CÓDIGOS JAVA

```
public class DatabaseConnector {
        private static DatabaseConnector instance;

        private DatabaseConnector() {}

        public static DatabaseConnector getInstance() {
        if (instance == null) {
        instance = new DatabaseConnector();
        }
        return instance;
        }

        public void connect() {
        System.out.println("Connecting to database...");
        }
}
```

AA. ¿Qué patrón de diseño se implementa aquí y cuál es su propósito principal?

a) Builder - Crear objetos complejos paso a paso.

b) Singleton - Asegurar una única instancia de la clase.

c) Factory - Crear objetos sin especificar la clase exacta.

d) Prototype - Clonar objetos existentes.

```
public class Rectangle {
	protected double width;
	protected double height;

	public void setWidth(double width) {
	this.width = width;
	}

	public void setHeight(double height) {
	this.height = height;
	}

	public double area() {
	return width * height;
	}
}
```

AB. ¿Qué principio SOLID podría violarse si se extiende esta clase para crear una subclase Square?

a) Single Responsibility Principle
b) Open/Closed Principle
c) Liskov Substitution Principle
d) Interface Segregation Principle

```
public class Connection {
	public Connection(String type) {
	   if (type.equals("MySQL")) {
			// Conexión a MySQL
	   } else if (type.equals("PostgreSQL")) {
			// Conexión a PostgreSQL
	   }
			// Más condiciones para otros tipos de bases de datos
	}
}
```

AC. ¿Qué cambio aplicarías para optimizar el siguiente código usando el patrón Factory?

a) Crear una clase ConnectionFactory que encapsule la creación de las conexiones.
b) Convertir la clase Connection en abstracta y usar subclases.
c) Implementar el patrón Singleton en la clase Connection.
d) Ninguna de las anteriores.

```
public interface Worker {
	void work();
	void eat();
}
```

AD. Si una clase Robot implementa Worker pero solo usa el método work(), ¿qué principio SOLID se está violando?

a) Single Responsibility Principle
b) Open/Closed Principle
c) Liskov Substitution Principle
d) Interface Segregation Principle

```
public class PaymentProcessor {
        public void processPayment(PaymentData data, boolean debug) {
        if (debug) {
        System.out.println("Processing payment: " + data.getAmount());
        }
        // Lógica para procesar el pago
        }
}
```

AE. ¿Cuál es el principal problema aquí y cómo lo mejorarías?

a) Violación del Single Responsibility Principle; separar la lógica de depuración y procesamiento.
b) Uso inapropiado del polimorfismo; implementar distintas clases para cada tipo de procesamiento.
c) Dependencia directa de una implementación; usar inyección de dependencias para mejorar la flexibilidad.
d) Hardcoding; remover la bandera de depuración y utilizar un logger.

```
public class OrderService {
        private EmailService emailService;

        public OrderService() {
        this.emailService = new EmailService();
        }

        public void confirmOrder(Order order) {
        // Confirmar orden
        emailService.sendEmail(order);
        }
}
```

AF. ¿Cómo deberías modificar OrderService para adherir al DIP?
a) Crear una fábrica que devuelva EmailService.
b) Cambiar EmailService por una interfaz y usar inyección de dependencias para la implementación.
c) Implementar el patrón Singleton en EmailService.
d) Ninguna de las anteriores.

```
class Iniciar {
	public static void main(String[] args) {
	Ambiente a = new Ambiente();
	a.ejecutar();
	a.cambiar();
	a.ejecutar();
	}
}

class Musico extends Object {
	public void tocar() {
	}
}

public class Guitarrista extends Musico {
	@Override
	public void tocar() {
	System.out.println("Guitarrista tocando.");
	}
}

public class Bajista extends Musico {
	@Override
	public void tocar() {
	System.out.println("Bajista tocando.");
	}
}

public class Ambiente {
	private Musico m = new Guitarrista();

	public void cambiar() {
	m = new Bajista();
	}

	public void ejecutar() {
	m.tocar();
	}
}
```

AG.
a) Guitarrista tocando.
Guitarrista tocando.
b) Bajista tocando.
Bajista tocando.
c) Guitarrista tocando.
Bajista tocando.
d) Bajista tocando.
Guitarrista tocando.

```
class Musico extends Object {
	public void tocar() {
	}
}

public class Guitarrista extends Musico {
	@Override
	public void tocar() {
	System.out.println("Guitarrista tocando.");
	}
}

public class Bajista extends Musico {
	@Override
	public void tocar() {
	System.out.println("Bajista tocando.");
	}
}

public class Ambiente {
	private Musico m = new Guitarrista();

	public void cambiar() {
	m = new Bajista();
	}

	public void ejecutar() {
	m.tocar();
	}
}
```

AH

a) Violación del Principio de Segregación de Interfaces (ISP) y del Principio de Sustitución de Liskov (LSP)

b) Violación del Principio de Responsabilidad Única (SRP) y del Principio de Sustitución de Liskov (LSP)

c) Violación del Principio de Inversión de Dependencias (DIP) y del Principio de Abierto/Cerrado (OCP)

d) Ninguna de las anteriores, el código está correctamente diseñado.

Pregunta: Considera el siguiente programa en Java. ¿Qué modificación harías para optimizar el código siguiendo el principio de abierto/cerrado?

```
class EjecutarReporte {
        public static void main(String[] args) {
        ReporteGenerador rg = new ReporteGenerador();
        rg.generarReporte("HTML");
        rg.generarReporte("PDF");
        }
}

class ReporteGenerador {
        public void generarReporte(String tipo) {
        if (tipo.equals("HTML")) {
        System.out.println("Generando reporte en formato HTML.");
        } else if (tipo.equals("PDF")) {
        System.out.println("Generando reporte en formato PDF.");
        }
        // Supongamos que hay varios otros formatos de reporte
        }
}
```

AI.
¿Cómo debería modificarse ReporteGenerador para que siga el principio de abierto/cerrado?

a) Añadiendo más condicionales if para cada nuevo formato de reporte.
b) Creando una clase abstracta o interfaz Reporte con un método generar(), y subclases concretas para cada formato de reporte.
c) Cambiando el tipo de reporte de String a enum para garantizar tipos seguros.
d) Manteniendo el código como está porque ya sigue el principio de abierto/cerrado.

AJ - Pregunta: Considera el siguiente programa en Java. ¿Qué implementación del método `crearAnimal` en la clase `Zoologico` utilizarías para optimizar el código siguiendo el patrón Factory Method y facilitar la adición de nuevos tipos de animales en el futuro?

```
import java.util.ArrayList;

abstract class Animal {
        public abstract void hacerSonido();
}

class Leon extends Animal {
        public void hacerSonido() {
        System.out.println("Roar");
        }
}

class Serpiente extends Animal {
        public void hacerSonido() {
        System.out.println("Sss");
        }
}
```

```
class Zoologico {
        private ArrayList<Animal> animales = new ArrayList<>();

        public void agregarAnimal(Animal animal) {
        animales.add(animal);
        }

        // Método a implementar
        public Animal crearAnimal(String tipo) {
        // Implementación pendiente
        }

        public void demostrarSonidos() {
        for (Animal animal : animales) {
        animal.hacerSonido();
        }
        }
}

public class DemoZoologico {
        public static void main(String[] args) {
        Zoologico zoo = new Zoologico();
        zoo.agregarAnimal(zoo.crearAnimal("Leon"));
        zoo.agregarAnimal(zoo.crearAnimal("Serpiente"));
        zoo.demostrarSonidos();
        }
}
```

¿Cuál sería la implementación adecuada para el método `crearAnimal`?

a)

```
public Animal crearAnimal(String tipo) {
        if (tipo.equals("Leon")) {
        return new Leon();
        } else if (tipo.equals("Serpiente")) {
        return new Serpiente();
        }
        return null;
}
```

b)

```
public Animal crearAnimal(String tipo) {
      switch (tipo) {
      case "Leon":
            return new Leon();
      case "Serpiente":
            return new Serpiente();
      default:
```

```
            throw new IllegalArgumentException("Tipo
desconocido");
      }
}
```

c)

```
public Animal crearAnimal(String tipo) {
      return switch (tipo) {
      case "Leon" -> new Leon();
      case "Serpiente" -> new Serpiente();
      default -> throw new IllegalArgumentException("Tipo
desconocido");
      };
}
```

d)

```
// Este código asume la existencia de un mapa o algún otro mecanismo para manejar la
instanciación.
public Animal crearAnimal(String tipo) {
      return AnimalFactory.getAnimal(tipo);
}
```

AK. Pregunta: Considera el siguiente programa en Java. Analiza la implementación de los patrones de diseño y los principios SOLID, y selecciona la opción que mejora y completa correctamente el código, teniendo en cuenta la integración con un `ResultSet`.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

interface ProcesadorDatos {
        void procesar(ResultSet resultSet) throws SQLException;
}

class ProcesadorUsuarios implements ProcesadorDatos {
        @Override
        public void procesar(ResultSet resultSet) throws SQLException {
          while (resultSet.next()) {
            System.out.println("Usuario: " + resultSet.getString("nombre"));
           }
        }
}

class ProcesadorProductos implements ProcesadorDatos {
        @Override
        public void procesar(ResultSet resultSet) throws SQLException {
        while (resultSet.next()) {
        System.out.println("Producto: " + resultSet.getString("nombre") + ", Precio: " +
resultSet.getDouble("precio"));
        }
        }
}

class FabricaProcesadores {
        public static ProcesadorDatos crearProcesador(String tipo) {
        if (tipo.equals("usuarios")) {
          return new ProcesadorUsuarios();
        } else if (tipo.equals("productos")) {
          return new ProcesadorProductos();
        } else {
          throw new IllegalArgumentException("Tipo de procesador no soportado.");
        }
        }
}

class ConexionDB {
        private static ConexionDB instancia;
        private Connection conexion;

        private ConexionDB() {
        try {
        this.conexion =
DriverManager.getConnection("jdbc:mysql://localhost:3306/mi_base_datos", "usuario",
```

```
"contraseña");
        } catch (SQLException e) {
        e.printStackTrace();
        }
        }

        public static ConexionDB getInstancia() {
        if (instancia == null) {
            instancia = new ConexionDB();
        }
        return instancia;
        }

        public void ejecutarConsulta(String consulta, String tipoProcesador) {
        try (Statement statement = conexion.createStatement();
        ResultSet resultSet = statement.executeQuery(consulta)) {

        ProcesadorDatos procesador =
FabricaProcesadores.crearProcesador(tipoProcesador);
        procesador.procesar(resultSet);

        } catch (SQLException e) {
        e.printStackTrace();
        }
        }
}

public class SistemaInformacion {
        public static void main(String[] args) {
        ConexionDB conexionDB = ConexionDB.getInstancia();
        conexionDB.ejecutarConsulta("SELECT * FROM usuarios", "usuarios");
        conexionDB.ejecutarConsulta("SELECT * FROM productos", "productos");
        }
}
```

AK
¿Cuál de las siguientes mejoras sería la mejor práctica para optimizar el código?

a) Hacer que ConexionDB permita múltiples instancias en lugar de un Singleton, para mejorar la escalabilidad en consultas concurrentes.

b) Modificar FabricaProcesadores para almacenar las instancias creadas en un ArrayList, evitando crear nuevos objetos en cada consulta y mejorando la eficiencia.

c) Convertir ProcesadorDatos en una clase abstracta en lugar de una interfaz, permitiendo reutilizar código común en todos los procesadores.

d) Separar la lógica de ConexionDB en dos clases: una que maneje la conexión (ConexionDB) y otra que maneje la ejecución de consultas (GestorConsultas), aplicando el Principio de Responsabilidad Única (SRP).

AL. Pregunta: Considera el siguiente programa en Java. Revisa la implementación del polimorfismo y la vinculación dinámica, y selecciona la opción que mejora el código adheriéndose a los principios SOLID.

```
abstract class Empleado {
	abstract void trabajar();
	abstract void reportar();
}

class Desarrollador extends Empleado {
	@Override
	void trabajar() {
	System.out.println("Desarrollando software.");
	}

	@Override
	void reportar() {
	System.out.println("Reportando el progreso del desarrollo.");
	}
}

class Gerente extends Empleado {
	@Override
	void trabajar() {
	System.out.println("Gestionando el equipo.");
	}

	@Override
	void reportar() {
	System.out.println("Reportando la gestión del equipo.");
	}
}

class Empresa {
	private List<Empleado> empleados;

	public Empresa() {
	this.empleados = new ArrayList<>();
	}

	public void agregarEmpleado(Empleado empleado) {
	empleados.add(empleado);
	}

	public void gestionarEmpresa() {
	for (Empleado empleado : empleados) {
	empleado.trabajar();
	empleado.reportar();
	}
	}
}

public class SistemaGestion {
	public static void main(String[] args) {
```

```
        Empresa empresa = new Empresa();
        empresa.agregarEmpleado(new Desarrollador());
        empresa.agregarEmpleado(new Gerente());
        empresa.gestionarEmpresa();
        }
}
```

Dada la estructura del programa, ¿cuál de las siguientes opciones representa la mejor mejora para adherirse a los principios SOLID mientras se mantiene el polimorfismo y la vinculación dinámica?

a) (Principio de Abierto/Cerrado - OCP)
Implementar una fábrica de Empleado que instancie Desarrollador o Gerente basándose en parámetros de entrada. Esto permite que el sistema pueda extenderse fácilmente con nuevos tipos de empleados sin modificar el código existente, manteniendo la flexibilidad y escalabilidad del diseño.

b) (Principio de Sustitución de Liskov - LSP)
Crear una interfaz Gestionable que contenga los métodos trabajar() y reportar(), y hacer que las clases Desarrollador y Gerente la implementen. Esto garantiza que las subclases puedan ser utilizadas en lugar de la superclase sin alterar el comportamiento esperado del programa.

c) (Principio de Segregación de Interfaces - ISP)
Separar las responsabilidades de trabajar() y reportar() en diferentes interfaces, permitiendo que cada clase implemente solo las interfaces necesarias. Esto evita que las clases estén obligadas a implementar métodos que no utilizan, asegurando que cada interfaz sea específica para su propósito.

d) (Principio de Responsabilidad Única - SRP)
Introducir un método descansar() en la clase abstracta Empleado, asegurando que todos los empleados tengan este comportamiento. Sin embargo, esto podría violar SRP si no todos los empleados requieren descanso, ya que la clase Empleado estaría asumiendo una responsabilidad adicional innecesaria.

AM. Pregunta: Analiza el siguiente programa Java completo que tiene varios problemas relacionados con los principios SOLID. Elige la opción de respuesta que mejor refactore el código aplicando estos principios.

```
import java.util.List;
import java.util.ArrayList;

class Estudiante {
        String nombre;
        List<String> cursos;

        public Estudiante(String nombre) {
        this.nombre = nombre;
        this.cursos = new ArrayList<>();
        }

        public void agregarCurso(String curso) {
        cursos.add(curso);
        }

        public void imprimirCursos() {
        System.out.println("Cursos de " + nombre + ":");
        for (String curso : cursos) {
        System.out.println(curso);
        }
        }

        public void guardarEstudiante() {
        // Código para guardar el estudiante en una base de datos
        System.out.println("Guardando " + nombre + " en la base de datos.");
        }
}

public class RegistroEstudiantes {
        public static void main(String[] args) {
        Estudiante estudiante = new Estudiante("Juan");
        estudiante.agregarCurso("Matemáticas");
        estudiante.agregarCurso("Historia");
        estudiante.imprimirCursos();
        estudiante.guardarEstudiante();
        }
}
```

El código anterior viola varios principios SOLID. ¿Cuál de las siguientes opciones refactoriza mejor el código respetando estos principios?

a) Aplicar el principio de inversión de dependencias (DIP):

```
class Estudiante {
        String nombre;
        List<String> cursos;
        BaseDatos baseDatos;

        public Estudiante(String nombre, BaseDatos baseDatos) {
```

```
        this.nombre = nombre;
        this.cursos = new ArrayList<>();
        this.baseDatos = baseDatos;
        }

        public void agregarCurso(String curso) {
        cursos.add(curso);
        }

        public void imprimirCursos() {
        System.out.println("Cursos de " + nombre + ":");
        for (String curso : cursos) {
        System.out.println(curso);
        }
        }

        public void guardarEstudiante() {
        baseDatos.guardar(this);
        }
}

interface BaseDatos {
        void guardar(Estudiante estudiante);
}

class EstudianteDB implements BaseDatos {
        public void guardar(Estudiante estudiante) {
        System.out.println("Guardando " + estudiante.nombre + " en la base de datos.");
        }
}
```

b) Refactorizar aplicando el principio abierto/cerrado (OCP) y el principio de responsabilidad única (SRP):

```
class Estudiante {
        String nombre;
        List<String> cursos;

        public Estudiante(String nombre) {
        this.nombre = nombre;
        this.cursos = new ArrayList<>();
        }

        public void agregarCurso(String curso) {
        cursos.add(curso);
        }

        public void imprimirCursos() {
        System.out.println("Cursos de " + nombre + ":");
```

```
        for (String curso : cursos) {
        System.out.println(curso);
        }
        }
}

class GestorEstudiante {
        Estudiante estudiante;

        public GestorEstudiante(Estudiante estudiante) {
        this.estudiante = estudiante;
        }

        public void guardarEstudiante() {
        // Código para guardar el estudiante en una base de datos
        System.out.println("Guardando " + estudiante.nombre + " en la base de datos.");
        }
}
```

c) Separar responsabilidades (SRP) y aplicar segregación de interfaces (ISP):

```
interface Imprimible {
        void imprimir();
}

interface Persistible {
        void guardar();
}

class Estudiante implements Imprimible {
        String nombre;
        List<String> cursos;

        public Estudiante(String nombre) {
        this.nombre = nombre;
        this.cursos = new ArrayList<>();
        }

        public void agregarCurso(String curso) {
        cursos.add(curso);
        }

        @Override
        public void imprimir() {
        System.out.println("Cursos de " + nombre + ":");
        for (String curso : cursos) {
        System.out.println(curso);
        }
        }
```

```
}

class EstudianteDB implements Persistible {
        Estudiante estudiante;

        public EstudianteDB(Estudiante estudiante) {
        this.estudiante = estudiante;
        }

        @Override
        public void guardar() {
        System.out.println("Guardando " + estudiante.nombre + " en la base de datos.");
        }
}
```

d) Ninguna opción es correcta.

AN. Pregunta: Analiza el siguiente programa Java que utiliza clases abstractas, polimorfismo y vinculación dinámica. Determina qué cambios serían adecuados para mejorar el diseño del código.

```
abstract class Vehiculo {
        abstract void arrancar();
}

class Coche extends Vehiculo {
        @Override
        void arrancar() {
        System.out.println("Coche arrancando");
        }
}

class Motocicleta extends Vehiculo {
        @Override
        void arrancar() {
        System.out.println("Motocicleta arrancando");
        }
}

public class PruebaVehiculos {
        public static void main(String[] args) {
        List<Vehiculo> vehiculos = Arrays.asList(new Coche(), new Motocicleta());
        for (Vehiculo vehiculo : vehiculos) {
        vehiculo.arrancar();
        }
        }
}
```

Considera el código proporcionado. ¿Cuál de las siguientes opciones refinaría el diseño del programa manteniendo el polimorfismo y la vinculación dinámica?

a) Implementar una interfaz en lugar de una clase abstracta:

```
interface Vehiculo {
        void arrancar();
}

class Coche implements Vehiculo {
        @Override
        public void arrancar() {
        System.out.println("Coche arrancando");
        }
}

class Motocicleta implements Vehiculo {
        @Override
        public void arrancar() {
        System.out.println("Motocicleta arrancando");
        }
}
```

```
public class PruebaVehiculos {
	public static void main(String[] args) {
	List<Vehiculo> vehiculos = Arrays.asList(new Coche(), new Motocicleta());
	for (Vehiculo vehiculo : vehiculos) {
	vehiculo.arrancar();
	}
	}
}
```

b) Agregar un método concreto a la clase abstracta:

```
abstract class Vehiculo {
	abstract void arrancar();

	void detener() {
	System.out.println("Vehículo detenido");
	}
}

class Coche extends Vehiculo {
	@Override
	void arrancar() {
	System.out.println("Coche arrancando");
	}
}

class Motocicleta extends Vehiculo {
	@Override
	void arrancar() {
	System.out.println("Motocicleta arrancando");
	}
}

public class PruebaVehiculos {
	public static void main(String[] args) {
	List<Vehiculo> vehiculos = Arrays.asList(new Coche(), new Motocicleta());
	for (Vehiculo vehiculo : vehiculos) {
	vehiculo.arrancar();
	vehiculo.detener();
	}
	}
}
```

c) Utilizar el polimorfismo para incluir más comportamientos:

```
abstract class Vehiculo {
```

```
        abstract void arrancar();
        abstract void tocarBocina();
}

class Coche extends Vehiculo {
        @Override
        void arrancar() {
        System.out.println("Coche arrancando");
        }

        @Override
        void tocarBocina() {
        System.out.println("Coche bocina: Beep beep!");
        }
}

class Motocicleta extends Vehiculo {
        @Override
        void arrancar() {
        System.out.println("Motocicleta arrancando");
        }

        @Override
        void tocarBocina() {
        System.out.println("Motocicleta bocina: Beep beep!");
        }
}

public class PruebaVehiculos {
        public static void main(String[] args) {
        List<Vehiculo> vehiculos = Arrays.asList(new Coche(), new Motocicleta());
        for (Vehiculo vehiculo : vehiculos) {
        vehiculo.arrancar();
        vehiculo.tocarBocina();
        }
        }
}
```

d) Eliminar la clase abstracta y utilizar clases concretas:

```
class Coche {
        void arrancar() {
        System.out.println("Coche arrancando");
        }
}

class Motocicleta {
        void arrancar() {
```

```
        System.out.println("Motocicleta arrancando");
        }
}

public class PruebaVehiculos {
        public static void main(String[] args) {
        List<Object> vehiculos = Arrays.asList(new Coche(), new Motocicleta());
        for (Object obj : vehiculos) {
        if (obj instanceof Coche) {
                ((Coche) obj).arrancar();
        } else if (obj instanceof Motocicleta) {
                ((Motocicleta) obj).arrancar();
        }
        }
        }
}
```

AÑ. La relación entre `Cancion` y `Categoria` en UML se representa como **una relación de asociación 1 a 1**, ya que **cada canción pertenece a una única categoría**.

📌 **Representación en Caracteres ASCII (Texto)**

lua
CopiarEditar

```
+------------+         +-------------+
|  Cancion   | 1 ----> |  Categoria  |
+------------+         +-------------+
| - titulo   |         | - nombre    |
| - artista  |         +-------------+
| - categoria : Categoria |
+------------+
```

Dado el siguiente diseño UML, donde una clase Cancion se relaciona con una única Categoria, ¿cómo debería implementarse la relación en Java?

a) Usar una relación de composición para asegurar que cada Cancion tenga obligatoriamente una Categoria y esta se destruya con la canción.

```
class Categoria {
    private String nombre;

    public Categoria(String nombre) {
        this.nombre = nombre;
    }

    public String getNombre() {
```

```
        return nombre;
    }
}

class Cancion {
    private String titulo;
    private String artista;
    private final Categoria categoria;

    public Cancion(String titulo, String artista, String nombreCategoria) {
        this.titulo = titulo;
        this.artista = artista;
        this.categoria = new Categoria(nombreCategoria); // Composición
    }

    public String getCategoria() {
        return categoria.getNombre();
    }
}
```

---

b) Usar una relación de agregación permitiendo que Categoria pueda existir independientemente de Cancion.

```
class Categoria {
    private String nombre;

    public Categoria(String nombre) {
        this.nombre = nombre;
    }

    public String getNombre() {
        return nombre;
    }
}

class Cancion {
    private String titulo;
    private String artista;
    private Categoria categoria;

    public Cancion(String titulo, String artista, Categoria categoria) {
        this.titulo = titulo;
        this.artista = artista;
        this.categoria = categoria; // Agregación
    }

    public String getCategoria() {
        return categoria.getNombre();
    }
}
```

---

c) Implementar la relación con herencia haciendo que Cancion extienda de Categoria.

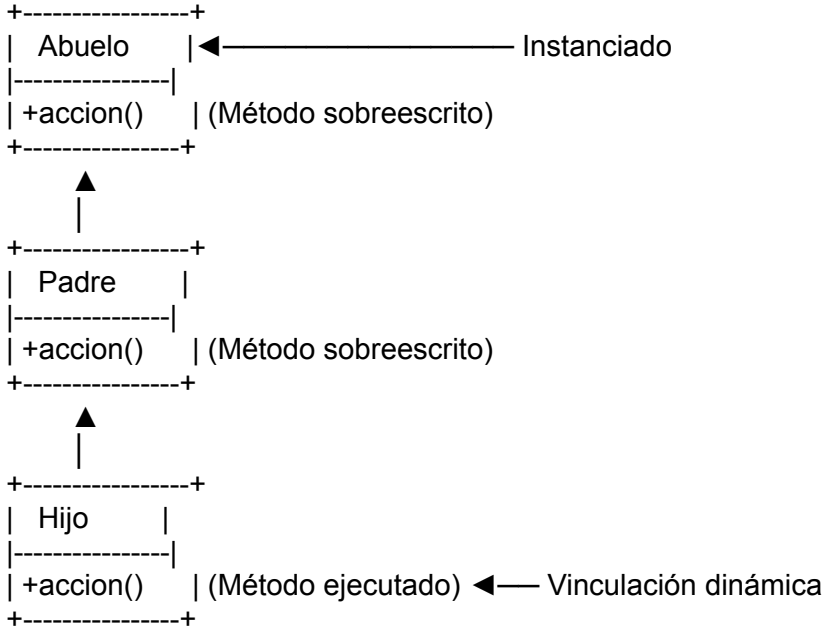
```
class Categoria {
    protected String nombre;
}

class Cancion extends Categoria {
    private String titulo;
    private String artista;

    public Cancion(String titulo, String artista, String nombreCategoria) {
        this.titulo = titulo;
        this.artista = artista;
        this.nombre = nombreCategoria; // Uso de herencia
    }

    public String getCategoria() {
        return nombre;
    }
}
```

---

d) Usar una relación de dependencia sin que Cancion tenga una referencia directa a Categoria.

```
class Categoria {
    private String nombre;

    public Categoria(String nombre) {
        this.nombre = nombre;
    }

    public String getNombre() {
        return nombre;
    }
}

class Cancion {
    private String titulo;
    private String artista;

    public Cancion(String titulo, String artista) {
        this.titulo = titulo;
        this.artista = artista;
    }

    public String getCategoria(Categoria categoria) {
        return categoria.getNombre();
    }
}
```

AO



Dado el siguiente código que implementa un sistema de herencia con vinculación dinámica en Java:

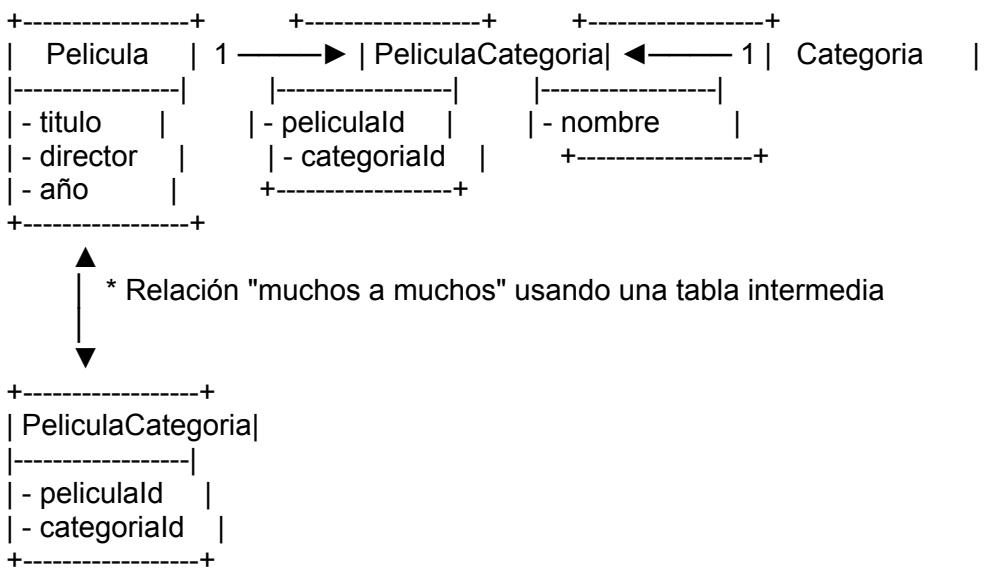
```
class Abuelo {
    void accion() {
        System.out.println("Acción del Hijo");
    }
}

class Padre extends Abuelo {
    @Override
    void accion() {
        System.out.println("Acción del Padre");
    }
}

class Hijo extends Padre {
    @Override
    void accion() {
        System.out.println("Acción del Abuelo");
    }
}

public class TestHerencia {
    public static void main(String[] args) {
        Abuelo ref = new Hijo();
        ref.accion();
    }
}
```

Pregunta:
¿Qué se imprimirá en la salida del programa al ejecutar el método `accion()` desde la referencia de tipo `Abuelo` que apunta a una instancia de `Hijo`?

---

a) Acción del Abuelo
b) Acción del Padre
c) Acción del Hijo
d) Error en tiempo de compilación

AP



Dado que **una película puede tener múltiples categorías** y **cada categoría puede incluir varias películas**, ¿cómo se debe implementar correctamente la relación en Java?

Opciones de Código Java

a) Usar una lista (`List<Categoria>`) dentro de `Pelicula` para representar la relación "muchos a muchos".

```
import java.util.List;
import java.util.ArrayList;

class Categoria {
    private String nombre;

    public Categoria(String nombre) {
        this.nombre = nombre;
    }
```

```
    public String getNombre() {
        return nombre;
    }
}

class Pelicula {
    private String titulo;
    private List<Categoria> categorias;

    public Pelicula(String titulo) {
        this.titulo = titulo;
        this.categorias = new ArrayList<>();
    }

    public void agregarCategoria(Categoria categoria) {
        categorias.add(categoria);
    }

    public List<Categoria> getCategorias() {
        return categorias;
    }
}
```

---

b) Usar una tabla intermedia `PeliculaCategoria` para representar la relación en una base de datos.

```
import java.util.List;

class Pelicula {
    private String titulo;

    public Pelicula(String titulo) {
        this.titulo = titulo;
    }

    public String getTitulo() {
        return titulo;
    }
}

class Categoria {
    private String nombre;

    public Categoria(String nombre) {
        this.nombre = nombre;
    }

    public String getNombre() {
        return nombre;
    }
}
```

```
// Clase intermedia para modelar la relación
class PeliculaCategoria {
    private Pelicula pelicula;
    private Categoria categoria;

    public PeliculaCategoria(Pelicula pelicula, Categoria categoria) {
        this.pelicula = pelicula;
        this.categoria = categoria;
    }

    public Pelicula getPelicula() {
        return pelicula;
    }

    public Categoria getCategoria() {
        return categoria;
    }
}
```

---

c) Usar herencia haciendo que `Pelicula` extienda de `Categoria`.

```
class Categoria {
    protected String nombre;
}

class Pelicula extends Categoria {
    private String titulo;

    public Pelicula(String titulo, String nombreCategoria) {
        this.titulo = titulo;
        this.nombre = nombreCategoria;
    }

    public String getCategoria() {
        return nombre;
    }
}
```

---

d) Relacionar `Categoria` con `Pelicula` sin listas, solo con una referencia única.

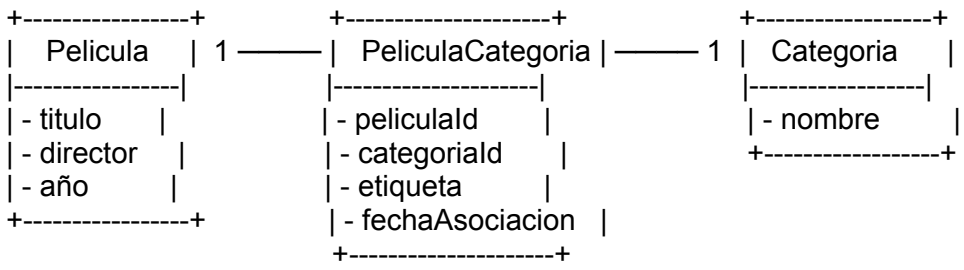
```
class Pelicula {
    private String titulo;
    private Categoria categoria; // Solo una categoría por película

    public Pelicula(String titulo, Categoria categoria) {
        this.titulo = titulo;
        this.categoria = categoria;
```

```
    }

    public String getCategoria() {
        return categoria.getNombre();
    }
}
```

📌 **Pregunta:**
Dado que **una película puede tener múltiples categorías** y **cada categoría puede incluir varias películas**, **¿cuál es la mejor manera de representar esta relación en Java?**

✔ **Opciones de Respuesta:**
a) Usar una `List<Categoria>` dentro de `Pelicula`.
b) Crear una clase `PeliculaCategoria` como tabla intermedia.
c) Hacer que `Pelicula` extienda de `Categoria`.
d) Relacionar `Pelicula` y `Categoria` con una referencia única en `Pelicula`.

AQ



¿Cuál es la mejor manera de implementar esta relación en Java?

Opciones de Código Java

a) Implementar una relación de agregación usando `List<Categoria>` en `Pelicula` (Sin clase intermedia)

```java
import java.util.List;
import java.util.ArrayList;

class Categoria {
    private String nombre;

    public Categoria(String nombre) {
        this.nombre = nombre;
```

```
    }

    public String getNombre() {
        return nombre;
    }
}

class Pelicula {
    private String titulo;
    private List<Categoria> categorias;

    public Pelicula(String titulo) {
        this.titulo = titulo;
        this.categorias = new ArrayList<>();
    }

    public void agregarCategoria(Categoria categoria) {
        categorias.add(categoria);
    }

    public List<Categoria> getCategorias() {
        return categorias;
    }

    public String getTitulo() {
        return titulo;
    }
}
```

---

b) Implementar la relación con una clase intermedia (`PeliculaCategoria`)

```
import java.util.ArrayList;
import java.util.List;

class Categoria {
    private String nombre;

    public Categoria(String nombre) {
        this.nombre = nombre;
    }

    public String getNombre() {
        return nombre;
    }
```

```
}

class Pelicula {
    private String titulo;

    public Pelicula(String titulo) {
        this.titulo = titulo;
    }

    public String getTitulo() {
        return titulo;
    }
}

// Clase intermedia que almacena información extra sobre la
relación
class PeliculaCategoria {
    private Pelicula pelicula;
    private Categoria categoria;
    private String etiqueta; // Ejemplo: "Principal", "Secundaria"
    private String fechaAsociacion;

    public PeliculaCategoria(Pelicula pelicula, Categoria
categoria, String etiqueta, String fechaAsociacion) {
        this.pelicula = pelicula;
        this.categoria = categoria;
        this.etiqueta = etiqueta;
        this.fechaAsociacion = fechaAsociacion;
    }

    public Pelicula getPelicula() {
        return pelicula;
    }

    public Categoria getCategoria() {
        return categoria;
    }

    public String getEtiqueta() {
        return etiqueta;
    }

    public String getFechaAsociacion() {
        return fechaAsociacion;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Pelicula peli1 = new Pelicula("El Padrino");
        Categoria drama = new Categoria("Drama");
        Categoria mafia = new Categoria("Mafia");

        List<PeliculaCategoria> relaciones = new ArrayList<>();
        relaciones.add(new PeliculaCategoria(peli1, drama,
"Principal", "2024-05-01"));
        relaciones.add(new PeliculaCategoria(peli1, mafia,
"Secundaria", "2024-05-02"));

        System.out.println("Película: " + peli1.getTitulo());
        for (PeliculaCategoria pc : relaciones) {
            System.out.println("Categoría: " +
pc.getCategoria().getNombre() +
                               ", Etiqueta: " + pc.getEtiqueta() +
                               ", Fecha Asociación: " +
pc.getFechaAsociacion());
        }
    }
}
```

---

c) Implementar la relación con herencia, haciendo que `Pelicula` extienda de `Categoria`

```
class Categoria {
    protected String nombre;
}

class Pelicula extends Categoria {
    private String titulo;

    public Pelicula(String titulo, String nombreCategoria) {
        this.titulo = titulo;
        this.nombre = nombreCategoria; // Uso de herencia
    }

    public String getCategoria() {
        return nombre;
    }
}
```

---

d) Relacionar `Categoria` con `Pelicula` sin listas, solo con una referencia única

```
class Pelicula {
    private String titulo;
    private Categoria categoria; // Solo una categoría por
película

    public Pelicula(String titulo, Categoria categoria) {
        this.titulo = titulo;
        this.categoria = categoria;
    }

    public String getCategoria() {
        return categoria.getNombre();
    }
}
```

AR

Pregunta Tipo Test: Uso de instanceof en Java

```
class Empleado {
   void trabajar() {
      System.out.println("Empleado trabajando.");
   }
}

class Gerente extends Empleado {
   void gestionar() {
      System.out.println("Gerente gestionando el equipo.");
   }
}

class Desarrollador extends Empleado {
   void programar() {
      System.out.println("Desarrollador escribiendo código.");
   }
}

public class Empresa {
   public static void main(String[] args) {
      Empleado e1 = new Gerente();
      Empleado e2 = new Desarrollador();
```

```
        if (e1 instanceof Gerente) {
            ((Gerente) e1).gestionar();
        }

        if (e2 instanceof Desarrollador) {
            ((Desarrollador) e2).programar();
        }
    }
}
```

📌 **Pregunta:**
¿Qué se imprimirá en la salida al ejecutar este código?

✔ Opciones de Respuesta:

a)

```
Empleado trabajando.

Empleado trabajando.
```

b)

```
Gerente gestionando el equipo.

Empleado trabajando.
```

c)

```
Gerente gestionando el equipo.

Desarrollador escribiendo código.
```

d) Error en tiempo de compilación debido a una conversión inválida.

AS

```
import java.util.ArrayList;

class Producto {
    private String nombre;

    public Producto(String nombre) {
        this.nombre = nombre;
    }

    public String getNombre() {
        return nombre;
    }
```

```
}

public class Supermercado {
    public static void main(String[] args) {
        ArrayList<Producto> productos = new ArrayList<>();

        productos.add(new Producto("Manzana"));
        productos.add(new Producto("Pan"));
        productos.add(new Producto("Leche"));

        productos.remove(1); // Elimina el producto en la posición 1

        for (Producto p : productos) {
            System.out.println(p.getNombre());
        }
    }
}
```

📌 **Pregunta:**
¿Qué se imprimirá en la salida al ejecutar este código?

---

a)
Manzana
Pan
Leche

b)
Manzana
Leche

c)
Pan
Leche

d) Error en tiempo de ejecución porque no se puede eliminar un elemento de ArrayList.