RE-APRENDIENDO OOP

Esta vez, bien

Antonio González Gea

Emmanuel Valverde Ramos

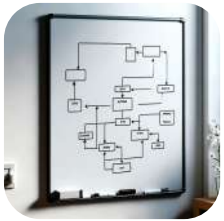
ÍNDICE

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

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.

index.sql

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

Declarativo

describimos el resultado esperado, sin detallar los pasos a seguir

=

index.php

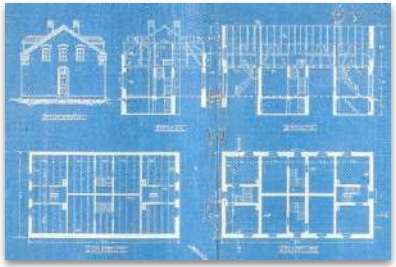
```php
<?php

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.

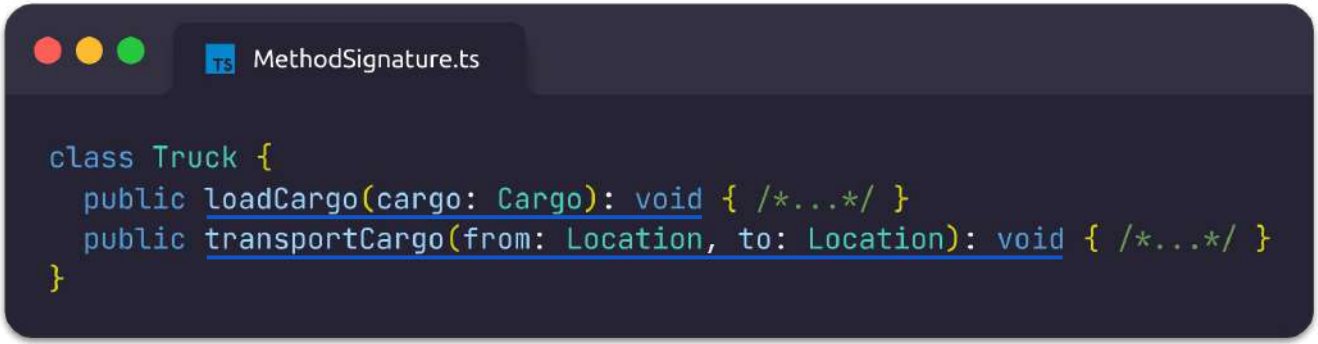
- cerrar la puerta
- encender la calefacción

Atributos

Son el estado y las características de nuestro objeto.

- metros cuadrados
- número de ventanas
- dirección

ESTRUCTURA BÁSICA DE LA OOP



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

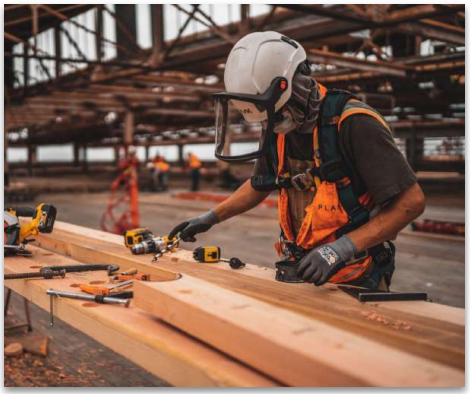
OOP

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

el enfoque del diseño evoluciona junto a los problemas

Diseñando Software Orientado a Objetos

se enfoca en los principios de diseño, filosofía y valores

HERRAMIENTAS Y 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.

Leyes de Lehman de la evolución del software


CONTEXTO HISTORICO

ESCUELAS DE OOP

Bjarne Stroustrup:

Abstraction!

Inheritance!

Polymorphism!

creador de C++

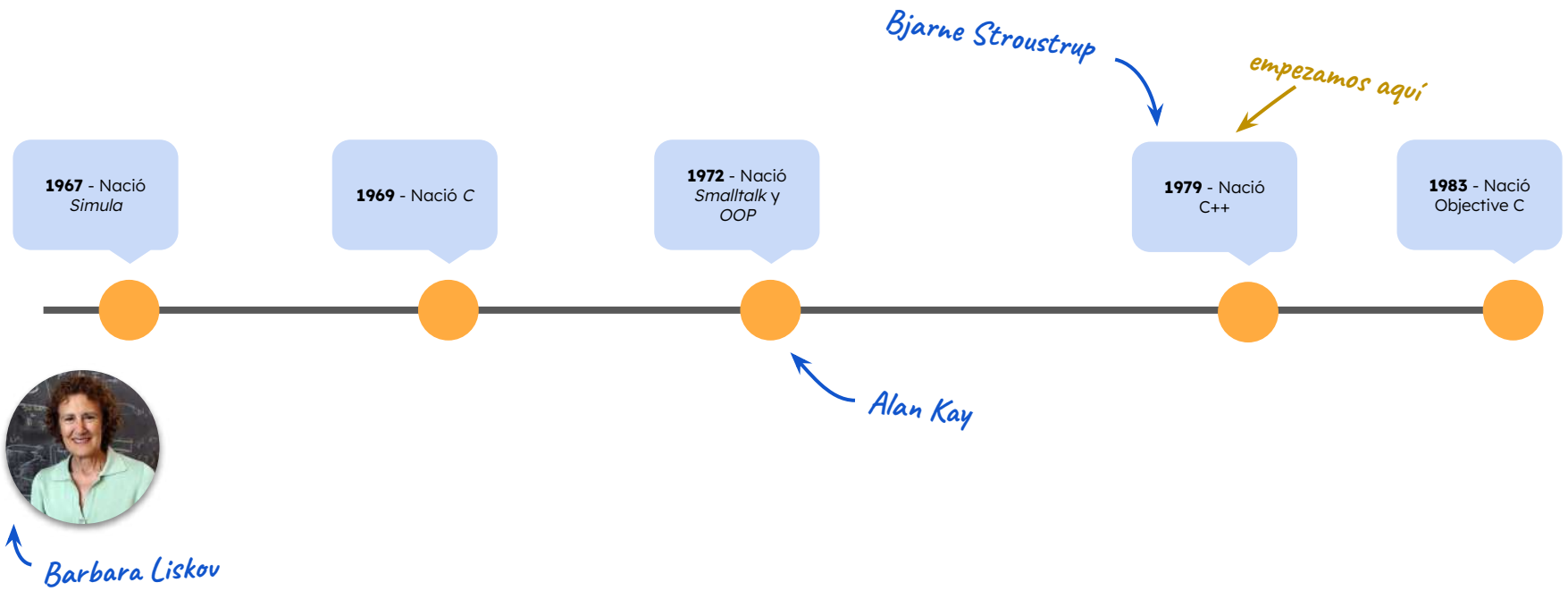
Alan Kay:

Everything is an object!

Objects communicate by sending and receiving messages!

creador de OOP y Smalltalk

LÍNEA TEMPORAL PARA OOP

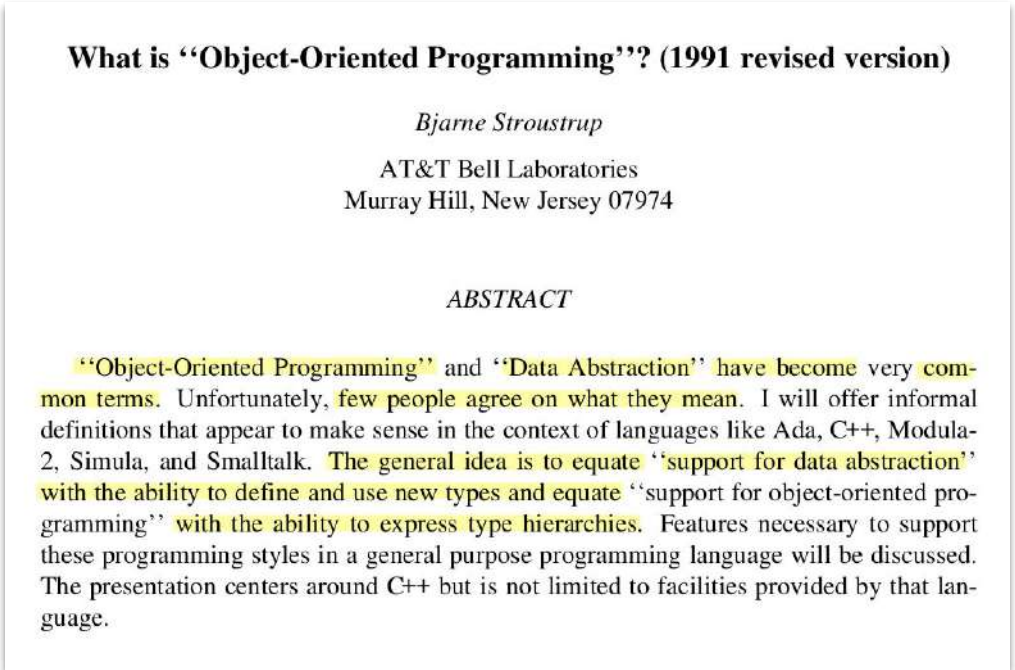


ESCUELA DE BJARNE STROUSTRUP

(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

http://www.stroustrup.com/oopsla.pdf

ESCUELA DE BJARNE STROUSTRUP

creador de C++

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

no estaba en la visión original de Bjarne

Abstracción

Encapsulación

Herencia

Polimorfismo

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.

concepto

Camión

contexto A

Transportista

Correos

representación A

```
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 { /*...*/ }
}
```

contexto B

Aseguradora

Allianz

representación B

```
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 { /*...*/ }
}
```

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

PILARES FUNDAMENTALES - ENCAPSULACIÓN

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

nos referimos a interfaces públicas

no confundir con el keyword interface

Puerta de un Garaje

DoorRemote.ts

```ts
class DoorRemote {
  private privateKey: string;
  private frequency: number;

  public openDoor(): void { /*...*/ }
}
```

complejidad interna oculta

fácil de usar

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

usando interfaces, clases abstractas o duck typing

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

interfaz uniforme

Rentable.ts

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

ProductDetails.ts

```ts
class ProductDetails {
  private product: Rentable;

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

código cliente

implementaciones

Book.ts

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

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

Newspaper.ts

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

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

CompactDisc.ts

```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.

herencia mal utilizada

Bird.php

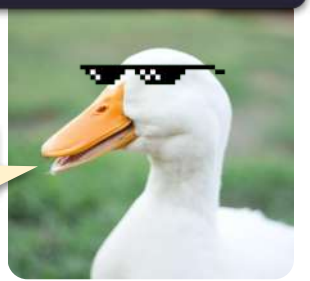
```php
<?php
class Bird {
  protected string $specieName;
  protected Peak $peakType;

  public function fly(): void { /*...*/ }
  public function dive(): void { /*...*/ }
}
```

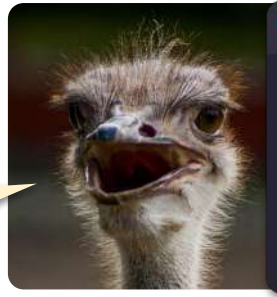
Chicken.php

```php
<?php
class Chicken extends Bird {
  protected string $specieName;
  protected Peak $peakType;

  public function fly(): void { /* can fly */ }
  public function dive(): void { /* can not dive */ }
}
```

Duck.php

```php
<?php
class Duck extends Bird {
  protected string $specieName;
  protected Peak $peakType;

  public function fly(): void { /* can fly */ }
  public function dive(): void { /* can dive */ }
}
```





Ostrich.php

```php
<?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.

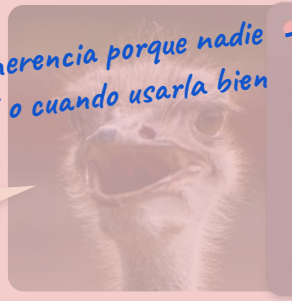
ambigüedad

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**".

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

¿Eh?



Ostrich.php

```php
<?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 */ }
}
```

la composición sobre herencia, el ISP y el SRP nos pueden ayudar

PILARES FUNDAMENTALES - HERENCIA COMPOSICIÓN

composition over inheritance

rasgos

CanFly.php

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

CanDive.php

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

Chicken.php

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

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

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

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

Ostrich.php

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

PILARES FUNDAMENTALES - HERENCIA COMPOSICIÓN

composition over inheritance

interfaces

CanFly.php

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

CanDive.php

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

Chicken.php

```php
<?php
class Chicken implements CanFly {
  protected string $specieName;
  protected Peak $peakType;

  public function fly(): void {
    /* how chickens fly */
  }
}
```

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

Duck.php

```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 */
  }
}
```

Ostrich.php

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

I believe I CanFly

¡Soy la forma de vida definitiva!

LÍNEA TEMPORAL PARA OOP

Bjarne Stroustrup

ahora vamos aquí

1967 - Nació *Simula*

1969 - Nació *C*

1972 - Nació *Smalltalk* y *OOP*

1979 - Nació C++

1983 - Nació Objective C

Barbara Liskov

Alan Kay

ESCUELA DE ALAN KAY

anteriormente biologo

creador de OOP y Smalltalk

(SmallTalk, Ruby)

> “ 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—

“Points of View - A Tribute to Alan Kay”

ESCUELA DE ALAN KAY

anteriormente biologo

creador de OOP y Smalltalk

Definición de Programación **Orientada a Objetos Original**:

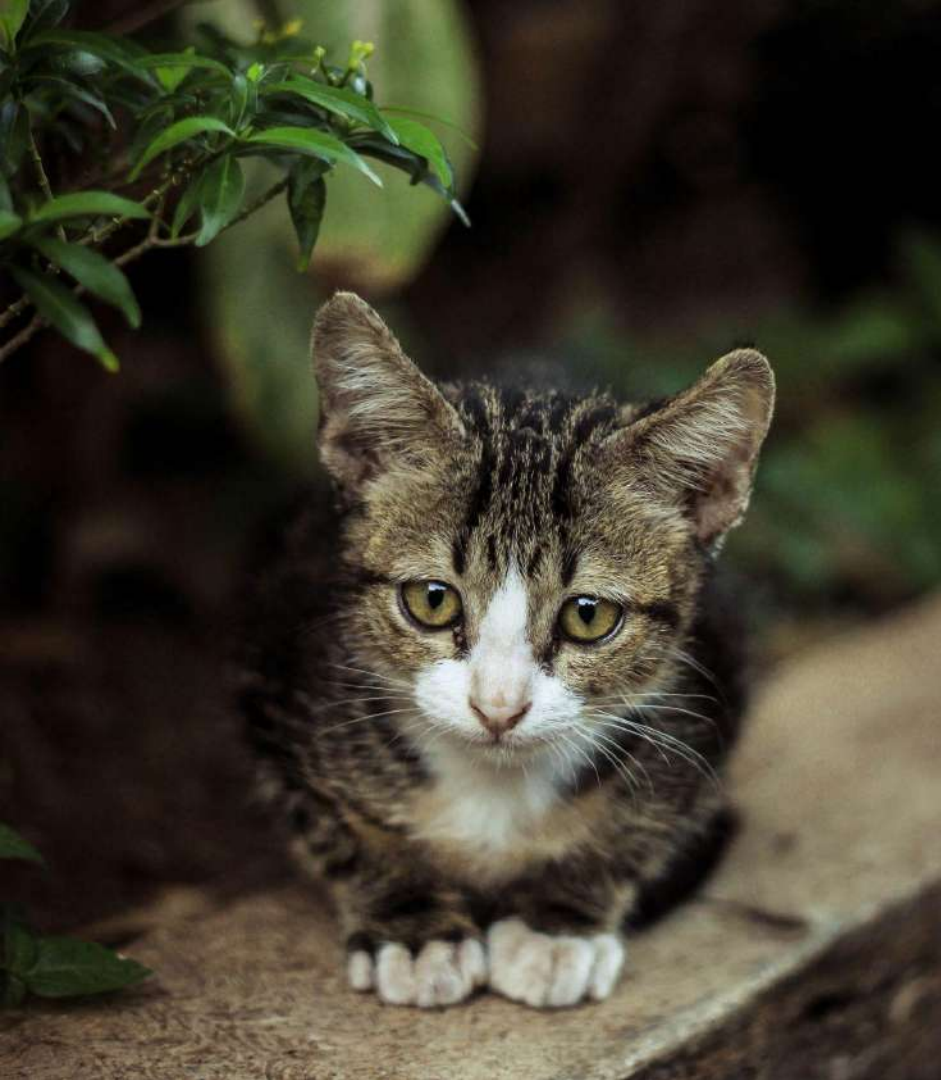
1. **Todo es un objeto**. diferente definición de objeto
2. Los objetos se comunican enviando y recibiendo **mensajes**.
3. Los objetos tienen su propia memoria (estado).
4. Cada **objeto** es una **instancia** de una **clase**.
5. La clase contiene el comportamiento compartido para sus instancias.
6. 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**.

Alan Kay did not invent objects

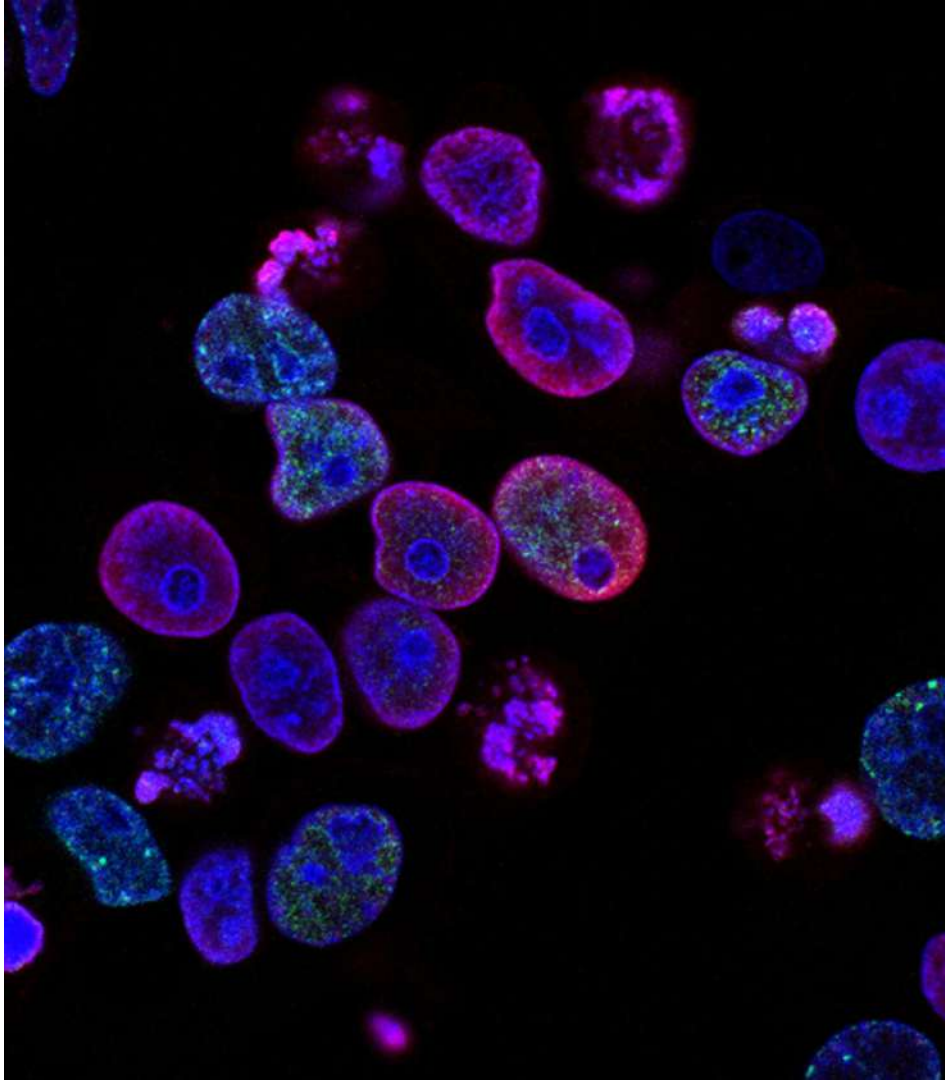


METÁFORA BIOLÓGICA

- Herencia
- Taxonomias

METÁFORA BIOLÓGICA

- Células
- Organismos



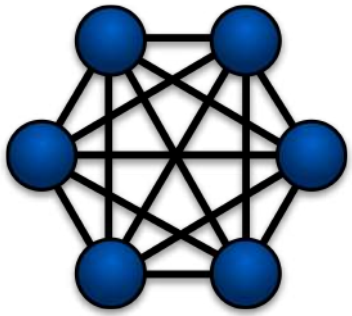
ELEMENTOS DE LA ESCUELA DE ALAN KAY

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**.



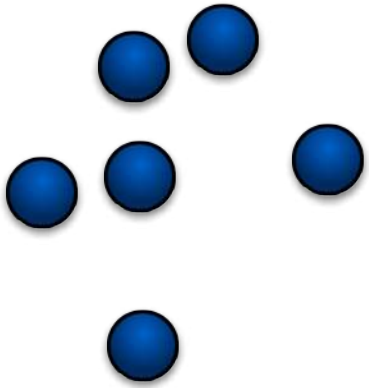
Red de Comunicaciones

The Early History Of Smalltalk by Alan Kay

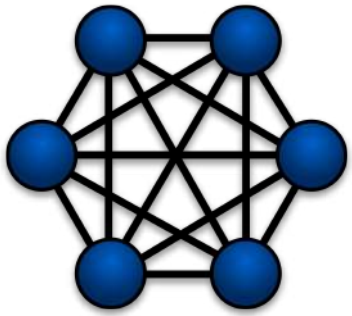
ELEMENTOS DE LA ESCUELA DE ALAN KAY

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.



Prototypes vs Classes - Alan Kay

ESCUELA DE ALAN KAY

creador de OOP y Smalltalk

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

Late binding

Pasar Mensajes

comunicación
mediante mensajes

Enlace Tardío

resolución de llamadas en
tiempo de ejecución

Encapsulación

proteger el estado interno
de una clase

PILARES FUNDAMENTALES - PASAR MENSAJES

messages.rb

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

messages.rb

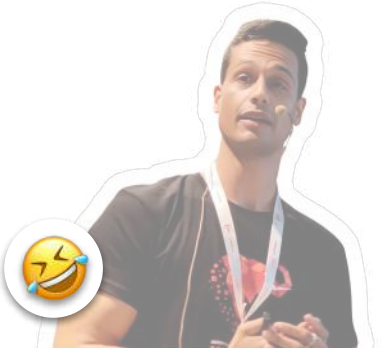
```ruby
thing.send(:do, some, stuff)
```

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

característica del lenguaje

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

traducirse

ensamblador o código máquina

creó el primer compilador

Grace Murray Hopper

HelloWorld.c

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

hello_world

```
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

característica del lenguaje

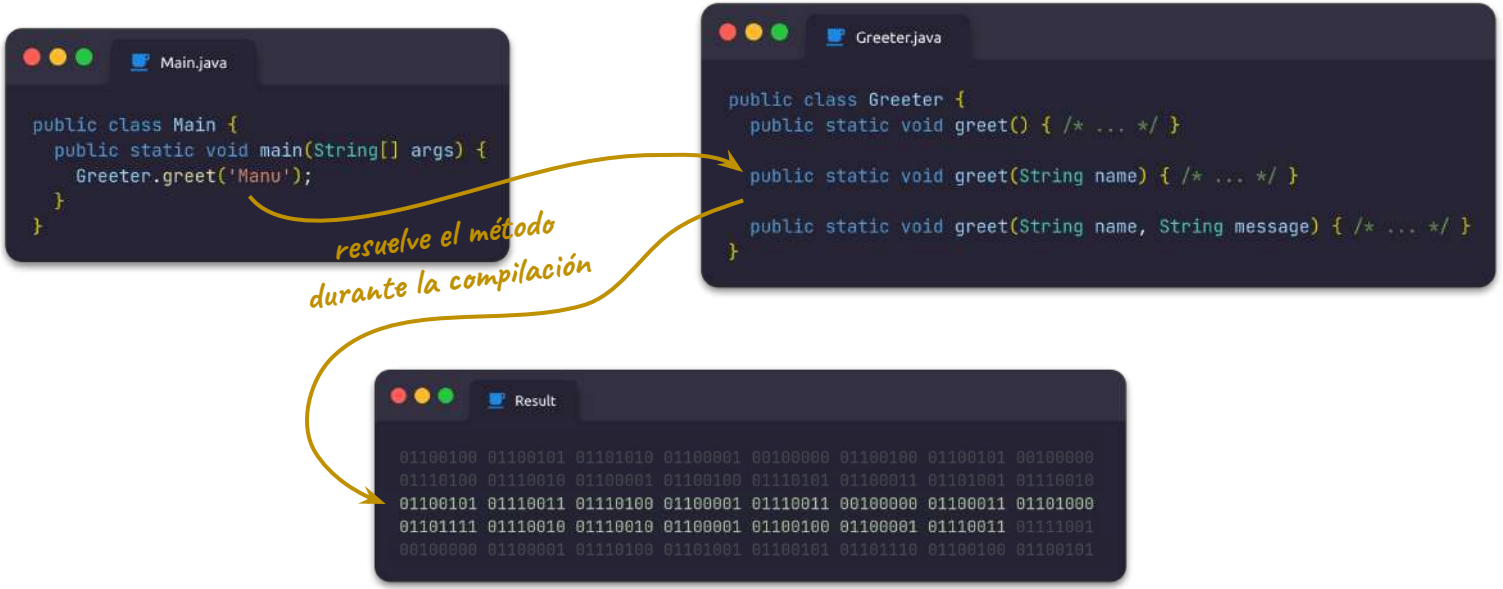
Lenguaje Interpretado: Necesitan un **programa** que va traduciendo nuestro código línea a línea.

interprete/traductor

PILARES FUNDAMENTALES - ENLACE TARDÍO

característica del lenguaje

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



PILARES FUNDAMENTALES - ENLACE TARDÍO

característica del lenguaje

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

Rentable.ts

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

ProductDetails.ts

```ts
class ProductDetails {
  private product: Rentable;

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

si esto no es un objeto concreto...

¿cual es la implementación de este método?

Book.ts

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

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

¿1?

Newspaper.ts

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

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

¿2?

CompactDisc.ts

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

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

¿3?

PILARES FUNDAMENTALES - ENCAPSULACIÓN

... la clave para crear sistemas *sostenible* y *escalable* es **diseñar cómo se comunican sus módulos...**

Alan Kay

Cohesión y Acoplamiento

provienen de programación estructurada

Kent Beck

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

Tweet

> **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, nunca 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.

las cosas que cambian juntas deben estar juntas

nadie tiene una nevera en el baño o una cama en la cocina espero...

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

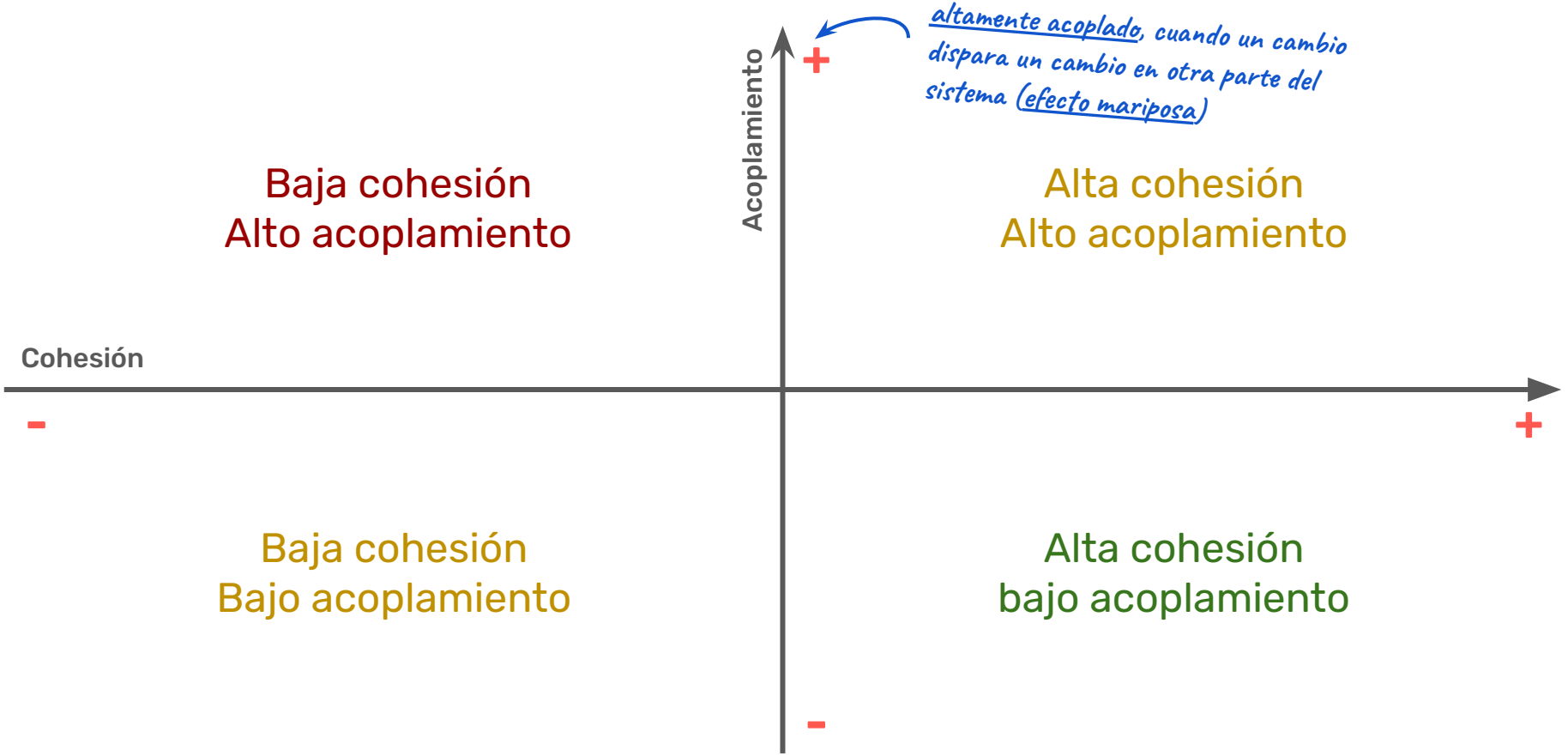
- Normalmente fácil de encontrar.
- Fácil de conseguir.
- Ayuda a encontrar desacoplamiento.

Acoplamiento

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

Tweet

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

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

customer.py

```
class Customer:
    name: str
    email: str
    phone: str
    address: Address
    orders: List[Order]
```

1:1

1:n

relaciones

RELACIONES - AGREGACIÓN

Un objeto está compuesto por otros objetos.

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

objetos relacionados

agregado

address.py

```
class Address:
    street: Street
    city: City
    postal_code: PostalCode
    country: Country
```

colaboradores

RELACIONES - COMPOSICIÓN

Un objeto está compuesto de otros objetos.

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

objeto principal

- rasgos
- interfaces
- clases abstractas

agregado

Chicken.php

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

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

característica (usando rasgos)

¡Si que puedo volar!

CanFly.php

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

implementación común de la característica

RELACIONES - COMPOSICIÓN

Un objeto está compuesto de otros objetos.

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

objeto principal

- rasgos
- interfaces
- clases abstractas

agregado

característica (usando interfaces)

contrato: si puedes volar, explícanos como

Chicken.php

```php
<?php
class Chicken implements CanFly {
  protected string $specieName;
  protected Peak $peakType;

  public function fly(): void {
    /* how chickens fly */
  }
}
```

implementación específica de la característica

CanFly.php

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

¡Si que puedo volar!

RELACIONES - DEPENDENCIA

objeto principal

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

- objetos secundarios
- dependencias
- colaboradores

cliente

GuessTheNumberGame.php

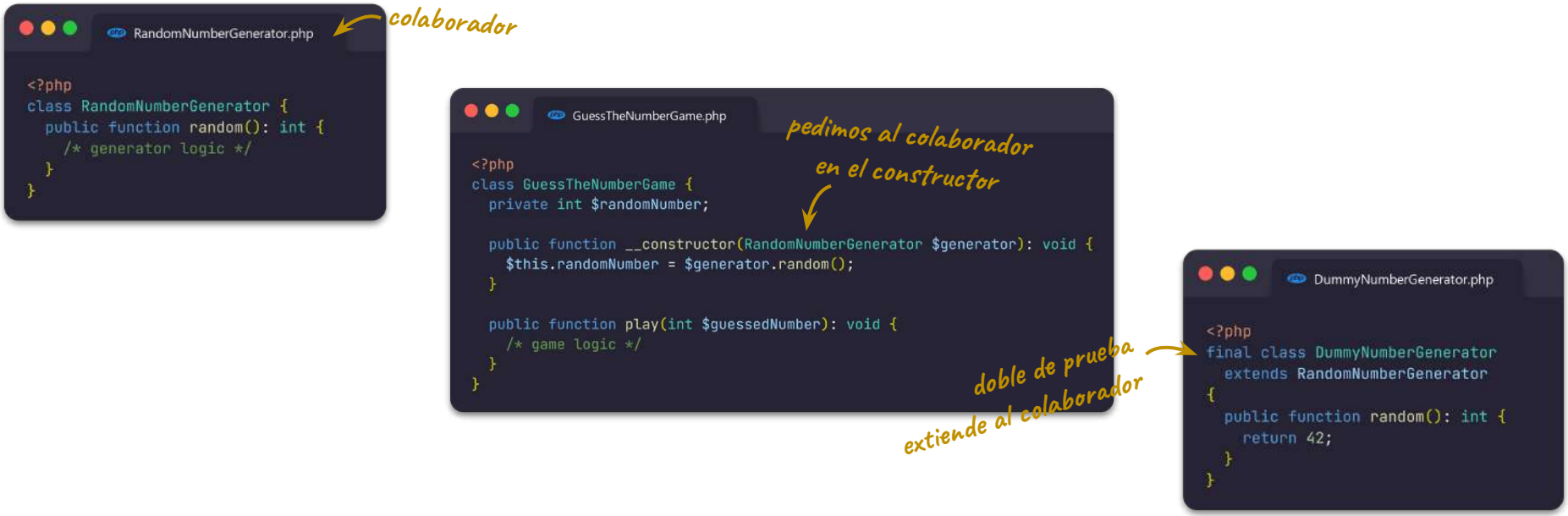
```php
<?php
class GuessTheNumberGame {
  private int $randomNumber;

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

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

colaborador: 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.



Inyección de dependencias - Beneficios

Reduce la gestión de las dependencias en de un módulo.

- Proporcionar al los objetos sus colaboradores.
- Código más limpio.
- Simplifica la configuración.
- Mejora los tests.

en lugar de crearlos ellos mismos

contenedores de dependencias

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

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

colaborador implementa el contrato

GuessTheNumberGame.php

```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

contrato

RandomNumberGenerator.php

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

doble de prueba implementa el contrato

DummyNumberGenerator.php

```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

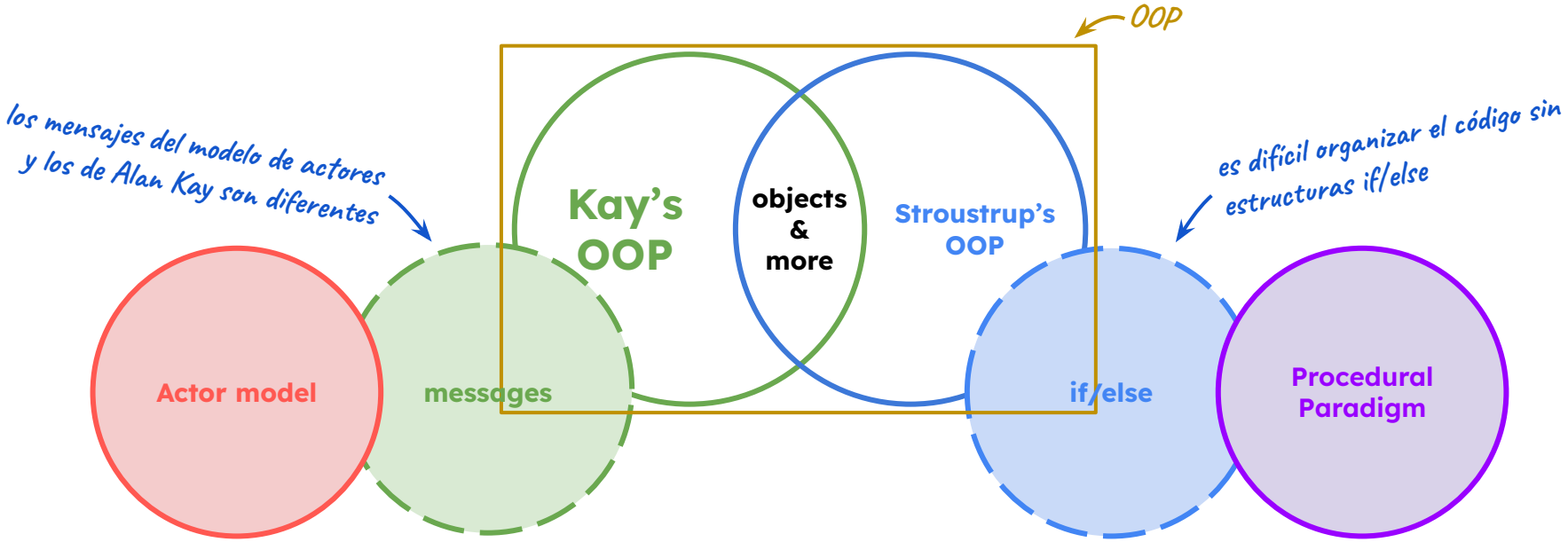
- Separar entre interfaces e implementaciones.
- Promueve el diseño modular.
- Reduce el acoplamiento.
- Mejora los tests.

implementaciones intercambiables

dobles de pruebas más simples

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

> Diferentes partes del programa se ocupan de intereses diferentes.

Fran Iglesias Gómez

Cada cosita en su lugar

Manuscripts of Edsger W. Djikstra

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

filter_users.php

```php
<?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')));
```

Separation of concerns

Llamada externa Procesamiento Salida

SEPARACIÓN DE INTERESES

filter_users.php

```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

Separation of concerns

Llamada externa Procesamiento Salida

SEPARACIÓN DE INTERESES

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

filter_users.php

```php
<?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

Separation of concerns

Llamada externa

Procesamiento

Salida

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.

Separation of concerns

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

```python
import sys

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

identificamos cada responsabilidad

Separation of concerns

Entrada Procesamiento Salida

Fran Iglesias Gómez

SEPARACIÓN DE INTERESES

sum_cli.py

```python
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

Separation of concerns

Entrada Procesamiento Salida

Fran Iglesias Gómez

SEPARACIÓN DE INTERESES

sum_cli.py

```python
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

Separation of concerns

Entrada Procesamiento Salida

SOLID

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

Single **R**esponsibility **P**rinciple

Open-**C**lose **P**rinciple

Liskov **S**ubstitution **P**rinciple

Interface **S**egregation **P**rinciple

Dependency **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.

¿QUÉ ES SOLID?

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

impacto de las dependencias

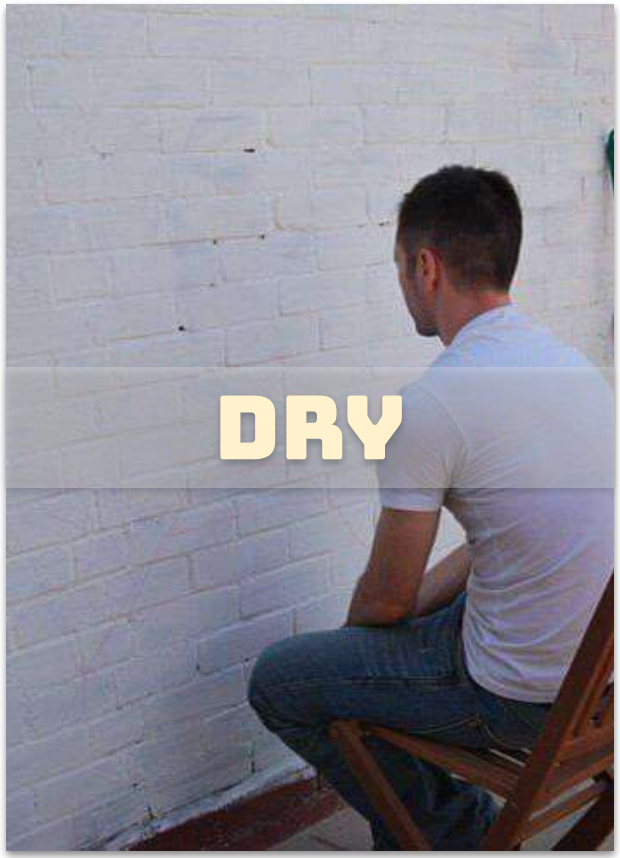
ayuda a gestionar la complejidad y tomar el control del coste del cambio teniendo en cuenta cohesión y acoplamiento

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

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

Uncle Bob SOLID principles

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

YAGNI

You Ain't Gonna Need It

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

no hables con extraños

DemetersLaw.php

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

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

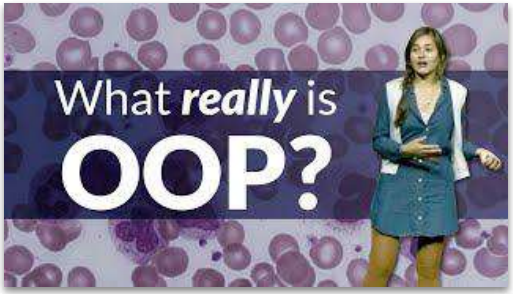
- **Solo explicar las herramientas**, no las disciplinas y **que relación tienen** entre sí.
- No tener en cuenta que las palabras **Orientación y Objetos**, son dos **terminos o visiones distintas**.
- No tener **contexto histórico**.
- **Malinterpretar las metáforas** de la orientación a objetos.
- 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

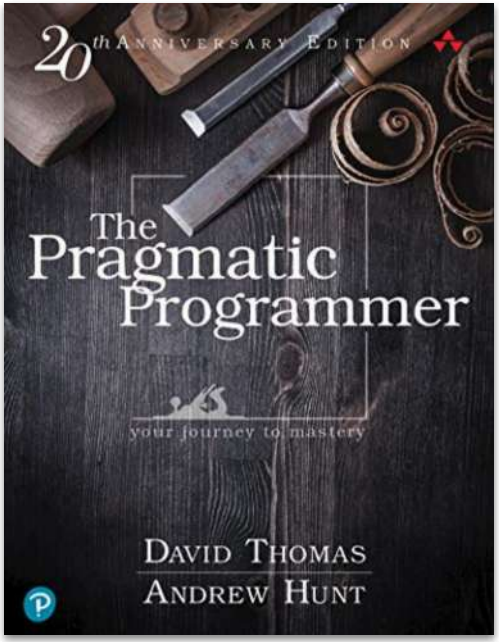
TL;DR There is no one "canonical" definition of OOP. There are **at least two** of them. From **my POV** there are two big definitions, which deserves separate names. I named them after biggest advocates. Those two definition have some differences and some commonalities.

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:

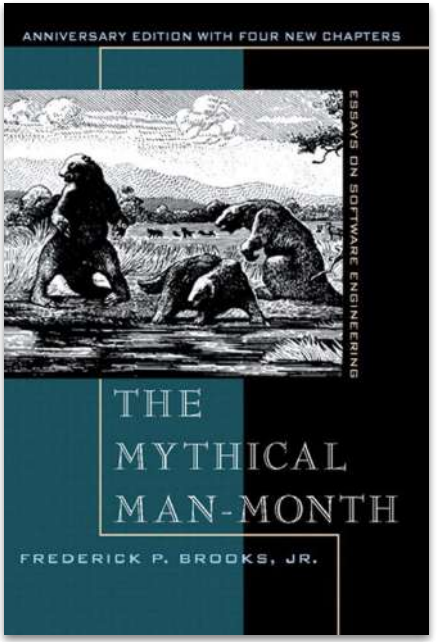
- Alan Kay's school or message focused
- Bjarne Stroustrup's school or class focused or "PolymorphismEncapsulationInheritance" in terms of c2

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

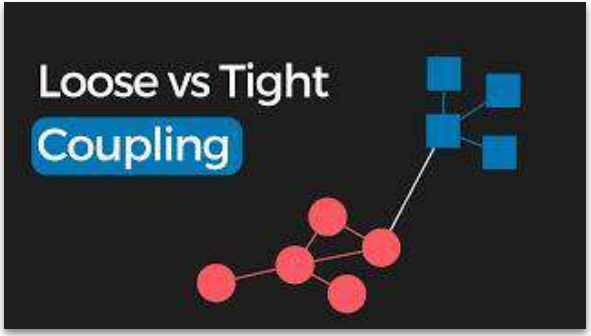


Mythical Man-Month, The: Essays on Software Engineering

Cohesión y acoplamiento

Coupling and Cohesion

Understanding Coupling and Cohesion



Loose vs Tight Coupling

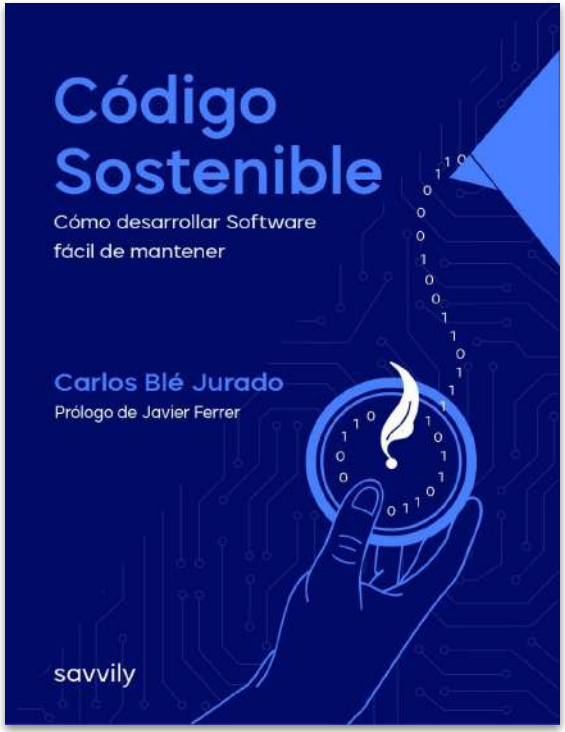
DISEÑO

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

CRC Cards (Class Responsibility Collaborator)

Por Manuel Perez

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,

CRC cards

CHARLAS SIMILARES

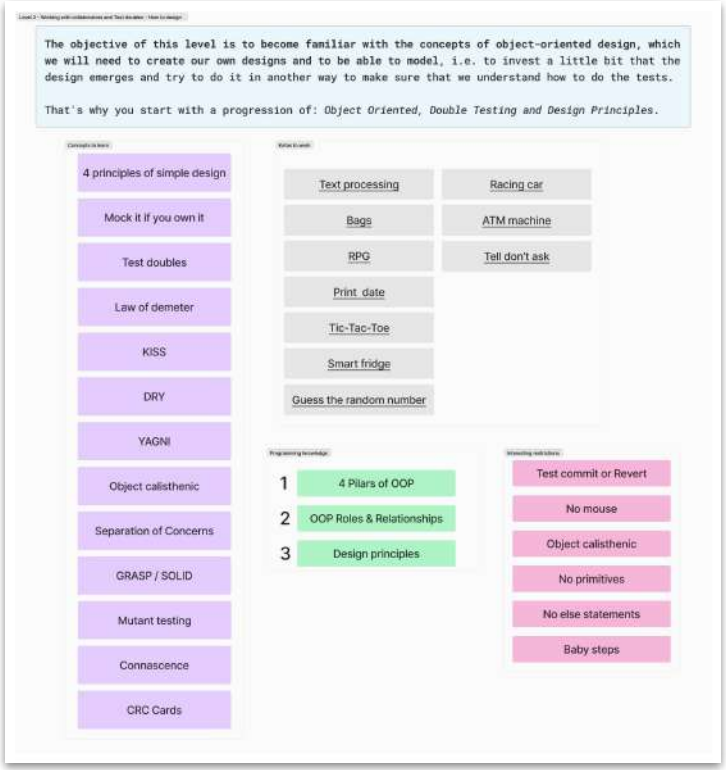
Lemi Orhan Ergin

thank you for the inspiration

Unlearn OOP: Deleted Scenes, Misconceptions, Unspoken Truths

LEARNING THROUGH KATAS

by Emmanuel Valverde



hay mas niveles! → Learning Through Katas

PILAR	DESCRIPCIÓN	CARACTERÍSTICAS
Abstracción	Representación de las características esenciales de un objeto sin incluir los detalles específicos irrelevantes en nuestro contexto.	- Modelado de objetos - Ocultamiento de información - Simplificación - Definición de interfaces
Encapsulación	Ocultamos el estado y el comportamiento interno de nuestros objetos para reducir la complejidad y exponemos un API simple de usar.	- Protección de datos - Ocultamiento de implementación - Control de acceso - Facilita el mantenimiento y evolución
Herencia	Mecanismo que permite que un clase hija herede atributos y métodos de una clase padre.	- Reutilización de código - Relación jerárquica - Polimorfismo - Herencia de comportamiento y estructura - Acoplamiento fuerte
Polimorfismo	Capacidad de un objeto para tomar diferentes formas y comportarse de diferentes maneras según la necesidad.	- Intercambiabilidad de objetos - Flexibilidad - Extensibilidad

RELACIÓN	DESCRIPCIÓN	CARACTERÍSTICAS
Asociación	Relación entre dos objetos donde uno tiene una referencia al otro de alguna manera.	- Dependencia flexible - Acoplamiento débil - Objetos independientes - Sin relación jerárquica - Pueden existir sin el otro objeto
Dependencia	Relación en la que un objeto utiliza o depende de otro objeto para llevar a cabo una operación.	- Dependencia flexible - Acoplamiento débil - Objetos independientes - No hay relación jerárquica - Uno depende del otro
Herencia	Relación en la que una clase hereda características (atributos y métodos) de otra clase.	- Reutilización de código - Relación jerárquica - Polimorfismo - Herencia de comportamiento y estructura - Acoplamiento fuerte
Implementación	Relación en la que una clase implementa una interfaz y proporciona una implementación concreta de sus métodos.	- Cumplimiento de contrato - Polimorfismo - Comportamiento específico - Acoplamiento flexible - Reutilización de interfaces
Agregación	Relación en la que un objeto contiene una colección de otros objetos, pero los objetos relacionados pueden existir independientemente del objeto principal.	- Relación "todo-parte" - Objetos independientes - Asociación flexible - Puede haber múltiples partes - Ciclo de vida independiente - Acoplamiento débil
Composición	Relación en la que un objeto contiene otros objetos como componentes, pero los objetos componentes no pueden existir sin el objeto principal.	- Relación "todo-parte" - Objetos dependientes - Asociación fuerte - Una sola parte - Ciclo de vida acoplado - Acoplamiento fuerte

PREGUNTAS

GRACIAS

👏