La madre de todas las herencias.

Según Sigmund Freud siempre ella tiene la culpa...

¿Sabemos qué es la herencia?

Perdón la herencia en el contexto de la POO (Programación Orientada a Objetos)

¿Sabemos qué es POO?

¿Sabemos qué es un paradigma?


TE IMPORTARÍA
IR AL GRANO
DE UNA VÉZ
memegenerator.es

En un minuto...

Paradigma: la forma de pensar acerca de algo.

- Paradigma Funcional: Abstracciones como resultado de pensar acerca de la función. (Este lo conocen bien).

- **Paradigma Orientado a Objetos:** Abstracciones como el resultado de pensar acerca de la forma, basado en clases o debido a que las clases definen características comunes a un conjunto de objetos.

¡Demonios!

¿Sabemos qué es la Abstracción?

Tres al vuelo...

- **Abstracción, Ocultamiento de Información y Encapsulación:** son conceptos diferentes pero fuertemente relacionados.
- La abstracción es la técnica que nos ayuda a identificar que es significativo y que no lo es de un objeto.
- El ocultamiento de Información es la técnica que nos permite enfatizar, hacer visibles, que aspectos son esenciales a la entidad y cuales no.
- La encapsulación es la técnica que nos permite con lo significativo y lo que no lo es, construir una entidad cohesiva, una capsula.

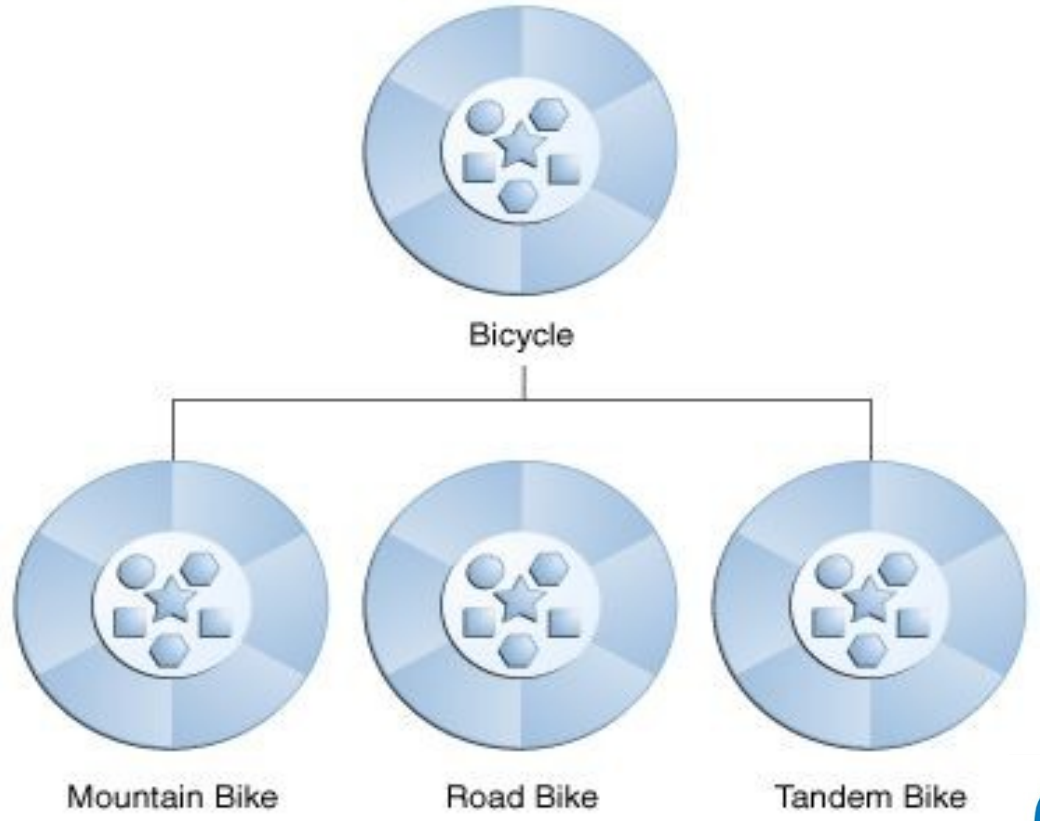
¿Podemos empezar?

¿Que era la herencia?

Herencia es cuando un objeto o clase se basa en otro objeto o clase, usando la misma implementación o comportamiento.

La herencia tiene cabida donde existe una relación “es un/una”...

¡Cuidado!, esta relación es unidireccional hacia la clase padre. Pongamos un ejemplo para verlo mejor:



La bicicleta simple arriba sería la super-clase, y los tipos concretos abajo, diríamos que son las subclases. Es decir el supertipo o abstracción es la super-clase de los subtipos o concreciones.

WTF? En C++ por favor.

Yo eliminando el código que creo que no sirve para nada

```
#include <iostream>

class Bicycle
{
    private:
        int _wheelsAmount;

    public:
        Bicycle() // constructor
        {
            _wheelsAmount = 2;
        }
        ~Bicycle() {} // destructor

        void setWheelsAmount(int wheelsAmount) { _wheelsAmount = wheelsAmount; }

        int getWheelsAmount() { return _wheelsAmount; }
};

class MountainBike : public Bicycle {};
class RoadBike : public Bicycle {};
class TandemBike : public Bicycle {};
```

¿Entonces cuál es el problema con la herencia?

La relación de “ser” vs “tener”.

La connotación de la relación “es un” es más fuerte de lo que creemos, porque ha de ser de por vida, ¿qué quiere decir esto? lo vemos mejor con un ejemplo:

Lean a Barbara...

A ti te encanta correr, y podríamos decir que “eres un runner”, pero claro, es posible que cuando tengas 10 nietos y 80 años prefieras dedicar tu tiempo a la fotografía por ejemplo y dejes de ser un runner. Pues ahí tienes una falsa relación “es un”.

Hubiera sido un error plantear la clase que representa tu persona como herencia de runner porque tú no eres un runner de por vida. Simplemente en ese momento de tu vida tenías un Rol. Recordar el verbo “Tener”.

La mala fama de la herencia no es por que sea mala per se, es porque en el 95% de los casos que vas a usar herencia, si no te planteas las preguntas correctas la vas a usar mal. Porque la herencia **engatusa**.

¡Alerta! Cuándo la herencia se vuelve en tu contra

La principal razón es bastante obvia aunque no lo parezca.

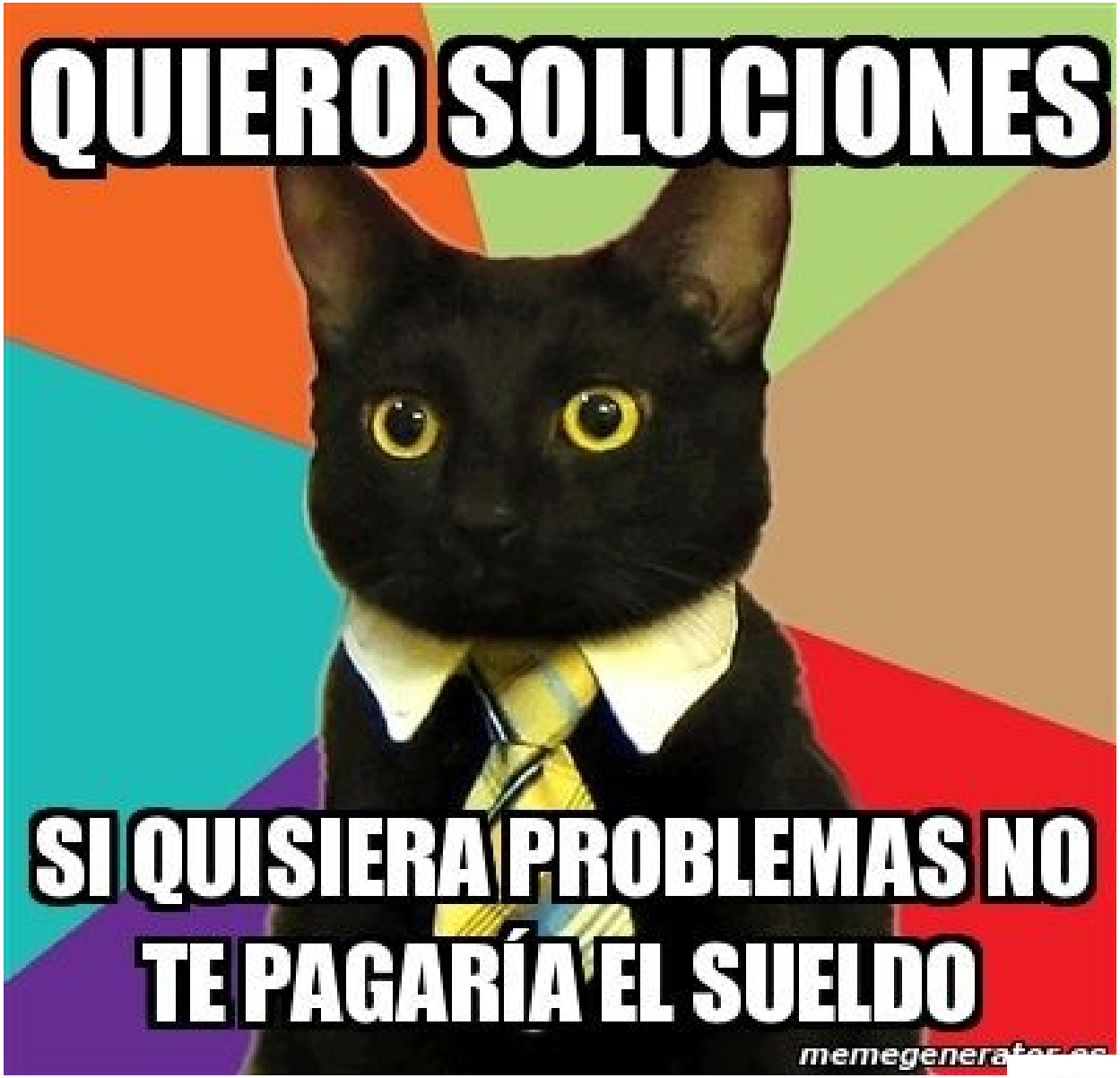
En un proyecto que se enmarca en un proceso de desarrollo ágil, donde los requisitos cambian constantemente y no están para nada bajo control, es imposible hacer suposiciones del tipo: “esta pantalla va a tener que realizar siempre esta acción”, “esta respuesta de servidor siempre tiene 4 listados”.

Borra de tu mente la palabra siempre, siempre guía de forma equivocada en nuestra mente a “es un” y eso guía a herencia. En un desarrollo donde todo cambia, la herencia puede dejar de cumplirse en cualquier momento.

Problemas Divertidos:

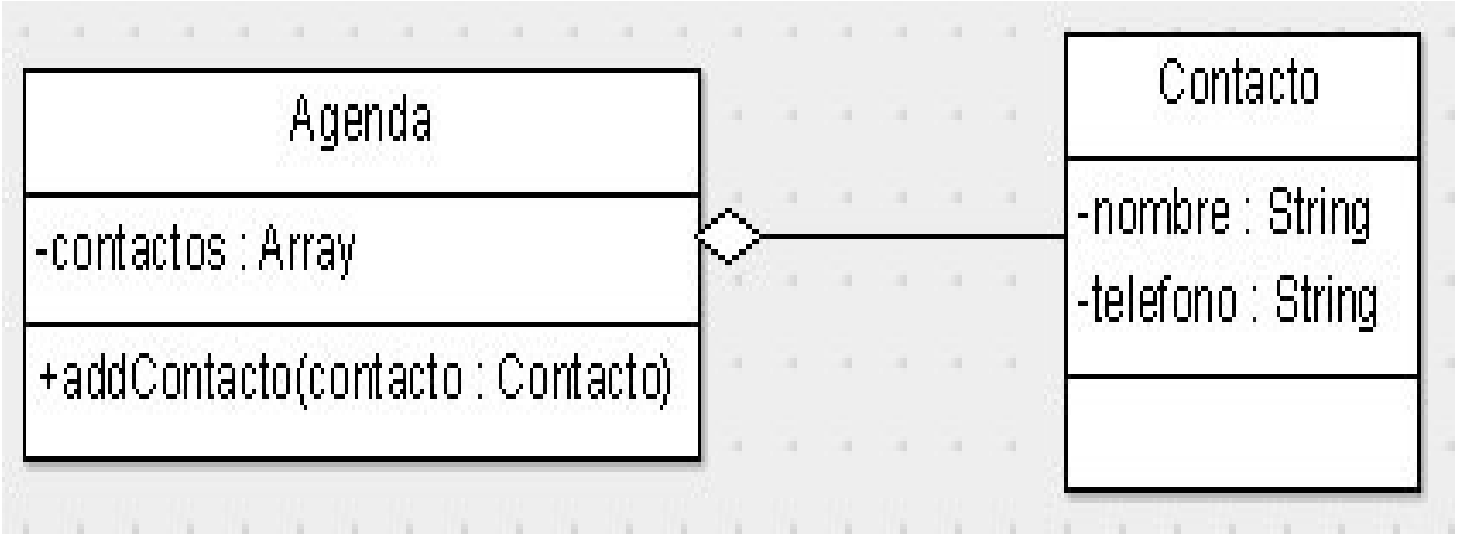
- “La fiesta de los override”.
- “El aluvión de métodos heredados inservibles”.
- “La herencia de 7 niveles”.
- “El quiero heredar y no puedo porque estoy sujeto a otra herencia”.

Problemas Divertidos: Herencia de Implementación

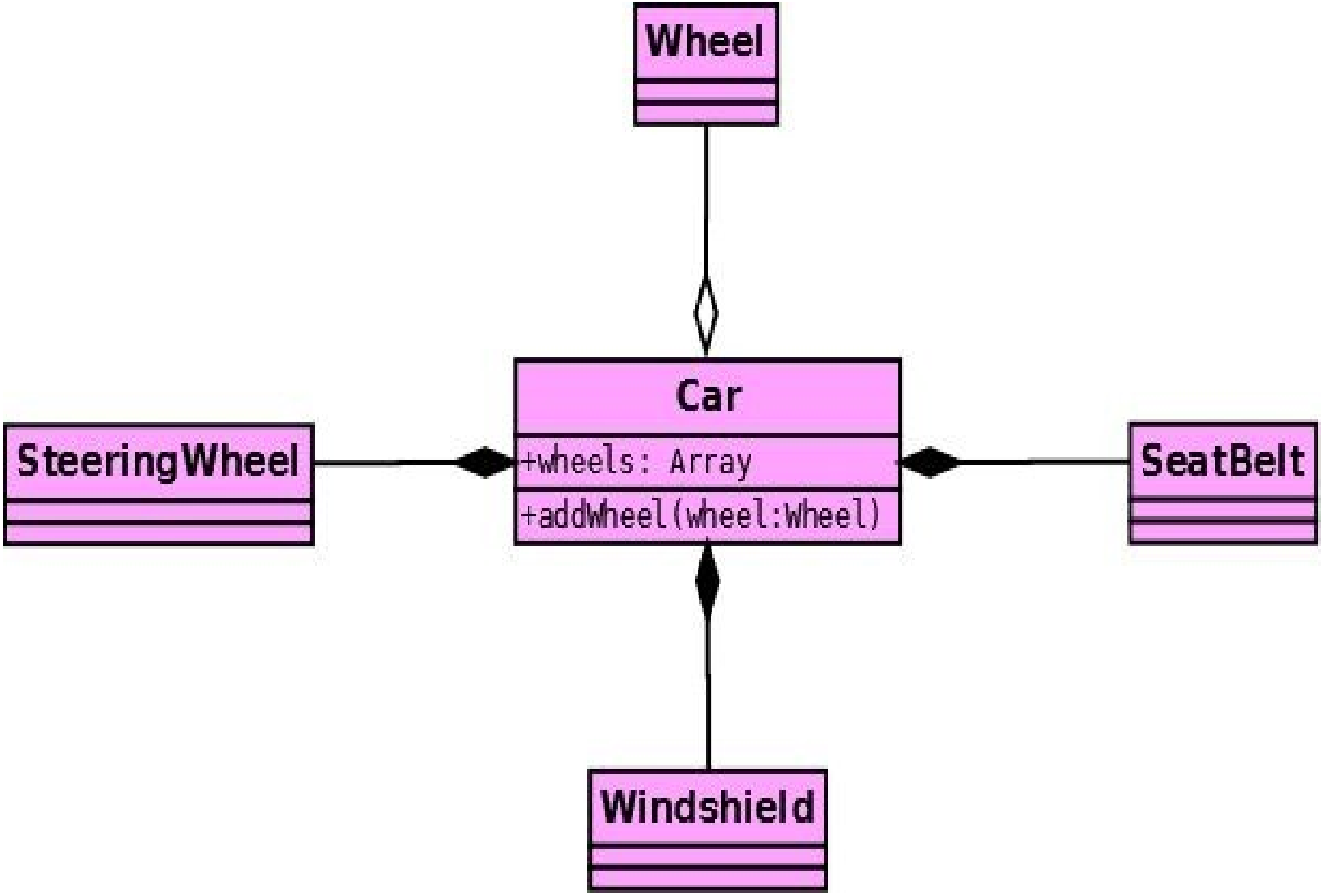

QUIERO SOLUCIONES
SI QUISIERA PROBLEMAS NO TE PAGARÍA EL SUELDO

COMPOSICIÓN Y DELEGACIÓN

Composición quiere decir que tenemos una instancia de una clase que contiene instancias de otras clases que implementan las funciones deseadas.



Es decir, estamos delegando las tareas que nos mandan a hacer a aquella pieza de código que sabe hacerlas. El código que ejecuta esa tarea concreta está sólo en esa pieza y todos delegan el ella para ejecutar dicha tarea. Por lo tanto estamos reutilizando código de nuevo.


Wheel
Car
+wheels: Array
+addWheel(wheel:Wheel)
SteeringWheel
SeatBelt
Windshield

En el caso anterior decimos que el coche esta compuesto por ruedas, volante, cinturones de seguridad, parabrisas... es decir, el coche tiene elementos o usa elementos para hacer todas las funciones que puede realizar. Delega sus responsabilidades en colaboradores designados para cada responsabilidad.

También hay un detalle en los conectores de UML¿Adivinen?

Todo parecen ventajas con la composición, ¿no?

Caso	Diseño basado en herencia	Diseño Basado en Composición
Inicio del desarrollo	Más rapido	Más lento
Diseño del software	Fácil pero pobre	Más complejo
Efectos de lado	Muchos. Además es fácil que aparezcan.	Se reducen y se localizan de forma sencilla
Adaptable a cambios	Cada vez es más difícil el cambio. Muchos niveles de herencia empiezan a generar "Spaguetti code"	Fácil de cambiar
Después de un año de desarrollo	Mucha gente ha pasado por el proyecto haciendo "su propia interpretación de la herencia"	Los desarrolladores que han pasado por el proyecto siguen una estrategia de composición.
Testing	Es difícil de escribir y mantener test por culpa del nivel de herencia y todas las sobreescrituras	Es fácil mantener los test, y escribir nuevos las piezas están acotadas y las fronteras claras.
Es fácil extender el software	Paradoja: lo que está diseñado para ser extendido no se puede extender de forma sencilla porque heredamos funcionalidad "basura"	Se extiende el software por composición de las piezas ya existentes y la incorporación de nuevas

Y ahora viene lo peor de la composición...

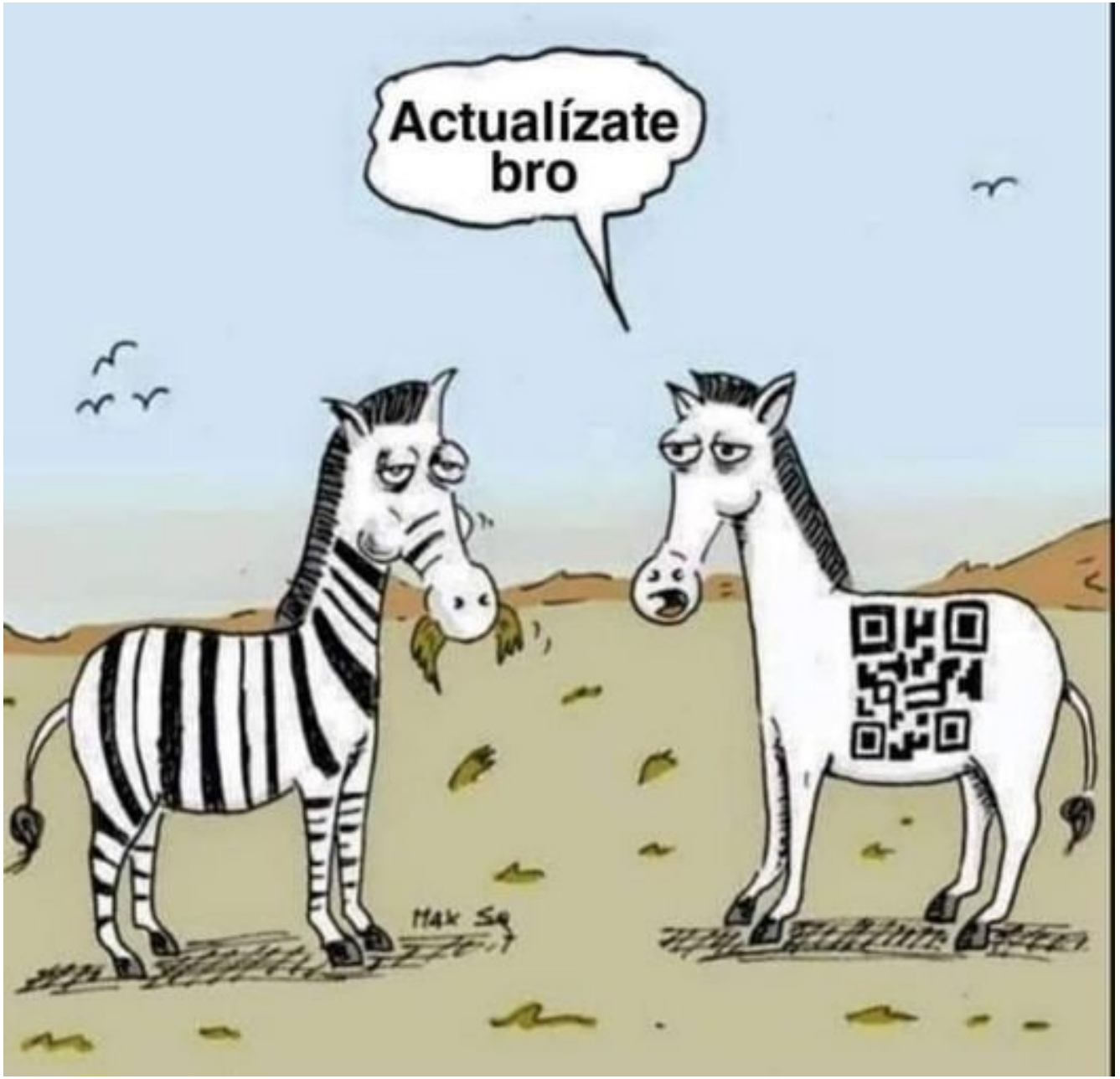
La composición por sí misma no es polimórfica.

En programación orientada a objetos, el polimorfismo se refiere a la propiedad por la que es posible enviar mensajes sintácticamente iguales a objetos de tipos distintos. El único requisito que deben cumplir los objetos que se utilizan de manera polimórfica es saber responder al mensaje que se les envía.


CALMA CALMÁ
QUE NO CUNDA EL PANICO !!

Con el uso de interfaces conseguimos traer el polimorfismo al mundo de la composición.

-Ah, pero si eso yo lo sabía....

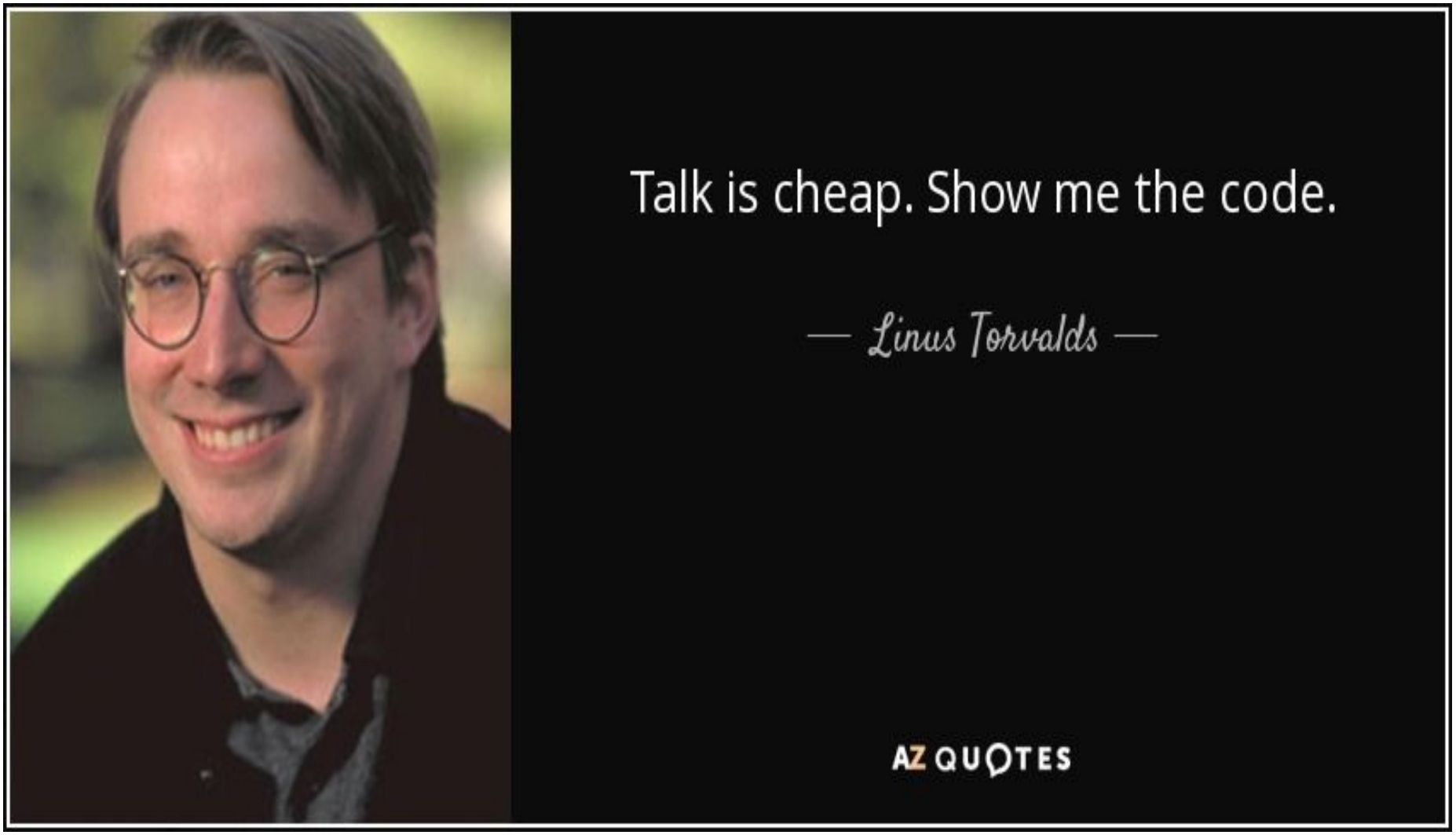

Actualízate bro

Recordar a Barbara Liskov la mujer detrás y delante de todo esto....

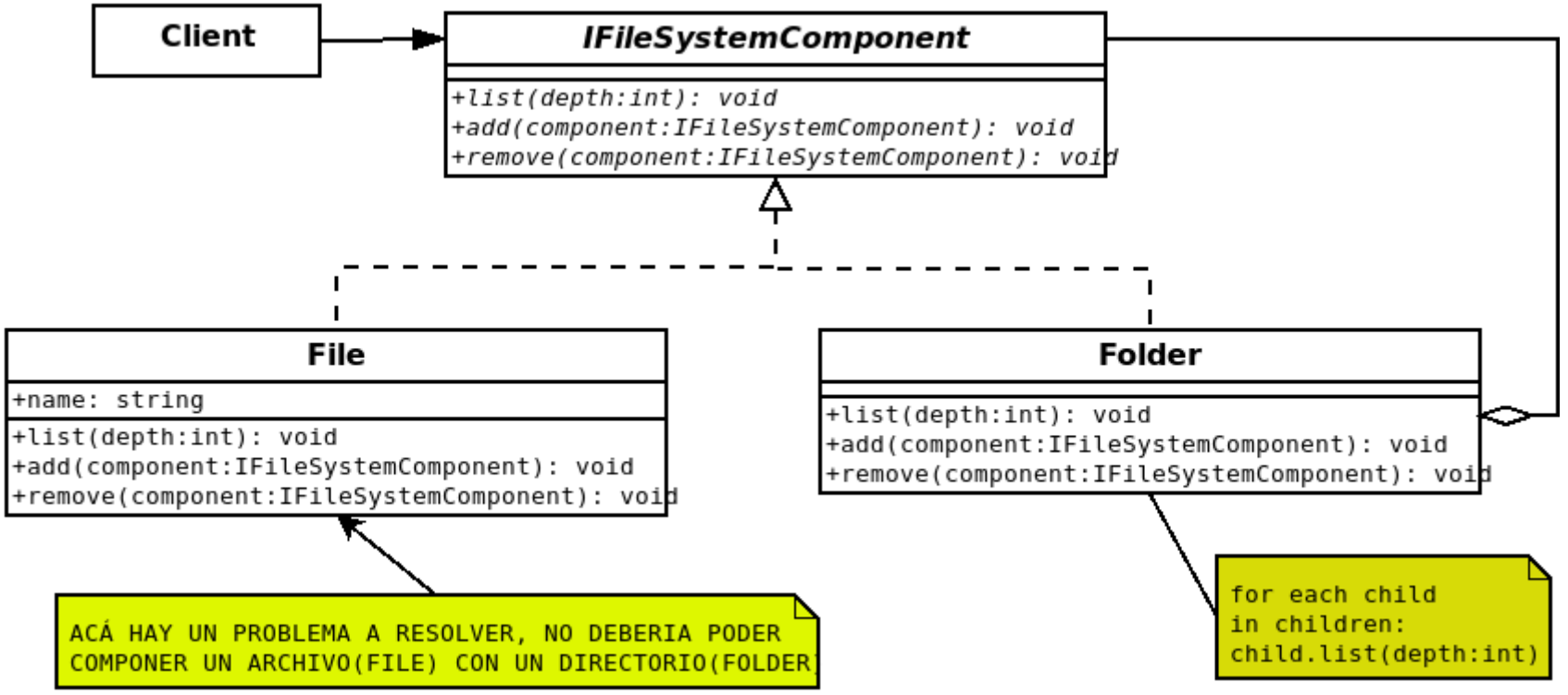
Lean a Barbara...

- Enlace a paper en español.
- Enlace a paper en inglés.

Con el uso de interfaces, podemos hacer que nuestros objetos compuestos se hagan pasar por la forma que nos venga bien. Y lo que es mejor, puedes implementar varias interfaces en cualquier momento o dejar de implementarlas con muchas más versatilidad que en la herencia, sin estar atado a nada más que los métodos que implemente esa interfaz concreta.


Talk is cheap. Show me the code.
— Linus Torvalds —
AZ QUOTES

COMPOSITE (UML)



Esta presentación “se compone con” y no podría ser nada sin el excelente artículo escrito por Sergio Martinez Rodríguez: “Herencia vs Composición ¿Tienes claro cuál es el rival más débil?” que pueden encontrar y deberían leer en:

https://devexperto.com/herencia-vs-composicion/

Y no se olviden lean a

Barbara...

ISFT