Recibe una cálida:

# Bienvenida!

Te estábamos esperando 😁







# Polimorfismo y principios básicos de diseño

Plan formativo: Desarrollo de Aplicaciones Full Stack Java Trainee V2.0





# HOJA DE RUTA

¿Cuáles skill conforman el programa?









## REPASO CLASE ANTERIOR



En la clase anterior trabajamos 📚:

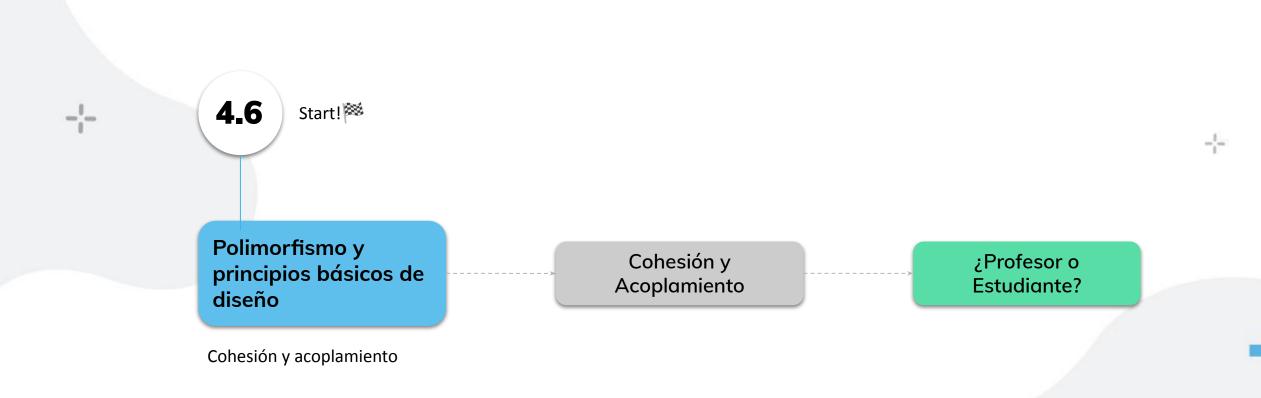
- Principios de diseño orientado a objetos
- Conceptos de mantenibilidad y reutilizabilidad







# LEARNING PATHWAY







# OBJETIVOS DE APRENDIZAJE

¿Qué aprenderemos?



Comprender la importancia e implementación de los conceptos de cohesión y acoplamiento.

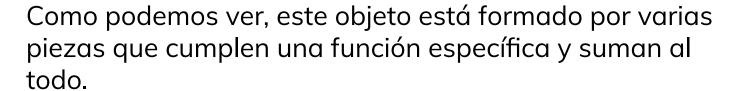








#### Piezas interconectadas: 🙌



#### Respondan en el chat o levantando la mano: 羔



- ¿Qué les sugiere esta imagen en términos de organización de componentes?
- ¿Cómo creen que se relaciona esta imagen con la programación Java?
- ¿Qué similitudes pueden encontrar entre la estructura de esta imagen y el código Java bien organizado?







# Cohesión



# Cohesión

La cohesión se refiere al **grado en que los miembros de una clase o componente de software están relacionados y trabajan juntos para lograr un objetivo específico**.

En otras palabras, una clase cohesiva se centra en una única responsabilidad o función.

#### Tipos de Cohesión:



- Cohesión Alta: Una clase con alta cohesión realiza una sola tarea o función bien definida.
   Es deseable, ya que facilita el mantenimiento y la reutilización del código.
- Cohesión Media: Una clase con cohesión media realiza varias tareas relacionadas, pero no de manera ideal. Puede ser necesario dividir la clase en clases más pequeñas y cohesivas.
- Cohesión **Baja**: Una clase con baja cohesión realiza tareas no relacionadas y suele ser difícil de entender y mantener.





#### Para lograr una alta cohesión:

- Asignar una única responsabilidad bien definida a cada clase.
- Mantener los elementos de una clase (atributos, métodos) enfocados en esa única responsabilidad.
- Dividir clases grandes en otras más pequeñas y cohesivas.
- Agrupar elementos lógicamente relacionados en una misma clase.
- No incluir funcionalidades que no estén estrictamente relacionadas con la responsabilidad principal de la clase.
- Evitar clases "utilidad" para funcionalidad variada no relacionada.
- Revisar continuamente que las clases sigan siendo cohesivas frente a cambios.

```
// Ejemplo de alta cohesión
class Calculator {
    public int add(int a, int b) {
        return a + b;
    public int subtract(int a, int b) {
        return a - b;
    public int multiply(int a, int b) {
        return a * b;
```









El acoplamiento se refiere a **la medida en que dos o más clases están interconectadas o dependen entre sí. Un acoplamiento bajo es deseable**, ya que reduce la dependencia entre clases y hace que el código sea más flexible y fácil de mantener.

#### Tipos de Acoplamiento:



- Acoplamiento Bajo: Las clases tienen poca o ninguna dependencia entre sí. Cambios en una clase tienen un impacto mínimo en otras clases.
- Acoplamiento Medio: Las clases tienen alguna dependencia entre sí, pero no están completamente entrelazadas. Se puede mejorar para reducir la dependencia.
- Acoplamiento **Alto**: Las clases están fuertemente interconectadas y dependen en gran medida unas de otras. Cambios en una clase pueden afectar muchas otras clases.





#### Para lograr un bajo acoplamiento:

- Definir interfaces o abstracciones que permitan la interacción sin depender de las implementaciones concretas.
- Cada clase debe tener una única responsabilidad bien definida y no debe mezclar múltiples responsabilidades.
- Organizar el código en módulos o paquetes independientes con responsabilidades claras.



- Evitar dependencias rígidas, como llamadas directas a métodos de otras clases.
- Usar variables locales o limitar el alcance de las variables al mínimo necesario.
- Reducir el uso de variables globales que pueden aumentar el acoplamiento.
- Mantener la lógica de negocio separada de la interfaz de usuario para facilitar cambios y pruebas.





```
><
```

```
// Mal: Acoplamiento alto
class Order {
   private MySQLDatabase database;
   public Order() {
        database = new MySQLDatabase();
   // ...
// Bien: Bajo acoplamiento
interface Database {
   void saveData();
class Order {
   private Database database;
    public Order(Database database) {
       this.database = database;
```





+

# Evaluación Integradora

¿Listos para un nuevo desafío? En esta clase comenzamos a construir nuestro....



Iremos completándolo progresivamente clase a clase.







## LIVE CODING

Ejemplo en vivo

#### Creando una Wallet cohesiva y de bajo acomplamiento:

Vamos a mejorar el diseño de la Wallet que gestiona el saldo y las transacciones de una billetera virtual. En este ejercicio, aplicaremos los principios de cohesión y acoplamiento para asegurarnos de que la clase sea modular y fácil de mantener.

#### Objetivo:

Diseñar la Wallet con alta cohesión y bajo acoplamiento para gestionar el saldo y las transacciones.





## LIVE CODING

Ejemplo en vivo

- 1. La clase CuentaBancaria debe tener una única responsabilidad: gestionar el saldo y las transacciones.
- 2. La clase CuentaBancaria debe proporcionar métodos para:
- Consultar el saldo actual.
- Realizar depósitos.
- Realizar retiros.
- Registrar transacciones.





## LIVE CODING

Ejemplo en vivo

- **3.** Crear una clase Transaccion con los atributos origen, destino y monto.
- **4.** Crear un método realizar() para la clase Transaccion que permita realizar la transacción.

Tiempo: 40 minutos







# Ejercicio N°1 ¿Profesor o Estudiante?





## ¿Profesor o Estudiante?

#### Aplicando cohesión y reduciendo el acomplamiento: 🙌

Para poder afianzar estos conceptos es necesario poder visualizarlos en código. A continuación veremos imágenes de un código determinado y deberás pensar las posibilidades para reducir su acoplamiento y aumentar su cohesión.

#### Consigna: 🚣

- 1. Crear las clases necesarias en tu IDE
- 2. Identificar los problemas de cohesión en las clases Student y Teacher.
- 3. Identificar la dependencia de la clase Student en la clase Teacher y viceversa.
- 4. Reorganizar el código para separar las responsabilidades de las clases y reducir el acoplamiento.
- 5. Crear una nueva clase, por ejemplo, Course, para gestionar la inscripción en cursos y separar esta funcionalidad de Student.

Tiempo : 35 minutos





#### ×

### **Manteniendo**

#### Analicemos el siguiente código

```
public class Student {
   private String name;
   private int age;
   private double gpa;
   private Teacher teacher;
   public Student(String name, int age, double gpa) {
       this.name = name;
       this.age = age;
       this.gpa = gpa;
   public void setTeacher(Teacher teacher) {
       this.teacher = teacher;
   3
   public void enrollInCourse(String courseName) {
       System.out.println(name + " has enrolled in " + courseName + ".");
       teacher.teachCourse(courseName);
```

```
public class Teacher {
    private String name;

public Teacher(String name) {
        this.name = name;
    }

public void teachCourse(String courseName) {
        System.out.println(name + " is teaching " + courseName + ".");
    }

public void provideFeedback() {
        System.out.println(name + " is providing feedback to the student.");
}
```





# ¿Profesor o Estudiante?

Como pudimos ver, el ejemplo tiene algunos "problemas"

#### Problemas de Cohesión:

- La clase Student tiene una responsabilidad mixta. Debería enfocarse en representar a un estudiante y su información académica, pero también está relacionada con la interacción con el profesor y la inscripción en cursos.
- La clase Teacher también tiene una responsabilidad mixta. Debería centrarse en la información del profesor, pero también se encarga de la enseñanza y la retroalimentación de los estudiantes.

#### Problemas de Acoplamiento:

• La clase Student y Teacher están fuertemente acopladas. Student depende de Teacher para realizar acciones relacionadas con cursos y calificaciones.





# ¿Alguna consulta?



## RESUMEN

¿Qué logramos en esta clase?



Comprender la importancia de los conceptos de cohesión y acoplamiento en la programación orientada a objetos.







## **#WorkingTime**

Continuemos ejercitando

#### ¡Antes de cerrar la clase! Te invitamos a: 👇 👇

1\_

- 1. Repasar nuevamente la grabación de esta clase
- 2. Revisar el material compartido en la plataforma de Moodle (lo que se vio en clase y algún ejercicio adicional)
  - a. Material 1 (Foro)
  - b. Lectura Módulo 4, Lección 6: página 8
- 3. Traer al próximo encuentro, todas tus dudas y consultas para verlas antes de iniciar nuevo tema.







-1-

# Muchas Gracias!

Nos vemos en la próxima clase 🤎



*M* alkemy

>:

Momento:

# Time-out!

**⊘**5 min.



