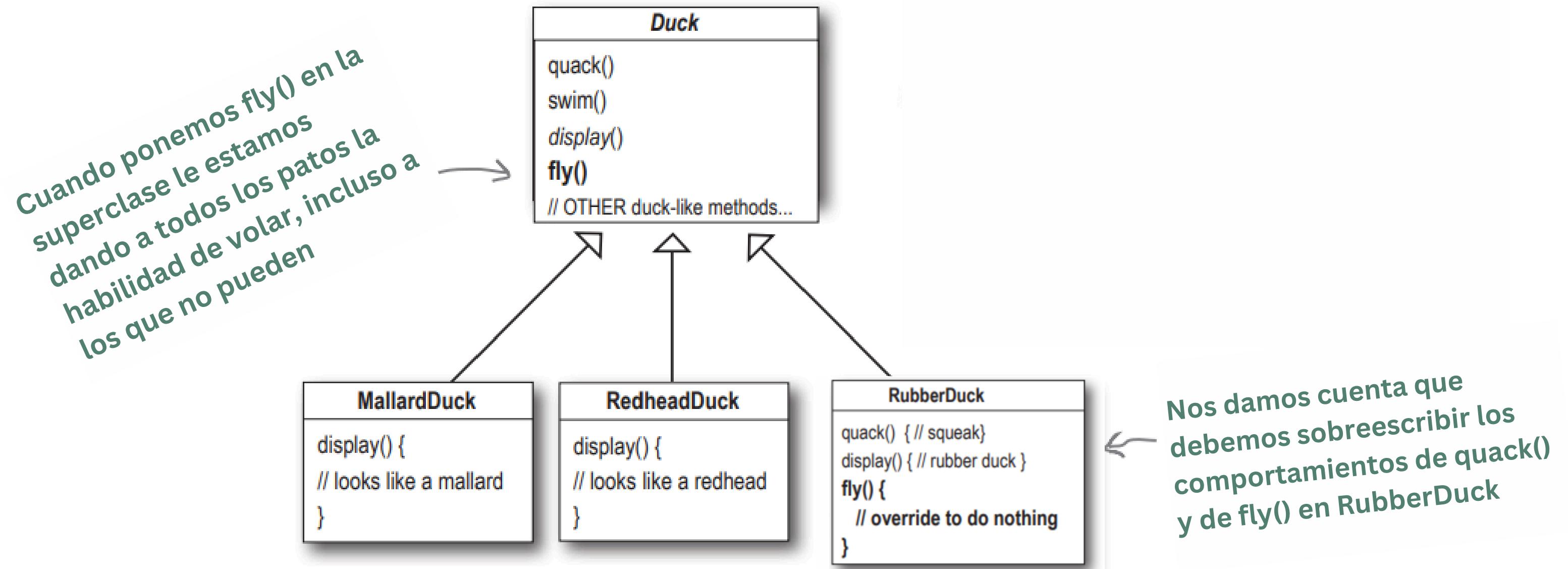
El juego ha ganado popularidad y ahora necesitamos **innovar**, por eso necesitamos algo nuevo...

Necesitamos que los patos vuelen

Pero parece que hoy es nuestro día de suerte, porque esto es tan sencillo como agregar un **método fly()** al **padre** y que **todos las subclases lo hereden**.



Pero cuando probamos el juego empezamos a notar cosas un poco... **raras**. Creíamos que nuestro planteamiento con la herencia era excelente gracias a la **reutilización**, pero esto ha generado un problema en lo que respecta al **mantenimiento**.



Y esto puede ocurrir con muchos mas ejemplos, podemos usar incluso el **DecoyDuck**, (o pato de mentira/señuelo), el cual tampoco hace quack, ni tampoco vuela.

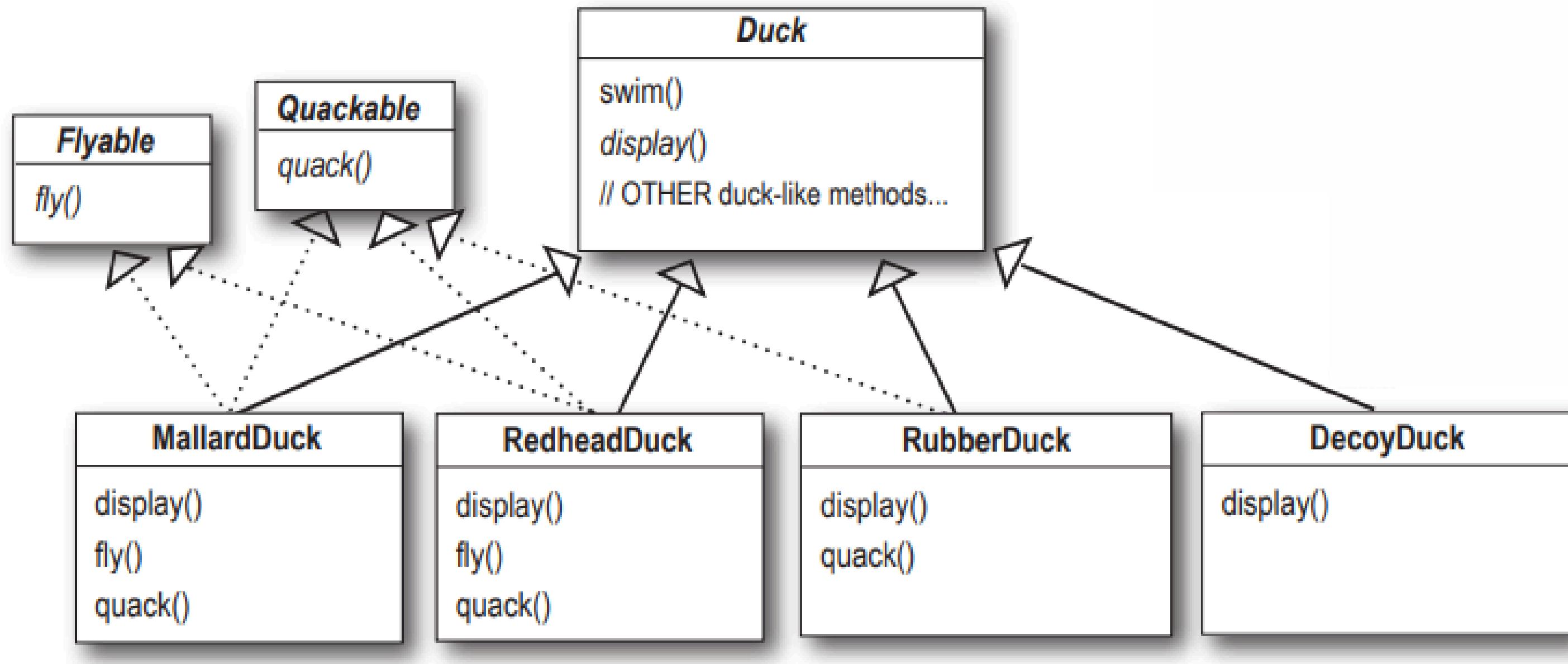
De esta manera nos damos cuenta que **la herencia no siempre es la solución**, y menos cuando los comportamientos son **propensos a cambiar**.

Si seguimos añadiendo nuevos tipos de pato, debemos modificar la implementación de esta nueva subclase. y pero aun, si agregan un nuevo comportamiento a la clase padre (por ejemplo, ahora queremos que los patos caminen), **debemos analizar todas y cada una de las subclases** buscando si este comportamiento cambia en alguno de los casos.

Ya sabemos la raíz del problema, es decir, sabemos que cambia, así que podemos **modificar** un poco nuestro planteamiento inicial



- Pista: una clase puede implementar más de una interfaz al mismo tiempo



Podría sacar **fly()** de la superclase Pato y crear una interfaz **Flyable** con un método **fly()**. De esa manera, solo los patos que se supone que vuelan implementarían esa interfaz y tendrían un método **fly()**... y, ya que estamos, también podría hacer un **Quackable**, ya que no todos los patos pueden hacer quack.

Entonces, se propone usar **interfaces (Flyable y Quackable)**. De esta manera, solo las clases que necesitan esos comportamientos los implementan.

El problema es que esta solución **tampoco es perfecta**: si varios tipos de patos vuelan de la misma manera, tendrías que **duplicar el código** del método **fly()** en cada una de sus clases, lo que es lo opuesto a la reutilización y también crea una "pesadilla de mantenimiento".

En TypeScript sería algo así:

```
class MallardDuck extends Duck implements Flyable, Quackable {  
    display(): void {  
        console.log('Mostrar un pato mallard');  
    }  
    fly(): void {  
        console.log('Volar con mis alas ✈️');  
    }  
    quack(): void {  
        console.log('Hacer quack! Quack!');  
    }  
}
```

```
class RedheadDuck extends Duck implements Flyable, Quackable {  
    display(): void {  
        console.log('Mostrar un pato cabeza roja');  
    }  
    fly(): void {  
        console.log('Volar con mis alas ✈️');  
    }  
    quack(): void {  
        console.log('Hacer quack! Quack!');  
    }  
}
```

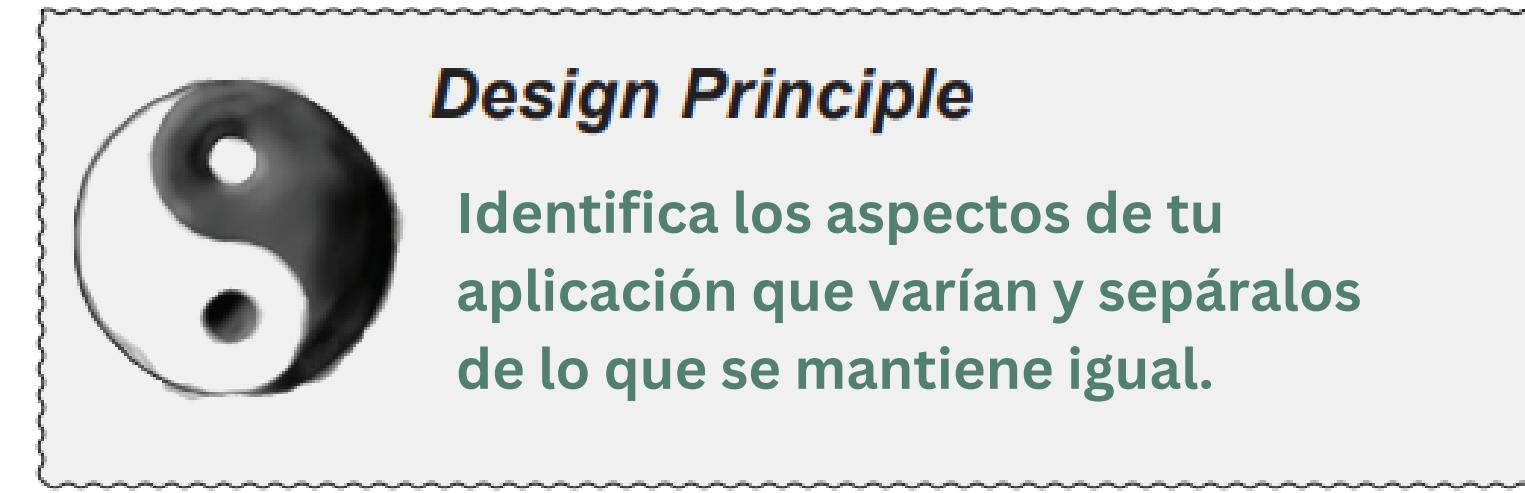


¿Vemos el código duplicado?

¿Están viendo las vueltas que estamos dando gracias a los cambios que ocurren en los requerimientos?

Esta solución es algo mejor a lo que teníamos inicialmente, pero seguimos teniendo problemas, y cuando existan cambios en los requerimientos estamos obligados a cambiar mucho código, y hay algo que debemos saber es que:

Mientras mayor cantidad de código sea propenso a cambiar, mas errores se pueden generar.

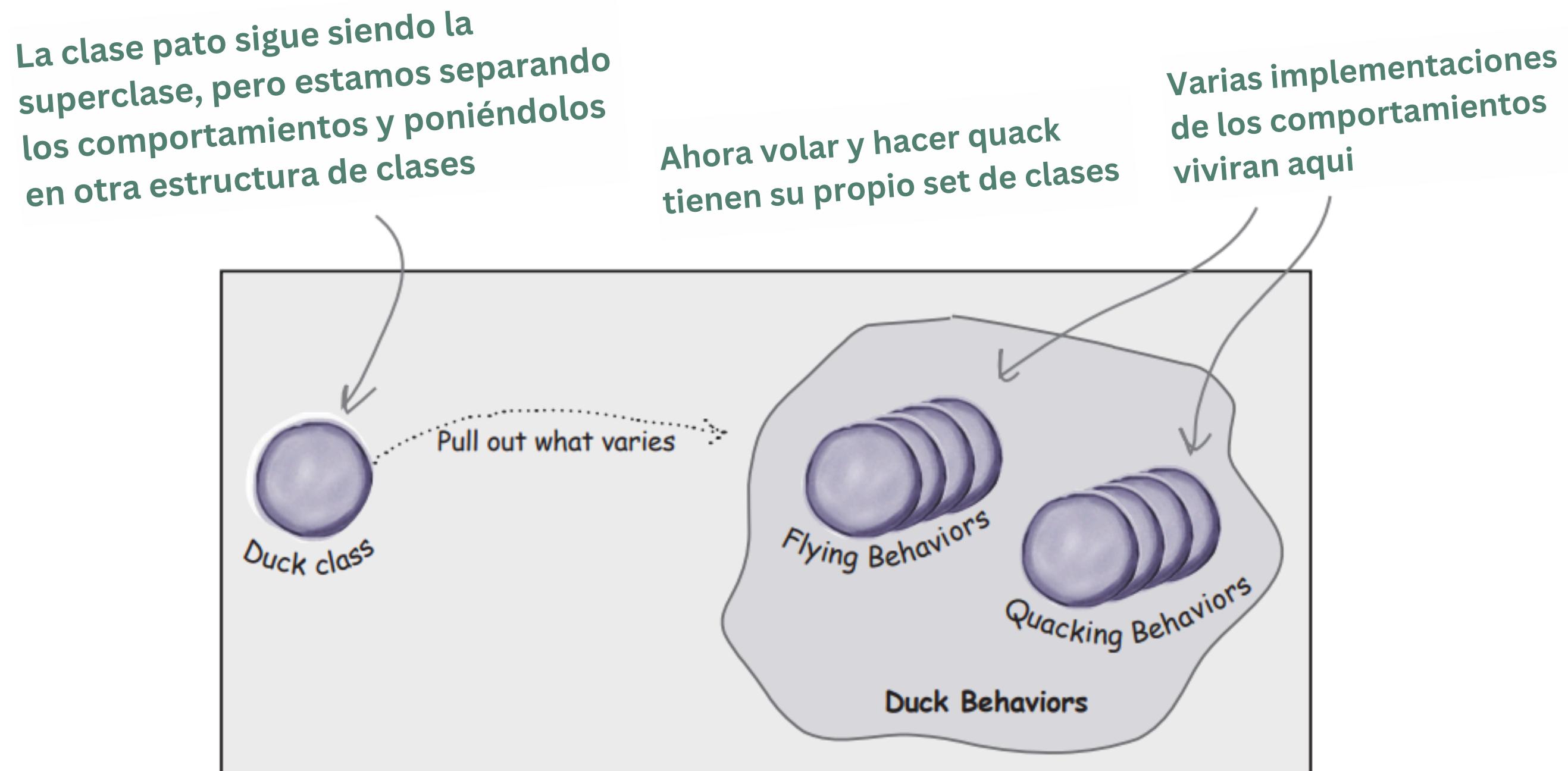


En otras palabras, toma las partes que varían y **encapsúlalas**, para que luego puedas modificarlas o extenderlas sin afectar aquellas que no cambian.

Tan simple como es, este concepto forma la base de casi todos los patrones de diseño. **Todos los patrones permiten que una parte de un sistema cambie de forma independiente a todas las demás partes.**

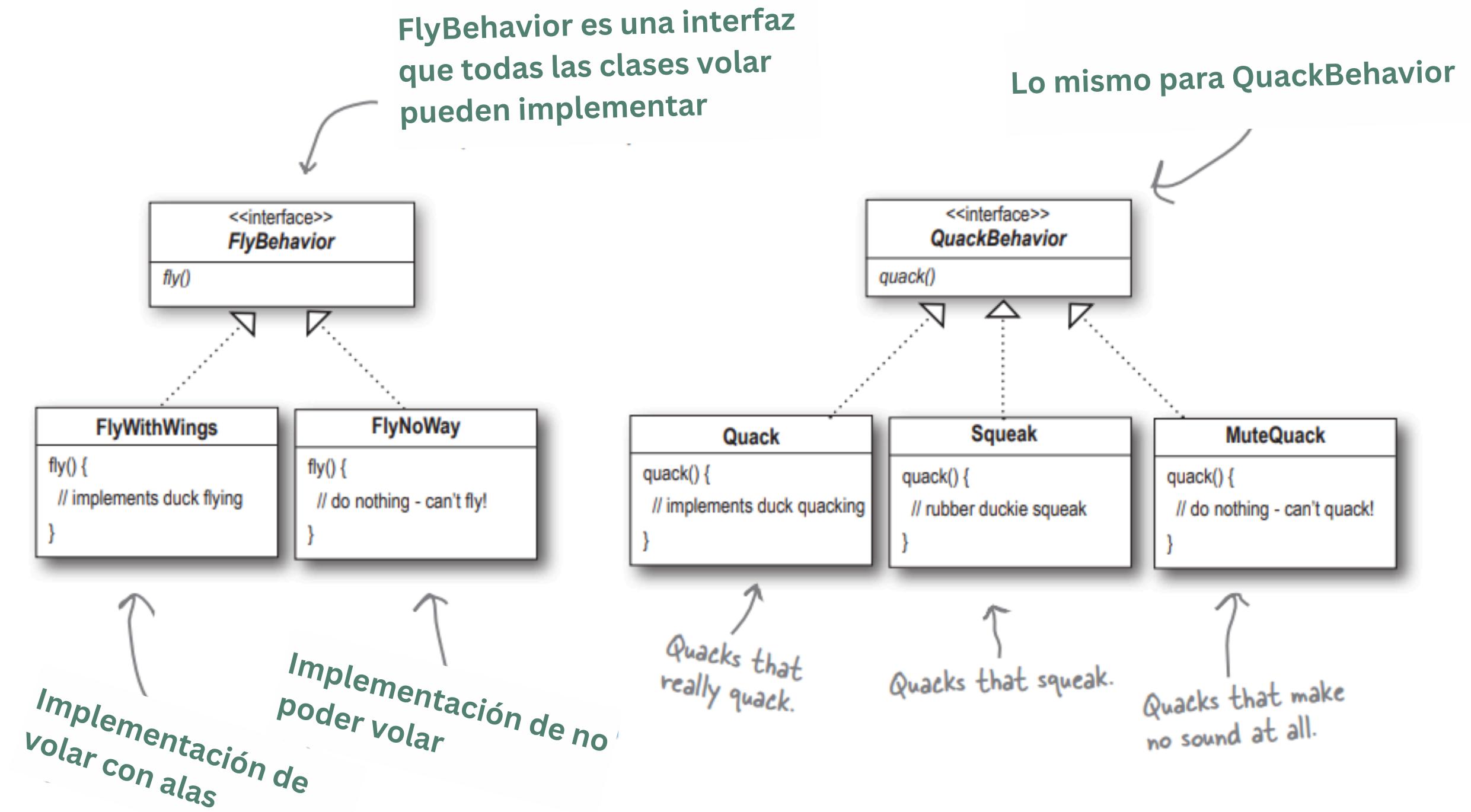
Separar las partes que cambian de las que se mantienen igual

Sabemos que **fly()** y **quack()** son las partes de la clase Pato que varían entre los distintos patos. Para separar estos comportamientos de la clase Pato, extraemos ambos métodos de la clase y crearemos un nuevo conjunto de clases para representar cada comportamiento.



Las **acciones** de los Patos (como **fly** o **quack**) se pondrán en "cajas" de código separadas. De esta manera, las clases **Duck** ya **no necesita saber cómo funciona cada acción por dentro**. Simplemente elige la "caja" que necesita usar y listo. Siempre y cuando el comportamiento cumpla con el contrato que se define (la Interfaz), la clase **Duck** podrá hacer uso de él sin problema.

Así otros tipos de objetos pueden reutilizar nuestros comportamientos de **fly** y **quack** porque estos ya no están ocultos dentro de nuestras clases **Duck**. Y podemos añadir nuevos comportamientos sin modificar ninguna de nuestros comportamiento existentes ni tocar ninguna de las clases **Duck** que los utilizan.



Integrar los comportamientos de Duck

Añadimos dos variables tipo **FlyBehavior** y **QuackBehavior** (recordemos que estas dos son interfaces). Cada **Duck** concreto asignará a estas variables un comportamiento específico en tiempo de ejecución, como **FlyWithWings** para volar y **Squeak** para graznar.

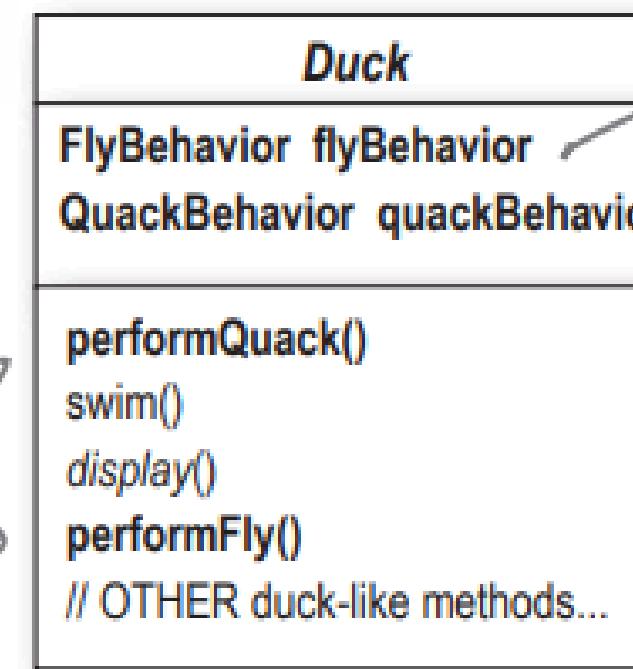
Eliminamos los métodos **fly()** y **quack()** de la clase **Duck** (y de cualquier subclase) porque hemos trasladado este comportamiento

Reemplazamos **fly()** y **quack()** en la clase **Duck** con dos métodos similares, **performFly()** y **performQuack()**.

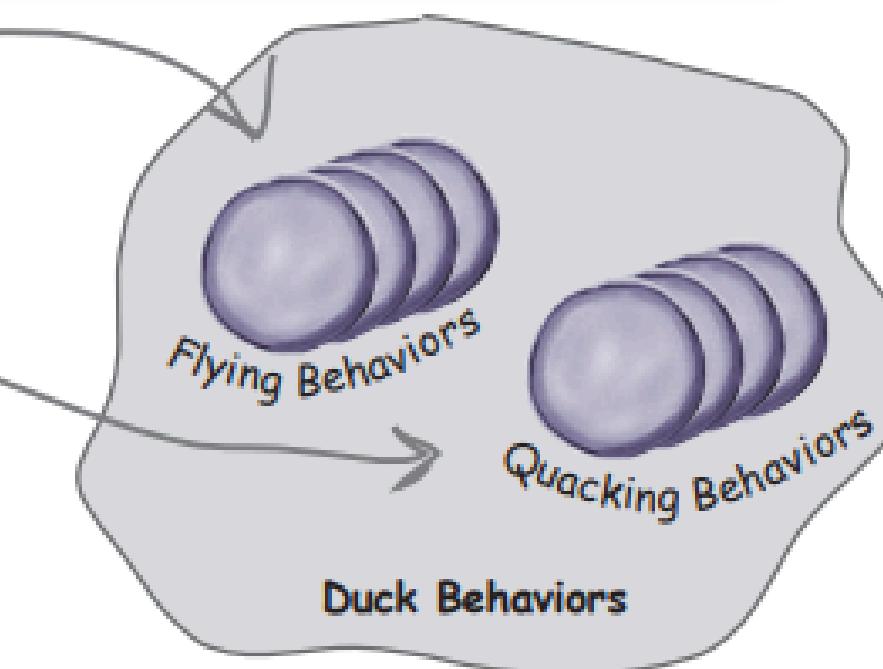
Añadimos setters **setFlyBehavior()** y **setQuackBehavior()**

El tipo de las variables de comportamientos son interfaces

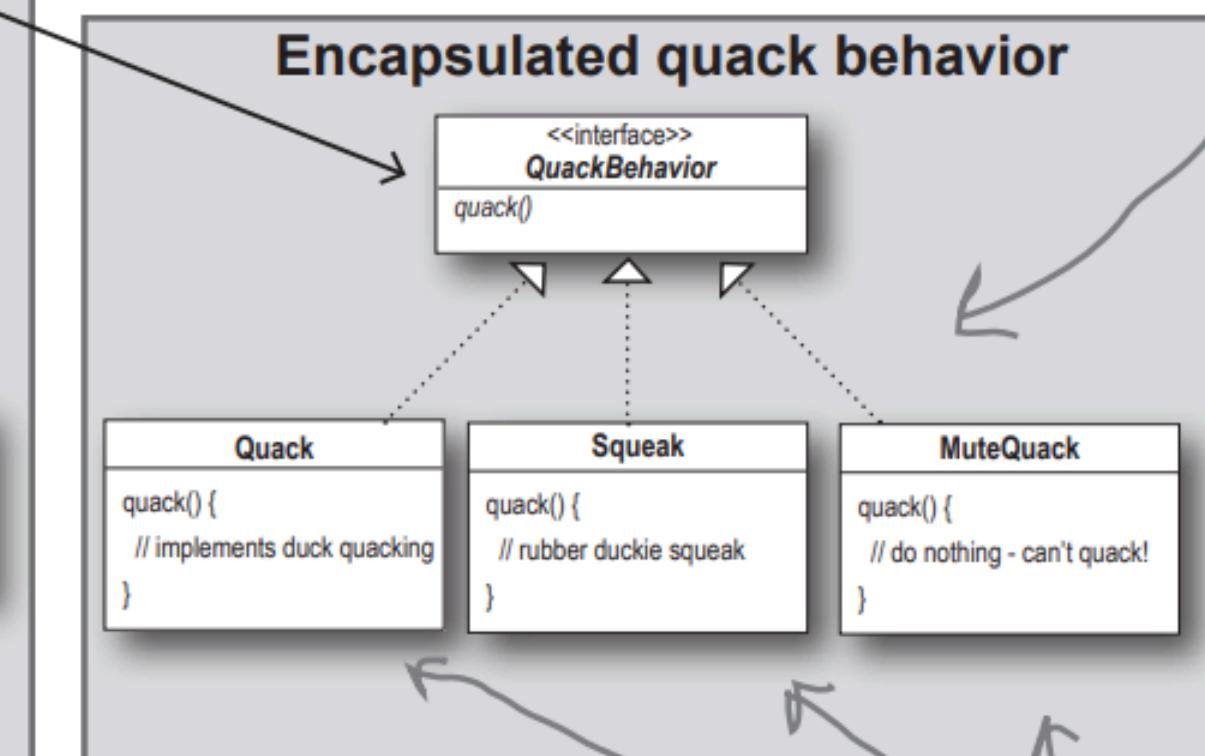
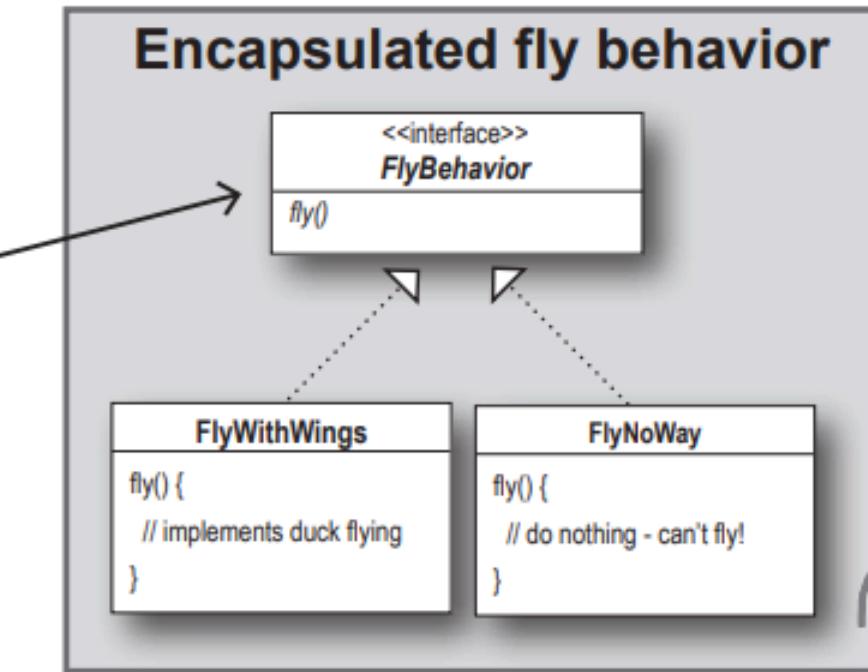
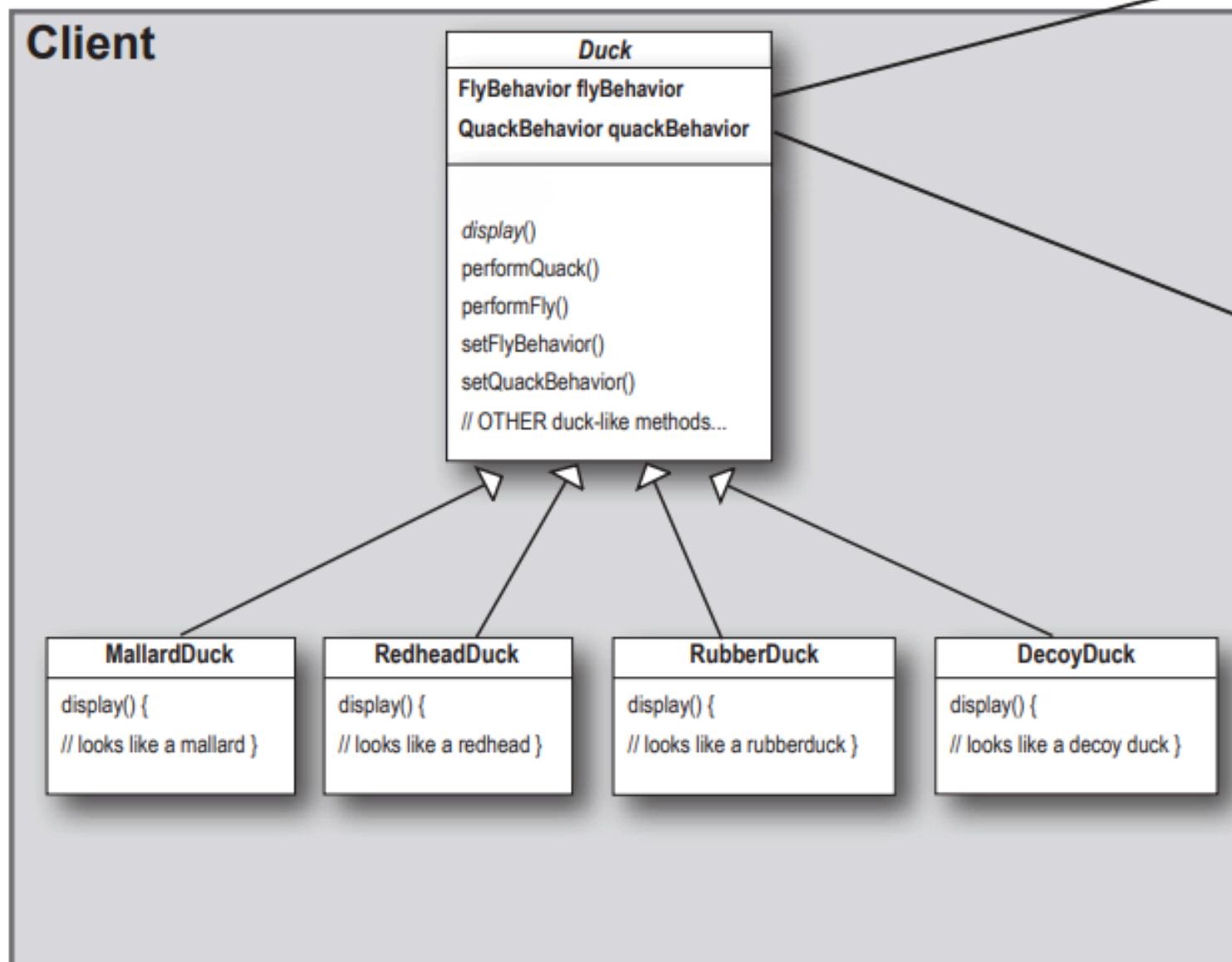
Métodos que reemplazan **fly()** y **quack()**



Las variables de comportamientos referencian comportamientos concretos en runtime



El cliente usa una familia de algoritmos encapsulada tanto para volar como para hacer quack



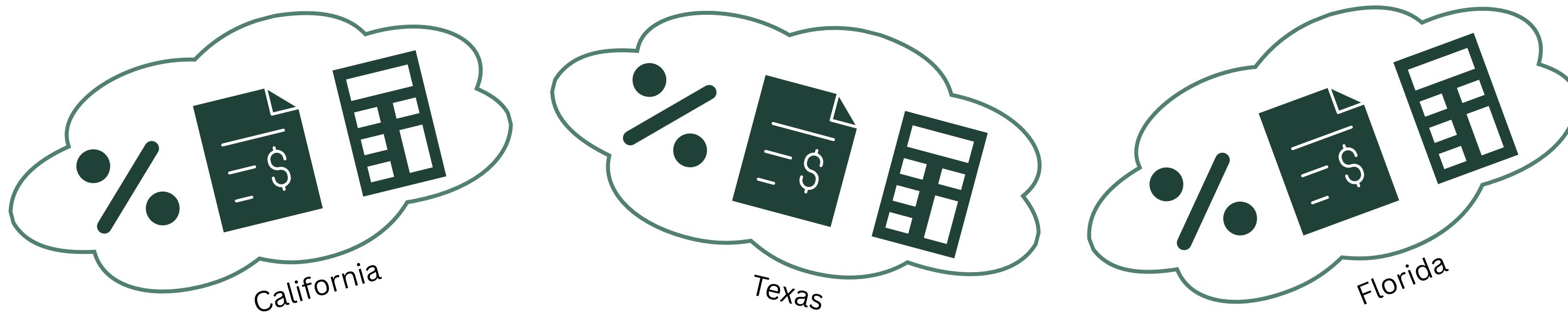
Piensa en cada set de comportamientos como una familia de algoritmos

Estos comportamientos “algoritmos” son intercambiables

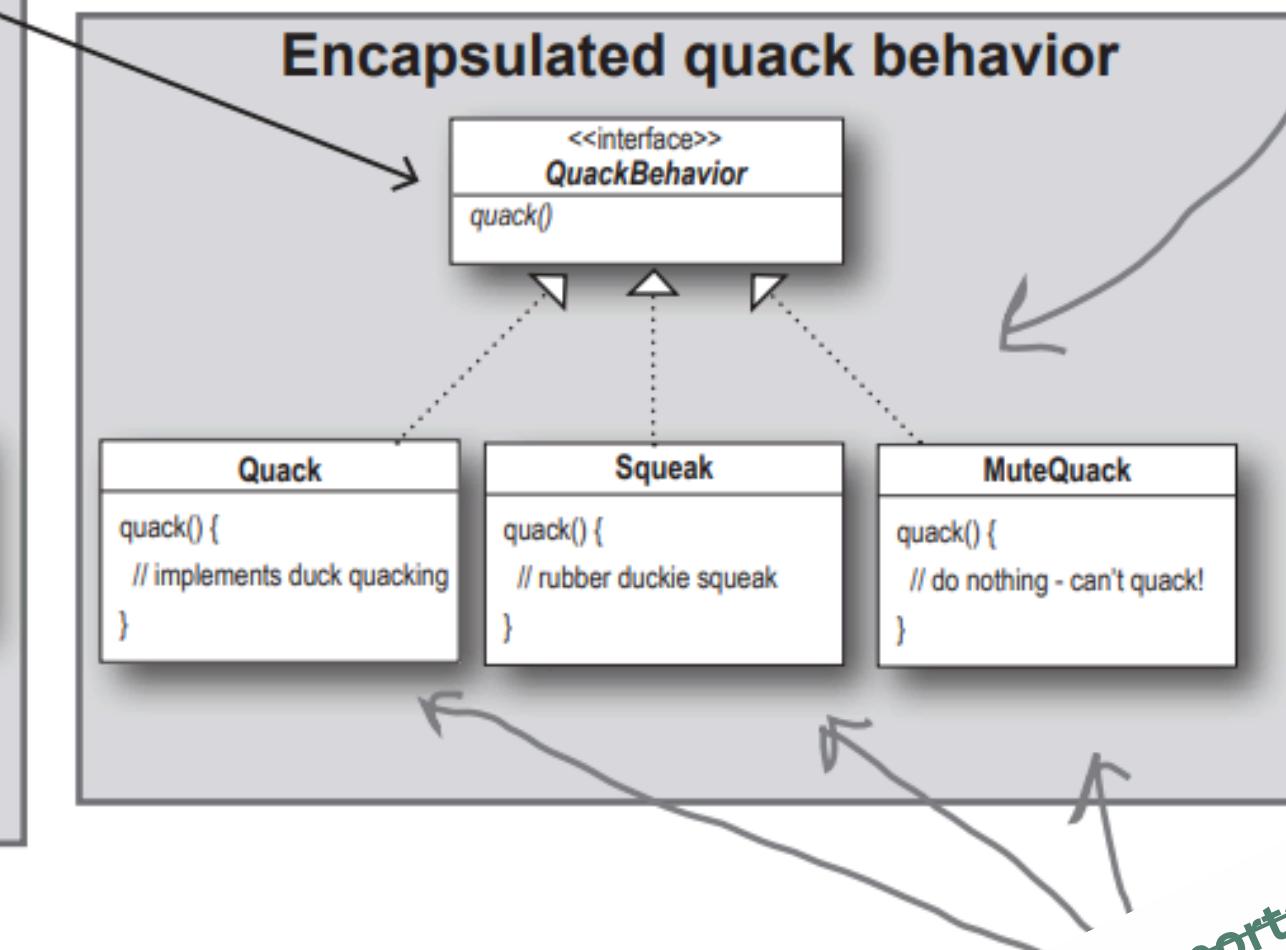
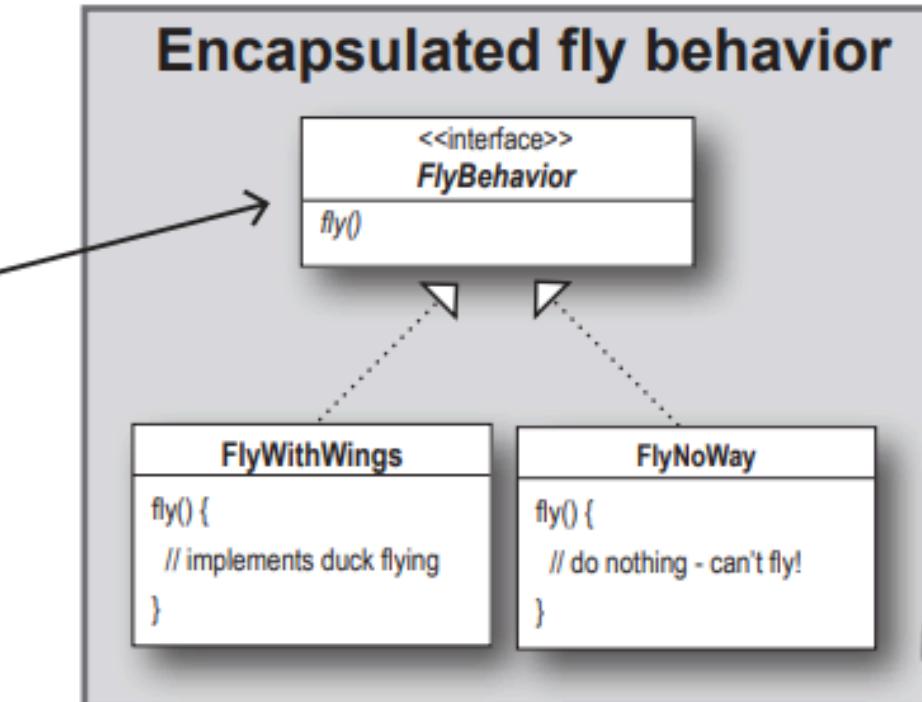
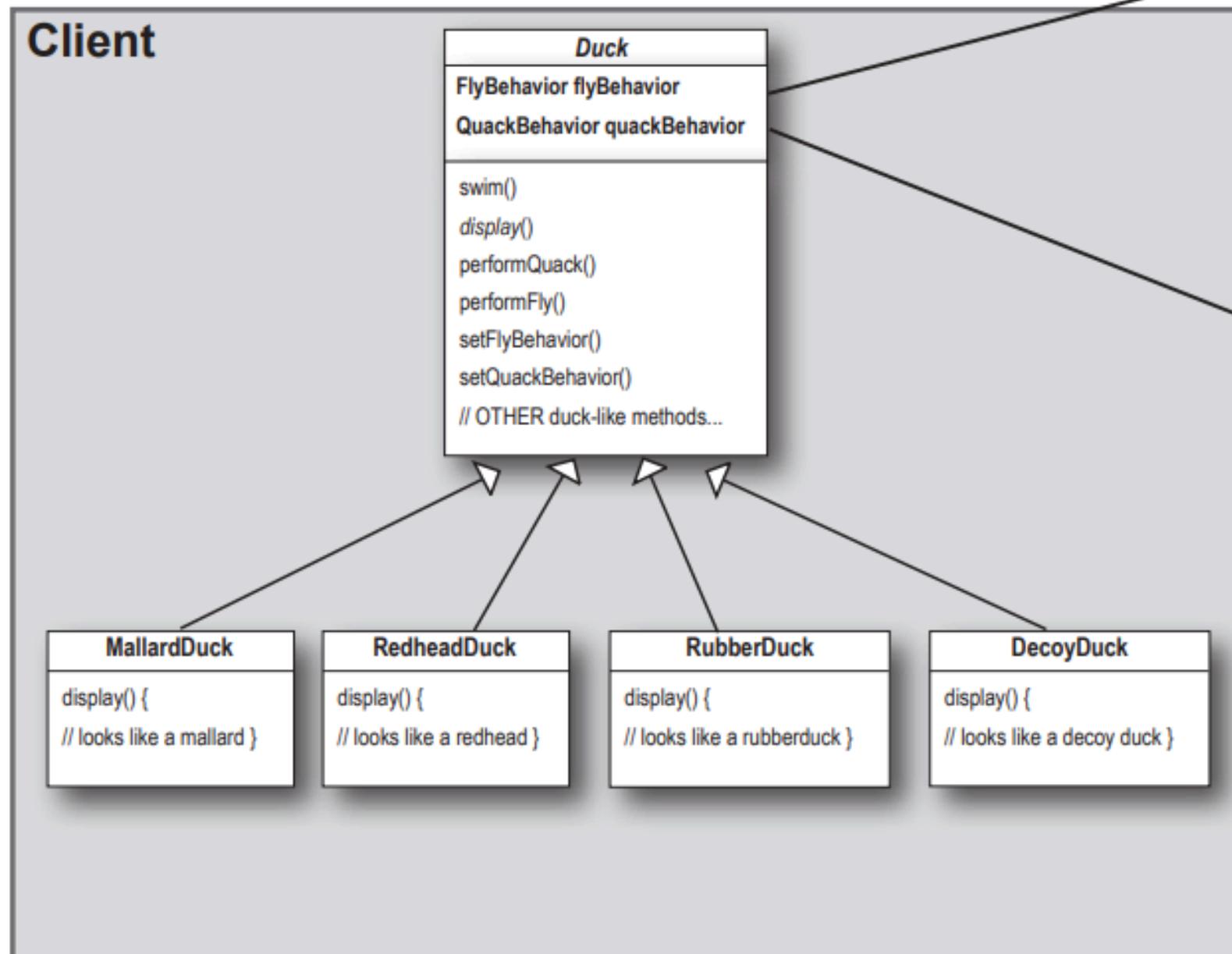
Hemos empezado a describir las cosas de manera **un poco diferente**. En lugar de pensar en los comportamientos del **Duck** como un conjunto de comportamientos, empezaremos a verlos como una **familia de algoritmos**.

En el diseño del juego, **los algoritmos representan cosas que haría un pato** (distintas formas de **fly** o **quack**).

Y podemos usar estas mismas técnicas para muchos otros casos. por **ejemplo**, *un conjunto de clases que implementen las diferentes maneras de calcular el impuesto sobre las ventas a nivel estatal en los distintos estados.*



El cliente usa una familia de algoritmos encapsulada tanto para volar como para hacer quack



Implementen el diagrama en código (TypeScript)
Y encapsulen el comportamiento de nadar (swim)

Piensa en cada set de comportamientos como una familia de algoritmos

Estos comportamientos “algoritmos” son intercambiables

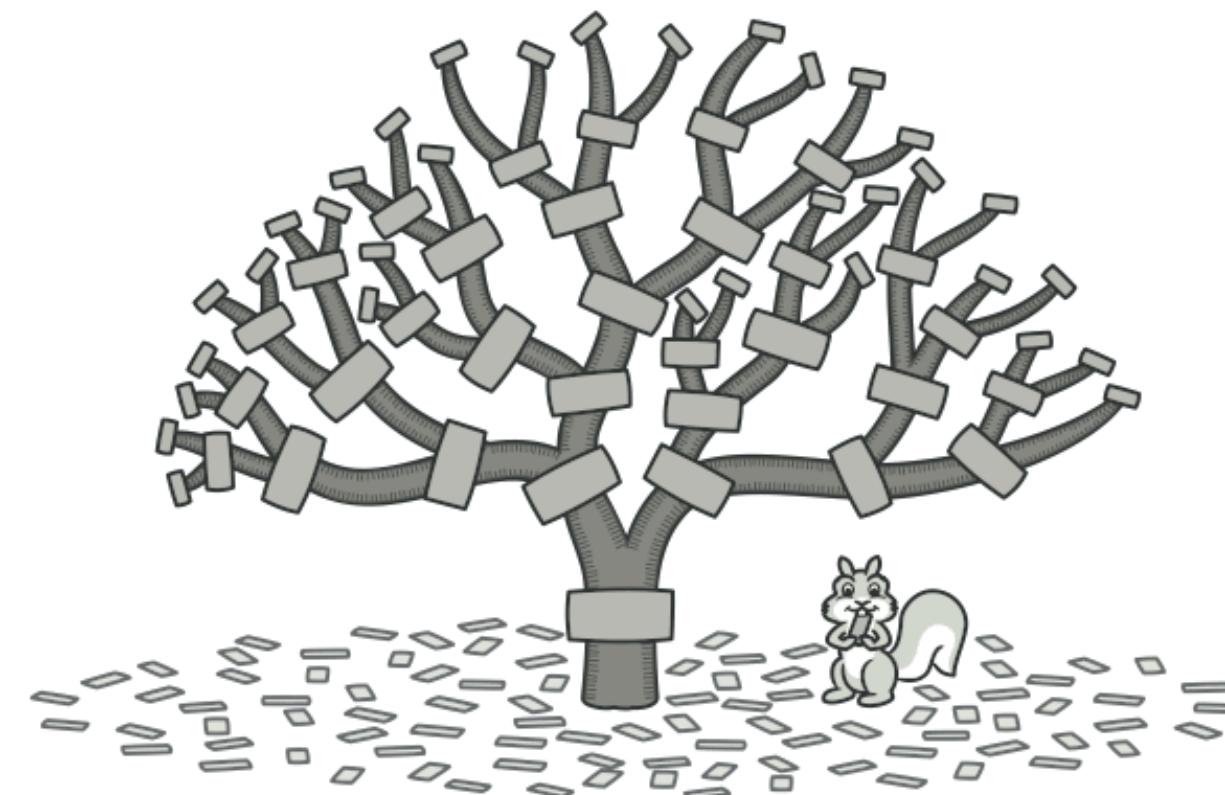
Composite

Patrón estructural que te permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales.

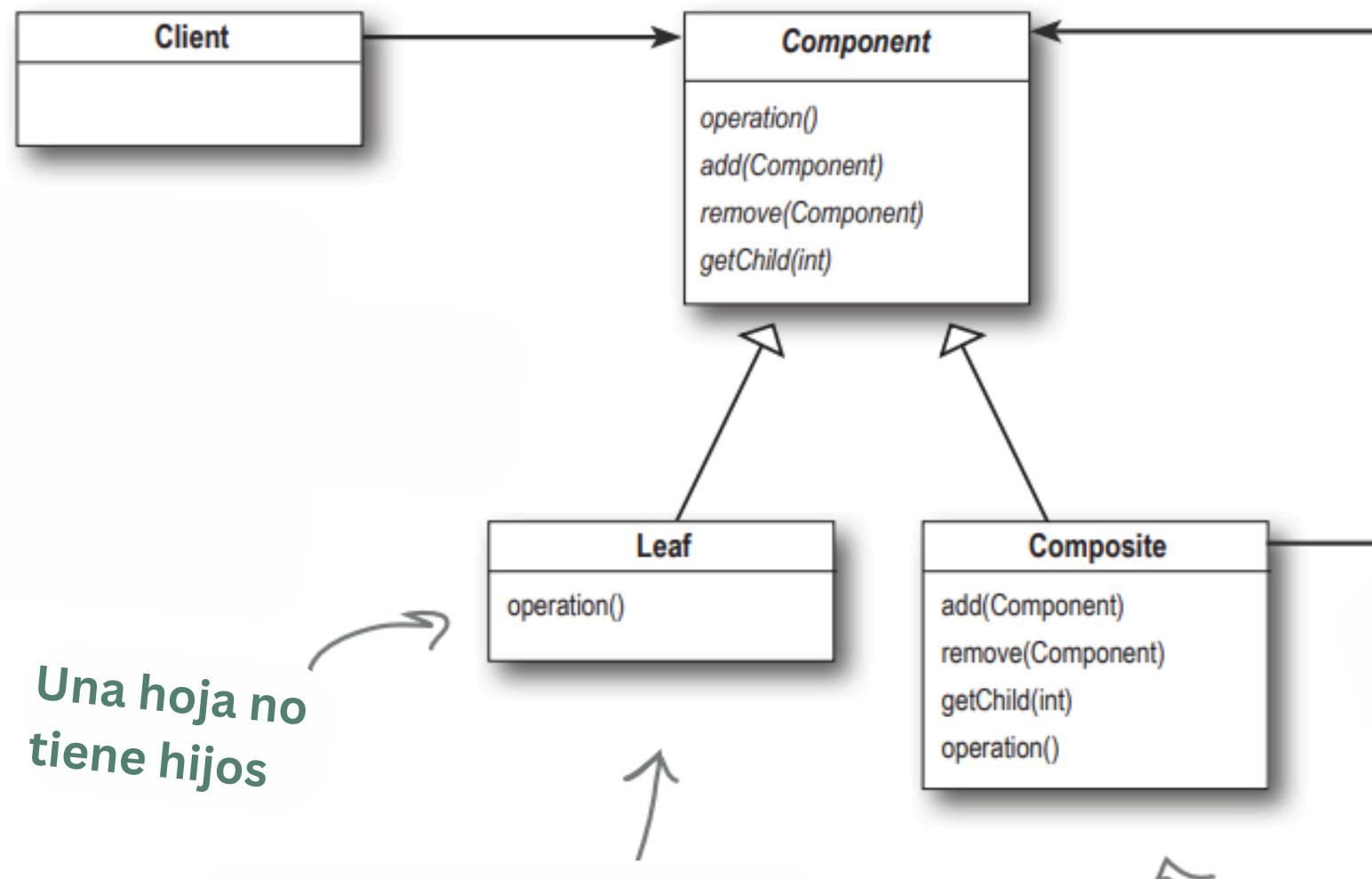
El patrón Composite nos permite construir estructuras de objetos en forma de árbol que contienen tanto composiciones de objetos como objetos individuales a modo de nodos.

En otras palabras, en la mayoría de los casos podemos ignorar las diferencias entre las composiciones de objetos y los objetos individuales.

Nuevamente buscamos: **Abstraccion**



El cliente usa la interfaz Component para manipular objetos en la composición



Una hoja no tiene hijos

Una hoja define el comportamiento para los elementos de la composición

El Componente define una Interfaz para todos los objetos en la composición: tanto composite como hojas

El componente puede implementar comportamientos por defecto para add(), remove(), getChild() y su/s operacion/es

- Un Composite contiene componentes.
- Los componentes vienen en dos tipos: composite y hojas.
- Un Composite alberga un conjunto de hijos; esos hijos pueden ser otros composite u otras hojas.

Cuando organizas los datos de esta manera, terminas con una estructura de árbol con un composite en la raíz y ramas de composite que crecen hasta llegar a las hojas.

¿Suena recursivo? Lo es.

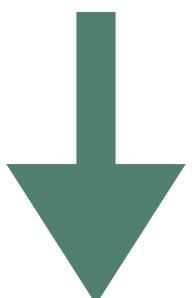
El rol del Composite es definir el comportamiento de los componentes que si tienen hijos

Partiendo del ejercicio

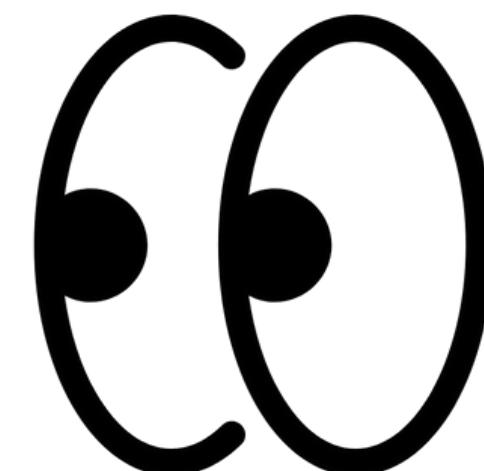
Considere todas las posibles expresiones aritméticas posibles, por ejemplo:

$$(5 + 10 + (40 - 10) + (50 * 10 * 3) - (500/10))$$

Un ejemplo
más fácil

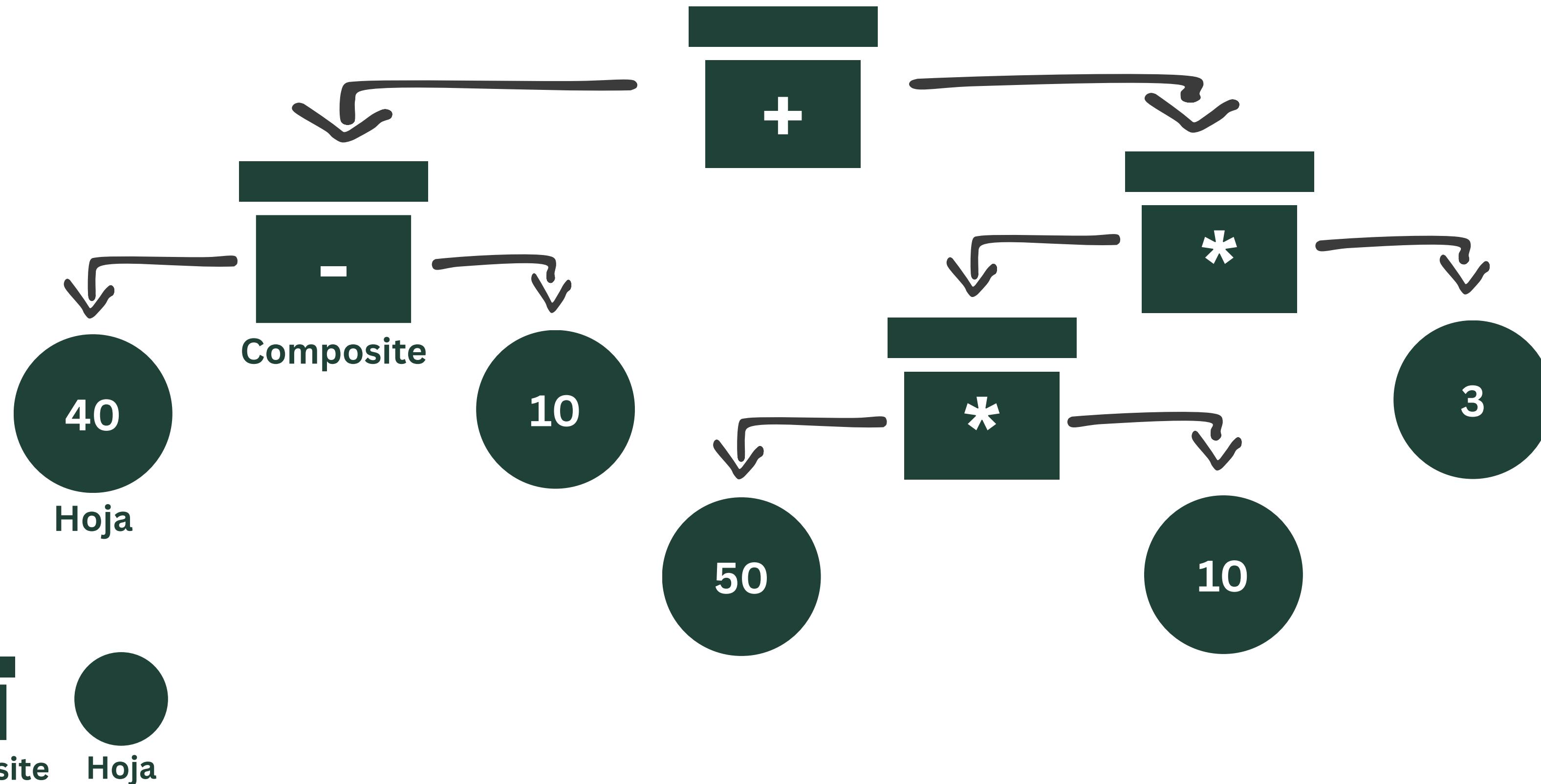


$$((40 - 10) + ((50 * 10) * 3))$$

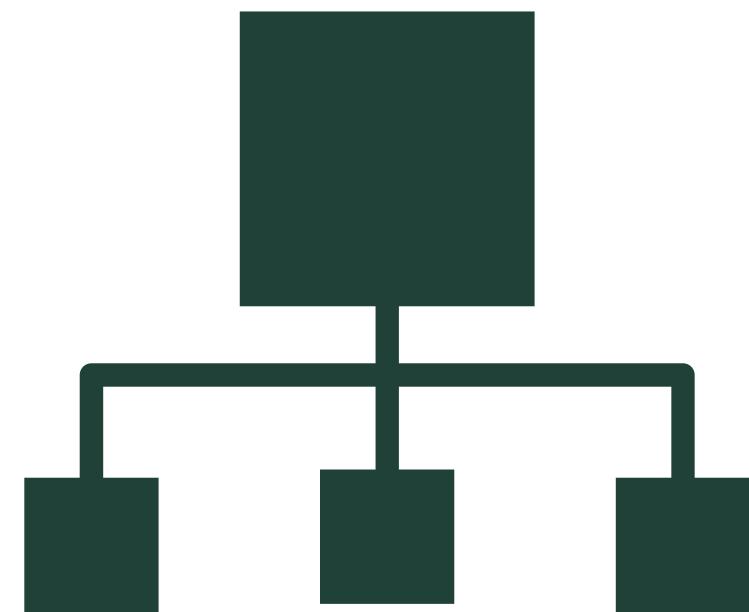


¿Podemos escribir esto de alguna otra manera?

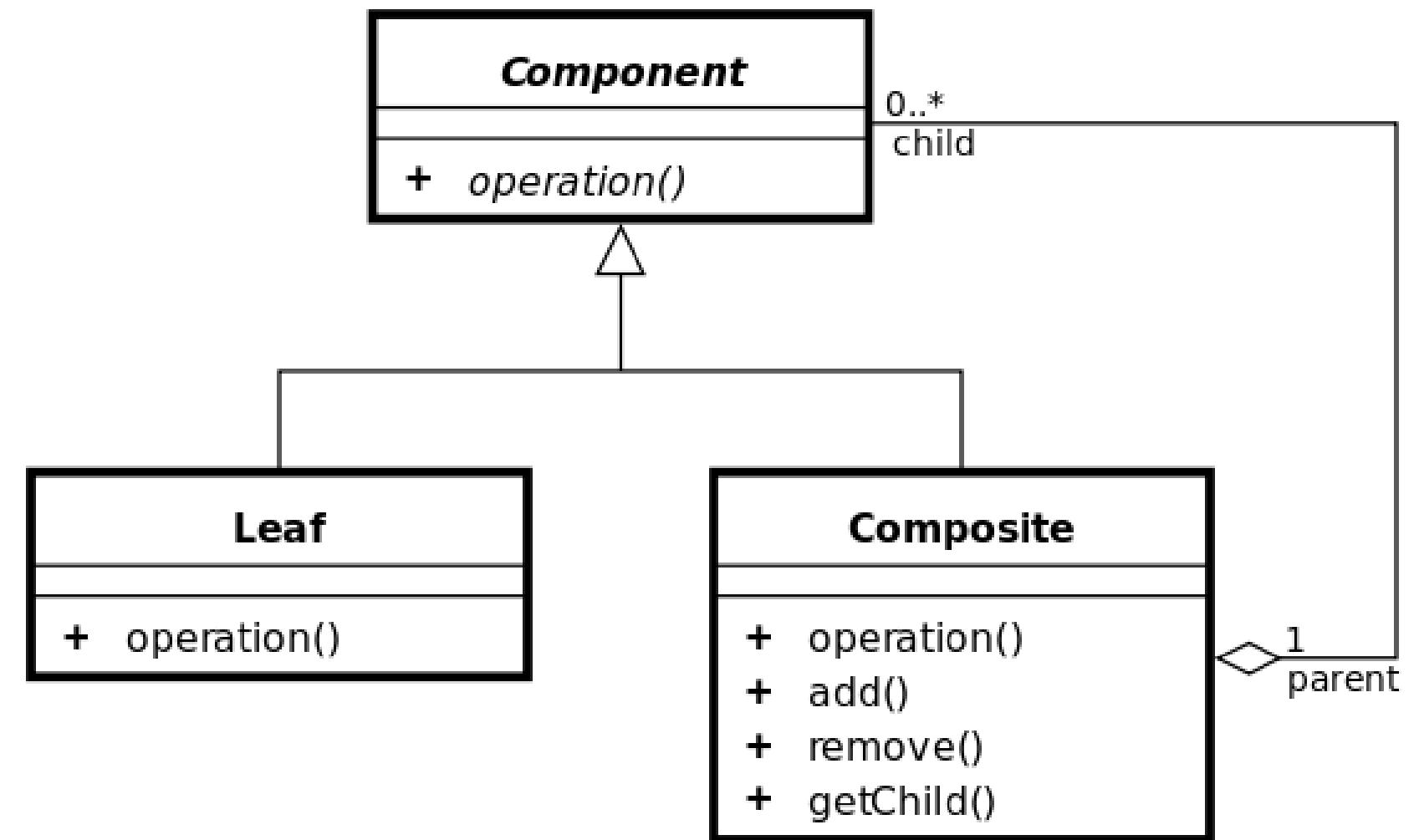
$$((40 - 10) + ((50 * 10) * 3))$$



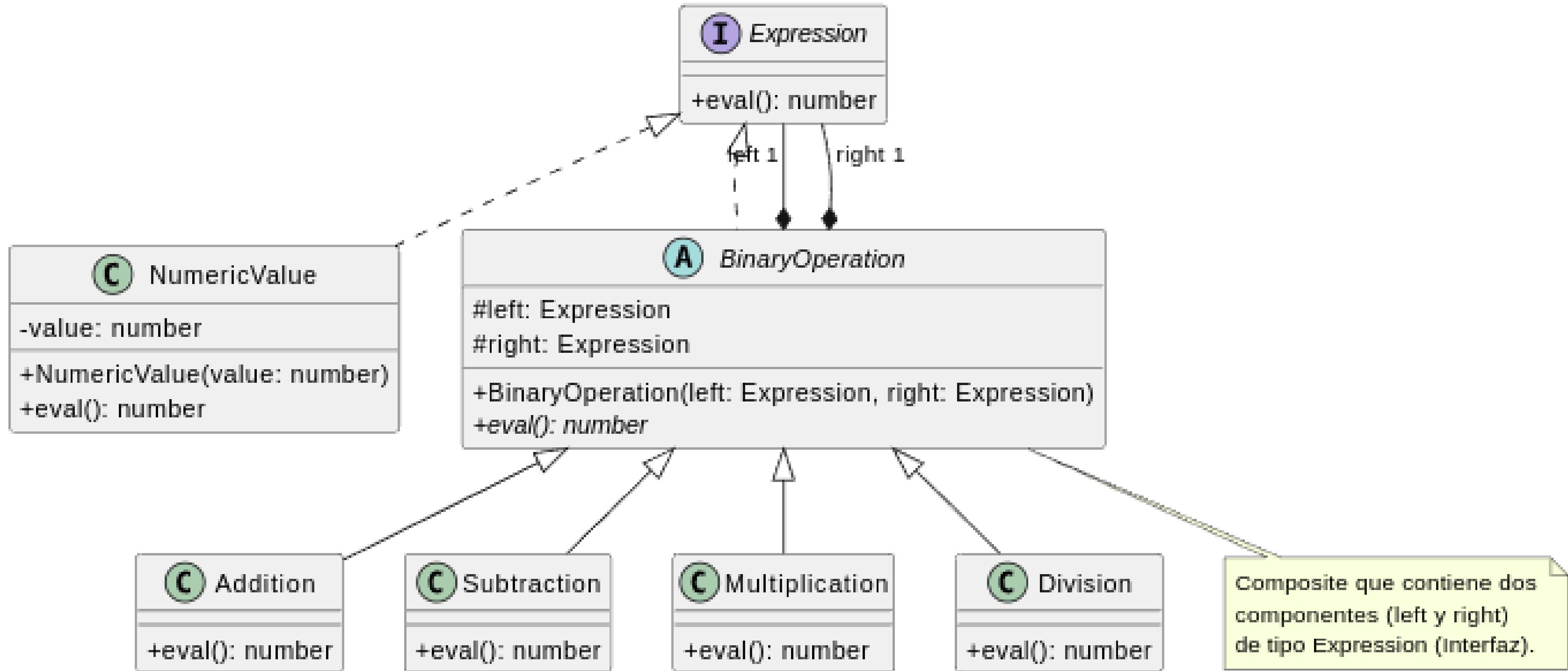
Vamos a diagramarlo



Podemos usar su diseño referencial

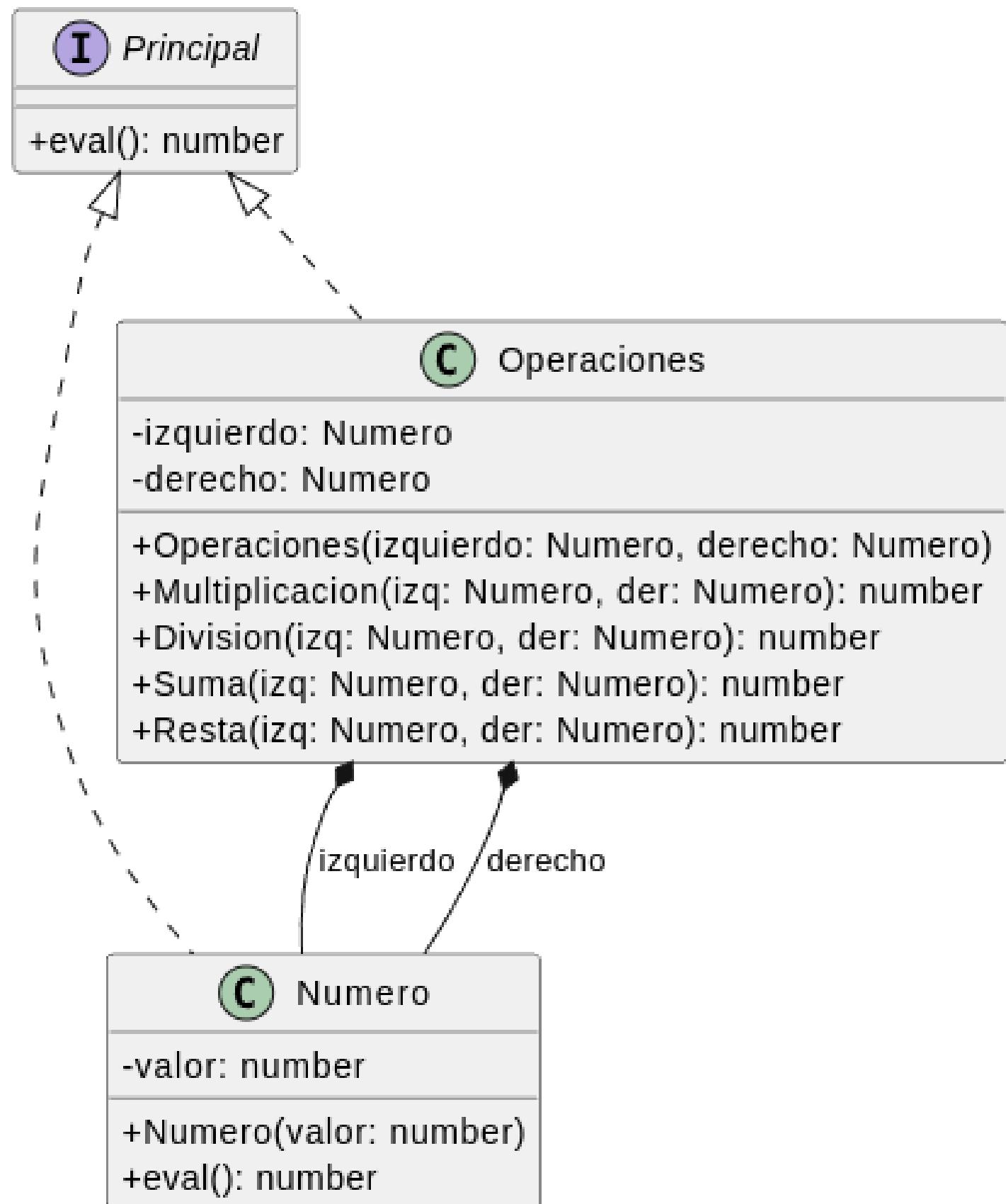
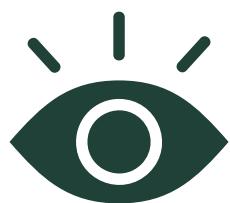


Patrón Composite — Expresiones Aritméticas

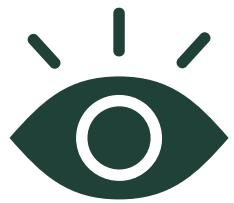


Vemos que podemos aplicar la **misma operación (eval())** sin necesidad de distinguir si estoy trabajando con un **solo elemento** o con una **composición de elementos**.

Cosas que vi



Cosas que vi

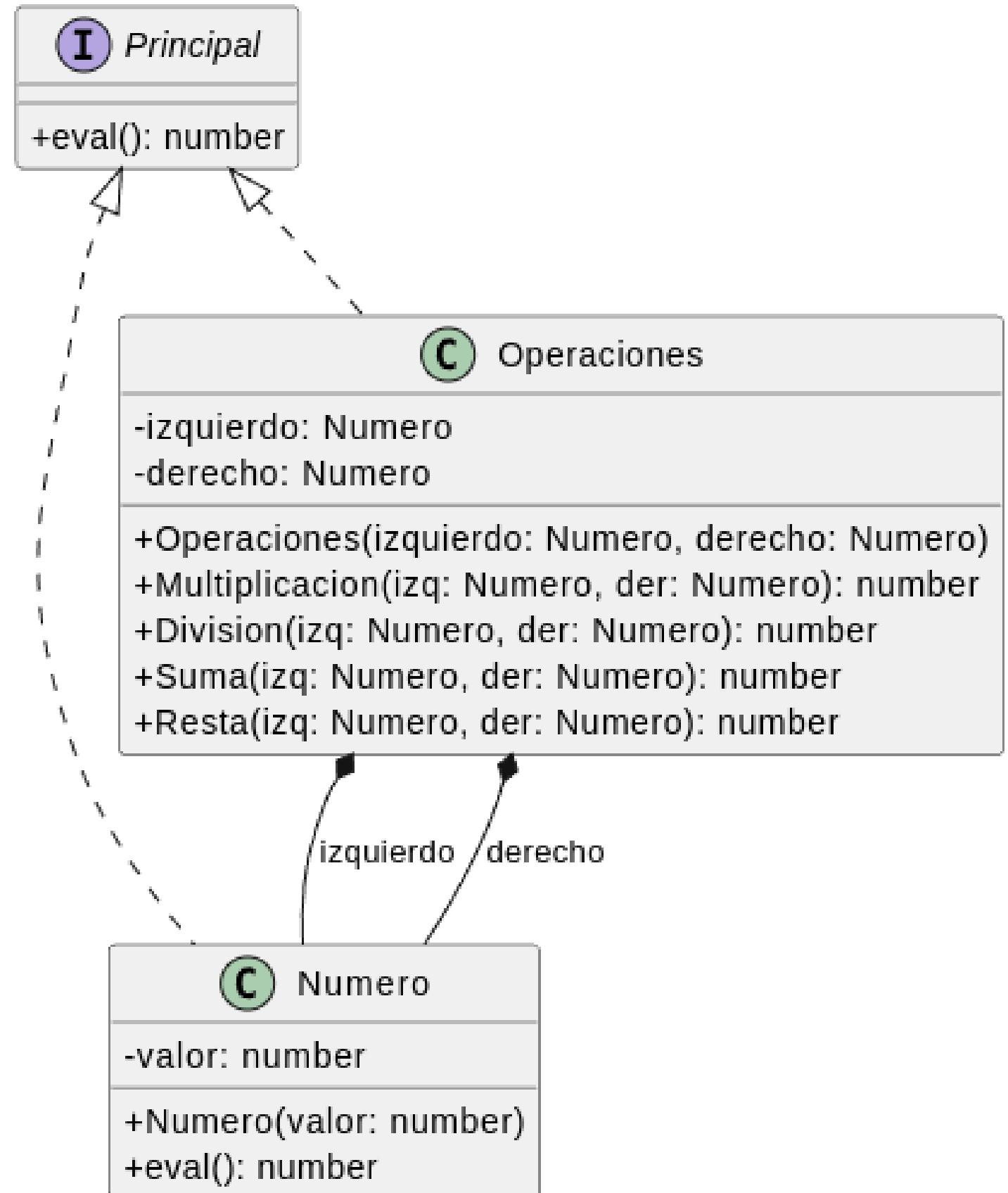


No se cumple el contrato de la interfaz:

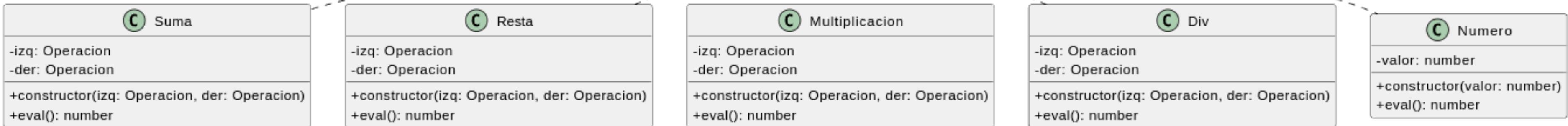
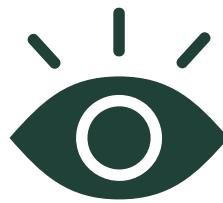
El Composite depende de la Hoja (referencias directas)

No se puede crear una estructura de árbol

La clase Operaciones no es un verdadero composite



Cosas que vi



La implementación es correcta y realmente cumple los requisitos del patron Composite

Interfaz Común (Operacion)

Composición Recursiva

Operación Uniforme

Aun así, tenemos ciertos detalles a tomar en cuenta:

Duplicación de código: Los campos izquierda y derecha, y el constructor se repiten 4 veces.

Violación del principio DRY (Don't Repeat Yourself).

Más difícil de mantener: Si quieres cambiar algo común a todas las operaciones binarias, tienes que tocarlo en 4 lugares.

Pero no todo tiene que ser perfecto desde el inicio, el **refactoring** es parte natural del proceso y estos patrones suelenemerger cuando identificamos repetición en la solución a distintos problemas

¿Que se pide con patrón?

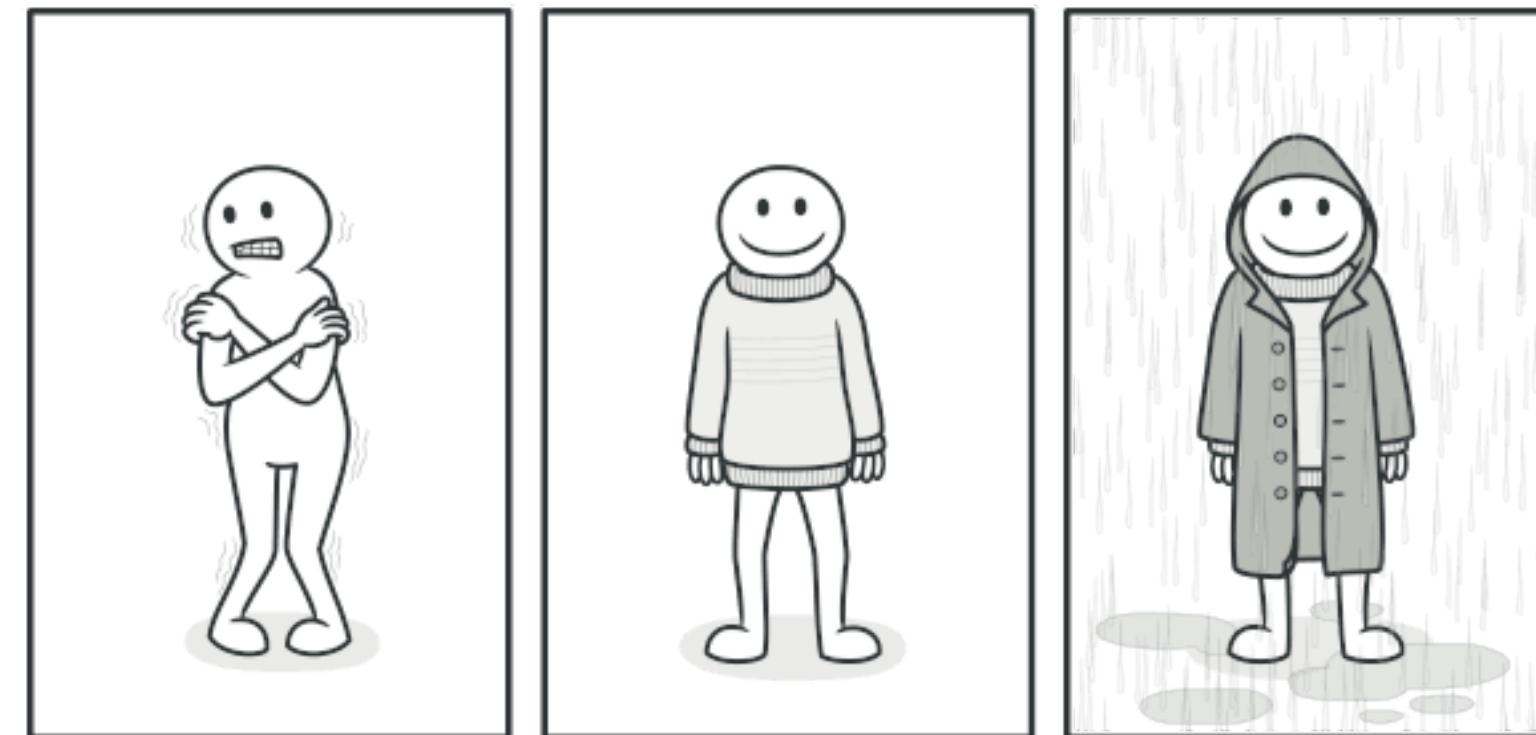
“Lo siento profe, tengo que repasar“

Decorator

Patrón estructural que permite agregar comportamientos adicionales a un objeto de forma dinámica, sin modificar su clase original ni afectar a otros objetos de la misma clase.

Permite extender la funcionalidad de un objeto "envolviéndolo" con otros objetos que agregan nuevas características, como capas que se van apilando una sobre otra.

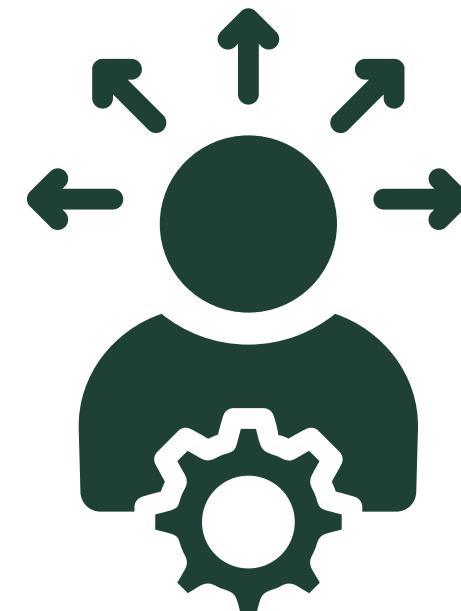
Café simple → Le agregas leche (**decorador**) → Le agregas azúcar (**otro decorador**)



Reexaminaremos el típico **uso excesivo de la herencia** y aprenderemos cómo "decorar" clases en tiempo de ejecución usando una forma de **composición de objetos**.

¿Por qué? Una vez que conozcamos las técnicas de decoración, serás capaz de dar a tus objetos (o a los de alguien más) nuevas responsabilidades **sin hacer ningún cambio** en el código de las clases subyacentes.

Ya no estamos hablando de cambiar comportamientos como en el caso de Strategy, ahora estamos hablando de **agregar o extender comportamientos**.

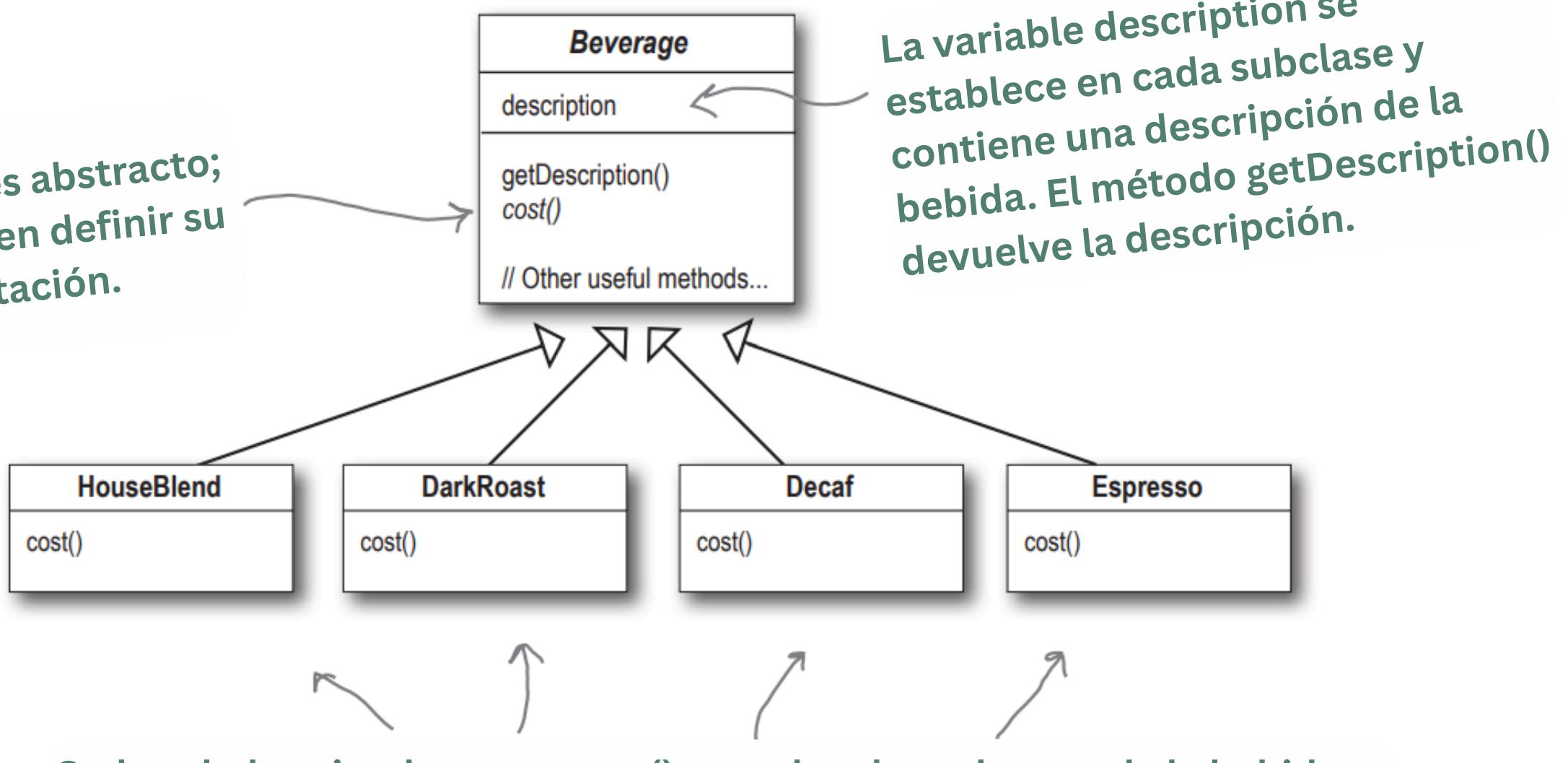




Páramo café ha crecido mucho en Venezuela, y como ya saben ha crecido de manera muy rápida, pero el equipo técnico esta sufriendo al actualizar su sistema de pedidos para que coincidan con su oferta de bebidas.

Cuando recién comenzaron su negocio, diseñaron sus clases de esta manera...

Beverage (Bebida) es una clase abstracta, de la cual se crean subclases para todas las bebidas ofrecidas en la cafetería.

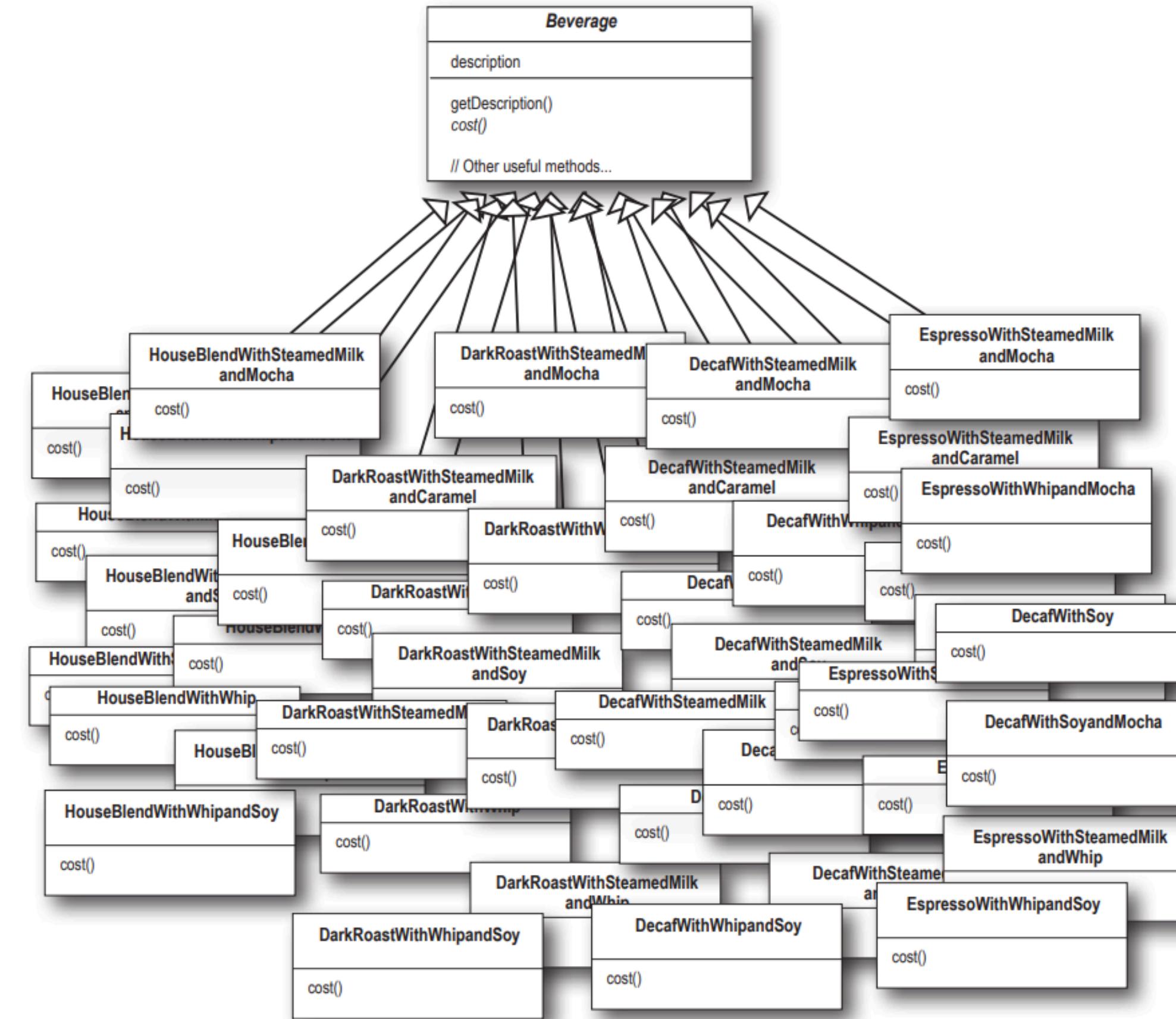


Aparte de tu café, también puedes pedir varios complementos, como leche de vaca, leche de soya y chocolate, y rematarlo todo con crema batida.

Como Páramo cobra un extra por cada complemento, necesitan urgentemente integrarlos a su sistema de pedidos.



Aquí está el primer intento



Cada método `cost()` calcula el costo del café junto con los otros condimentos en el pedido.

Es bastante obvio que Páramo ha creado una pesadilla de mantenimiento para sí mismos.

¿Qué pasa cuando el precio de la leche sube?

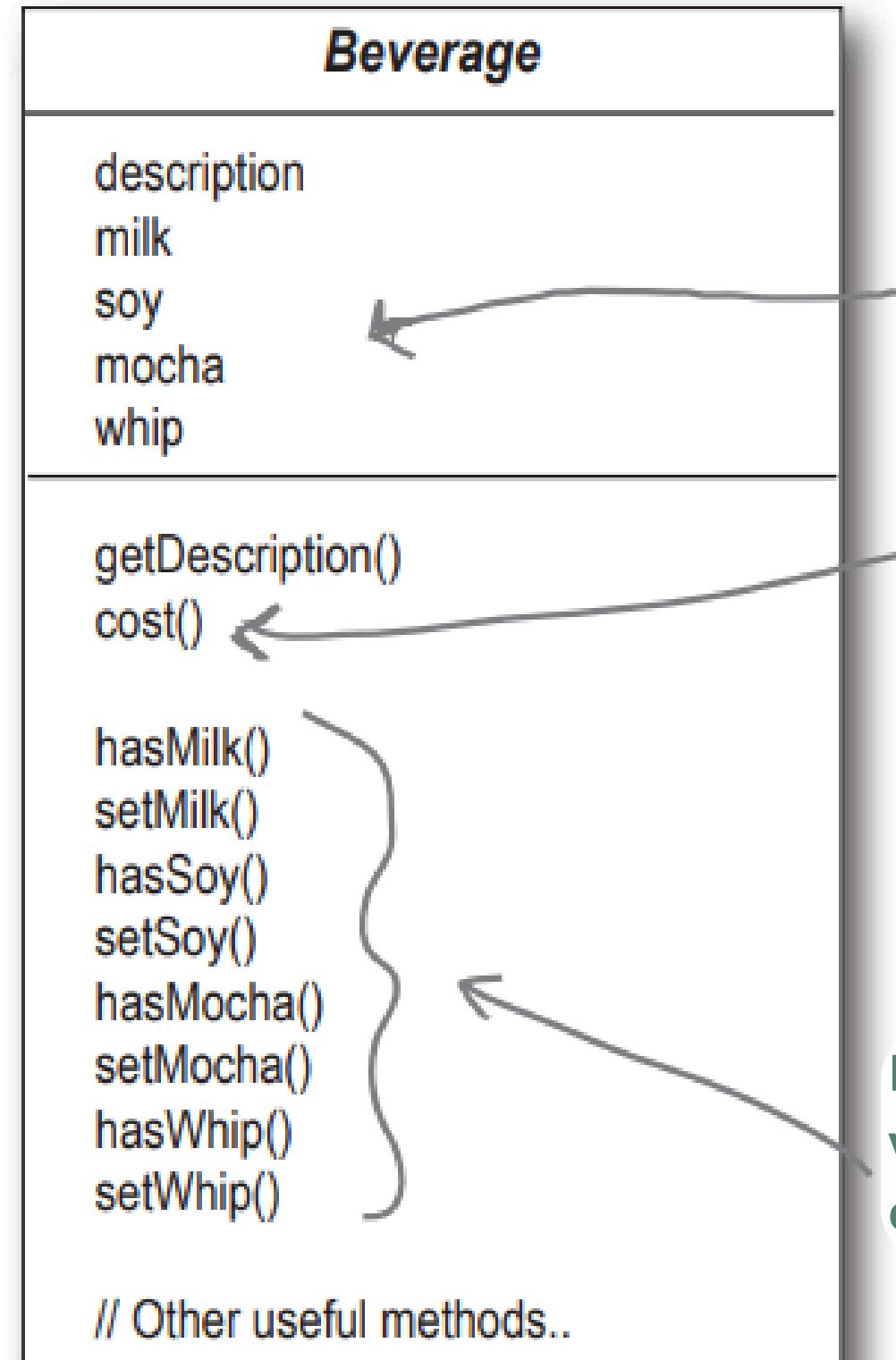
¿Qué hacen cuando añaden una nueva cobertura de caramelo?

Entonces...



¿Por qué necesitamos todas estas clases?

¿No podemos simplemente usar variables y herencia en la superclase para llevar un registro de los condimentos?



Nuevos valores booleanos para cada condimento.

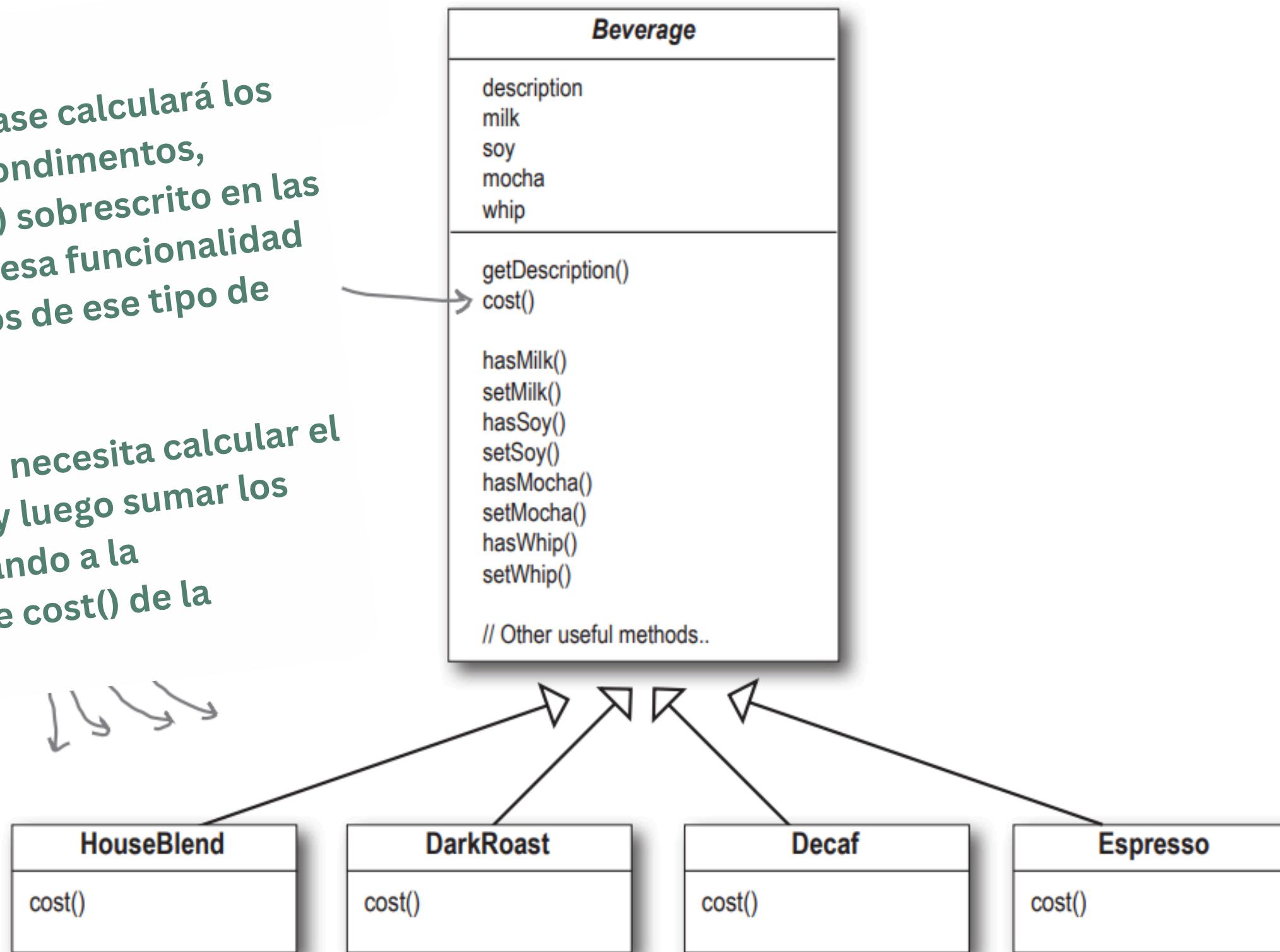
Ahora implementaremos `cost()` en `Beverage` (en lugar de mantenerlo abstracto), para que pueda calcular los costos asociados con los condimentos para una instancia de bebida en particular. Las subclases seguirán sobrescribiendo `cost()`, pero también invocarán la versión de la superclase para que puedan calcular el costo total de la bebida básica más los costos de los condimentos añadidos.

Estos obtienen y establecen los valores booleanos para los condimentos. (getters y setters).

Ahora agregamos las subclases:

El `cost()` de la superclase calculará los costos de todos los condimentos, mientras que el `cost()` sobrescrito en las subclases extenderá esa funcionalidad para incluir los costos de ese tipo de bebida específica.

Cada método `cost()` necesita calcular el costo de la bebida y luego sumar los condimentos llamando a la implementación de `cost()` de la superclase.



Ejercicio

Define los métodos cost()

```
abstract class Beverage {
    cost(): number {
        // ...
    }
}

class DarkRoast extends Beverage {
    constructor() {
        super();
        this.description = "Tostado Oscuro de Calidad";
    }

    cost(): number {
        // ...
    }
}
```

Solución

Seguro ya lo sabías...

Esta implementación puede traer problemas a futuro:

- Los cambios de precio en los condimentos nos obligarán a modificar código existente.
- Los nuevos condimentos nos obligarán a agregar nuevos métodos y a modificar el método **cost()** en la superclase.
- Podemos tener nuevas bebidas. Para algunas de estas bebidas (¿té helado?), los condimentos pueden no ser apropiados, sin embargo, la subclase **Tea** seguirá heredando métodos como **hasWhip()**.
- ¿Qué pasa si un cliente quiere un doble moca?

```
class DarkRoast extends Beverage {
    public DarkRoast() {
        this.description = "Tostado Oscuro de Calidad";
    }

    // Sobrescribe el método cost
    public cost(): number {
        // El costo base se suma al costo de los condimentos calculado por
        // la superclase
        return 1.99 + super.cost();
    }
}
```

```
class Beverage {
    // Variables de costos
    public milkCost: number = 0.10;
    public soyCost: number = 0.15;
    // mas...

    // Variables booleanas...
    // getters y setters...

    // Método que calcula el costo total de los condimentos
    public cost(): number {
        let condimentCost: number = 0.0;

        if (this.hasMilk()) {
            condimentCost += this.milkCost;
        }
        if (this.hasSoy()) {
            condimentCost += this.soyCost;
        }
        if (this.hasMocha()) {
            condimentCost += this.mochaCost;
        }
        if (this.hasWhip()) {
            condimentCost += this.whipCost;
        }
        return condimentCost;
    }
}
```



Gurú: Ha pasado algún tiempo desde nuestro último encuentro. ¿Has estado en profunda meditación sobre la herencia?

Estudiante: Sí, Gurú. Si bien la herencia es poderosa, he aprendido que no siempre conduce a los diseños más flexibles o fáciles de mantener.

Gurú: Ah, sí, has progresado. Entonces, dime, mi estudiante, ¿cómo lograrás la reutilización si no es a través de la herencia?

Estudiante: Gurú, he aprendido que hay maneras de "heredar" comportamiento en tiempo de ejecución a través de la **composición** y la **delegación**.

Gurú: Por favor, continúa...

Estudiante: Cuando heredo comportamiento mediante la creación de subclases, ese comportamiento se establece estáticamente en tiempo de compilación. Además, todas las subclases deben heredar el mismo comportamiento. Sin embargo, si puedo extender el comportamiento de un objeto mediante la composición, puedo hacerlo dinámicamente en tiempo de ejecución.

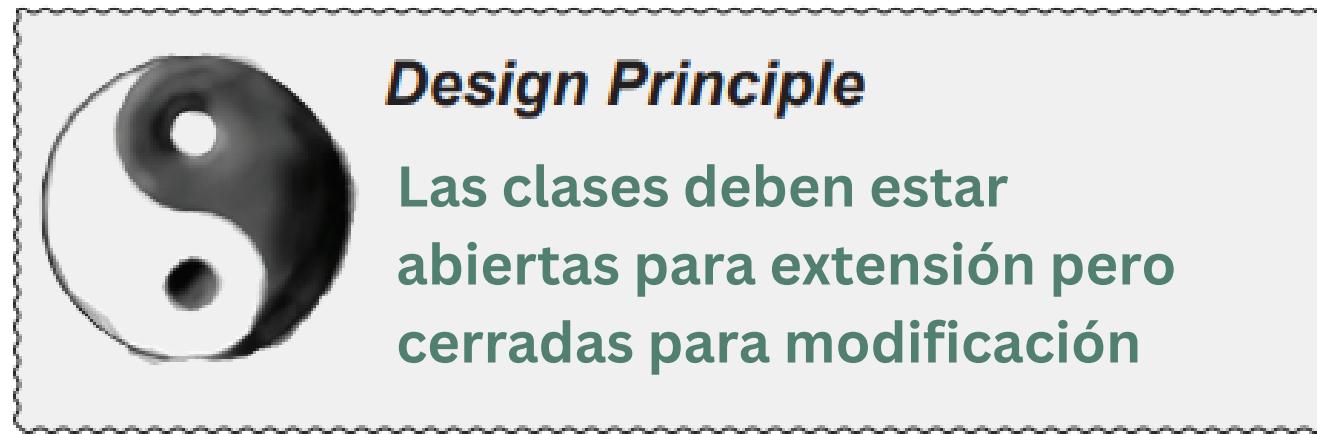
Gurú: Muy bien; estás empezando a ver el poder de la composición.

Estudiante: Sí, es posible para mí añadir múltiples nuevas responsabilidades a los objetos a través de esta técnica, incluidas responsabilidades en las que ni siquiera pensó el diseñador de la superclase. ¡Y no tengo que tocar su código!

Gurú: ¿Qué has aprendido sobre el efecto de la composición en el mantenimiento de tu código?

Estudiante: Bueno, a eso me refería. Al componer objetos dinámicamente, puedo añadir nueva funcionalidad escribiendo código nuevo en lugar de modificar código existente. Debido a que no estoy cambiando el código ya existente, las posibilidades de introducir errores o causar efectos secundarios no deseados en código preexistente se reducen considerablemente.

Gurú: Muy bien. Suficiente por hoy. Quiero que vayas y medites más sobre este tema... Recuerda, **el código debe estar cerrado (a la modificación) como la flor de loto por la tarde, pero abierto (a la extensión) como la flor de loto por la mañana.**

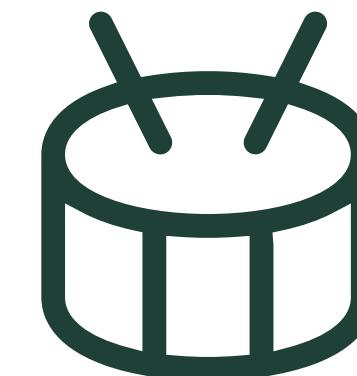


Nuestro objetivo es permitir que las clases sean **fácilmente extensibles** para incorporar nuevo comportamiento sin necesidad de modificar el código existente.

¿Qué obtenemos si logramos esto?

Diseños que son resistentes al cambio y lo suficientemente flexibles como para incorporar nueva funcionalidad para satisfacer requisitos cambiantes.

Si bien puede parecer una contradicción, existen técnicas que permiten que el código **sea extendido**



Y así es que presentamos a nuestro patron...

Decorator en acción

Así que, aquí está lo que haremos: comenzaremos con una bebida y la "**decoraremos**" con los condimentos en tiempo de ejecución. **Por ejemplo**, si el cliente quiere un **Tostado Oscuro con Moca y Crema Batida**, entonces haremos lo siguiente:

1. Comenzar con un objeto **DarkRoast** (Tostado Oscuro).
2. Decorarlo con un objeto **Mocha** (Moca).
3. Decorarlo con un objeto **Whip** (Crema Batida).
4. Llamar al método `cost()` y depender de la delegación para sumar los costos de los condimentos.

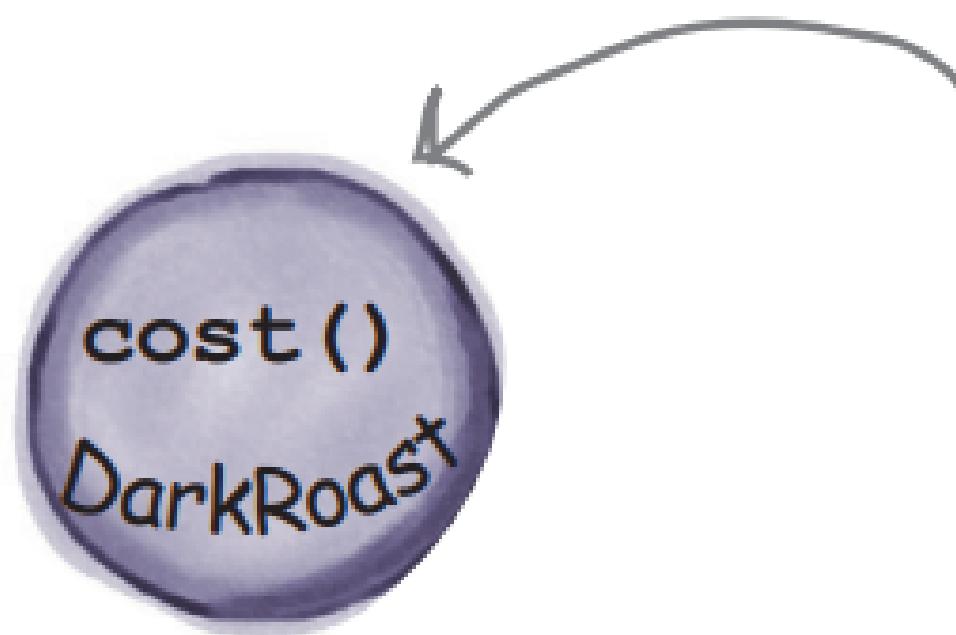
¿Cómo se "**decora**" un objeto y cómo entra en juego la **delegación**?

Piensa en los **objetos decoradores** como "**envoltorios**" (**wrappers**).



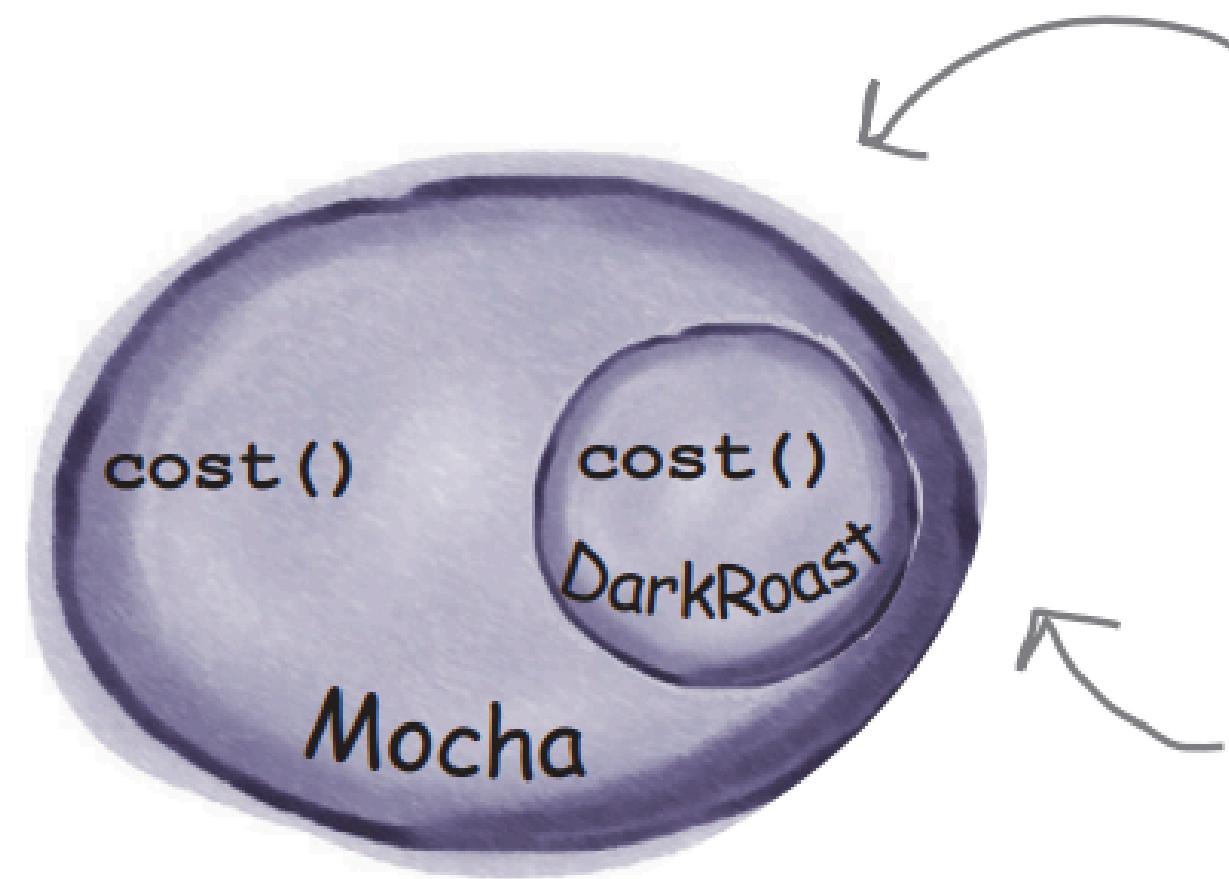
Veamos cómo funciona esto...

1. Comenzamos con nuestro objeto **DarkRoast**



Recuerda que DarkRoast hereda de Beverage y tiene un método cost() que calcula el costo de la bebida.

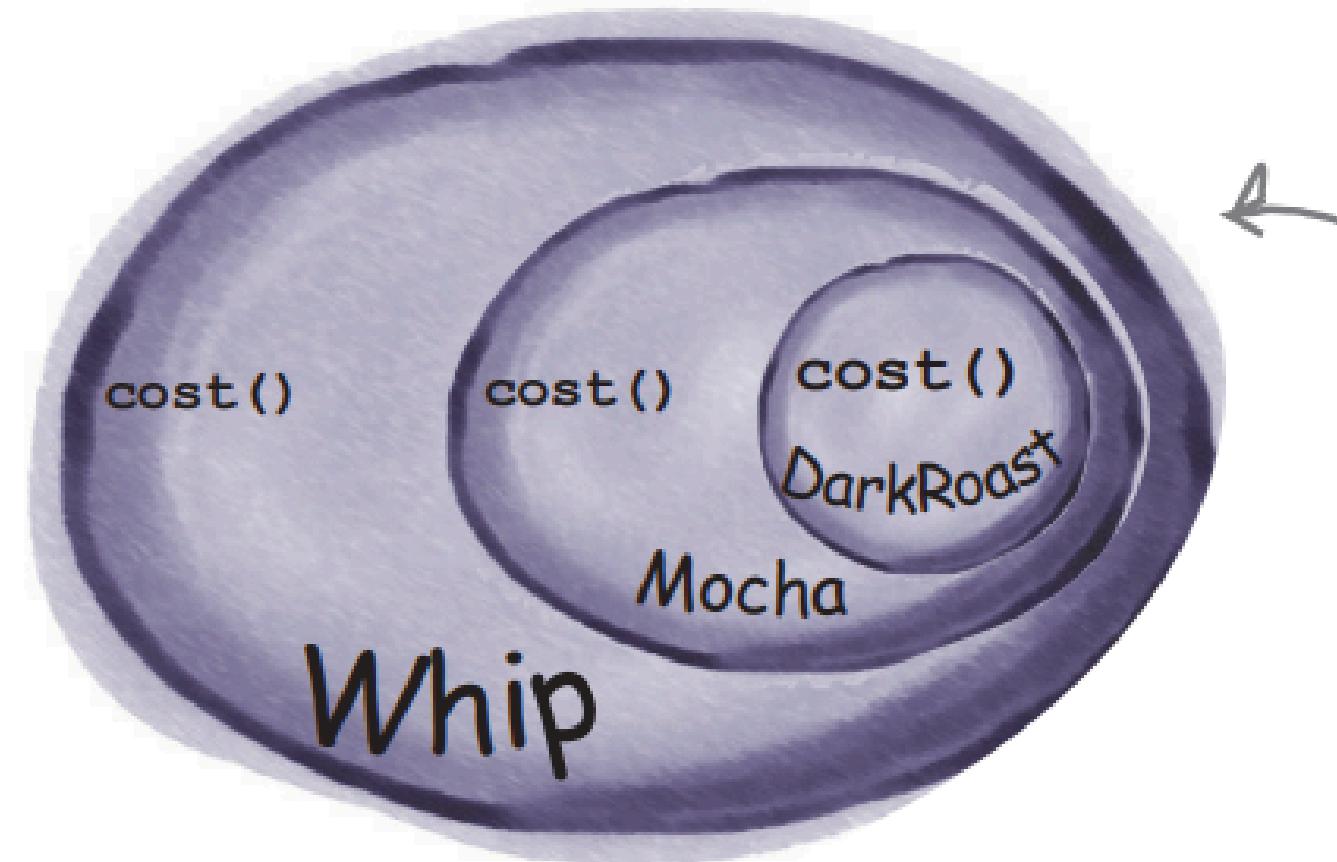
2. El cliente quiere **Moca**, así que creamos un objeto **Mocha** y lo envolvemos alrededor del **DarkRoast**.



El objeto **Mocha** es un decorador. Su tipo imita al objeto que está decorando; en este caso, una **Beverage** (Bebida). (Con "imita", nos referimos a que es del mismo tipo.)

Entonces, Mocha también tiene un método **cost()**, y a través del polimorfismo podemos tratar cualquier **Beverage** envuelta en Mocha como si fuera una **Beverage** también (porque Mocha es un subtipo de **Beverage**).

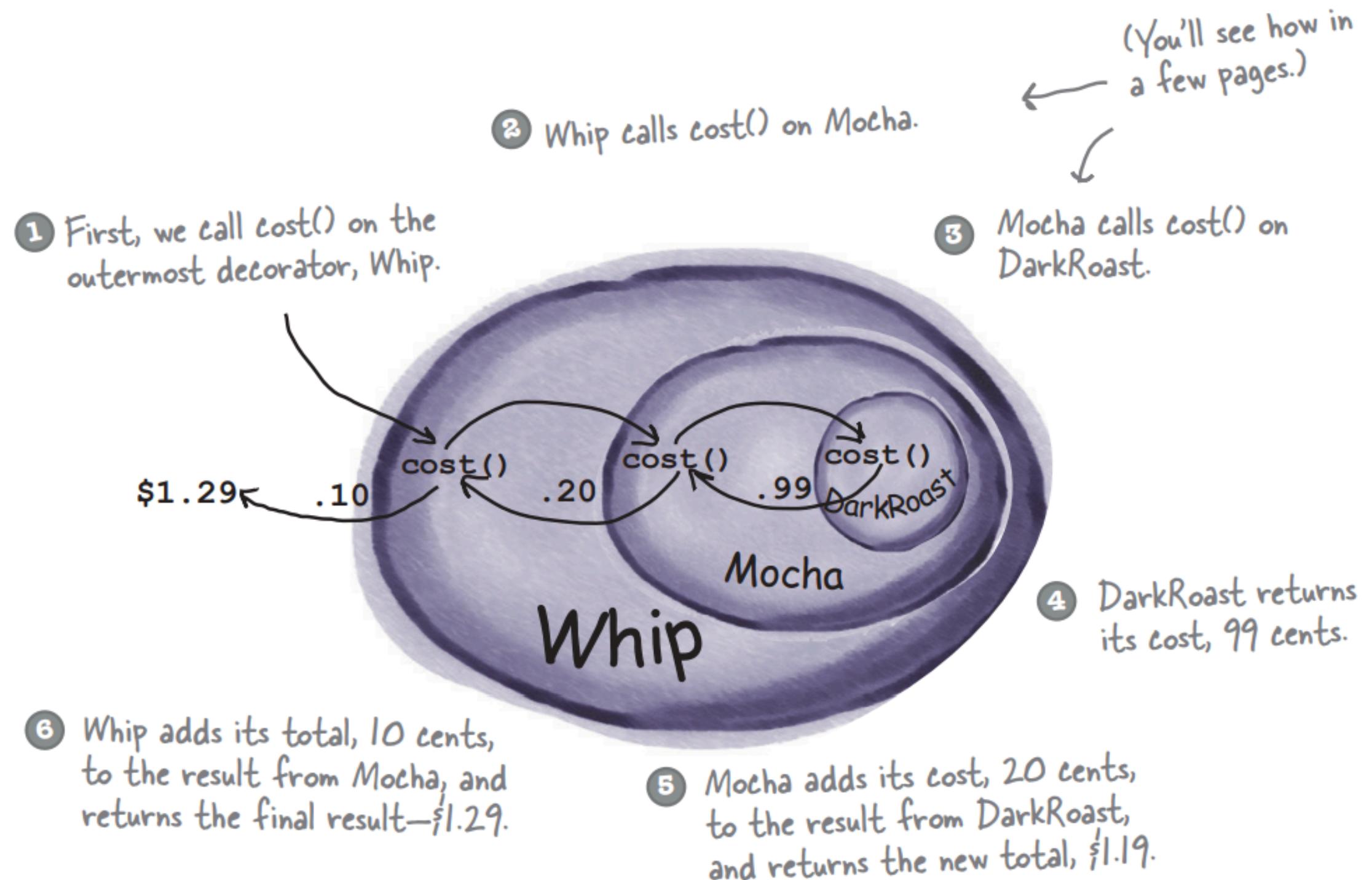
3. El cliente también quiere **Crema Batida**, así que creamos un decorador **Whip** (Crema Batida) y envolvemos el **Mocha** con él.



Whip (Crema Batida) es un decorador, por lo que también imita el tipo de DarkRoast e incluye un método cost().

Por lo tanto, un DarkRoast envuelto en Mocha y Whip sigue siendo una Beverage (Bebida) y podemos hacer cualquier cosa con él que haríamos con un DarkRoast, incluyendo llamar a su método cost().

4. Ahora es el momento de calcular el costo para el cliente. Hacemos esto llamando al método cost() en el decorador más externo, que es **Whip**. **Whip** va a delegar el cálculo del costo al objeto que decora. Y así sucesivamente.



1. Primero, llamamos a `cost()` en el decorador más externo, **Whip** (Crema Batida).
2. **Whip** llama a `cost()` en **Mocha**.
3. **Mocha** llama a `cost()` en **DarkRoast** (Tostado Oscuro).
4. **DarkRoast** devuelve su costo, 99 centavos.
5. **Mocha** añade su costo, 20 centavos, al resultado de **DarkRoast**, y devuelve el nuevo total, \$1.19.
6. **Whip** añade su costo total, 10 centavos, al resultado de **Mocha**, y devuelve el resultado final, \$1.29.

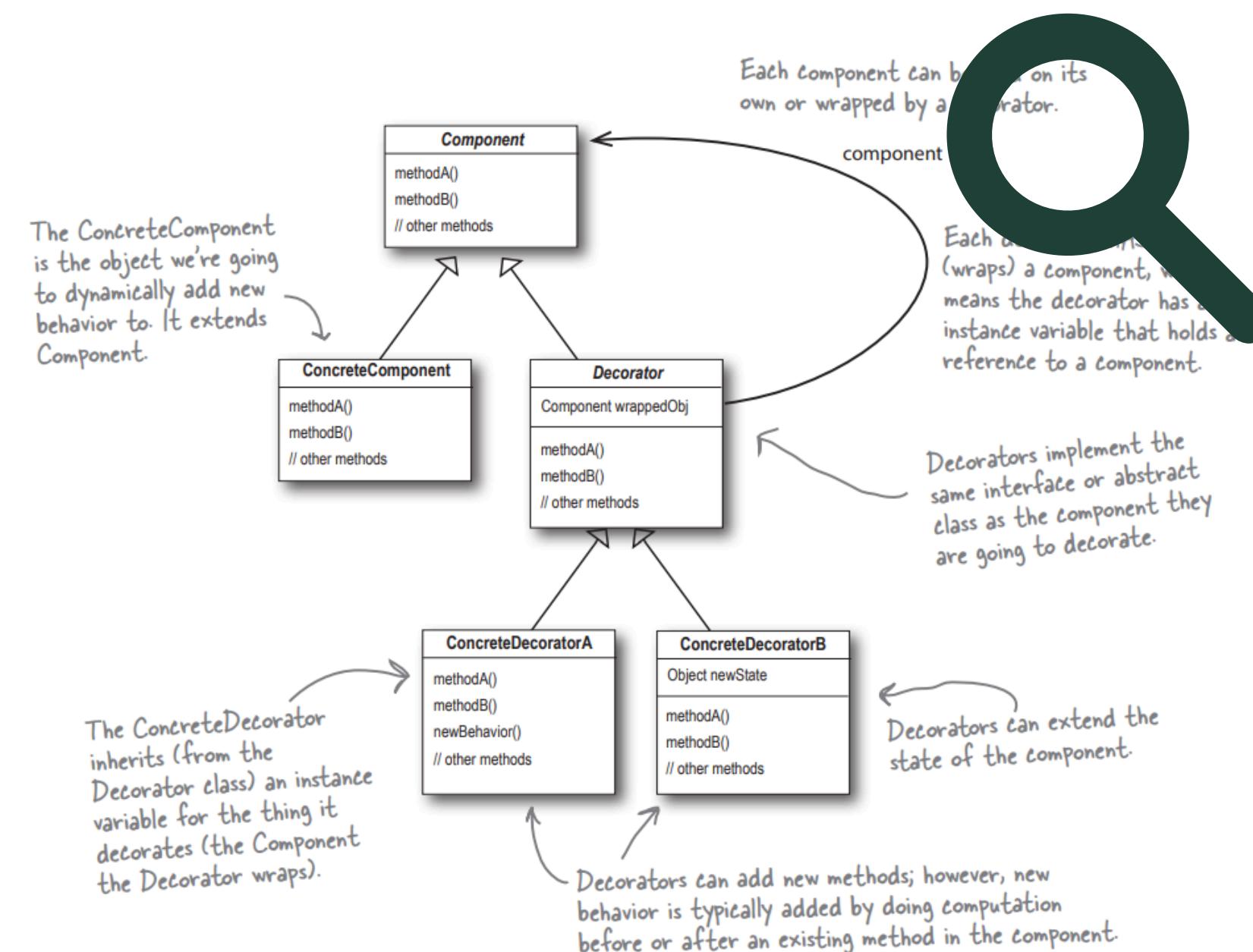
Esto es lo que sabemos hasta ahora sobre los decoradores

- Los Decoradores tienen el mismo supertipo que los objetos que decoran.
- Puedes usar uno o más decoradores para envolver un objeto.
- Dado que el decorador tiene el mismo supertipo que el objeto que decora, podemos pasar un objeto decorado en lugar del objeto original.
- **El decorador añade su propio comportamiento antes y/o después de delegar al objeto que decora para que realice el resto del trabajo.**
- Los objetos pueden ser decorados en cualquier momento, por lo que podemos decorarlos dinámicamente en tiempo de ejecución con tantos decoradores como queramos.

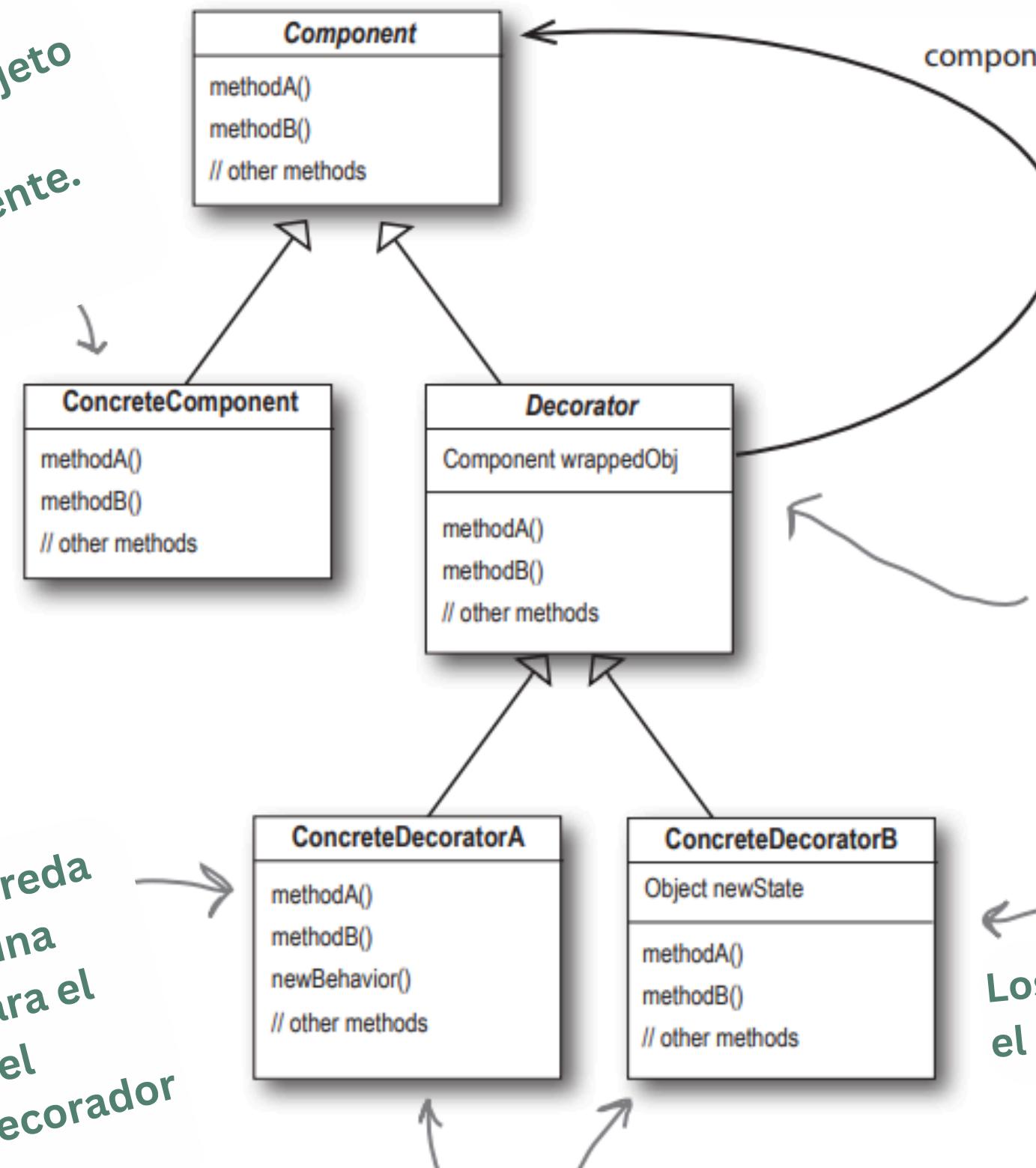


Definición

El **Patrón Decorador** asigna responsabilidades adicionales a un objeto de forma dinámica. Los Decoradores proporcionan una alternativa flexible a la creación de subclases para extender la funcionalidad.



El ConcreteComponent es el objeto al que vamos a añadirle nuevo comportamiento dinámicamente. Extiende a Component.



El ConcreteDecorator hereda (de la clase Decorator) una variable de instancia para el elemento que decora (el Componente que el Decorador envuelve).

Cada componente puede usarse por sí mismo o envuelto por un decorador.

Cada decorador TIENE UN (HAS-A) componente (lo envuelve), lo que significa que el decorador tiene una variable que mantiene una referencia a un componente.

Los decoradores implementan la misma interfaz o clase abstracta que el componente que van a decorar.

Los decoradores pueden extender el estado del componente.

Los decoradores pueden añadir nuevos métodos; sin embargo, el nuevo comportamiento se añade típicamente realizando cálculos antes o después de un método existente en el componente.

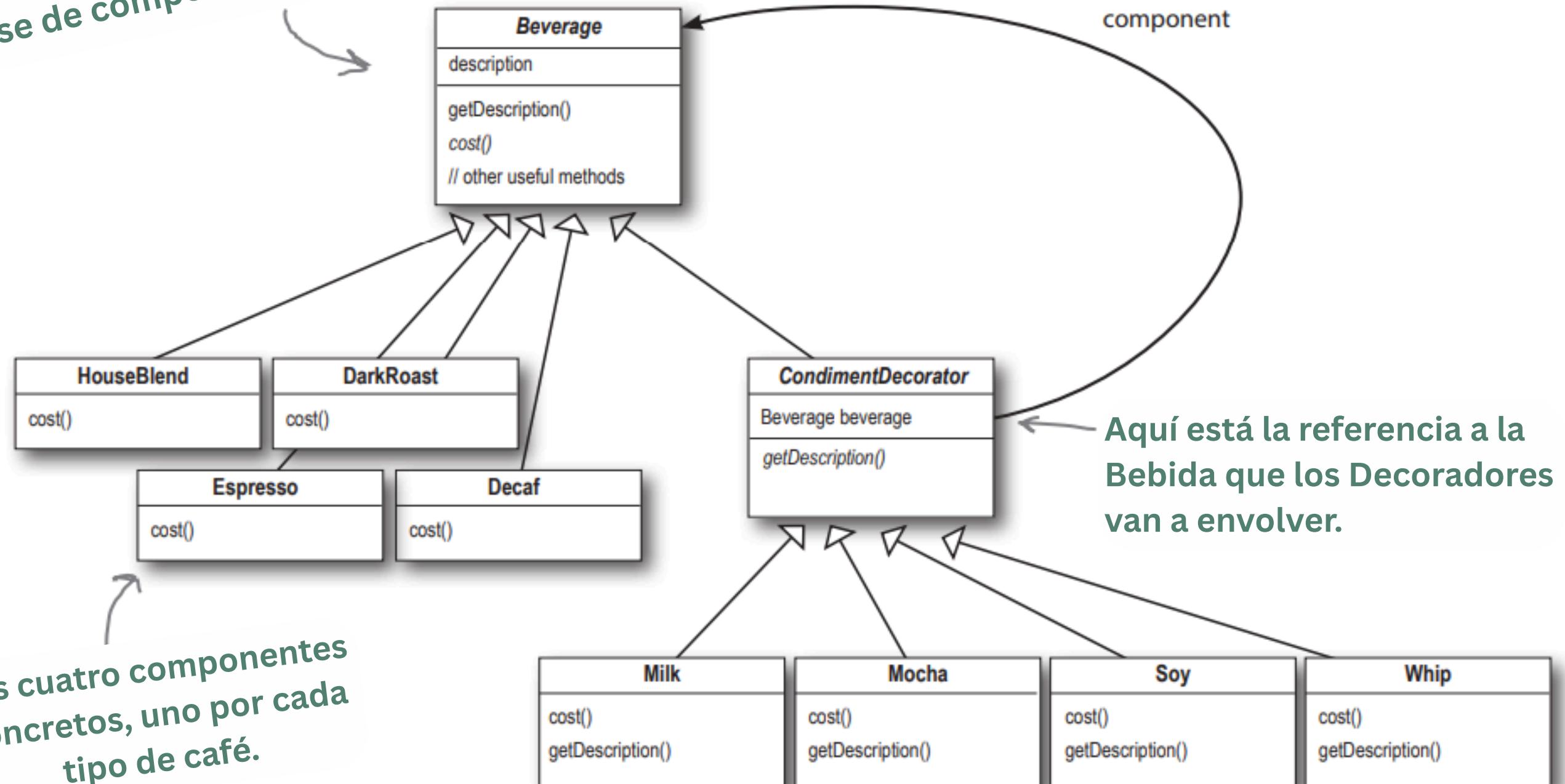
Decorando Páramo...

Es importante destacar que estamos usando la **herencia** para lograr la **coincidencia de tipos**, pero **no** estamos usando la **herencia** para obtener el comportamiento.

Cuando componemos un decorador con un componente, estamos añadiendo nuevo comportamiento.

Estamos adquiriendo nuevo comportamiento **no por heredarlo de una superclase, sino por componer objetos entre sí.**

Beverage (Bebida) actúa como nuestra clase de componente abstracta.



Los cuatro componentes concretos, uno por cada tipo de café.

CondimentDecorator
Beverage beverage
get>Description()

component

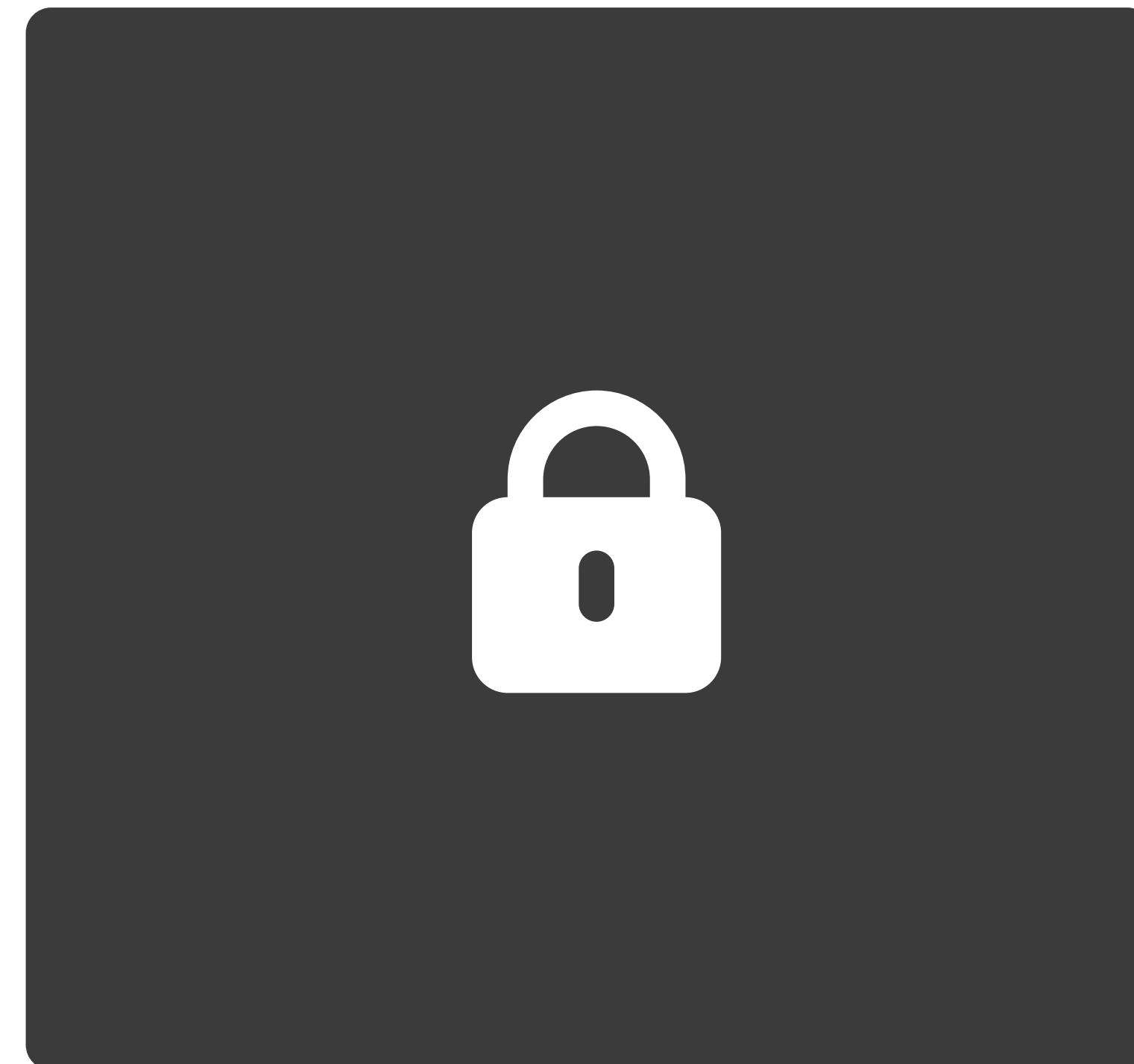
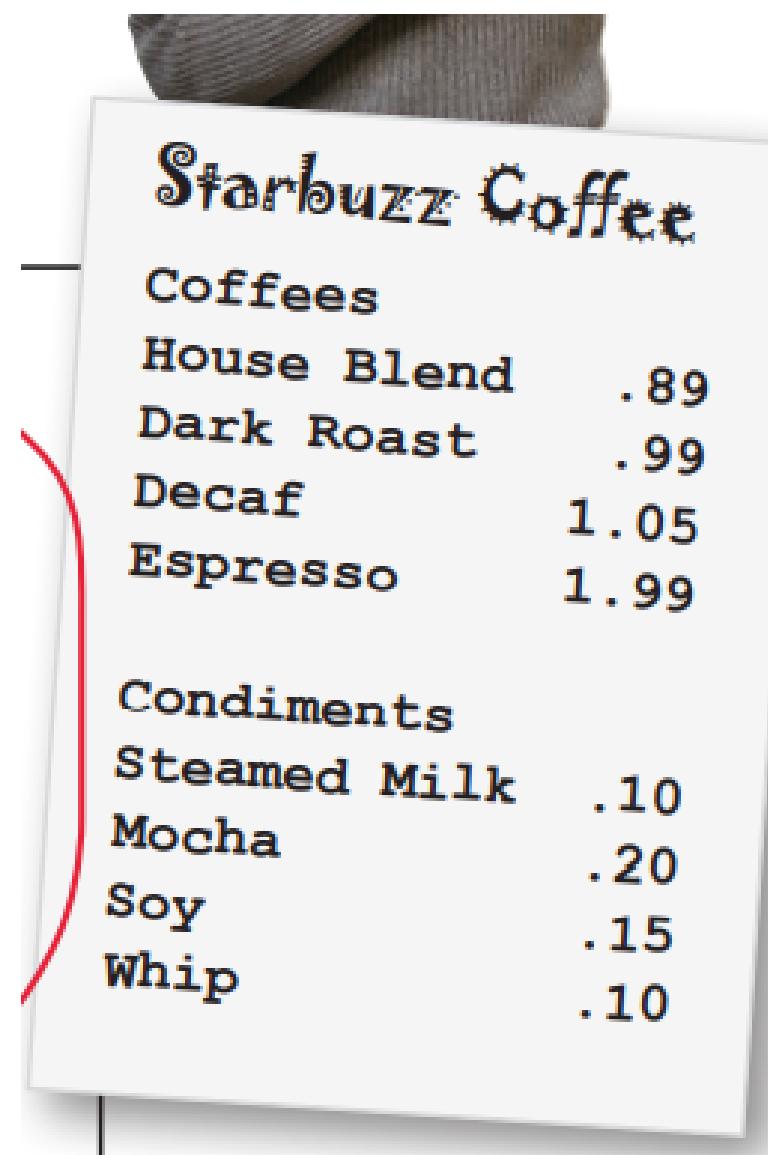
Aquí está la referencia a la Bebida que los Decoradores van a envolver.

Milk
cost()
getDescription()
Mocha
cost()
getDescription()
Soy
cost()
getDescription()
Whip
cost()
getDescription()

Y aquí están nuestros decoradores de condimentos; noten que necesitan implementar no solo cost() sino también getDescription(). Veremos por qué en un momento...

Ejercicio

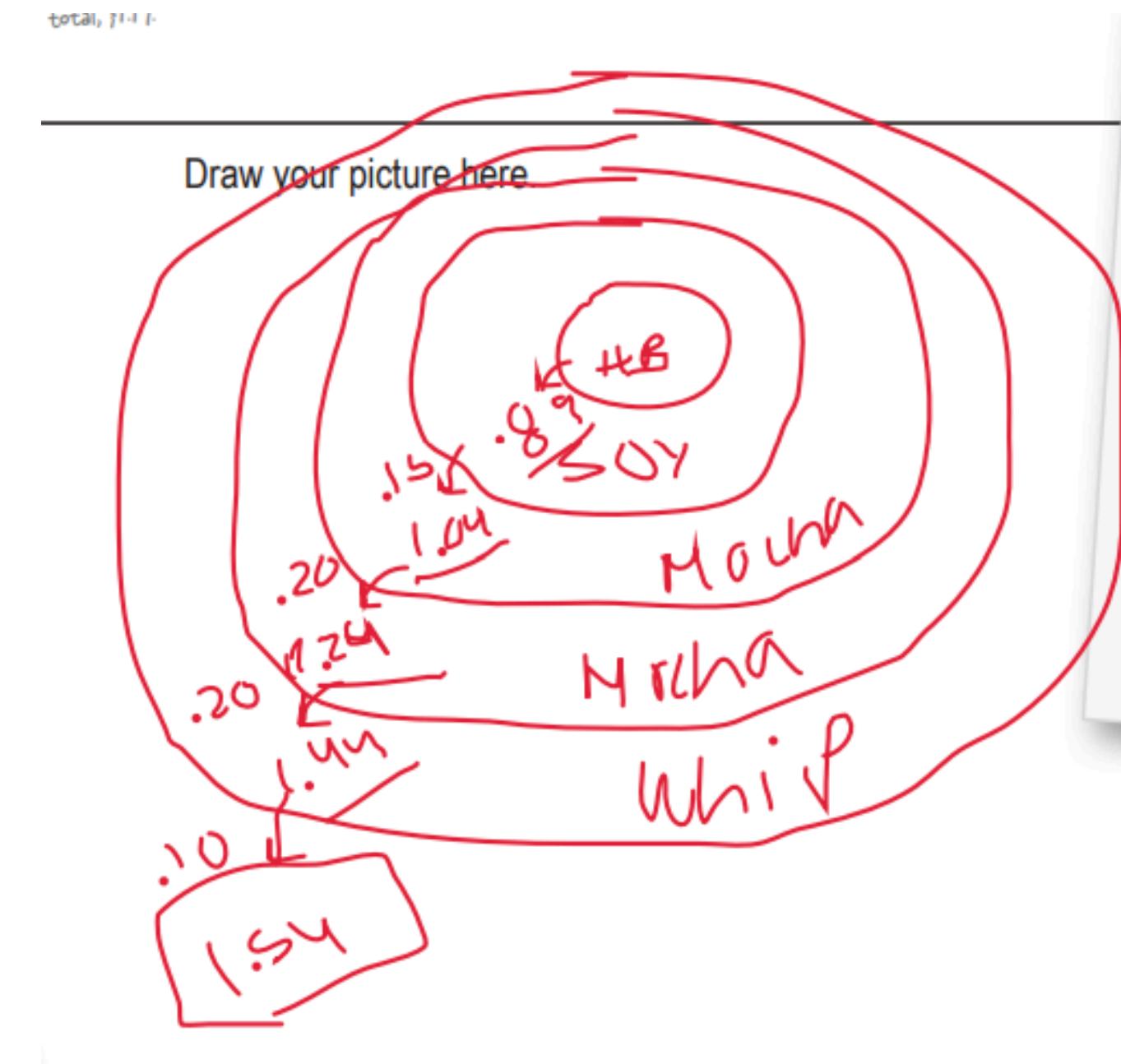
Nos llega una orden de “**double mocha soy latte with whip (Decaf)**”, ve la lista de precios y dibuja ese objeto como lo hicimos anteriormente:



Ejercicio

Nos llega una orden de “**double mocha soy latte with whip**”, ve la lista de precios y dibuja ese objeto como lo hicimos anteriormente:

Starbuzz Coffee	
Coffees	
House Blend	.89
Dark Roast	.99
Decaf	1.05
Espresso	1.99
Condiments	
Steamed Milk	.10
Mocha	.20
Soy	.15
Whip	.10



Arreglando Páramo...

```
abstract class Beverage {
    description: string = "Bebida Desconocida";

    //getDescription ya está implementado, falta implementar cost() en
    las subclases.
    public getDescription(): string {
        return this.description;
    }

    abstract cost(): number;
}

// Necesitamos que sea intercambiable con Beverage, por eso la
// extendemos
abstract class CondimentDecorator extends Beverage {
    // Beverage que cada Decorador va a envolver. El decorador puede
    // envolver cualquier bebida.
    protected beverage: Beverage;

    // Re-declaramos getDescription como abstracto para FORZAR a las
    // subclases (los condimentos concretos) a implementar este método.
    public abstract getDescription(): string;

    // El método abstracto cost() se hereda de Beverage y no necesita
    // ser re-declarado aquí.
}
```

TypeScript

```
// Extendemos la clase abstracta Beverage
class Espresso extends Beverage {
    constructor() {
        super();
        this.description = "Espresso";
    }

    // Sencillamente retornamos el costo
    public cost(): number {
        return 1.99;
    }
}

class HouseBlend extends Beverage {
    constructor() {
        this.description = "Café de la Casa";
    }

    public cost(): number {
        return 0.89;
    }
}
```

```
// Mocha es un decorador, así que extendemos CondimentDecorator
class Mocha extends CondimentDecorator {
    // El constructor toma la bebida que se va a envolver.
    constructor(beverage: Beverage) {
        super();
        this.beverage = beverage;
    }

    public getDescription(): string {
        // Delega a la bebida envuelta y añade la descripción de este condimento.
        return this.beverage.getDescription() + ", Mocha";
    }

    public cost(): number {
        // Delega a la bebida envuelta para obtener su costo y luego añade el costo de este condimento.
        return this.beverage.cost() + 0.20;
    }
}
```

TypeScript

Queremos que nuestra descripción incluya no solo la bebida —“Tostado Oscuro”— sino también cada elemento que la decora (por ejemplo, “Tostado Oscuro, Mocha”). Así que primero delegamos en el objeto que estamos decorando para obtener su descripción, y luego añadimos “, Mocha” a esa descripción.

También necesitamos calcular el costo de nuestra bebida con Mocha.

Primero, delegamos la llamada al objeto que estamos decorando para que pueda calcular su costo; luego, añadimos el costo del Mocha al resultado.

```
// Espresso sin condimentos
let beverage1: Beverage = new Espresso();
console.log(` ${beverage1.getDescription()} ${beverage1.cost()}$`);

// DarkRoast con doble moca y crema batida
let beverage2: Beverage = new DarkRoast();
beverage2 = new Mocha(beverage2);
beverage2 = new Mocha(beverage2);
beverage2 = new Whip(beverage2);
console.log(beverage2.getDescription());
console.log(beverage2.cost() + "$");

// HouseBlend con Soya, Moca y Crema Batida.
let beverage3: Beverage = new HouseBlend();
beverage3 = new Soy(beverage3);
beverage3 = new Mocha(beverage3);
beverage3 = new Whip(beverage3);
console.log(` ${beverage3.getDescription()} ${beverage3.cost()}$`);
```

```
% java StarbuzzCoffee
Espresso $1.99
Dark Roast Coffee, Mocha, Mocha, Whip $1.49
House Blend Coffee, Soy, Mocha, Whip $1.34
%
```

Fin