

Introducción a los Patrones de Diseño

Serie de Diseño de Software

Me llamo Daniel

Vivo en Yucatán, México
y escribo código 

¿Alguien dijo Diseño de Software?

Te escuchamos y tenemos grandes planes para ti 😈

Introducción a los patrones de diseño

Patrones
creacionales

Patrones
estructurales

Patrones de
comportamiento



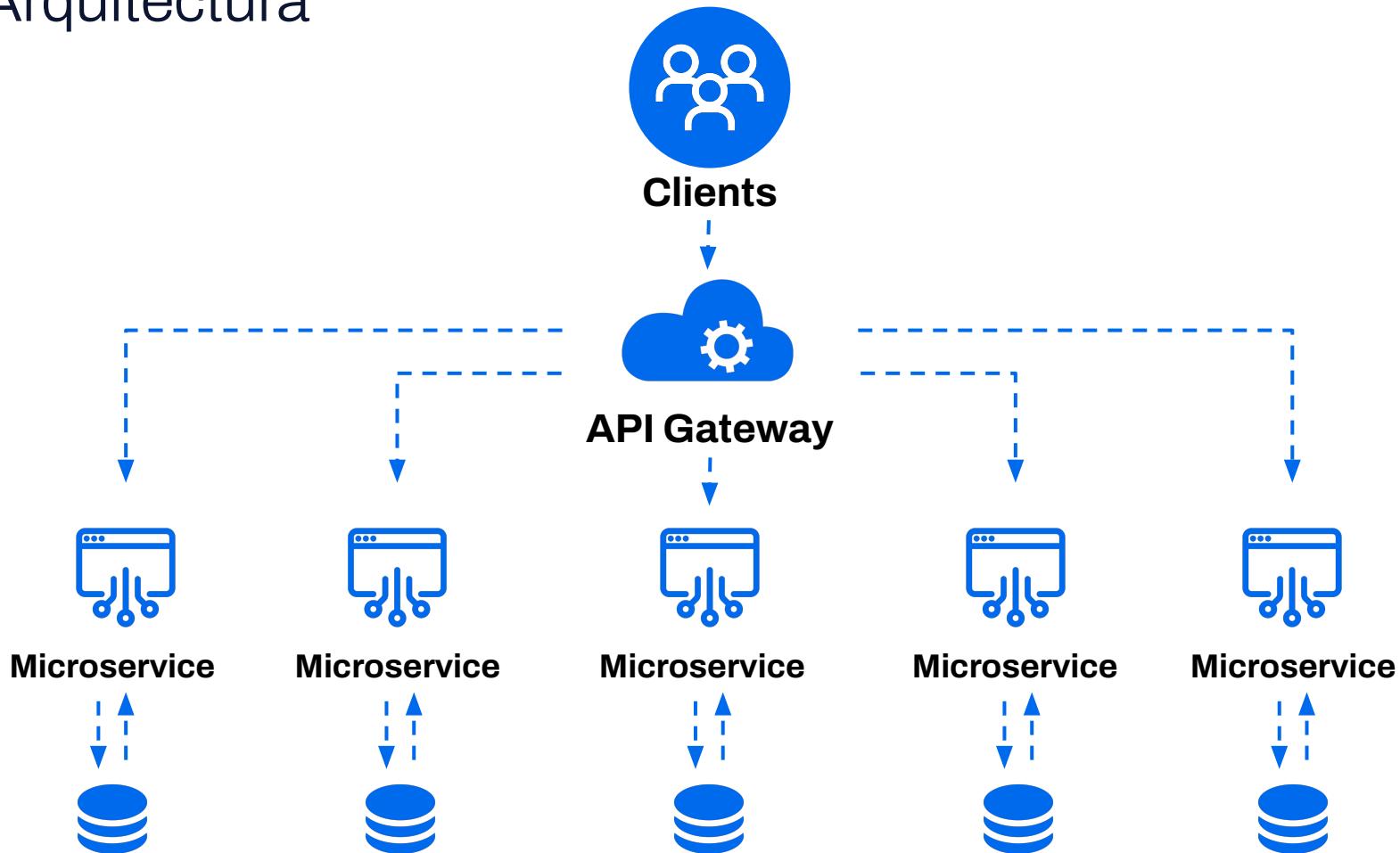
¡Estás aquí!

¿Por qué Diseño de Software?

- El diseño implica un proceso de creación y de desarrollo para producir un nuevo objeto.
- Nuestro objeto es el Software.
- Diseñar Software implica pensar en un modelo que describa de la mejor forma posible cómo funcionará nuestra aplicación.

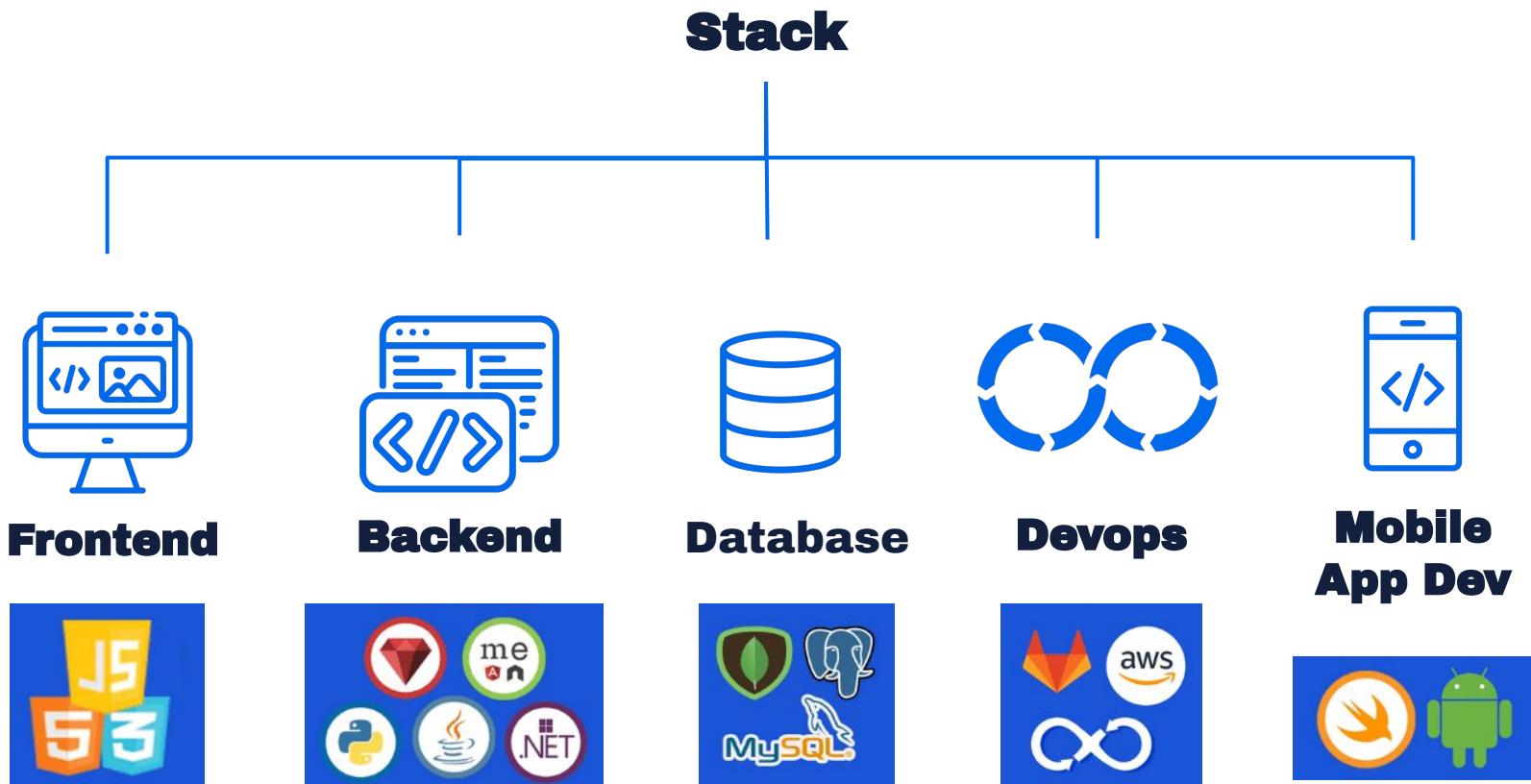
Algunos elementos del diseño...

Arquitectura



Algunos elementos del diseño...

Elección del stack



Fin de la clase 1

Ingeniería de Software



¿Qué es ingeniería?

“

Engineering is the designing, testing and building of machines, structures and processes using maths and science.

- University of Bath, UK

”

¿Qué es Ingeniería de Software?

“

Software engineering is the application of a systematic, disciplined, quantifiable approach to the design, development, operation and maintenance of software.

- IEEE 2010

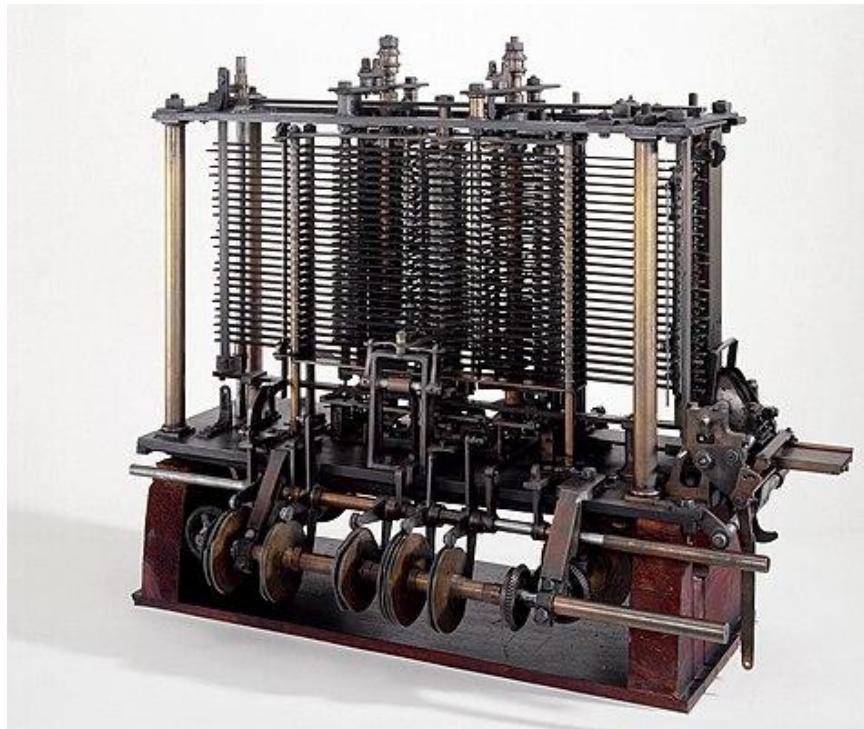
”

**¿Dónde nació
la Ingeniería de
Software? 00**

Un poco de historia



La Máquina Analítica, considerada como la primera computadora, fue diseñada y parcialmente construida por el inventor inglés Charles Babbage en el siglo XIX. Realizaba sus operaciones con ruedas y engranajes. Anticipó ideas como: unidad de cálculos, unidad de memoria y la unidad de entrada/salida de datos.



Un poco de historia



En agosto de 1966, el presidente de la Association for Computing Machinery (ACM por sus siglas) Anthony Gervin Oettinger, utilizó el término en el volumen 9 del diario ACM Communications en una nota destinada a los miembros.



Un poco de historia



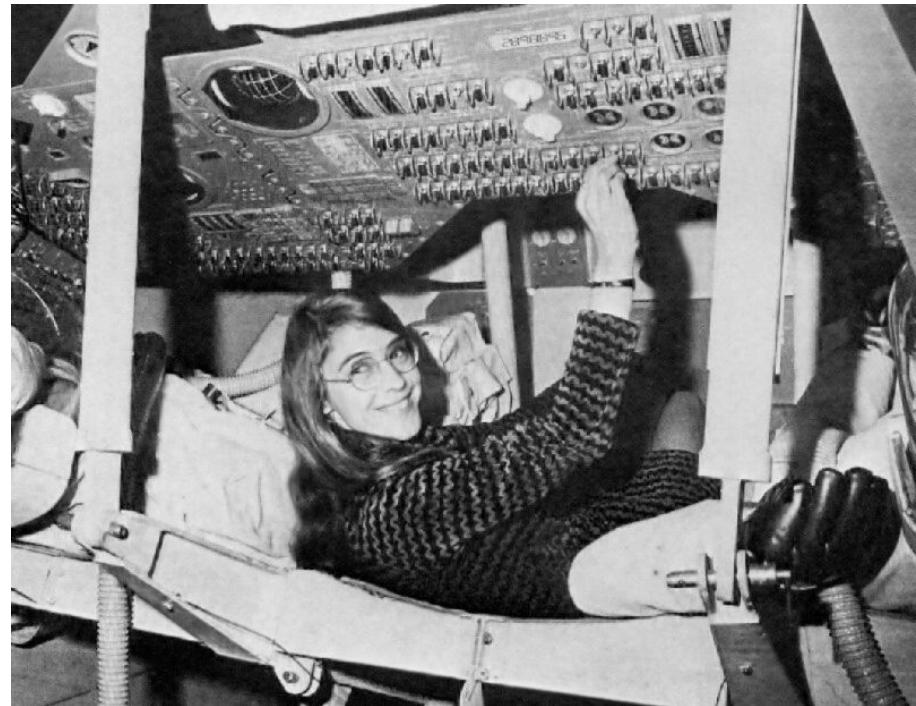
En 1968, La OTAN organizó las Conferencias sobre Ingeniería de Software en donde una de las ponencias dictada por el profesor Friedrich Bauer fue titulada “Software Engineering”.



Un poco de historia



Sin embargo, muchas fuentes le atribuyen la acuñación del término a la ingeniera y científica de la computación Margaret Heafield Hamilton, que buscaba darle legitimidad a la disciplina. Margaret fue la encargada de desarrollar el software de vuelo para el programa Apolo de la NASA.



Sources:

Daphne Weld Nichols, CC BY-SA 3.0 <<https://creativecommons.org/licenses/by-sa/3.0/>>, via Wikimedia Commons
Draper Laboratory, Public domain, via Wikimedia Commons

Fin de la clase 2

Software Development Life Cycle

SDLC para los cuates.

Procesos hasta en la sopa...

El **SDLC** puede ser entendido como un **proceso de varias fases** cuyo propósito es la producción de software de:

- Alta calidad
- Costos bajos
- Menor tiempo posible



The image shows a close-up of a computer monitor displaying a large block of code. The code is written in a programming language, possibly PHP, and includes HTML and CSS elements. It features color-coded syntax highlighting where different parts of the code are highlighted in various colors like red, green, blue, and yellow. The code is organized into several lines, with some lines being longer than others, creating a visual representation of the complexity of the software development process.

Fases conocidas

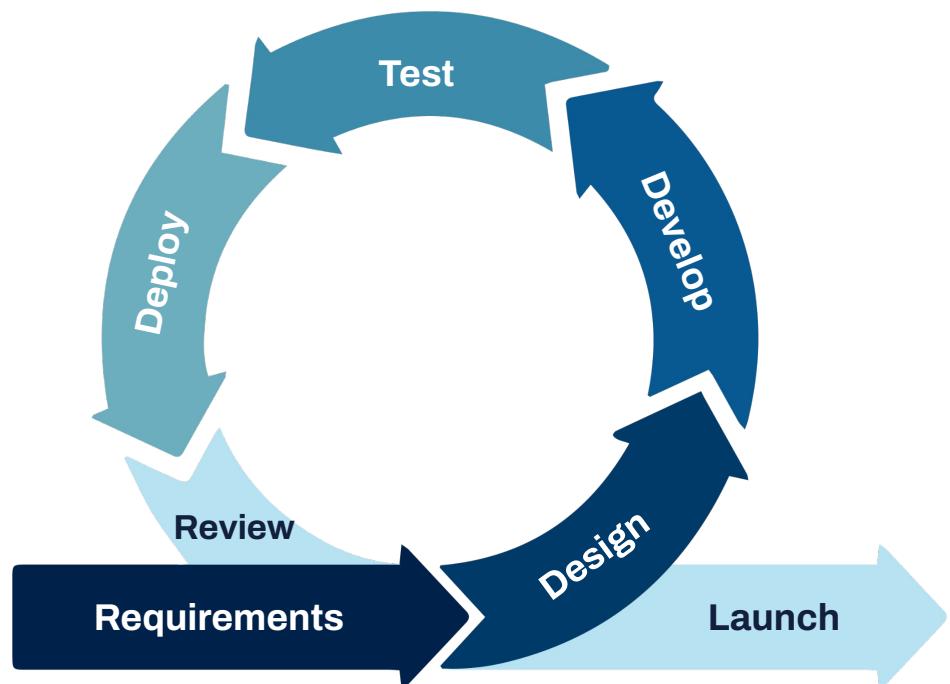
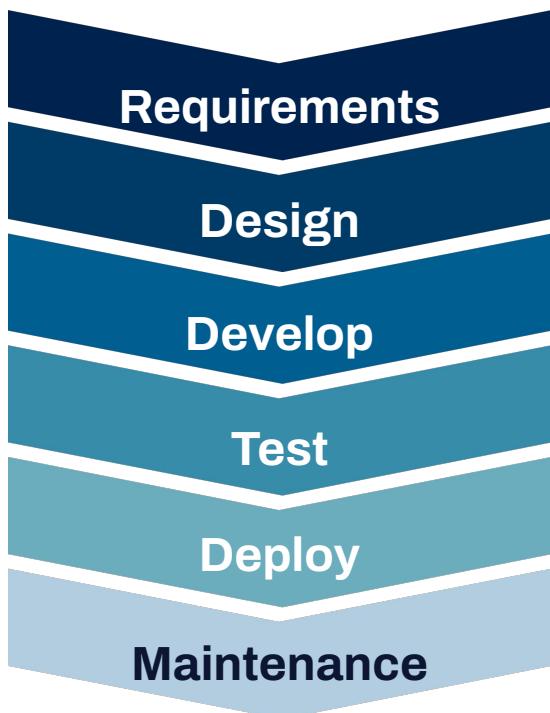
1. Planeación/Requerimientos 
2. Diseño 
3. Implementación/Ejecución/Desarrollo 
4. Pruebas 
5. Despliegue (A.K.A Deploy 😊) 
6. Mantenimiento 

Implementaciones del SDLC

Waterfall

vs.

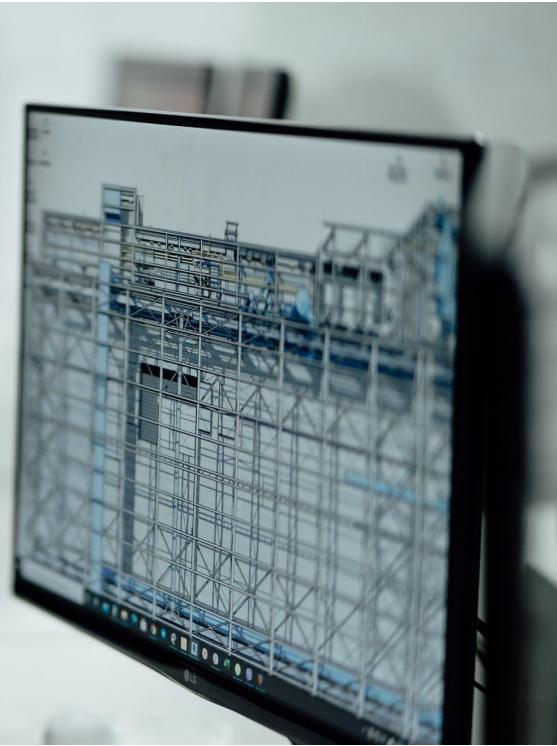
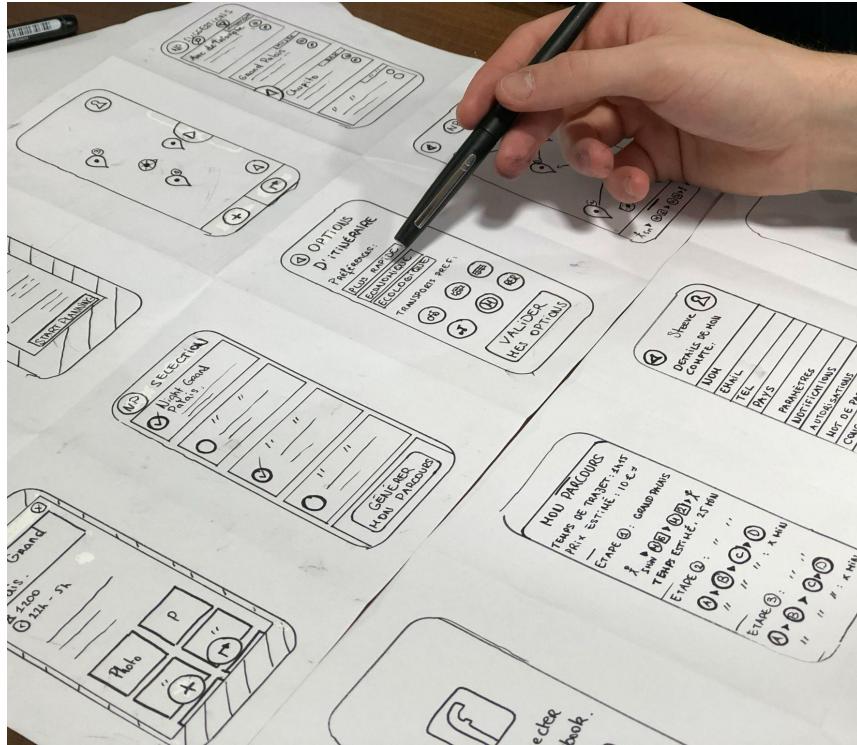
Agile



Fin de la clase 3

Fase de diseño de Software (el cafecito ☕)

Tipos de diseño



Algunos elementos de un buen diseño de Software

1. Modularidad
2. Tolerancia a fallos
3. Robustez
4. Seguridad
5. Usabilidad
6. Reusabilidad
7. Extensibilidad



Fin de la clase 4

Reusabilidad



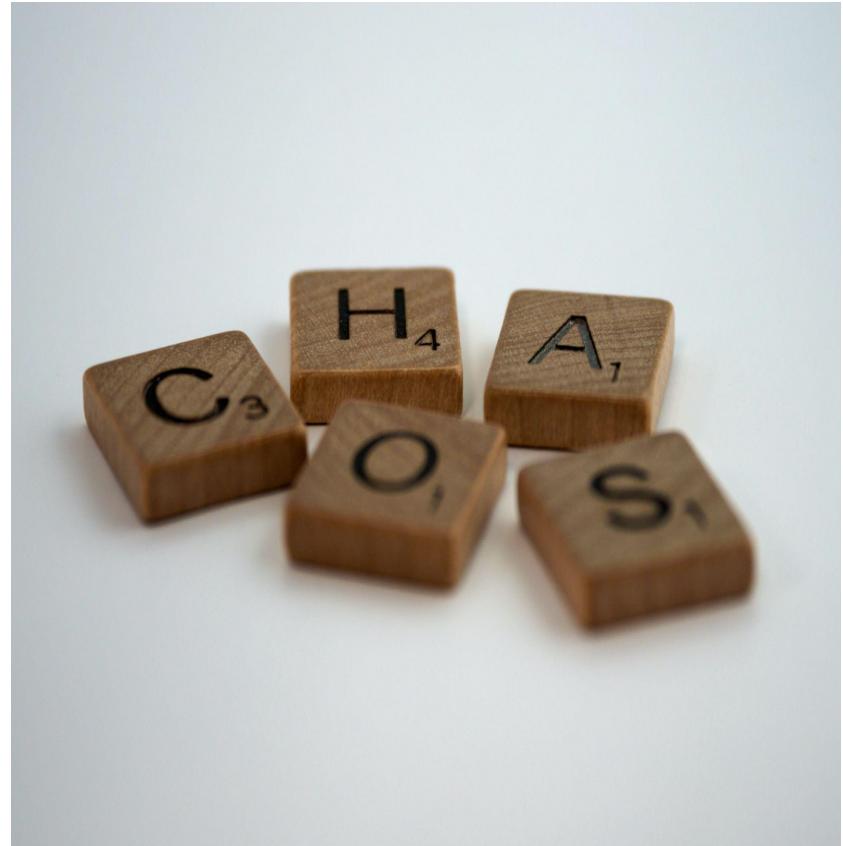
Reusabilidad: lo genial

- Reducción de:
 - Costos
 - Tiempos para lanzar un producto
- Libera recursos para tareas más cruciales como el marketing.
- ¿Por qué reinventar la rueda?



Reusabilidad: lo complejo

- Eliminar la duplicación y crear una abstracción (DRY)
 - Algunas partes no deberían ser abstraídas.
 - El código puede volverse innecesariamente complejo (acoplamiento).



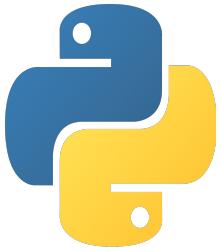
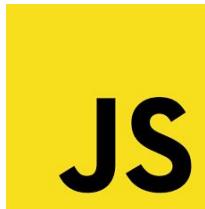
Niveles de reusabilidad





Clases / Funciones

A.K.A Ingredientes

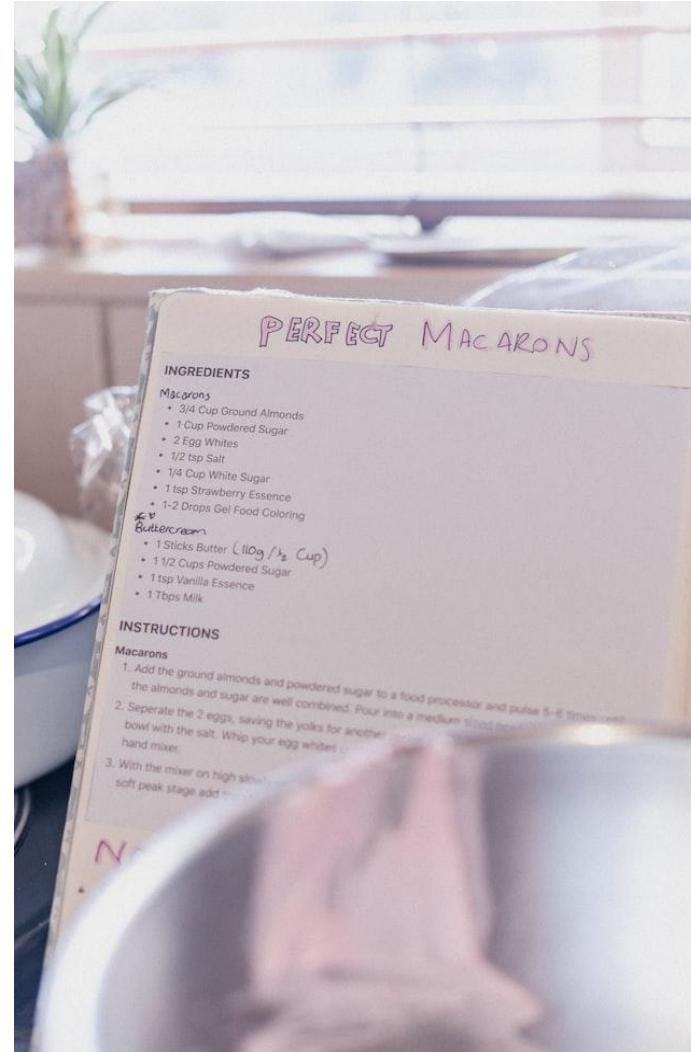


Sources:
Marisol Benitez on Unsplash



Patrones de diseño

A.K.A Receta



Sources:

Sincerely Media on Unsplash

Frameworks

A.K.A Lasagna congelada

django

NEXT.js

RAILS



Niveles de reusabilidad (Este slide no va)

Erich Gamma¹, uno de los fundadores de las ideas detrás de los Patrones de Diseño, señala que existen tres niveles de reusabilidad.

Frameworks (Rails, Django, Angular)



Patrones de Diseño



Reusabilidad de Clases/Funciones (Class User)

¹ <https://www.artima.com/articles/erich-gamma-on-flexibility-and-reuse>



```
import 'http' from 'HttpModule'

class UserService:

    private url = '/api/users'

    public method getAll() is
        return http.get(url)
```

UserService

- url: string

+method getAll

depends

HttpModule



```
import 'httpGateway' from 'HttpGateway'

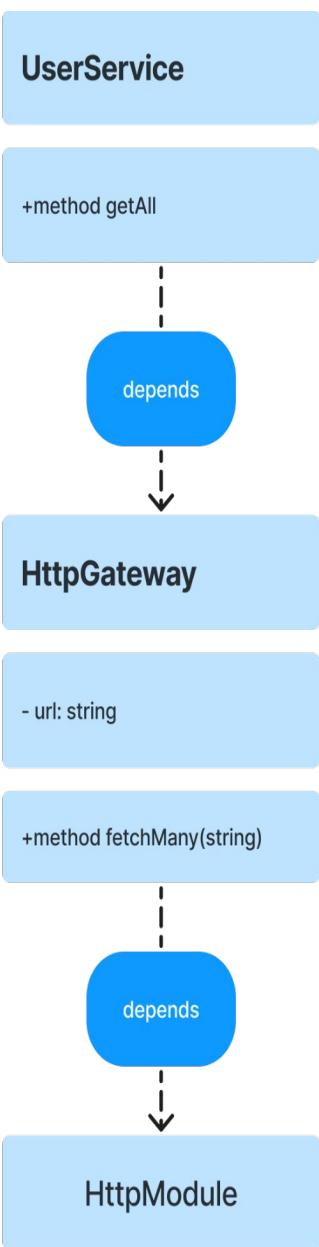
class UserService:
    public method getAll() is
        return httpGateway.fetchMany('users')

// HttpGateway file
import 'http' from 'HttpModule'

class HttpGateway:

    private url = '/api'

    public method fetchMany(resource) is
        // The url will be /api/users
        return http.get(url + '/' + resource)
```



Fin de la clase 5

Extensibilidad 🙌

El cambio es la única constante en el universo
(y en la vida de un programador).

Extensibilidad: lo genial

- Puntos de extensión/cambio sencillos y comprensibles.
- Reglas de cambio definidas por contratos/interfaces.



Extensibilidad: los retos

- Tener una buena comprensión del problema.
- Consenso en los estándares y/o reglas que guíen el desarrollo.





```
import 'httpGateway' from 'HttpGateway'

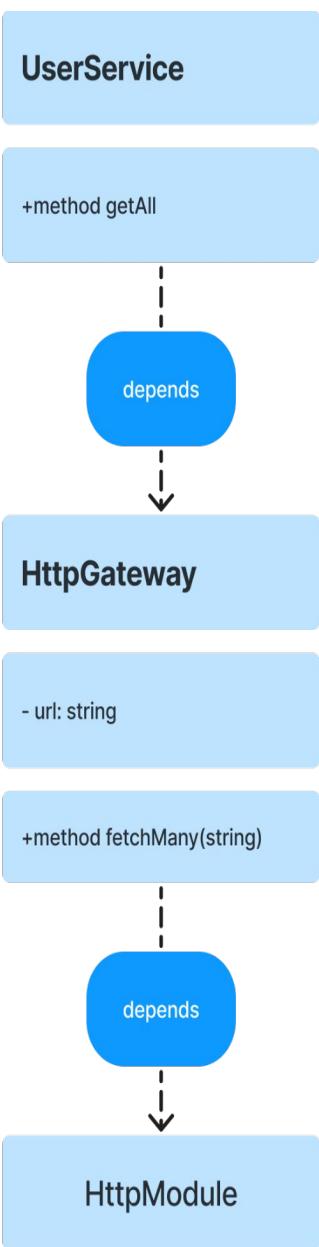
class UserService:
    public method getAll() is
        return httpGateway.fetchMany('users')

// HttpGateway file
import 'http' from 'HttpModule'

class HttpGateway:

    private url = '/api'

    public method fetchMany(resource) is
        // The url will be /api/users
        return http.get(url + '/' + resource)
```





```
interface Gateway {  
    public method fetchMany(resource)  
    public method fetchOne(resource, idToFetch)  
}
```

<<Interface>>

Gateway

+method fetchMany(string)
+method fetchOne(string, string)



```
import 'http' from 'HttpModule'

class RestGateway implements Gateway:

    private url = '/api'

    public method fetchMany(resource) is
        // The url will be /api/{resource}
        return http.get(url + '/' + resource)

    public method fetchOne(resource, idToFetch) is
        // The url will be /api/{resource}/:idToFetch
        return http.get(url + '/' + resource + '/' + idToFetch)
```

<<Interface>>

Gateway

+method fetchMany(string)
+method fetchOne(string, string)

implements

RestGateway

- url: string

+method fetchMany(...)
+method fetchOne(...)



```
import 'http' from 'HttpModule'

class GraphqlGateway implements Gateway:

    private url = '/api/graphql'

    public method fetchMany(resource) is
        // The url will be /api/graphql
        var body = buildBody({ resource })
        return http.post(url, body)

    public method fetchOne(resource, idToFetch) is
        // The url will be /api/graphql
        var body = buildBody({ resource, idToFetch })
        return http.post(url, body)
```

<<Interface>>

Gateway

+method fetchMany(string)
+method fetchOne(string, string)



GraphqlGateway

- url: string

+method fetchMany(...)
+method fetchOne(...)

```

import 'Gateway' from 'Gateway'
import 'GraphQLGateway' from 'GraphqlGateway'

class UserService:

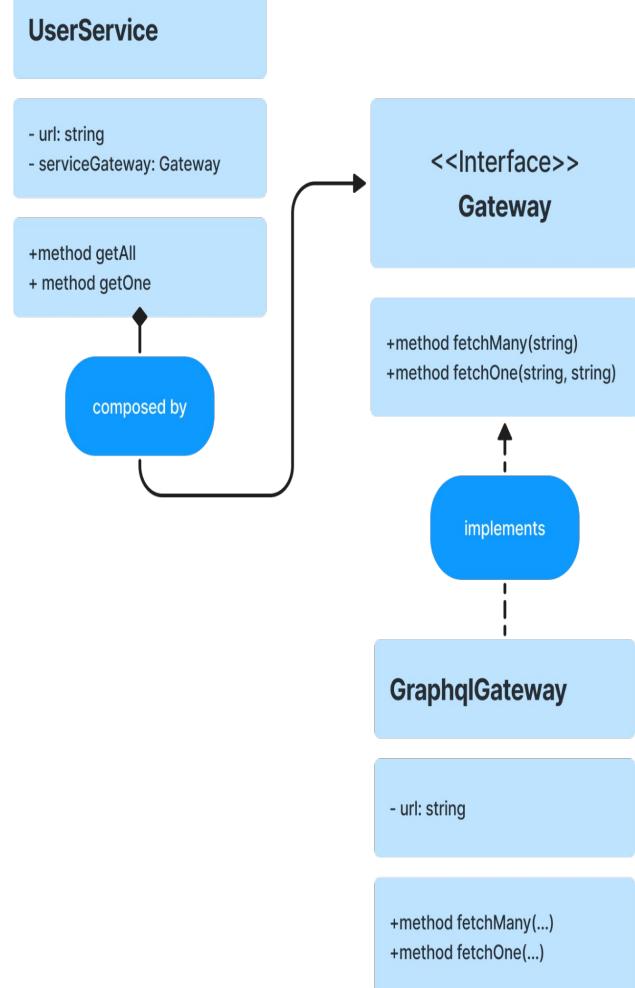
    private serviceGateway: Gateway

    public method constructor() is
        serviceGateway = new GraphqlGateway()

    public method getAll() is
        return serviceGateway.fetchMany('users')

    public method getOne(userId) is
        return serviceGateway.fetchOne('users', userId)

```



Fin de la clase 6

Nos enfocamos en...

- 1. Reusabilidad**
- 2. Extensibilidad**
- 3. Sencillez**



Proceso para llegar a una solución...

- 1.** Pensamos el problema.
- 2.** Implementamos algo sencillo/intuitivo.
- 3.** Nuevos requerimientos.
- 4.** Planear pensando en reutilizar/extender.
- 5.** Regresar al paso 2 y repetir hasta lograr estabilidad.



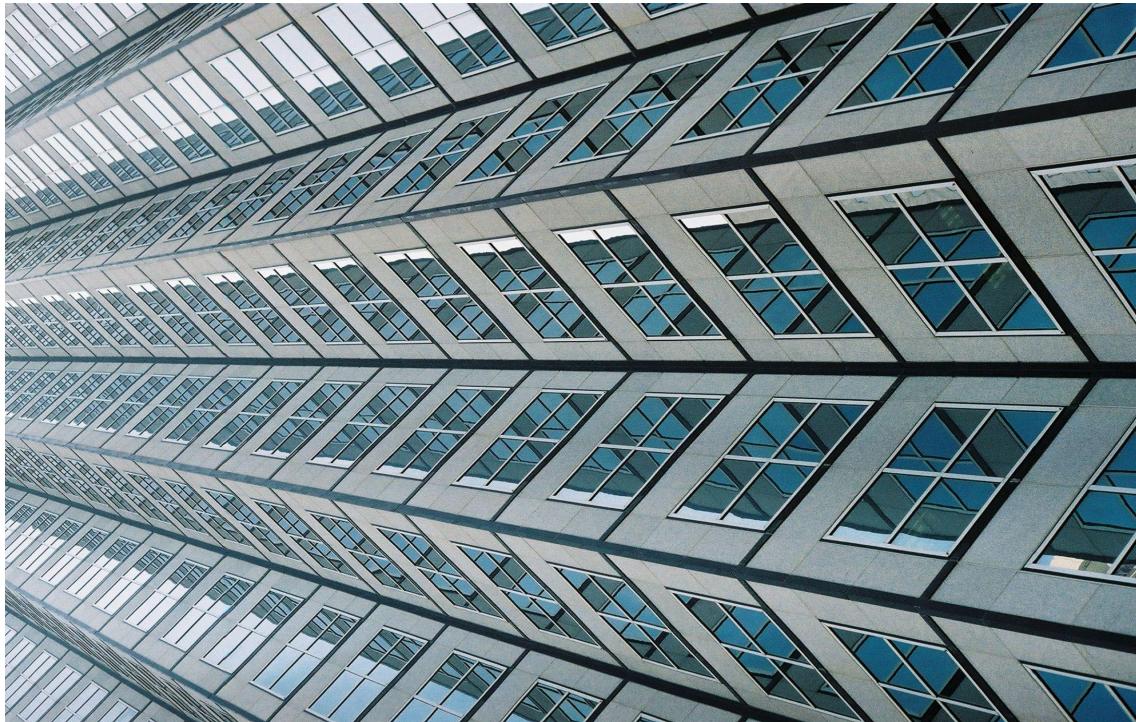
Fin de la clase 7

Patrones de diseño



¿Qué son los patrones de diseño?

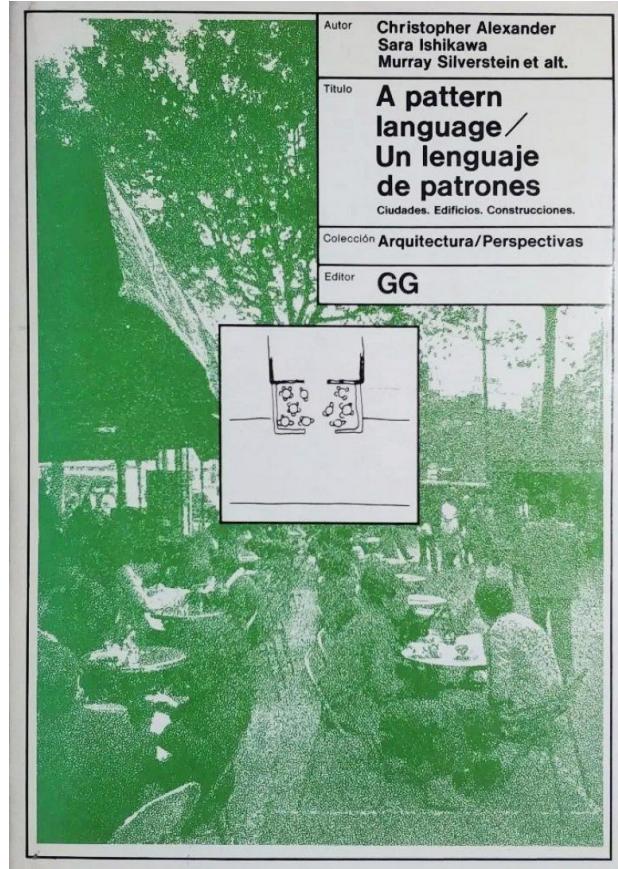
Soluciones habituales a problemas comunes en el diseño de Software.



Un poco de historia



El concepto de los “patrones” se originó como un concepto arquitectónico por Christopher Alexander, en 1977 en su libro “Un Lenguaje de patrones”.



Un poco de historia

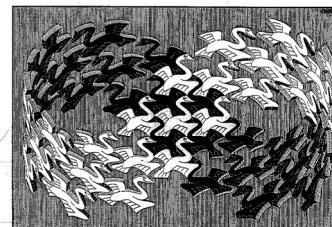


La idea fue recogida por cuatro autores (Gang of Four) Erich Gamma, John Vlissides, Ralph Johnson y Richard Helm, que en 1994 lanzan el libro “Patrones de diseño: Elementos Reutilizables en el Software Orientado a Objetos”

Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Cover art © 1994 M.C. Escher / Cordon Art - Baam - Holland. All rights reserved.

Foreword by Grady Booch



**¿Debo aprender
patrones de diseño?**



Por dos razones...

Son una caja de herramientas de soluciones a problemas comunes en el diseño de software.

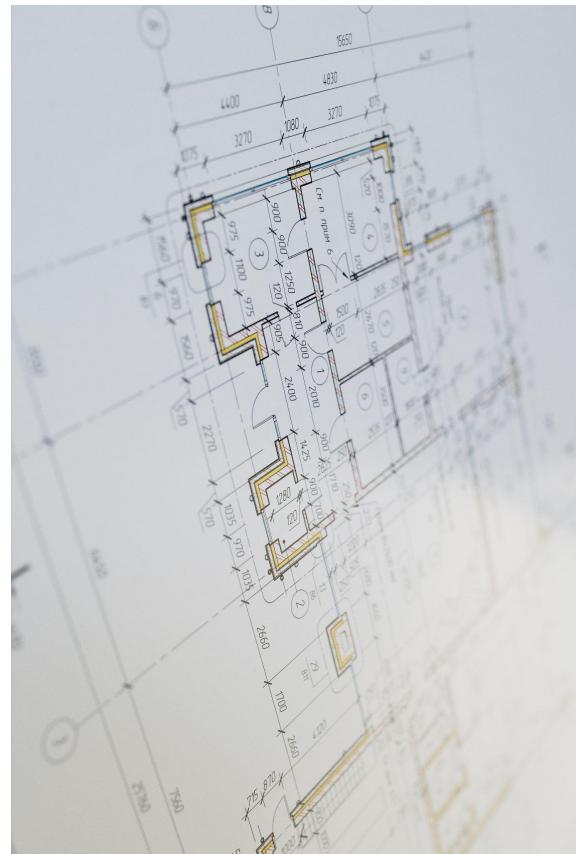


Definen un lenguaje que tú y tu equipo usarán para comunicarse de forma eficiente.



A considerar...

- Pueden ser vistos como planos prefabricados.
- Un patrón no es una biblioteca que podamos cortar, pegar y utilizar.
- Un patrón no es un algoritmo
 - Un algoritmo nos dirá el orden de los pasos.
 - Un patrón nos dirá el resultado y sus funciones.



Fin de la clase 8

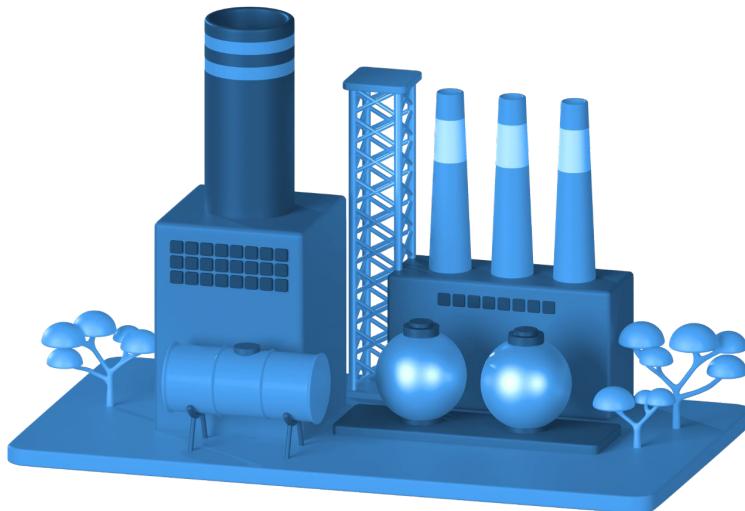
Categorías de patrones de diseño



Patrones creacionales

Su función es proveer mecanismos para la creación de objetos que favorecen la flexibilidad y reusabilidad del código.

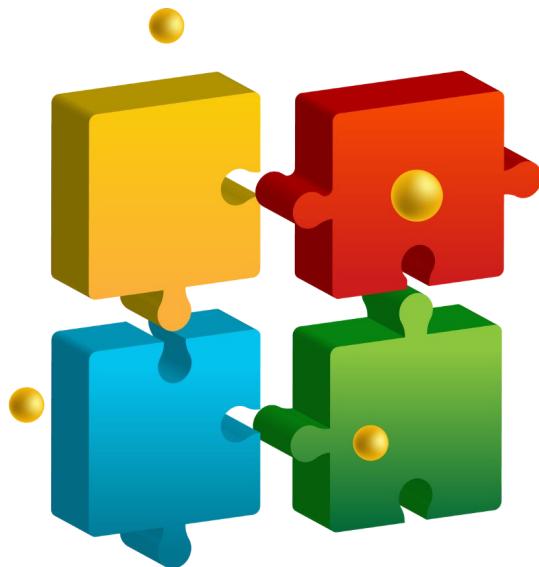
Ejemplos: Builder, Factory Method y Singleton.



Patrones estructurales

Su función es determinar cómo las clases y objetos se combinan/ensamblan para formar estructuras más grandes.

Ejemplos: Adapter, Decorator, Facade y Proxy.



Patrones de comportamiento

Su función es tratar con procedimientos que se encargan de implementar una comunicación y asignación de responsabilidades efectiva entre objetos.

Ejemplos: Command, Iterator, Observer y Strategy.



Fin de la clase 9

POO y patrones de diseño



Hablemos de Objetos y Clases

Un paradigma de muchos objetos...

La Programación Orientada a Objetos (POO para los amigos) basa el diseño del software alrededor de la figura de los objetos, elementos que contienen propiedades y comportamiento.

Class
Person

-firstName: string
+lastName: string

+method run
+method walk

Pilares de la POO



Fin de la clase 10

Herencia y Composición



Algunos detallitos...

1. La subclase y la superclase están estrechamente acopladas.
2. Una subclase no puede reducir la interfaz de la superclase.
3. Tratando de reusar el mayor código posible podemos crear clasificaciones (taxonomías) de clases complejas.

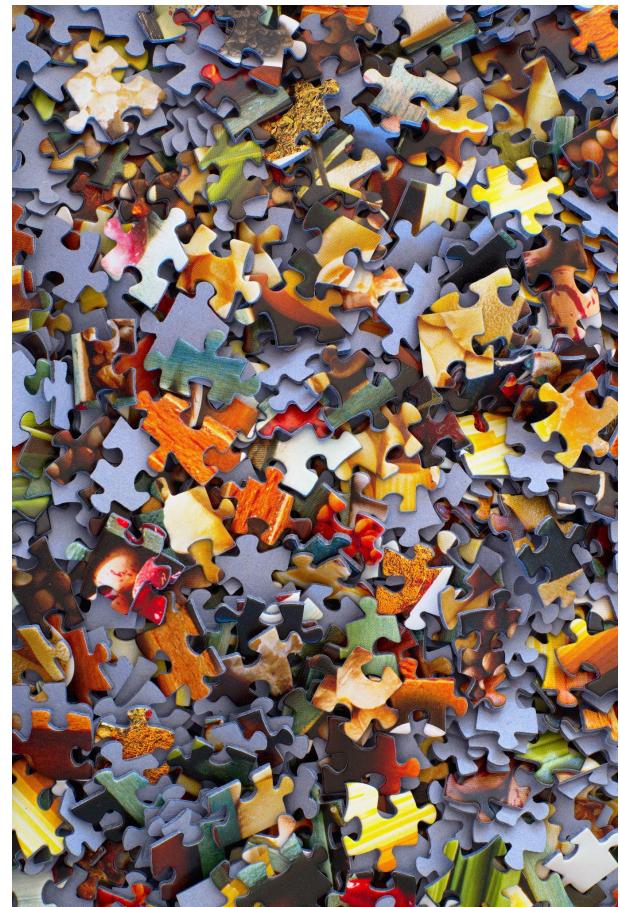


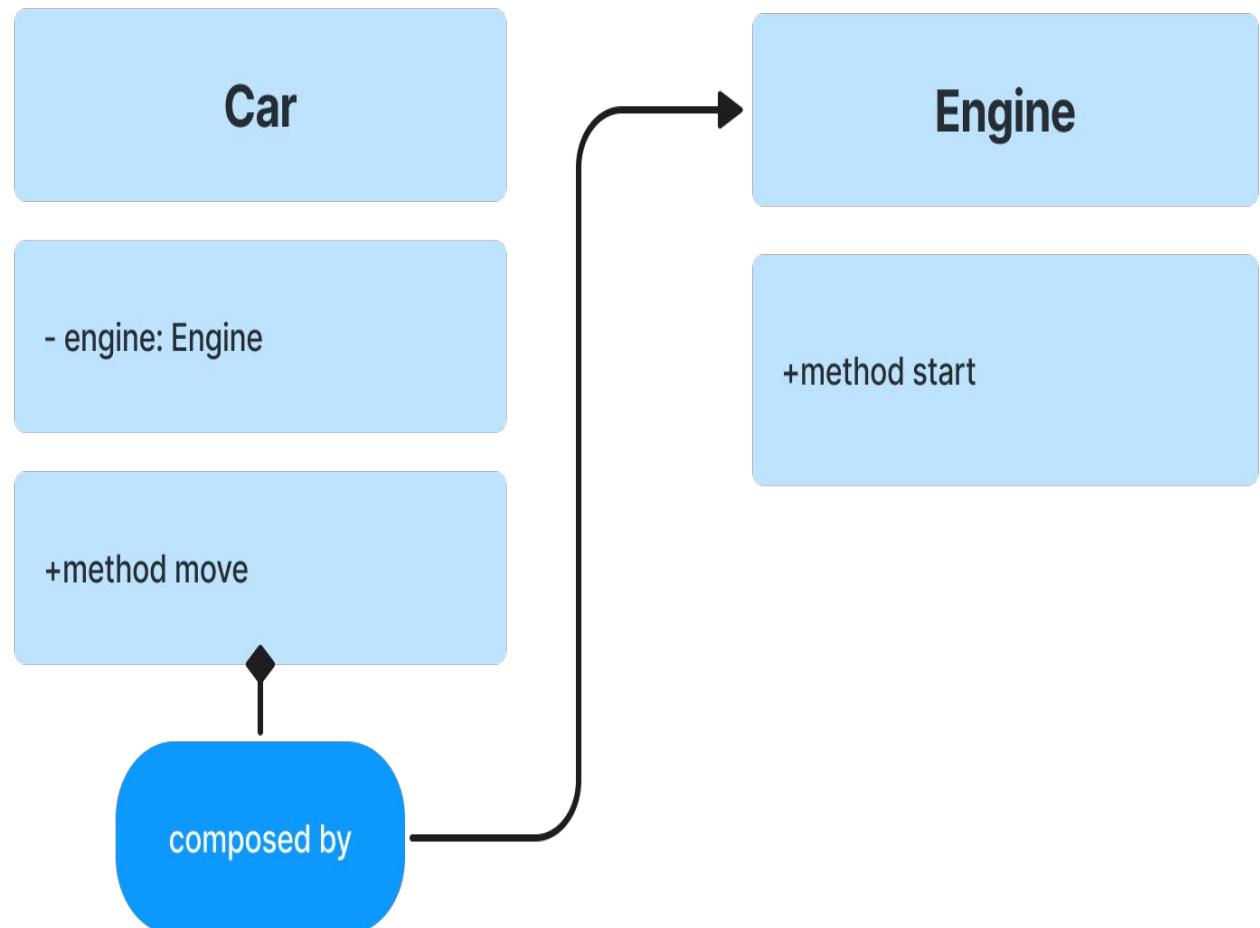
**¿Composición
al rescate?**



¿Cómo definir la composición?

Una relación entre dos clases, en donde una de ellas o ambas, hace referencia a objetos de la otra clase en sus propiedades.





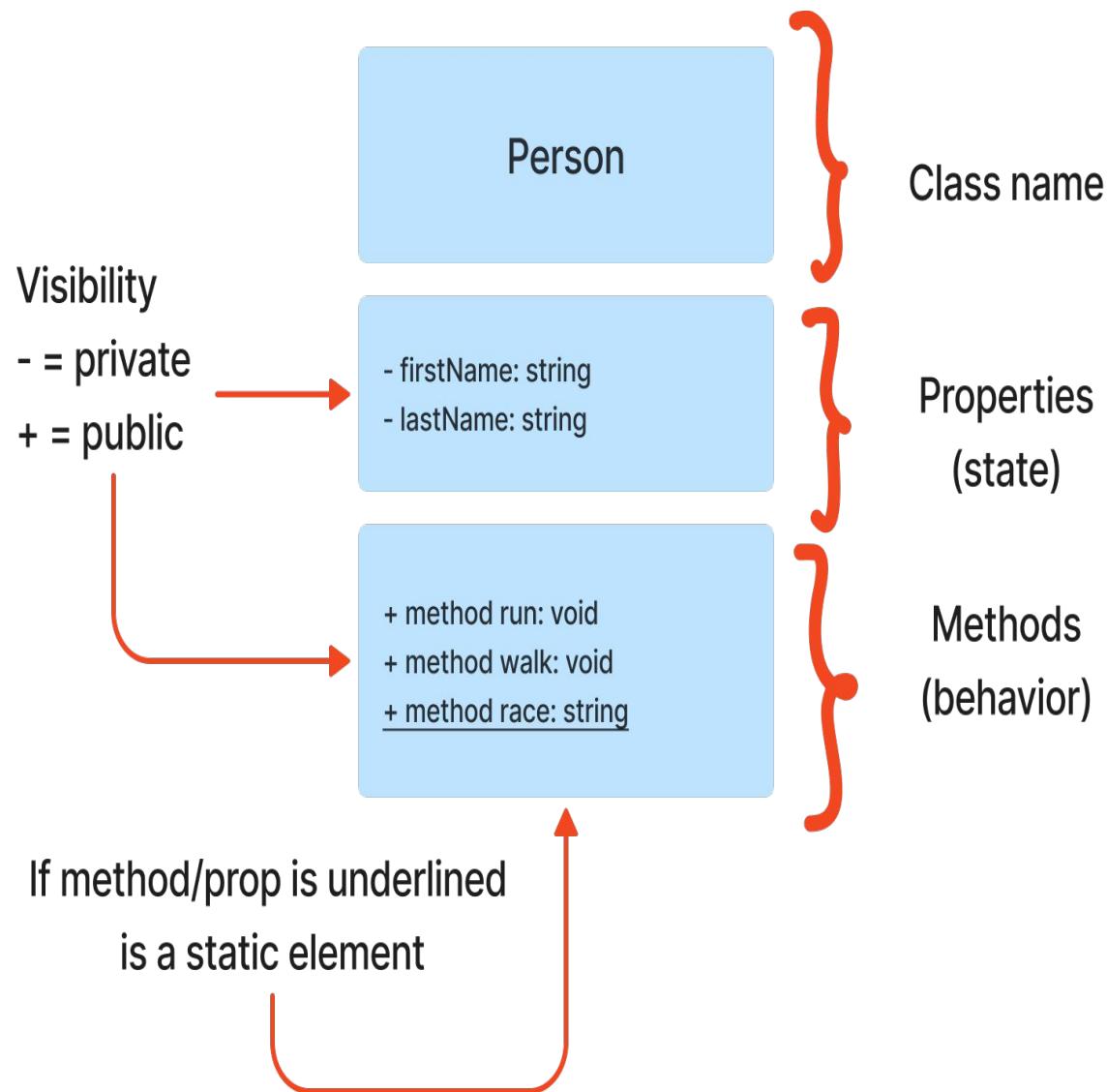
Fin de la clase 11

Relaciones entre clases



UML

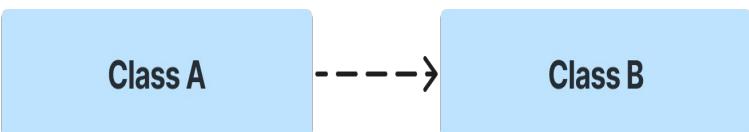
Unified Model Language



Relación de Dependencia

Hay una dependencia entre dos clases si al realizar cambios en alguna de ellas resulta en modificaciones en la otra.

Diagrama de clases



```
class Builder:  
    public method build(hammer: Tool) is  
        // If useIt method is renamed our code will fail  
        hammer.useIt()
```

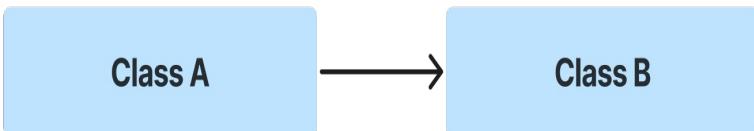
A screenshot of a code editor window with a dark theme. At the top, there are three colored circular icons: red, yellow, and green. Below them, the Java code for a Builder pattern is displayed. The code defines a class named 'Builder' with a single method 'build'. This method takes a parameter 'hammer' of type 'Tool'. Inside the method, a comment states: '// If useIt method is renamed our code will fail'. Following the comment, the code calls the 'useIt()' method on the 'hammer' object. The code is color-coded: 'Builder' and 'method' are in yellow, 'build' is in blue, 'hammer' is in orange, 'Tool' is in purple, and the comment and 'useIt()' call are in cyan.

Relación de Asociación

Hay una asociación entre clases si uno de ellos tiene acceso de forma permanente a los objetos de la otra. Esta relación está representada usualmente como un atributo de clase.

```
● ● ●  
class Builder:  
    // Hammer is available for all the class methods  
    private ownHammer: Tool  
  
    public method build() is  
        // If useIt method is renamed our code will fail  
        ownHammer.useIt()
```

Diagrama de clases



Relación de Implementación

La relación de implementación se da cuando una clase define su comportamiento con base en el definido en una interfaz.

Diagrama de clases

Class A



Class B

```
● ● ●

interface TextFormatter {
    public method format(text)
}

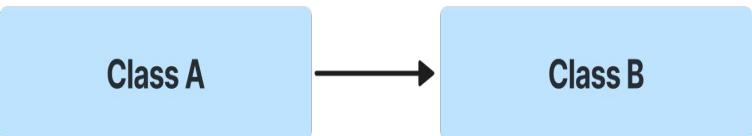
// Both classes based their behaviour in the interface
class PlainTextFormatter implements TextFormatter:
    public format(text) is
        print("PLAIN TEXT: " + text)

class ShortTextFormatter implements TextFormatter:
    public format(text) is
        print("SHORT TEXT: " + text)
```

Relación de Herencia

Hay una relación de herencia, cuando una clase es capaz de utilizar la misma interfaz e implementación de otra, pero con la posibilidad de extender su comportamiento.

Diagrama de clases



```
class Node:  
    private text: string  
    private next: Node  
  
class LinkedList:  
  
    private head: Node  
    private tail: Node  
  
    public method append(textToInsert) is  
        // Insert an element in the list  
  
    public method remove(textToDelete) is  
        // Delete the desired element in the list  
  
    // A queue that uses a linked list internally  
class QueueLinkedList extends LinkedList:  
  
    public method enqueue(text) is  
        // Call the method defined in the parent  
        parent.append(text)  
  
    public method dequeue(text) is  
        // Call the method defined in the parent  
        parent.remove(text)
```

A code snippet illustrating inheritance. It shows the definition of a Node class with private attributes for text and next, and a LinkedList class that extends Node. The LinkedList class has its own private attributes for head and tail, and two public methods: append and remove. Below it, a QueueLinkedList class is defined as an extension of LinkedList, overriding the enqueue and dequeue methods to call the corresponding methods on the parent class (LinkedList).

Relación de Agregación

Hay una relación de agregación entre clases si el objeto generado por una de ellas "tiene" uno o más objetos de la otra. La clase contenedora no gestiona el ciclo de vida de los objetos.

Diagrama de clases

Class A



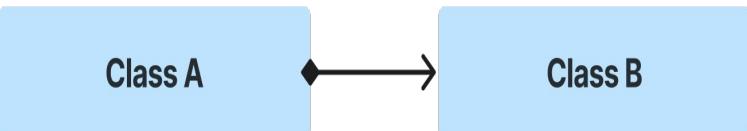
Class B

```
class Vehicle:  
  
    private engine: Engine  
    private wheels: Array<Wheel>  
  
    // We pass the objects as params,  
    // the class doesn't create the objects  
    constructor(newEngine, wheelsList) is  
        engine = newEngine  
        wheels = wheelsList  
  
    public method verifyWheels() is  
        for wheel in wheels:  
            var isWheelReady = wheel.verify()  
            if(!isWheelReady) return false  
        return true  
  
    public method turnOn() is  
        var allWheelsReady = verifyWheels()  
        if (!allWheelsReady) return  
        engine.start()
```

Relación de Composición

Hay una relación de composición entre clases si el objeto generado por una de ellas "consiste" de uno o más objetos de la otra. La clase contenedora gestiona el ciclo de vida de los objetos.

Diagrama de clases



```
class Vehicle:
```

```
    private engine: Engine  
    private wheels: Array<Wheel>
```

```
    constructor() is  
        engine = new Engine()  
        // This returns a list of wheels  
        wheels = new Wheel.get(4)
```

```
    public method verifyWheels() is  
        for wheel in wheels:  
            var isWheelReady = wheel.verify()  
            if(!isWheelReady) return false  
        return true
```

```
    public method turnOn() is  
        var allWheelsReady = verifyWheels()  
        if (!allWheelsReady) return  
        engine.start()
```

Fin de la clase 12

Mis dos centavos



A considerar...

1. Necesitas aprender sobre patrones de diseño para avanzar en el desarrollo de Software.
2. Conocimiento sobre patrones de diseño como requisito para posiciones en la industria.
3. Los patrones de diseño deben de dejar de ser enseñados usando POO y ser enseñados con paradigmas con mayor presencia (funcional) en el mercado.
4. Los patrones de diseño no solo te enseñan a solucionar un problema, sino también a pensar con creatividad.

Examen ✎

Review 👍

News! 😎

Gracias totales ❤️