

# RE-APRENDIENDO OOP

Esta vez, bien



→ Antonio González Gea



Emmanuel Valverde Ramos →

# ÍNDICE

1. [Introducción](#)
2. [Contexto Historico](#)
  - 2.1. [Escuela de Bjarne Stroustrup](#)
  - 2.2. [Escuela de Alan Kay](#)
3. [Diseño orientado a objetos](#)
4. [Conclusiones](#)
5. [Preguntas](#)

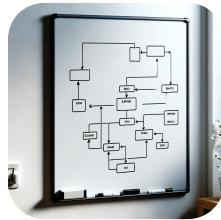


The background image is a wide-angle photograph of a volcanic landscape at night. In the center, a large, dark volcano with a sharp peak rises against a deep blue night sky filled with numerous stars. The foreground is a dark, hilly terrain with sparse vegetation and some distant lights from buildings or campfires. The overall mood is mysterious and serene.

# INTRODUCCIÓN

# ¿QUÉ SON LOS PARADIGMAS DE PROGRAMACIÓN?

Son un enfoque o un estilo de utilizar la programación para resolver problemas.



Estructurada



Orientado a  
Objetos



Funcional



Modular



Imperativa



Procedural



Orientado a  
Prototipos



Declarativa

# UN EJEMPLO PRÁCTICO

Cada paradigma tiene sus propias reglas, conceptos y técnicas que guían el proceso de desarrollo de software.



```
SELECT
    email
FROM
    users
WHERE
    email IS NOT NULL;
```

↑  
**Declarativo**  
*describimos el resultado esperado, sin detallar los pasos a seguir*

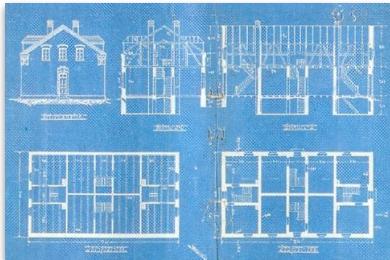


```
<?php

function getUserEmails($users)
{
    $emails = [];
    for ($i = 0; $i < count($users); $i++) {
        $user = $users[$i];
        if ($user["email"] !== null) {
            $emails[] = $user["email"];
        }
    }
    return $emails;
}
```

↑  
**Imperativo**  
*describimos los pasos a seguir para lograr el resultado esperado*

# ESTRUCTURA BÁSICA DE LA OOP



*new*



## Clases

Son las **instrucciones** que definen el comportamiento de un objeto.

*mediante métodos  
y atributos*



## Objetos

Resultado de seguir las **instrucciones** de una clase.

*concretamente las de un método mágico llamado constructor*



## Métodos

**Acciones** que se pueden realizar con nuestro objeto.

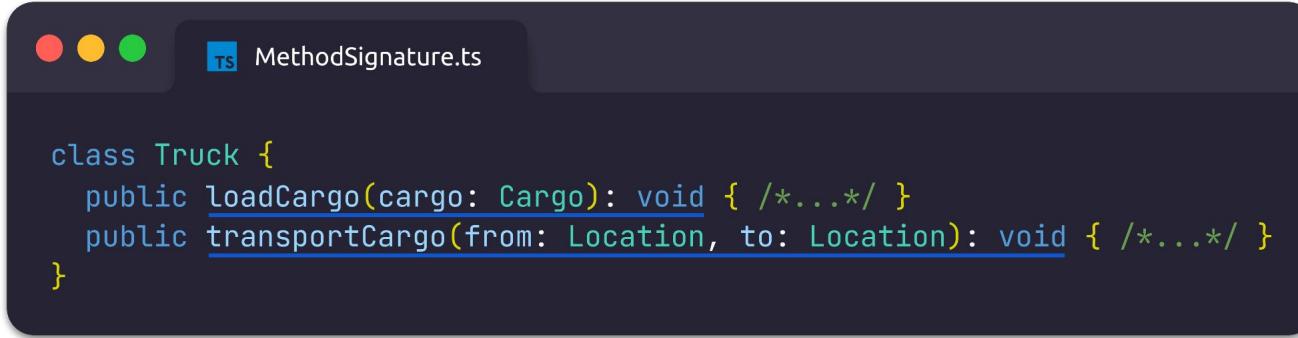
- cerrar la puerta
- encender la calefacción

## Atributos

Son **el estado y las características** de nuestro objeto.

- metros cuadrados
- número de ventanas
- dirección

# ESTRUCTURA BÁSICA DE LA OOP



```
MethodSignature.ts

class Truck {
    public loadCargo(cargo: Cargo): void { /*...*/ }
    public transportCargo(from: Location, to: Location): void { /*...*/ }
}
```

## Firma de un método

- + nombre del método
- + número y tipos de sus argumentos
- + tipo de retorno

# LA RAZÓN POR LA QUE OOP SE ENSEÑA MAL

algo que tiene múltiples  
significados o interpretaciones

La Programación Orientada a Objetos es un **término polisémico** que tiene diferentes visiones de **orientación** y **programación** ya que los términos evolucionan a medida que lo hacen los problemas.

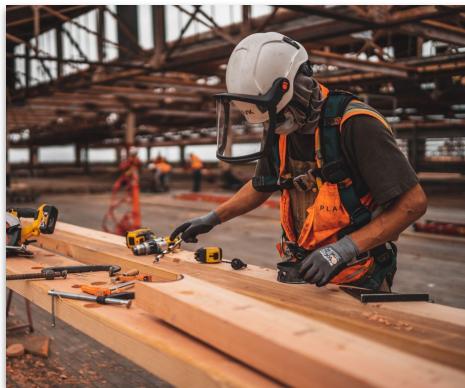
esta es la razón por la que hay tanta  
confusión en las diferentes definiciones,  
teorías y prácticas relacionadas con OOP

# OOP

## Programación Orientada a Objetos en la práctica tiene ambos significados

diferentes visiones sobre un mismo tema

contextos y límites claros



Estilo de Programación  
Orientado a Objetos

foco en la organización, estructura y  
construcción del código



Diseñando Software  
Orientado a Objetos

el enfoque del diseño evoluciona  
junto a los problemas

# HERRAMIENTAS Y DISCIPLINAS



## HERRAMIENTAS



## DISCIPLINAS

Características del lenguaje que utilizamos para alcanzar nuestros objetivos.

**No están ligadas 1:1** con los conceptos de las disciplinas.

- Clases, Interfaces, Abstractas, Rasgos, Accesibilidad, Visibilidad...

Resuelven las necesidades de nuestros stakeholders con código sostenible y de calidad.

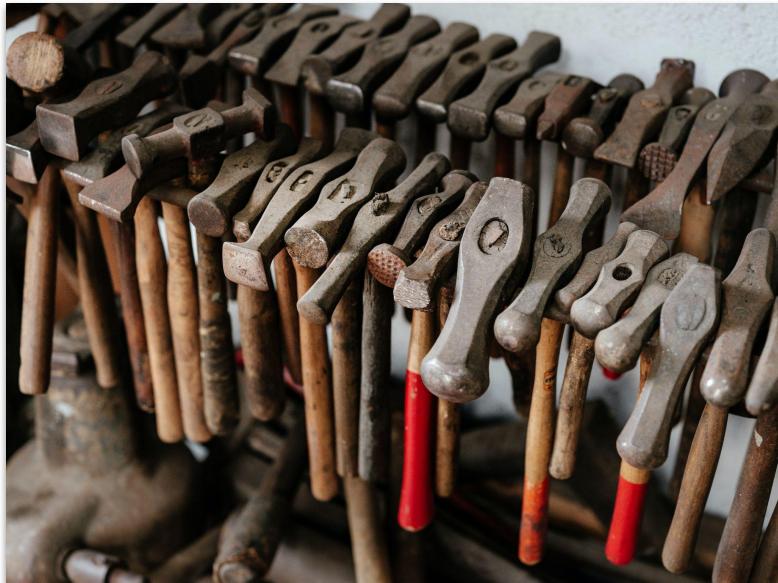
Nos permiten hacer entregas constantes y de valor con seguridad.

- Diseño, Valores, Principios, Pilares Fundamentales, Relaciones...

# EL MARTILLO DE MASLOW

Abraham Maslow

“ If all you have is a hammer, everything looks like a nail. ”



Conociendo **sólo las herramientas** podrías hacer código Orientado a Objetos, pero la pregunta es...

*¿Será código mantenible en el tiempo?*

# LEYES DE LA EVOLUCIÓN DEL SOFTWARE - CAMBIO CONTINUO

todo en software  
tiende a cambiar



Meir M. Lehman

Un programa que se usa **en un entorno real necesariamente**  
**debe cambiar** o se volverá progresivamente menos útil y

menos satisfactorio para el usuario.



# CONTEXTO HISTORICO

# ESCUELAS DE OOP

Bjarne Stroustrup:

Abstraction!  
Inheritance!  
Polymorphism!

creador de C++



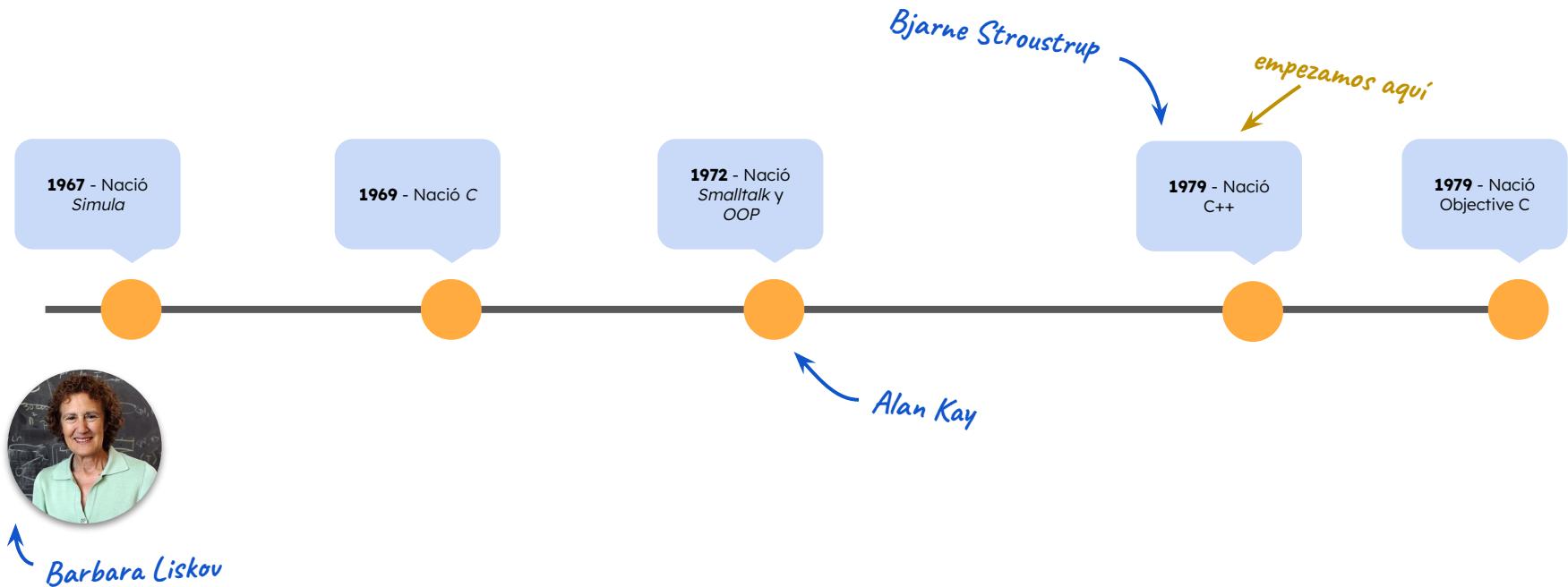
Alan Kay:

Everything is an object!  
Objects communicate by sending and receiving messages!

creador de OOP y Smalltalk



# LÍNEA TEMPORAL PARA OOP



# ESCUELA DE Bjarne Stroustrup

(C++, Java)

creador de C++

“Object-oriented programming is a technique for programming - a paradigm for writing "good" programs for a set of problems.”

## What is “Object-Oriented Programming”? (1991 revised version)

*Bjarne Stroustrup*

AT&T Bell Laboratories  
Murray Hill, New Jersey 07974

### *ABSTRACT*

“Object-Oriented Programming” and “Data Abstraction” have become very common terms. Unfortunately, few people agree on what they mean. I will offer informal definitions that appear to make sense in the context of languages like Ada, C++, Modula-2, Simula, and Smalltalk. The general idea is to equate “support for data abstraction” with the ability to define and use new types and equate “support for object-oriented programming” with the ability to express type hierarchies. Features necessary to support these programming styles in a general purpose programming language will be discussed. The presentation centers around C++ but is not limited to facilities provided by that language.

<https://www.stroustrup.com/whatis.pdf>

# ESCUELA DE Bjarne Stroustrup

*creador de C++*

Desde su **punto de vista original** la Orientación a Objetos debería soportar los siguientes puntos:

1. **Abstracción**: proporciona algún **tipo de clases** y objetos.
2. **Herencia**: permite crear nuevas abstracciones a partir de las existentes.
3. **Polimorfismo**: proporciona alguna forma de vinculación en tiempo de ejecución.

*late binding*

# ESCUELA DE BJARNE STROUSTRUP

creador de C++

**Las disciplinas** de esta escuela, conocidas como los 4 pilares de OOP.



Abstracción



Encapsulación



Herencia



Polimorfismo



✓ *no estaba en la visión  
original de Bjarne*

# ESCUELA DE BJARNE STROUSTRUP

creador de C++

Las disciplinas de esta escuela, conocidas como los 4 pilares de OOP.

Siempre nos enfocamos en el “**CÓMO**” funcionan estos conceptos, no en “**POR QUÉ**” necesitamos estos conceptos y en “**CUÁNDO**” deberíamos o no usarlos.

la gente culpa a la herencia o al polimorfismo  
porque nadie sabe porqué o cuando usarla

# PILARES FUNDAMENTALES - ABSTRACCIÓN

palabra polisémica ↑

no confundir con clases abstractas

Un concepto tiene más de un significado o representación según el contexto.



```
index.ts
```

```
class Truck {  
    private cargoCapacity: number;  
    private fuelTankCapacity: string;  
    private plateNumber: string;  
  
    public loadCargo(cargo: Cargo): void { /*...*/ }  
    public transportCargo(from: Location, to: Location): void { /*...*/ }  
}
```

nuestro objetivo es lograr la  
forma más simple de  
representar un objeto dado  
nuestro contexto

```
index.ts
```

```
class Truck {  
    private brand: string;  
    private model: string;  
    private cylinderCapacity: number;  
    private color: string;  
    private policyNumber: string;  
    private plateNumber: string;  
  
    public getPolicyPrice(): number { /*...*/ }  
    public fileClaim(): number { /*...*/ }  
}
```

# PILARES FUNDAMENTALES - ENCAPSULACIÓN

Comportamiento y estado ocultos tras una **API sencilla**. Exponemos solamente lo que necesitamos utilizar.

Puerta de un Garaje

```
DoorRemote.ts
```

```
class DoorRemote {  
    private privateKey: string;  
    private frequency: number;  
  
    public openDoor(): void { /*...*/ }  
}
```

fácil de usar

complejidad  
interna oculta

no confundir con el  
keyword interface

nos referimos a  
interfaces públicas



aprietas un botón y  
se abre la puerta!

# HERRAMIENTAS PARA LA ENCAPSULACIÓN



## Visibilidad

¿Quiénes necesitan acceder a los métodos y atributos del objeto?

- **public:** desde cualquier parte
- **private:** solo desde dentro
- **protected:** también por sus hijos

## Accesibilidad

¿Nuestro método necesita utilizar el **estado** del objeto?

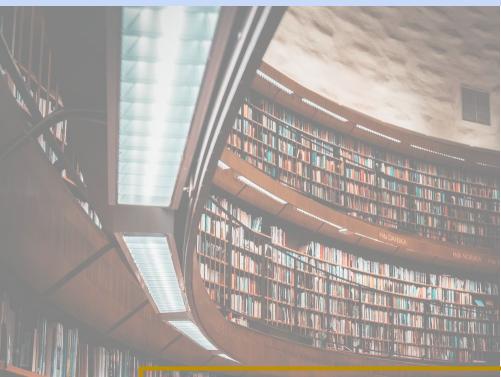
- **static:** no requiere una instancia
- **instance:** requiere una instancia

a.k.a. dinámico  
normalmente no tiene keyword

# PILARES FUNDAMENTALES - POLIMORFISMO

Capacidad de tratar objetos de diferentes clases de **forma uniforme**.

usando interfaces, clases abstractas o duck typing



interfaz uniforme

implementaciones

```
renteable.ts
```

```
interface Rentable {
    public ean(): EanId;
    public rent(): void;
}
```

```
ProductDetails.ts
```

```
class ProductDetails {
    private product: Rentable;

    public rent(): void {
        this.product.rent();
    }
}
```

código cliente

```
Book.ts
```

```
class Book implements Rentable {
    private isbn: ISBN;

    public ean(): EanId { /* bar code */ }
    public rent(): void { /* ... */ }
}
```

```
Newspaper.ts
```

```
class Newspaper implements Rentable {
    private issn: ISSN;

    public ean(): EanId { /* bar code */ }
    public rent(): void { /* ... */ }
}
```

```
CompactDisc.ts
```

```
class CompactDisc implements Rentable {
    private ean: EanId;

    public ean(): EanId { /* bar code */ }
    public rent(): void { /* ... */ }
}
```

# HERRAMIENTAS PARA EL POLIMORFISMO



## Interfaces

Literalmente **contratos**, las clases que las implementan deben cumplir lo que estas dicen.

↑  
definen cómo debe ser una clase



## Duck Typing

Si parece un **pato**.  
Si se comporta como un **pato**.

Debe ser un **pato**.

↑  
¿puede graznar?  
¡es un pato!



## Rasgos

Añaden métodos y atributos a una clase.

↑  
*se puede saber si un objeto tiene un rasgo o no*



## Clases Abstractas

Combinan características de la **herencia** y las **interfaces**.

según el lenguaje:

- no se pueden instanciar
- implementan métodos y atributos
- tienen métodos abstractos
- puedes extender una o varias

# PILARES FUNDAMENTALES - HERENCIA

Permite crear clases derivadas que heredan los atributos y métodos de la clase base.

```
<?php  
class Chicken extends Bird {  
    protected string $specieName;  
    protected Peak $peakType;  
  
    public function fly(): void { /* can fly */ }  
    public function dive(): void { /* can not dive */ }  
}
```



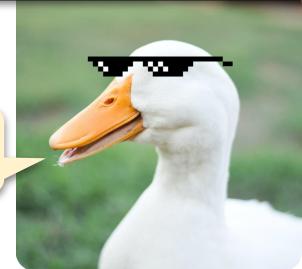
¿Eh?

```
<?php  
class Bird {  
    protected string $specieName;  
    protected Peak $peakType;  
  
    public function fly(): void { /* can fly */ }  
    public function dive(): void { /* can dive */ }  
}
```



herencia  
mal utilizada

```
<?php  
class Duck extends Bird {  
    protected string $specieName;  
    protected Peak $peakType;  
  
    public function fly(): void { /* can fly */ }  
    public function dive(): void { /* can dive */ }  
}
```



¡Soy la forma de  
vida definitiva!

```
<?php  
class Ostrich extends Bird {  
    protected string $specieName;  
    protected Peak $peakType;  
  
    public function fly(): void { /* can not fly */ }  
    public function dive(): void { /* can not dive */ }  
}
```

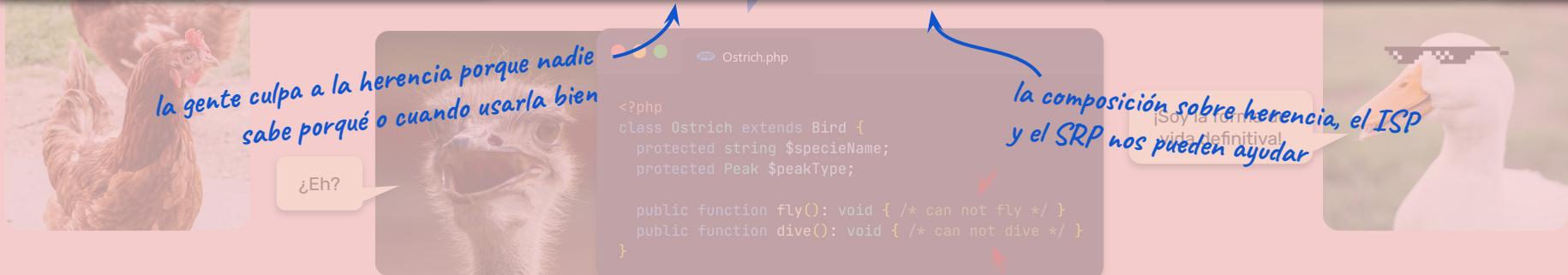
# PILARES FUNDAMENTALES - HERENCIA

Permite crear clases derivadas que heredan los atributos y métodos de la clase base.

ambigüedad

herencia

La herencia **AUMENTA LA ABSTRACCIÓN**, la abstracción en sí no es mala, pero siempre **AÑADE MÁS COMPLEJIDAD**, y la complejidad mal usada puede traer el problema de "**COMPLEJIDAD ACCIDENTAL**".



# PILARES FUNDAMENTALES - COMPOSICIÓN

*rasgos*

```
<?php
trait CanFly {
    public function fly(): void { /*...*/ }
}
```

```
<?php
trait CanDive {
    public function dive(): void { /*...*/ }
}
```

*composition over inheritance*

Chicken.php

```
<?php
class Chicken {
    use CanFly; ←

    protected string $specieName;
    protected Peak $peakType;
}
```



I believe I  
CanFly

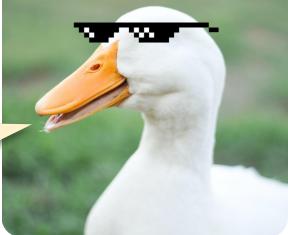
al componerlos con sus rasgos  
obtendrán la implementación de los  
métodos correspondientes

se puede componer con interfaces  
si el lenguaje no soporta rasgos

Duck.php

```
<?php
class Duck {
    use CanFly;
    use CanDive;

    protected string $specieName;
    protected Peak $peakType;
}
```



¡Soy la forma de  
vida definitiva!



```
<?php
class Ostrich {
    protected string $specieName;
    protected Peak $peakType;
}
```

# PILARES FUNDAMENTALES - COMPOSICIÓN

interfaces

Chicken.php

```
<?php  
class Chicken implements CanFly {  
    protected string $specieName;  
    protected Peak $peakType;  
  
    public function fly(): void {  
        /* how chickens fly */  
    }  
}
```



I believe I  
CanFly

```
CanFly.php  
<?php  
interface CanFly {  
    public function fly(): void;  
}  
  
CanDive.php  
<?php  
interface CanDive {  
    public function dive(): void;  
}
```

al componerlos con interfaces cada  
clase implementará su comportamiento

también se puede componer con clases  
abstractas o duck typing si el lenguaje  
tampoco soporta las interfaces

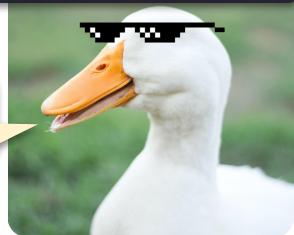


```
Ostrich.php  
<?php  
class Ostrich {  
    protected string $specieName;  
    protected Peak $peakType;  
}
```

Duck.php

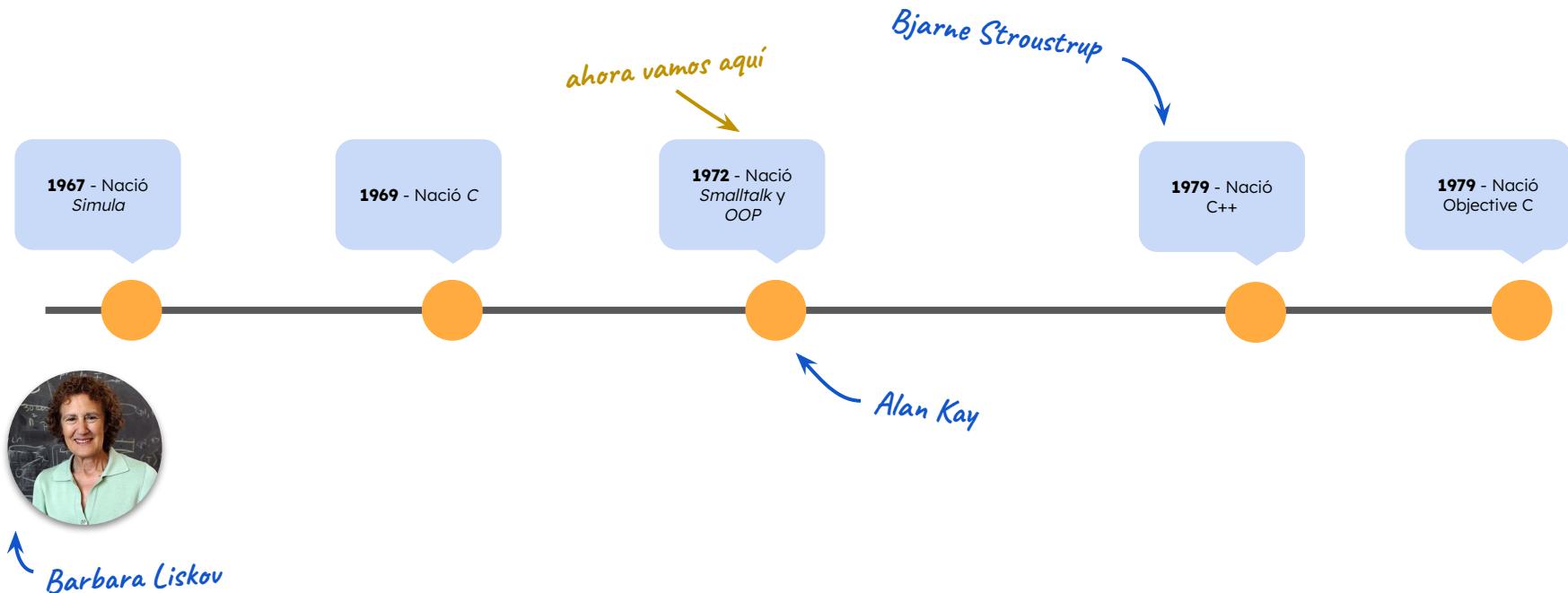
```
<?php  
class Duck implements CanFly, CanDive {  
    protected string $specieName;  
    protected Peak $peakType;  
  
    public function fly(): void {  
        /* how ducks fly */  
    }  
  
    public function dive(): void {  
        /* how ducks dive */  
    }  
}
```

¡Soy la forma de  
vida definitiva!



composition over  
inheritance

# LÍNEA TEMPORAL PARA OOP



# ESCUELA DE ALAN KAY

(SmallTalk, Ruby)

← anteriormente biólogo

← creador de OOP y Smalltalk

“ I made up the term ‘**object-oriented**’, and I can tell you I didn’t have C++ in mind.

Most of the practitioners of object-oriented programming spend much of their time dealing with class hierarchies, deriving specialized classes from general classes. In languages like Java (the ones that Alan and I hate, because of their confusion of strong type checking with object orientation) specialized classes have more data, more methods, and more overrides compared to their parent classes. The specialized classes are more tightly bound to implementation tricks, and are also the classes that are actually used by programmers.

This way of thinking unconsciously has led most object-oriented programmers to be highly resistant to adding or modifying methods of the root classes—in whatever language, the root class is often called “Object.” The intuition, I guess, is that modifying the root classes risks breaking everything—

# ESCUELA DE ALAN KAY

← anteriormente biólogo

← creador de OOP y Smalltalk

## Definición de Programación Orientada a Objetos Original:

1. **Todo es un objeto.** ← diferente definición de objeto
2. Los objetos se comunican enviando y recibiendo **mensajes**.
3. Los objetos tienen su propia memoria (estado).
4. Cada **objeto** es una **instancia** de una **clase**.
5. La clase contiene el comportamiento compartido para sus instancias.
6. Para evaluar una lista de programas, **el control se pasa al primer objeto y el resto se trata como su mensaje**.

# EL CONCEPTO DE OBJETO



Alan Kay

“ I'm sorry that I long ago coined the term “**objects**” for this topic because it **gets many people to focus on the lesser idea**. ”

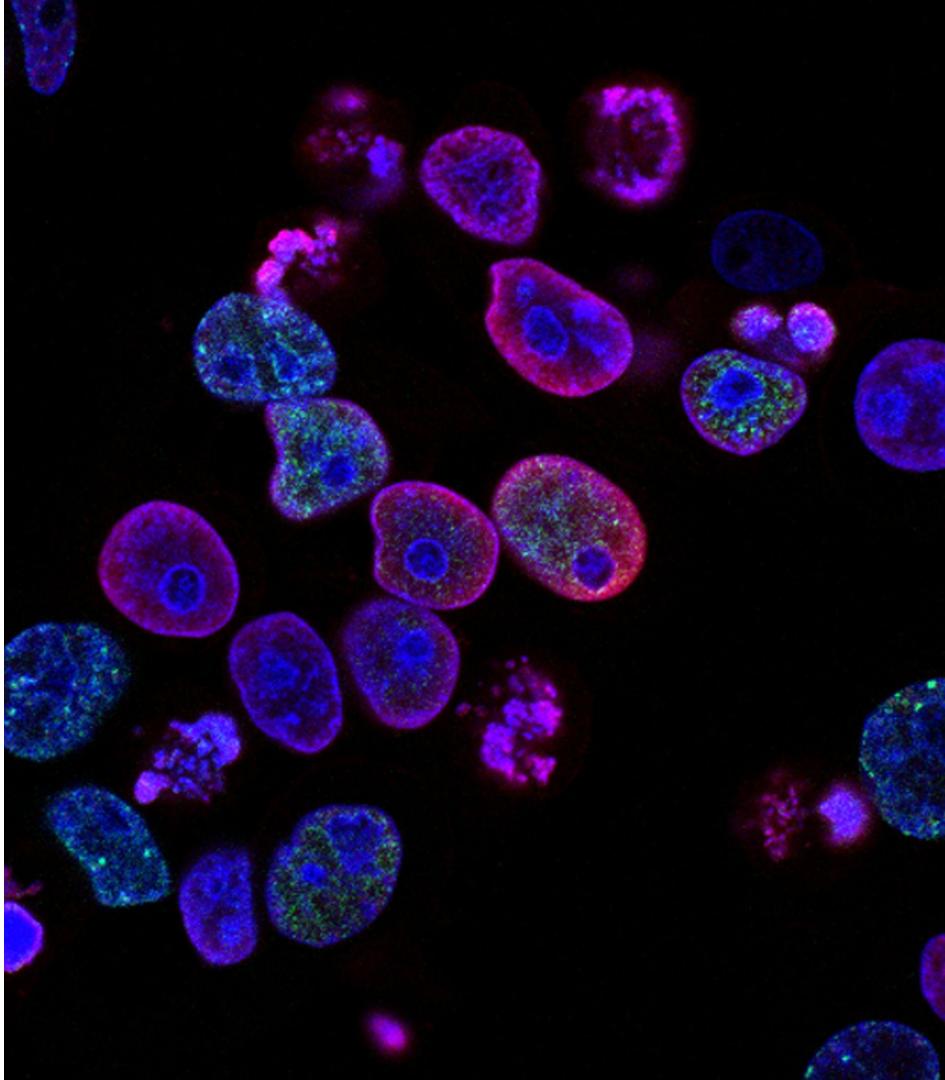


# METÁFORA BIOLÓGICA

- Herencia
- Taxonomías

# METÁFORA BIOLÓGICA

- Células
- Organismos



# ELEMENTOS DE LA ESCUELA DE ALAN KAY

→ *creador de OOP y Smalltalk*

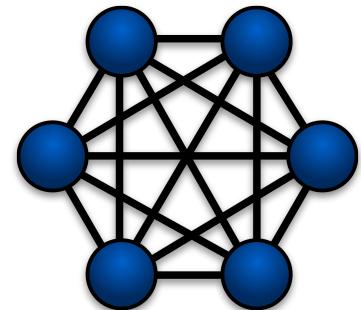
... el objetivo de la programación orientada a objetos es **no tener que preocuparse de lo que hay dentro de un objeto.**

Los objetos creados en máquinas diferentes y con lenguajes diferentes deberían poder comunicarse entre sí...



Red Neuronal

Alan Kay imaginó los objetos como máquinas **independientes** que actúan como **células biológicas**, operando en **su propio estado aislado**, y comunicándose mediante el **paso de mensajes**.

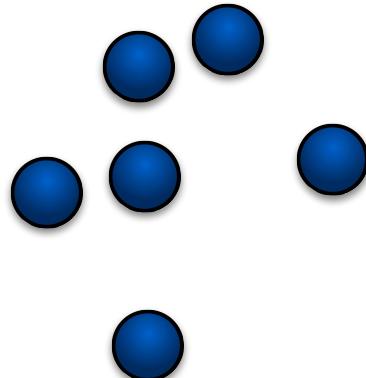


Red de Comunicaciones

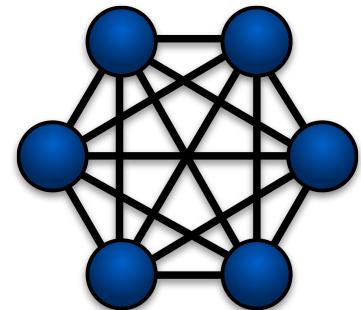
# ELEMENTOS DE LA ESCUELA DE ALAN KAY

creador de OOP y Smalltalk

la clave para crear sistemas *sostenible* y *escalable* es **diseñar cómo se comunican sus módulos**, más que cuáles deben ser sus propiedades y comportamientos internos.



En lugar de diseñar sistemas grandes diseñamos sistemas pequeños y combinamos estos pequeños sistemas para que resuelvan problemas de forma conjunta.



# ESCUELA DE ALAN KAY

creador de OOP y Smalltalk

Las disciplinas propuestas por Alan Kay son pasar mensajes, enlace tardío y encapsulación.



## Pasar Mensajes

comunicación  
mediante mensajes



## Enlace Tardío

resolución de llamadas en  
tiempo de ejecución



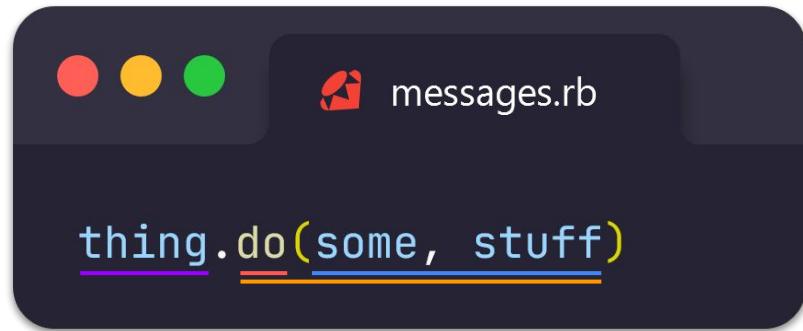
## Encapsulación

proteger el estado interno  
de una clase



Late binding

# PILARES FUNDAMENTALES - PASAR MENSAJES



messages.rb

```
thing.do(some, stuff)
```



messages.rb

```
thing.send(:do, some, stuff)
```

*con ruby queda claro que se  
está enviando un mensaje*



Destinatario



Mensaje



Nombre método

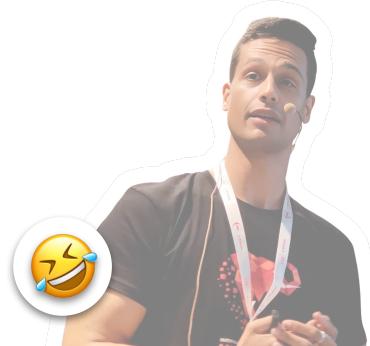


Argumentos





A ver como me las  
ingenio para explicar  
el **siguiente pilar...**



# PILARES FUNDAMENTALES - ENLACE TARDÍO

**Lenguaje Compilado:** Necesita **compilarse** a un **lenguaje** que pueda ser ejecutado por un ordenador.

*traducirse* ↗

*característica  
del lenguaje*

*ensamblador o  
código máquina*



```
#include <stdio.h>
int main() {
    printf("Hello, World!");
    return 0;
}
```

*creó el primer  
compilador*



*Grace Murray Hopper*



```
01100101 01110010 01100101 01110011
00100000 01100010 01100001 01110011
01110100 01100001 01101110 01110100
01100101 00100000 01100110 01110010
01101001 01101011 01101001 00100000
01110000 01101111 01110010 00100000
01110100 01110010 01100001 01100100
01110101 01100011 01101001 01110010
00100000 01100101 01110011 01110100
01101111
```

*agora sim entendo!*



# PILARES FUNDAMENTALES - ENLACE TARDÍO

**Lenguaje Interpretado:** Necesitan un **programa** que va traduciendo nuestro código línea a línea.

característica  
del lenguaje



# PILARES FUNDAMENTALES - ENLACE TARDÍO

característica  
del lenguaje

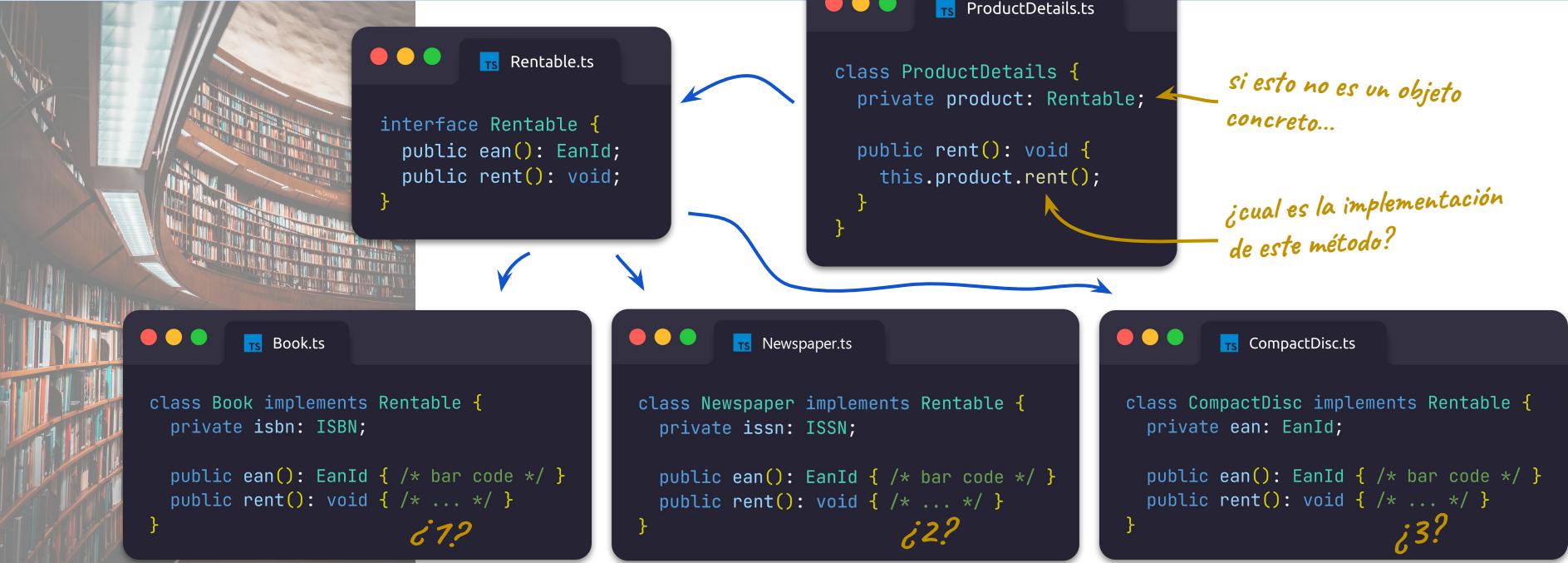
**Enlace Temprano:** El compilador resuelve las llamadas a métodos durante la compilación.



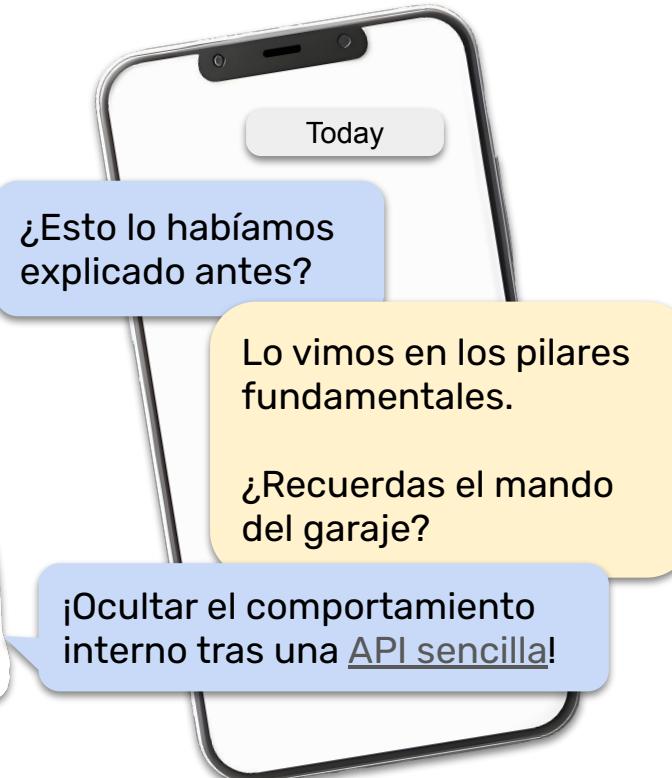
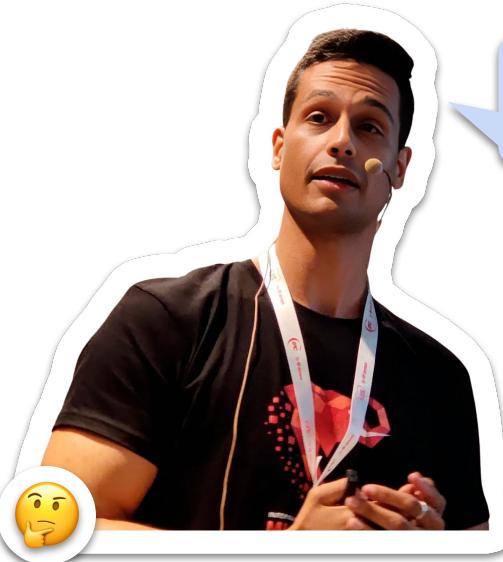
# PILARES FUNDAMENTALES - ENLACE TARDÍO

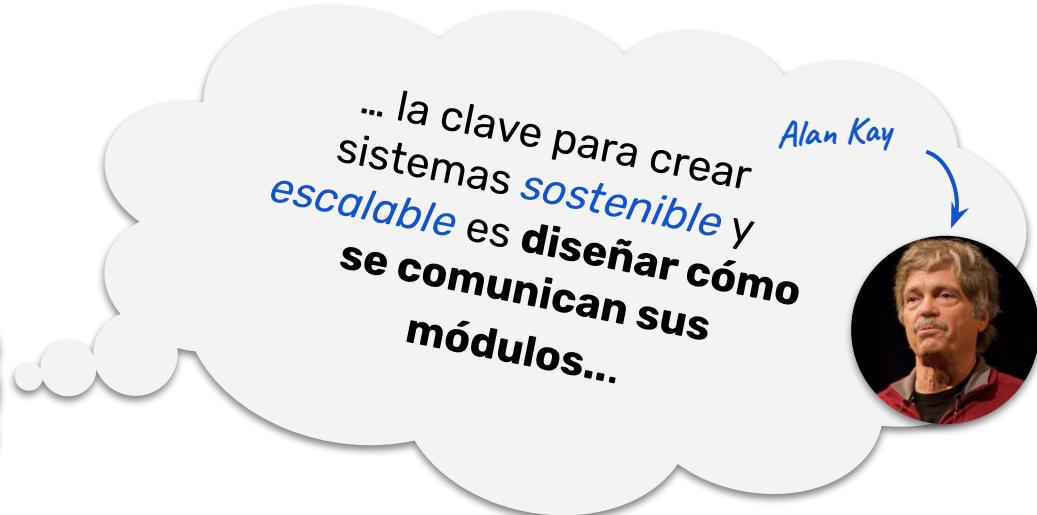
carácteristica  
del lenguaje

Determina la implementación específica de un método en tiempo de ejecución.



# PILARES FUNDAMENTALES - ENCAPSULACIÓN





# COHESIÓN Y ACOPLAMIENTO



Kent Beck

“

**Coupling** and **cohesion** are the laws of physics of software design.

”

[Tweet](#)

provienen de programación  
estructurada

“

**Coupling and cohesion are simply measures of the complexity of computer code, not from the perspective of the computers executing the programs but that of human beings trying to understand the code.** To understand any program, whether to create it or to correct it or to change it

”



Larry Constantine

# ACOPLAMIENTO

Es la forma y el nivel de interdependencia entre los distintos módulos de nuestro software.



bajo acoplamiento



una clase sólo tiene que cambiar si cambian sus requisitos, nevera porque cambien sus dependencias



imagina tener que cambiar de teléfono si se te estropea la batería

OOPS!

# COHESIÓN

Se refiere al grado en que los elementos de un mismo módulo permanecen juntos.



# DIFERENCIAS ENTRE COHESIÓN Y ACOPLAMIENTO

## COHESIÓN

1. Describe la **relación entre elementos de un mismo módulo** o contexto.
2. Está focalizada en **fortalecer la coherencia dentro del módulo** o contexto.
3. La cohesión debe ser **alta**.

## ACOPLAMIENTO

1. Describe la **relación entre diferentes módulos** o contextos.
2. Se focaliza en las **dependencias entre módulos** o contextos.
3. El acoplamiento debe ser **bajo**.

# COHESIÓN VS ACOPLAMIENTO

## COHESIÓN

- Normalmente fácil de encontrar.
- Fácil de conseguir.
- Ayuda a encontrar desacoplamiento.

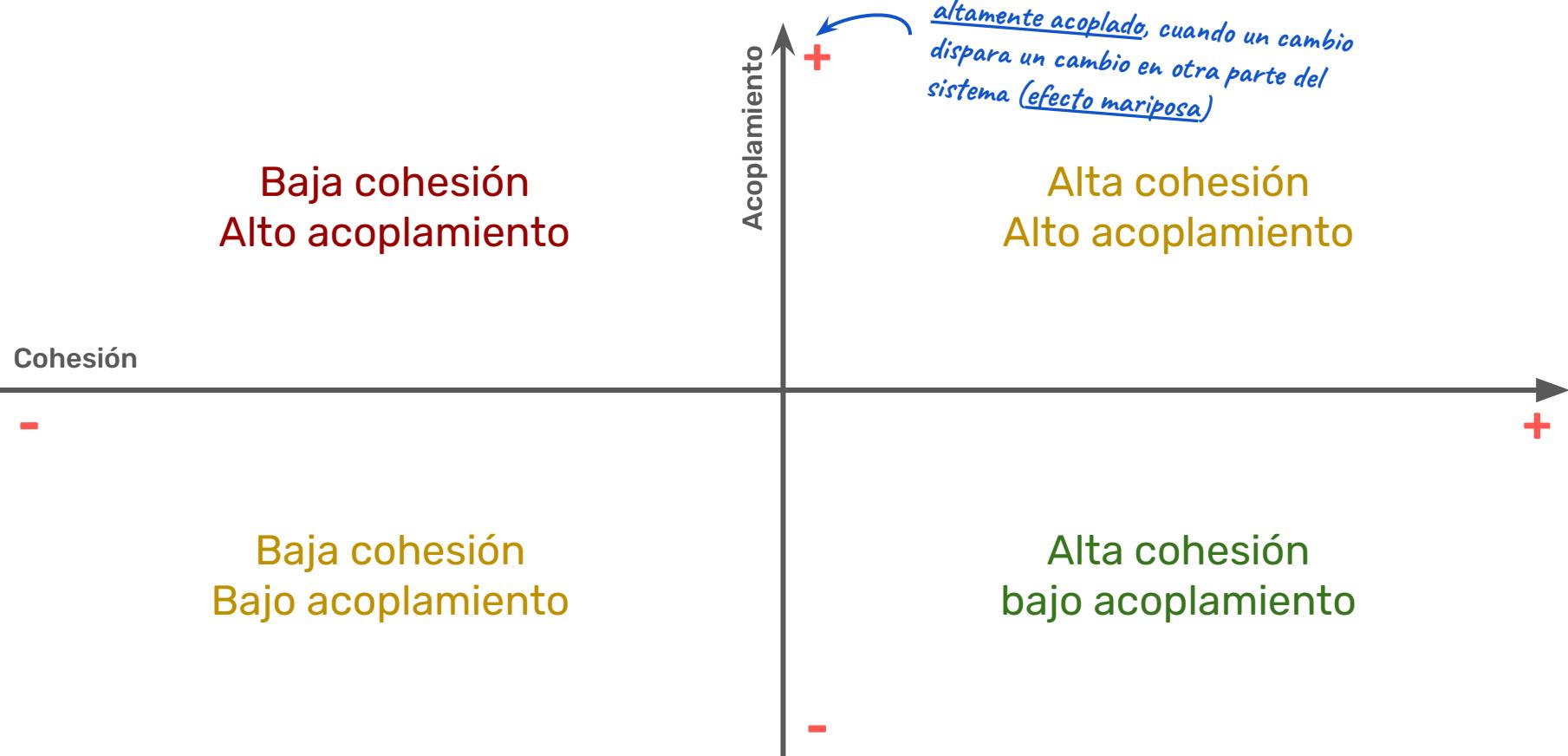
## ACOPLAMIENTO

- Puede ser difícil de encontrar.
- Puede ser realmente difícil y/o caro de eliminar.



Kent Beck

# COHESIÓN Y ACOPLAMIENTO



# RELACIONES

**Relaciones:** Forma que las clases interactúan entre sí.

- Asociación
- Dependencia
- Agregación
- Composición
- Herencia
- Implementación

mención  
previa

→ todavía hay más



# RELACIONES - ASOCIACIÓN

Dos o más objetos están relacionados entre sí.

Es genérica y **flexible**, soporta bidireccionalidad y cualquier **cardinalidad**.

*si un objeto cambia el otro no  
se ve afectado*



```
class Customer:  
    name: str  
    email: str  
    phone: str  
    1:1 → address: Address  
    1:n → orders: List[Order] }
```

*relaciones*

*1:1, 1:n, n:m...*



# RELACIONES - AGREGACIÓN

Un objeto está compuesto por otros objetos.

El **agregado** no puede existir sin sus **colaboradores**.



# RELACIONES - COMPOSICIÓN

Un objeto está compuesto de otros objetos.

El **objeto compuesto obtiene características** según sus **componentes**.

Diagram illustrating the Composition design pattern:

- Chicken.php** (Main Window):
  - Contains code for the `Chicken` class.
  - The `use CanFly;` statement is annotated with "agregado" (added) and "(usando rasgos)" (using traits).
  - The `CanFly` trait implementation is shown in a separate window.
  - A callout from the chicken says: ¡Si que puedo volar!
- CanFly.php** (Trait Window):
  - Shows the `CanFly` trait definition with its `fly()` method implementation.

Annotations in yellow:

- An arrow points from the `Chicken.php` window to the `CanFly` trait implementation with the text "implementación común de la característica" (common implementation of the characteristic).
- An arrow points from the `Chicken.php` window to the `use CanFly;` line with the text "agregado" (added).
- An arrow points from the `Chicken.php` window to the `CanFly` trait implementation with the text "(usando rasgos)" (using traits).
- An arrow points from the `CanFly` trait implementation back to the `Chicken.php` window with the text "característica" (characteristic).
- A blue callout bubble points to the `CanFly` trait implementation with the text "- rasgos" (traits), "- interfaces" (interfaces), and "- clases abstractas" (abstract classes).
- An arrow points from the `Chicken.php` window to the "rasgos" annotation with the text "objeto principal" (main object).

# RELACIONES - COMPOSICIÓN

Un objeto está compuesto de otros objetos.

El **objeto compuesto obtiene características** según sus **componentes**.

Diagrama que ilustra la composición en PHP:

Diagrama conceptual:

- objeto principal**: Representado por un cuadro superior que contiene el archivo `Chicken.php`.
- agregado**: Representado por un cuadro inferior que contiene el archivo `CanFly.php`.
- contrato**: Se refiere al contrato que el agregado cumple, mencionando que si puedes volar, expílanos como.

Diagrama de diseño:

- Característica (usando interfaces)**: Se apunta a la implementación de la interfaz `CanFly` en la clase `Chicken`.
- Implementación específica de la característica**: Se apunta a la implementación del método `fly()` en la clase `Chicken`.

Imagen y diálogo:



¡Si que puedo volar!

Código de `Chicken.php`:

```
<?php  
class Chicken implements CanFly {  
    protected string $specieName;  
    protected Peak $peakType;  
  
    public function fly(): void {  
        /* how chickens fly */  
    }  
}
```

Código de `CanFly.php`:

```
<?php  
interface CanFly {  
    public function fly(): void;  
}
```

# RELACIONES - DEPENDENCIA

objeto principal  
↓

El **cliente** requiere de sus **proveedores** para desempeñar su función.



- objetos secundarios
- dependencias
- colaboradores

cliente

```
<?php
class GuessTheNumberGame {
    private int $randomNumber;

    public function __constructor(RandomNumberGenerator $generator): void {
        $this.randomNumber = $generator.random();
    }

    public function play(int $guessedNumber): void {
        /* game logic */
    }
}
```

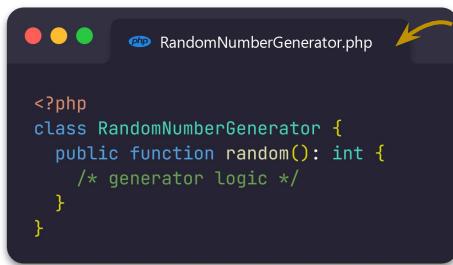
colaborador:  
el cliente lo necesita  
para cumplir su función



Guess the random number

# INYECCIÓN DE DEPENDENCIAS - EJEMPLO

Reduce la gestión de las dependencias en de un módulo.



```
<?php
class RandomNumberGenerator {
    public function random(): int {
        /* generator logic */
    }
}
```

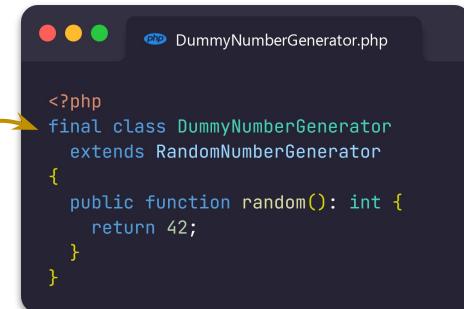


pedimos al colaborador  
en el constructor

```
<?php
class GuessTheNumberGame {
    private int $randomNumber;

    public function __constructor(RandomNumberGenerator $generator): void {
        $this.randomNumber = $generator.random();
    }

    public function play(int $guessedNumber): void {
        /* game logic */
    }
}
```



doble de prueba

extiende al colaborador

```
<?php
final class DummyNumberGenerator
    extends RandomNumberGenerator
{
    public function random(): int {
        return 42;
    }
}
```

# INYECCIÓN DE DEPENDENCIAS - BENEFICIOS

Reduce la gestión de las dependencias en de un módulo.

- Proporcionar al los objetos sus colaboradores.  
*en lugar de crearlos ellos mismos*
- Código más limpio.
- Simplifica la configuración.  
*contenedores de dependencias*
- Mejora los tests.  
*dobles de pruebas*

# INVERSIÓN DE DEPENDENCIAS - EJEMPLO

Inversión de  
Control (IoC)

"

No nos llames;  
nosotros te llamaremos

"



esa recruiter que  
nunca te llamó

```
RangeRandomNumberGenerator.php
```

```
<?php
final class RangeRandomNumberGenerator
    implements RandomNumberGenerator {
    private int $min;
    private int $max;

    public function random(): int {
        /* generator logic */
    }
}
```

colaborador  
implementa el contrato

```
RandomNumberGenerator.php
```

```
<?php
interface RandomNumberGenerator {
    public function generate(): int;
}
```

contrato

```
GuessTheNumberGame.php
```

```
<?php
class GuessTheNumberGame {
    private int $randomNumber;

    public function __constructor(RandomNumberGenerator $generator): void {
        $this->randomNumber = $generator->random();
    }

    public function play(int $guessedNumber): void {
        /* game logic */
    }
}
```

colaborador interfaz

doble de prueba  
implementa el contrato

```
DummyNumberGenerator.php
```

```
<?php
final class DummyNumberGenerator
    implements RandomNumberGenerator {
    public function random(): int {
        return 42;
    }
}
```

# INVERSIÓN DE DEPENDENCIAS - BENEFICIOS

Inversión de  
Control (IoC)

“  
No nos llames;  
nosotros te llamaremos



“  
esa recruiter que  
nunca te llamó

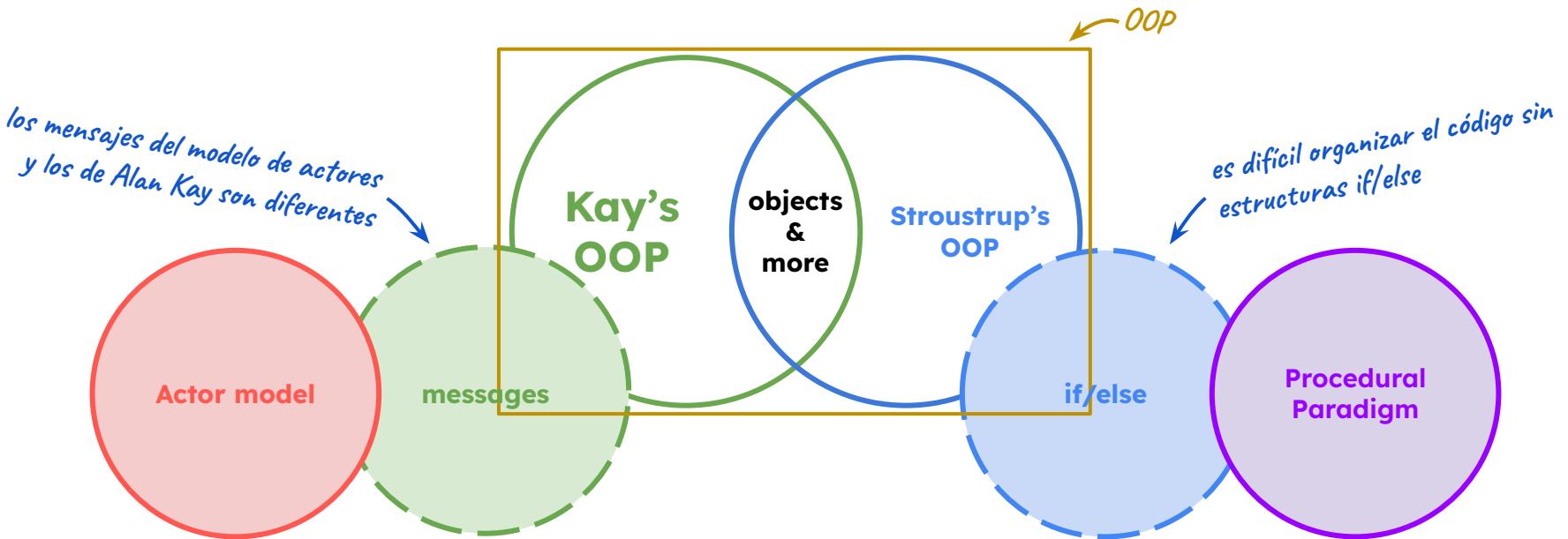
- Separar entre interfaces e implementaciones.
- Promueve el diseño modular.
- Reduce el acoplamiento.
- Mejora los tests.

↑  
dobles de pruebas  
más simples

←  
implementaciones  
intercambiables

# DIFERENCIAS ENTRE LAS ESCUELAS

Realmente no hay diferencias, se enfocan en puntos distintos del mismo paradigma, y ambos **conforman la Orientación a Objetos**.



disciplinas

# DISEÑO ORIENTADO A OBJETOS



# SEPARACIÓN DE INTERESES

Separation of Concerns (SoC)



E. W. Dijkstra



Fran Iglesias Gómez



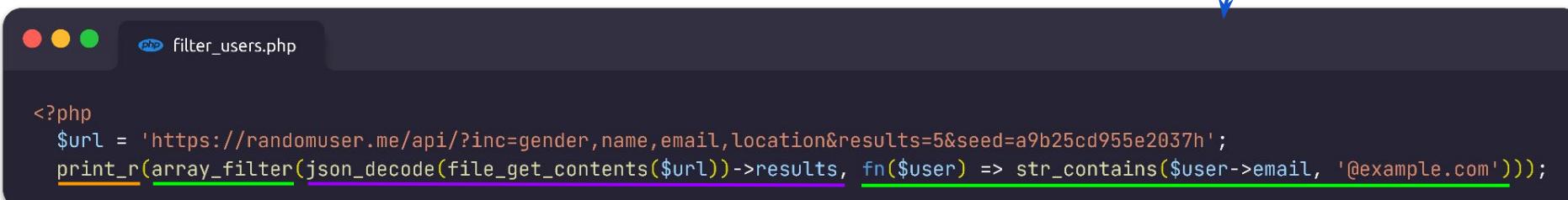
“Diferentes partes del programa se ocupan de intereses diferentes.”

”

# SEPARACIÓN DE INTERESES

Programa que se descarga un listado de usuarios de una fuente externa, se queda solo con los que cumplen un criterio concreto y luego muestra el resultado por pantalla.

*identificamos cada responsabilidad*



```
<?php
$url = 'https://randomuser.me/api/?inc=gender,name,email,location&results=5&seed=a9b25cd955e2037h';
print_r(array_filter(json_decode(file_get_contents($url))->results, fn($user) => str_contains($user->email, '@example.com')));
```



# SEPARACIÓN DE INTERESES

```
filter_users.php

<?php
$url = 'https://randomuser.me/api/?inc=gender,name,email,location&results=5&seed=a9b25cd955e2037h';
$users = json_decode(file_get_contents($url))->results;
$filterUsersByEmail = array_filter($response, fn($user):stdClass => str_contains($user->email, '@example.com'));
print_r($filterUsersByEmail);
```

separamos las responsabilidades en distintas  
líneas, como si fueran pasos atómicos



# SEPARACIÓN DE INTERESES

ahora los lugares de cambio para cada responsabilidad empiezan a estar más localizados

```
<?php  
  
function getUsers() {  
    $url = 'https://randomuser.me/...';  
    return json_decode(file_get_contents($url))->results;  
}  
  
function filterUsersByEmail($users) {  
    return array_filter(  
        $users,  
        fn($user) => str_contains($user->email, '@example.com')  
    );  
}  
  
function showUsers($users) {  
    print_r($users);  
}  
  
$users = getUsers();  
$filterUsersByEmail = filterUsersByEmail($users);  
showUsers($filterUsersByEmail);
```

acoplamiento con los datos

módulos = funciones



# SEPARACIÓN DE INTERESES

*Separation of Concerns (SoC)*

1. Identificar cada responsabilidad.
2. Separar responsabilidades en **distintas líneas** haciendo que cada operación sea atómica.  


*relacionado con la simetría del código*
3. Separamos las responsabilidades de en módulos individuales.



Fran Iglesias Gómez  
↓

# SEPARACIÓN DE INTERESES

Programa que suma todos los números que se le pasen como argumento para después mostrar el resultado por pantalla.

```
● ● ● sum_cli.py
import sys

if __name__ == '__main__':
    print(sum(map(int, sys.argv[1:])))
```

identificamos cada responsabilidad





Fran Iglesias Gómez

# SEPARACIÓN DE INTERESES



sum\_cli.py

```
import sys

if __name__ == '__main__':
    numbers_to_sum = map(int, sys.argv[1:])
    sum_result = sum(numbers_to_sum)
    print(sum_result)
```

separamos las responsabilidades en distintas  
líneas, como si fueran pasos atómicos



Entrada



Procesamiento



Salida



Fran Iglesias Gómez

# SEPARACIÓN DE INTERESES

```
● ○ ● sum_cli.py  
def get_numbers_to_sum():  
    return map(int, sys.argv[1:])  
  
def sum_numbers():  
    return sum(numbers_to_sum)  
  
def show_result():  
    print(sum_result)  
  
if __name__ == '__main__':  
    numbers_to_sum = get_numbers_to_sum()  
    sum_result = sum_numbers()  
    show_result()
```

módulos = funciones

# SOLID



Robert C. Martin  
a.k.a. Uncle Bob

**S**ingle **R**esponsibility **P**rinciple

**O**pen-**C**lose **P**rinciple

**L**iskov **S**ubstitution **P**rinciple

**I**nterface **S**egregation **P**rinciple

**D**ependency **I**nversion **P**rinciple



Barbara Liskov

# ¿QUÉ **NO** ES SOLID?

- *SOLID* **no** va sobre **diseño simple**, sino sobre “un mejor” diseño.
- *SOLID* **no** va sobre **entender OOP mejor**, sino sobre entender el diseño mejor.
- *SOLID* **no** es un **objetivo**, es una guía para intentar tener “un mejor” diseño.
- *SOLID* **no** son los **únicos “principios”** que pueden ayudarnos a eso, existen **muchos otros**.

como GRASP

# ¿QUÉ ES SOLID?

Limitar el impacto de los **cambios**, intentando que los cambios sean **fáciles de conseguir**.



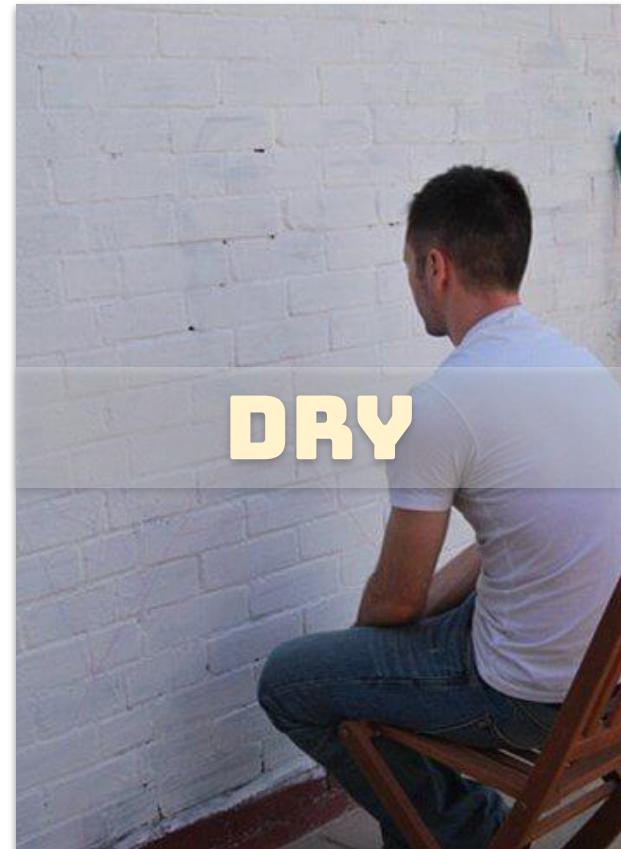
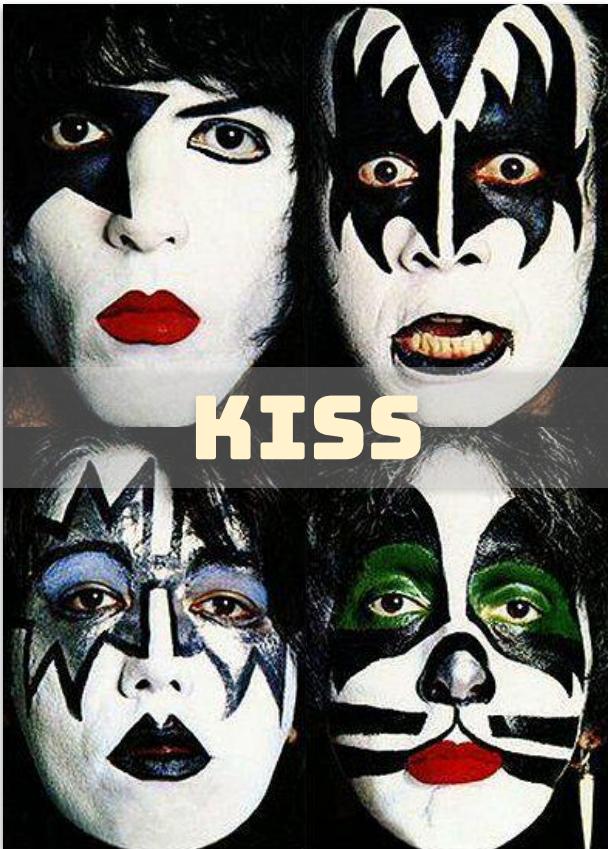
Robert C. Martin  
a.k.a. Uncle Bob

Una manera de manejar las dependencias, **invirtiéndolas**, de forma que el código que generemos sea sostenible y prevenga fragilidad, no escalabilidad y rigidez.

ayuda a gestionar la complejidad y tomar el control del coste del cambio teniendo en cuenta cohesión y acoplamiento

impacto de las dependencias

# PRÁCTICAS PARA SIMPLIFICAR EL CÓDIGO





# KISS

*Keep It Simple, Stupid*

La mayoría de los sistemas **funcionan mejor si se simplifican** que si se complican; por tanto, la **simplicidad debe ser un objetivo clave en el diseño**.



*Clarence Leonard ("Kelly") Johnson*

KISS

You Ain't Gonna Need It → **YAGNI**

Implementa las cosas **cuando las necesites**, nunca cuando creas que las necesitas.



← Ron Jeffries

mencionado por primera vez en la primera edición del libro eXtreme Programming, en la segunda edición sustituido por "incremental design"





**DRY** ↗ *Don't Repeat Yourself*

✓ mencionado en el libro  
*Pragmatic Programmers*

↗ Andrew Hunt



Cada pieza de conocimiento debe tener una representación inequívoca y autorizada en un sistema. La idea es **evitar duplicidad de código**.

en contraposición existe la ley de las tres repeticiones o la triangulación, para evitar abstracciones prematuras



↗  
Dave Thomas

# LEY DE DEMETER

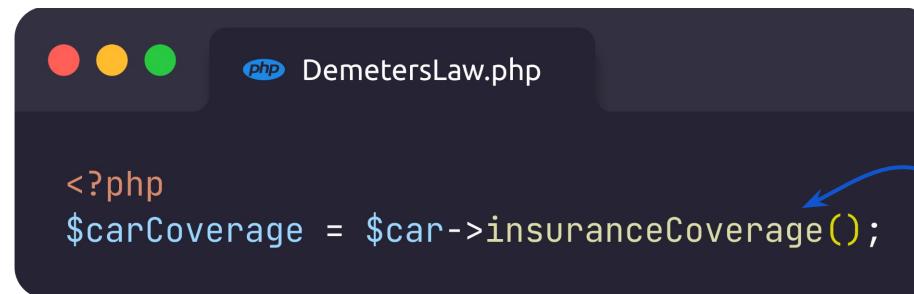
*principio de menor conocimiento*

Un módulo no debe conocer las entrañas de los objetos que manipula.



DemetersLaw.php

```
<?php  
$carCoverage = $car->insurance->policy->coverage();
```



DemetersLaw.php

```
<?php  
$carCoverage = $car->insuranceCoverage();
```



*no hables con extraños*

*Tell, Don't Ask*



OOP es simple, si  
lo explicas bien



← Khaby Lame



# CONCLUSIONES

# CONCLUSIONES

- La **OOP** fue pensada desde **puntos de vista diferentes**, con el tiempo el término **ha evolucionado** y se han nutrido entre sí, llegando a ser lo que hoy en día conocemos como OOP.
- Tener un **punto de vista histórico**, de lo que queremos aprender, siempre nos da una **perspectiva diferente**.
- La visión de **Alan Kay** sobre la OOP se parecía de alguna forma al **Paradigma de Programación Funcional**.
- Las **herramientas** sin las disciplinas funcionan, aunque **funcionan mejor junto a las disciplinas**.

# **ERRORES A LA HORA DE ENSEÑAR OOP**

- **Solo explicar las herramientas**, no las disciplinas y **que relación tienen entre sí**.
- No tener en cuenta que las palabras **Orientación y Objetos**, son dos **terminos o visiones distintas**.
- No tener **contexto histórico**.
- **Malinterpretar las metáforas** de la orientación a objetos.
- Ignorar la **visión de Alan Kay** sobre OOP.
- No apoyar la enseñanza de OOP con **principios y prácticas**.
- No dar suficiente importancia a las **relaciones entre objetos**.
- No enfocar la enseñanza en responder al “**¿Por qué?**” y al “**¿Cuándo?**”.

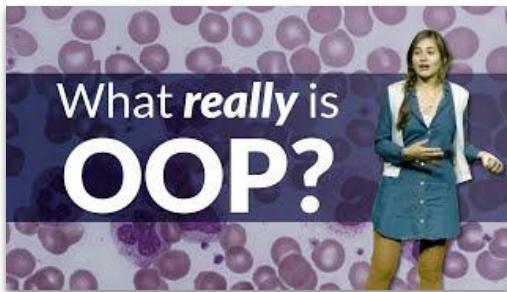
# ¿QUÉ NO ES OOP?

- OOP **NO** es **MODELAR EL MUNDO REAL**.
- OOP **NO** es una **MANERA DE PROGRAMAR SIMILAR A COMO PENSAMOS**.



# RECURSOS

# OBJECT ORIENTED PROGRAMMING



[Object Oriented Programming is  
not what I thought - Talk by  
Anjana Vakil](#)



[Object Oriented Programming –  
MIT](#)



[OOP SOLID - "Uncle" Bob Martin](#)

# OBJECT ORIENTED PROGRAMMING THINKING

## TWO BIG SCHOOLS OF OBJECT-ORIENTED PROGRAMMING

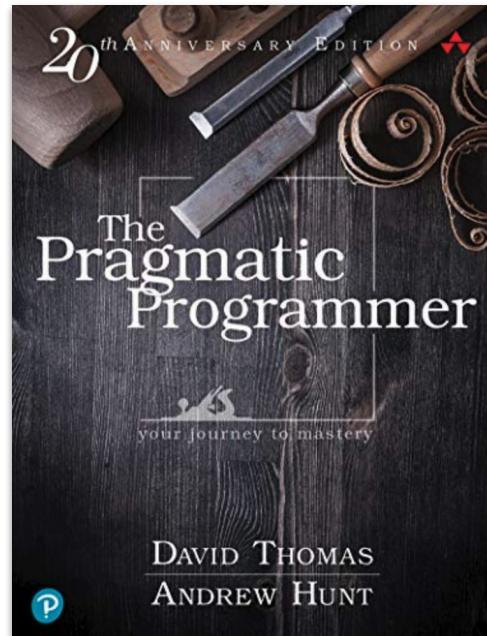
TL;DR There is no one "canonical" definition of OOP. There are at least two of them. From my POV there are two big definitions, which deserves separate names. I named them after biggest advocates. Those two definition have some differences and some commonalities.

I was about to get into another argue on OOP on the internet. I decided to do some research on the subject. I found more than one definition of OOP. Some of the definitions are variations of the same "tune", some are useless. On my opinion there are two main directions of thoughts in this domain, which from two schools of OOP. School as in school of thought. The same way as there were philosophical schools in ancient Greece. The same way there are two schools of OOP:

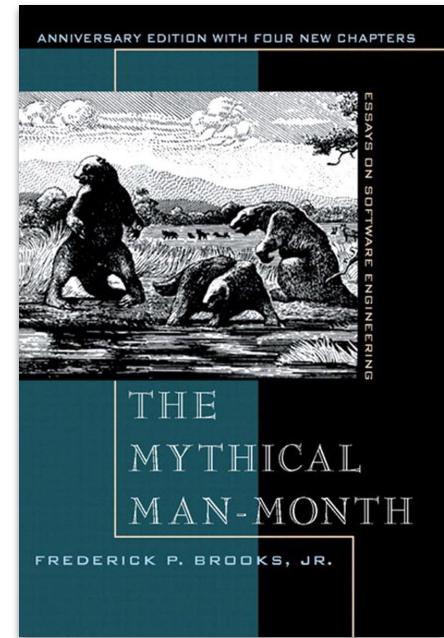
- Alan Kay's school or message focused
- Bjarne Stroustrup's school or class focused or "PolymorphismEncapsulationInheritance" in terms of c2

I must emphasize that this article is not about what is best way to do OOP, because such discussion will lead to holy war. Instead purpose of this article is to help to recognize the fact that there are two ways to think about OOP. I believe this will help to build more constructive discussions about OOP. Next time you will start discussion on OOP or OOP vs make sure you know which school of OOP you are talking about.

[Two big schools of object-oriented programming](#)



[The Pragmatic Programmer:  
Your Journey To Mastery](#)



[Mythical Man-Month, The:  
Essays on Software  
Engineering](#)

# COHESIÓN Y ACOPLAMIENTO



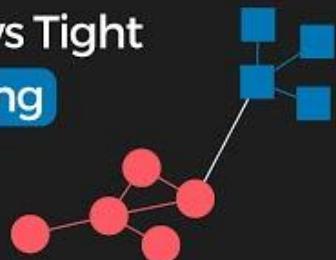
"I will contend that conceptual integrity is the most important consideration in system design. It is better to have a system omit certain anomalous features and improvements; but to reflect one set of design ideas, than to have one that contains many independent and uncoordinated ideas."

– Fred Brooks  
The Mythical Man-Month



[Coupling and Cohesion](#)

Loose vs Tight  
Coupling



[Understanding Coupling and Cohesion](#)

[Loose vs Tight Coupling](#)

# DISEÑO

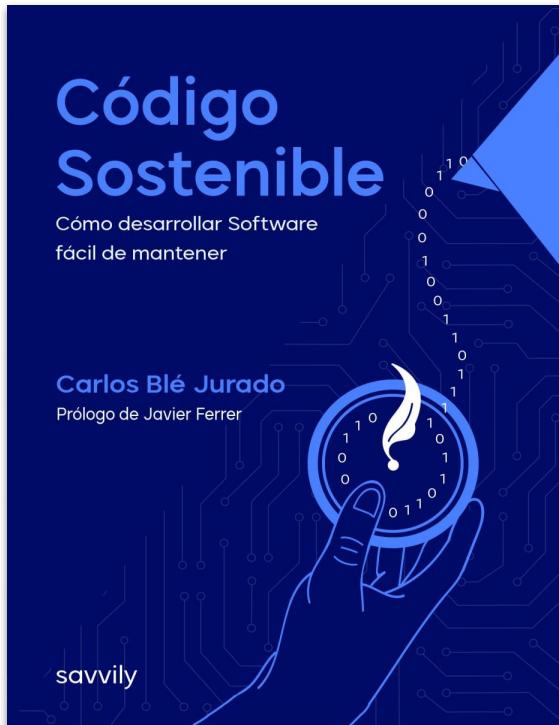


[Diseño en zapatillas - Fran Iglesias](#)

[Las 5 reglas del CÓDIGO SOSTENIBLE](#)

[Serie Programador Senior](#)

# DISEÑO



[Código Sostenible](#)

## Principios de diseño

Los principios de diseño nos proporcionan criterios para evaluar nuestras decisiones de diseño de software, ayudándonos a diseñar sistemas cohesivos, con bajo acoplamiento y fáciles de entender y hacer evolucionar.

Artículos etiquetados con design-principles

- Métodos largos (25 Jan 2022)
- Principio de separación de intereses (28 Jul 2021)
- De abstracciones y duplicaciones (23 Jan 2021)
- 99 bottles of OOP. Sandi Metz. (07 Jan 2021)
- Practical Object Oriented Design. Sandi Metz. (02 Jan 2021)
- Trabajar con legacy y entender el dominio 3 (27 Sep 2020)
- Trabajar con legacy y entender el dominio 2 (22 Sep 2020)
- Trabajar con legacy y entender el dominio (19 Sep 2020)
- Más allá de SOLID, los principios olvidados (12 Sep 2020)
- Más allá de SOLID, los consejos prácticos (30 Aug 2020)
- Más allá de SOLID, los cimientos (15 Aug 2020)

[Blog de Fran Iglesias](#)

# DISEÑO



The image shows the header of the LeanMind website. The logo 'LeanMind' with the tagline 'EFFECTIVE DEVELOPMENT' is on the left. A navigation bar with links 'ACERCA', 'CASOS DE ÉXITO', 'EMPLEO', 'FORMACIÓN', 'BLOG' (which is highlighted in blue), 'CONTACTO', and 'EN' follows. Below the header is a blog post thumbnail. The thumbnail has a dark blue background with white text. It features the word 'BDD' and 'In house' on the left, and some binary code ('1', '0', '0', '0', '1') on the right. A small orange button labeled 'MÁS INFORMACIÓN' is at the bottom right of the thumbnail.

BDD  
In house

1 0 0 0 1

MÁS INFORMACIÓN

## CRC Cards (Class Responsibility Collaborator)

Por [Manuel Pérez](#)

Quería compartirles esta herramienta que vimos en una kata con Codesai y Audience que me parece una buena alternativa y a la vez complemento de los clásicos UML a la hora de poner sobre la mesa lo que tenemos en la cabeza cuando vamos a diseñar.

### ¿QUÉ SON LAS TARJETAS CRC?

Las tarjetas CRC (Class Responsibility Collaborator) es una actividad en grupo de modelado orientado a objetos en la cual el equipo puede manifestar y debatir ideas acerca del diseño de un sistema. Hace especial énfasis en la simplicidad, comunicación y límites de un sistema. Se suele utilizar en las primeras fases del desarrollo de una historia como una

[CRC cards](#)

# CHARLAS SIMILARES



Lemi Orhan Ergin  
thank you for the  
inspiration

# LEARNING THROUGH KATAS

by Emmanuel Valverde

**Level 1 : Emerging design**

The objective of this level is to become familiar with all the basic concepts of Test-Driven Development and to create a solid foundation that can then be used.

This is why it starts with a progression from conditionals, algorithms to object-oriented programming.

Concepts to learn:	Katas to work:
Test desiderata	FizzBuzz
Test-Driven Development Cycle	Anagram
Fake it until you make it	Manhattan distance
3 Repetition rule (Triangulation)	Leap year
Baby steps	Roman calculator
Transformation priority premise (TPP)	Employee Report
Parallel Change Refactoring	Rock paper scissors
Test parameterization	Code syntax
Test fixture	Roman numeral
Classicist TDD	Bowling
Example mapping	Mastermind
To-Do List Technique	Stack
	String calculator
	Simple mars rover
	Christmas tree
	Potter kata

Programming knowledge:	Interesting restrictions:
1 Conditionals	Test commit or Revert
2 Algorithms	No mouse
3 Object Oriented Programming	Object calisthenic
4 Immutability	No else statements
5 Public vs Publish API	

**Level 2 : Working with collaborators and Test doubles - How to design**

The objective of this level is to become familiar with the concepts of object-oriented design, which we will need to create our own designs and to be able to model, i.e. to invest a little bit that the design emerges and try to do it in another way to make sure that we understand how to do the tests.

That's why you start with a progression of: Object Oriented, Double Testing and Design Principles.

Concepts to learn:	Katas to work:
4 principles of simple design	Text processing
Mock it if you own it	Racing car
Test doubles	Bags
Law of demeter	ATM machine
KISS	RPG
DRY	Tell don't ask
YAGNI	Print date
Object calisthenic	Tic-Tac-Toe
Separation of Concerns	Smart fridge
GRASP / SOLID	Guess the random number
Mutant testing	
Connascence	
CRC Cards	

Programming knowledge:	Interesting restrictions:
1 4 Pilaras of OOP	Test commit or Revert
2 OOP Roles & Relationships	No mouse
3 Design principles	Object calisthenic
	No primitives
	No else statements
	Baby steps

hay mas niveles!

→ Learning Through Katas

<b>PILAR</b>	<b>DESCRIPCIÓN</b>	<b>CARACTERÍSTICAS</b>
<b>Abstracción</b>	Representación de las características esenciales de un objeto sin incluir los detalles específicos irrelevantes en nuestro contexto.	<ul style="list-style-type: none"> <li>- Modelado de objetos</li> <li>- Ocultamiento de información</li> <li>- Simplificación</li> <li>- Definición de interfaces</li> </ul>
<b>Encapsulación</b>	Ocultamos el estado y el comportamiento interno de nuestros objetos para reducir la complejidad y exponemos un API simple de usar.	<ul style="list-style-type: none"> <li>- Protección de datos</li> <li>- Ocultamiento de implementación</li> <li>- Control de acceso</li> <li>- Facilita el mantenimiento y evolución</li> </ul>
<b>Herencia</b>	Mecanismo que permite que un clase hija herede atributos y métodos de una clase padre.	<ul style="list-style-type: none"> <li>- Reutilización de código</li> <li>- Relación jerárquica</li> <li>- Polimorfismo</li> <li>- Herencia de comportamiento y estructura</li> <li>- Acoplamiento fuerte</li> </ul>
<b>Polimorfismo</b>	Capacidad de un objeto para tomar diferentes formas y comportarse de diferentes maneras según la necesidad.	<ul style="list-style-type: none"> <li>- Intercambiabilidad de objetos</li> <li>- Flexibilidad</li> <li>- Extensibilidad</li> </ul>

<b>RELACIÓN</b>	<b>DESCRIPCIÓN</b>	<b>CARACTERÍSTICAS</b>
<b>Asociación</b>	Relación entre dos objetos donde uno tiene una referencia al otro de alguna manera.	<ul style="list-style-type: none"> <li>- Dependencia flexible</li> <li>- Acoplamiento débil</li> <li>- Objetos independientes</li> <li>- Sin relación jerárquica</li> <li>- Pueden existir sin el otro objeto</li> </ul>
<b>Dependencia</b>	Relación en la que un objeto utiliza o depende de otro objeto para llevar a cabo una operación.	<ul style="list-style-type: none"> <li>- Dependencia flexible</li> <li>- Acoplamiento débil</li> <li>- Objetos independientes</li> <li>- No hay relación jerárquica</li> <li>- Uno depende del otro</li> </ul>
<b>Herencia</b>	Relación en la que una clase hereda características (atributos y métodos) de otra clase.	<ul style="list-style-type: none"> <li>- Reutilización de código</li> <li>- Relación jerárquica</li> <li>- Polimorfismo</li> <li>- Herencia de comportamiento y estructura</li> <li>- Acoplamiento fuerte</li> </ul>
<b>Implementación</b>	Relación en la que una clase implementa una interfaz y proporciona una implementación concreta de sus métodos.	<ul style="list-style-type: none"> <li>- Cumplimiento de contrato</li> <li>- Polimorfismo</li> <li>- Comportamiento específico</li> <li>- Acoplamiento flexible</li> <li>- Reutilización de interfaces</li> </ul>
<b>Agregación</b>	Relación en la que un objeto contiene una colección de otros objetos, pero los objetos relacionados pueden existir independientemente del objeto principal.	<ul style="list-style-type: none"> <li>- Relación "todo-parte"</li> <li>- Objetos independientes</li> <li>- Asociación flexible</li> <li>- Puede haber múltiples partes</li> <li>- Ciclo de vida independiente</li> <li>- Acoplamiento débil</li> </ul>
<b>Composición</b>	Relación en la que un objeto contiene otros objetos como componentes, pero los objetos componentes no pueden existir sin el objeto principal.	<ul style="list-style-type: none"> <li>- Relación "todo-parte"</li> <li>- Objetos dependientes</li> <li>- Asociación fuerte</li> <li>- Una sola parte</li> <li>- Ciclo de vida acoplado</li> <li>- Acoplamiento fuerte</li> </ul>

# PREGUNTAS



THANK YOU

GRACIAS

