PROGRAMACIÓN II

Trabajo Práctico 6: Colecciones y Sistema de Stock

OBJETIVO GENERAL

Desarrollar estructuras de datos dinámicas en Java mediante el uso de colecciones (**ArrayList**) y enumeraciones (**enum**), implementando un sistema de stock con funcionalidades progresivas que refuerzan conceptos clave de la programación orientada a objetos..

MARCO TEÓRICO

Concepto	Aplicación en el proyecto
ArrayList	Estructura principal para almacenar productos en el inventario.
Enumeraciones (enum)	Representan las categorías de productos con valores predefinidos.
Relaciones 1 a N	Relación entre Inventario (1) y múltiples Productos (N).
Métodos en enum	Inclusión de descripciones dentro del enum para mejorar legibilidad.
Ciclo for-each	Recorre colecciones de productos para listado, búsqueda o filtrado.
Búsqueda y filtrado	Por ID y por categoría, aplicando condiciones.
Ordenamientos y reportes	Permiten organizar la información y mostrar estadísticas útiles.
Encapsulamiento	Restringir el acceso directo a los atributos de una clase

Caso Práctico 1

1. Descripción general

Se debe desarrollar un sistema de stock que permita gestionar productos en una tienda, controlando su disponibilidad, precios y categorías. La información se modelará utilizando clases, colecciones dinámicas y enumeraciones en Java.

2. Clases a implementar **Clase Producto**

Atributos:

- **id (String)** → Identificador único del producto.
- **nombre (String)** → Nombre del producto.
- **precio (double)** → Precio del producto.
- **cantidad (int)** → Cantidad en stock.
- **categoria (CategoriaProducto)** → Categoría del producto.

Métodos:

- **mostrarInfo()** → Muestra en consola la información del producto.

Enum CategoriaProducto

Valores:

- ALIMENTOS
- ELECTRONICA
- ROPA
- HOGAR

Método adicional:

java **public enum**

CategoriaProducto {

ALIMENTOS("Productos comestibles"),

ELECTRONICA("Dispositivos electrónicos"),

ROPA("Prendas de vestir"),

HOGAR("Artículos para el hogar");

private final String descripcion;

CategoriaProducto(String descripcion) {

this.descripcion = descripcion;

}

public String getDescripcion() {

return descripcion;

}

}

Clase Inventario

Atributo:

- **ArrayList<Producto> productos** Métodos requeridos:

- **agregarProducto(Producto p)**
- **listarProductos()**
- **buscarProductoPorId(String id)**
- **eliminarProducto(String id)**
- **actualizarStock(String id, int nuevaCantidad)**
- **filtrarPorCategoria(CategoriaProducto categoria)**
- **obtenerTotalStock()**
- **obtenerProductoConMayorStock()**
- **filtrarProductosPorPrecio(double min, double max)**
- **mostrarCategoriasDisponibles()**

3. Tareas a realizar

1. Crear al menos cinco productos con diferentes categorías y agregarlos al inventario.
2. Listar todos los productos mostrando su información y categoría.
3. Buscar un producto por ID y mostrar su información.
4. Filtrar y mostrar productos que pertenezcan a una categoría específica.
5. Eliminar un producto por su ID y listar los productos restantes.
6. Actualizar el stock de un producto existente.
7. Mostrar el total de stock disponible.
8. Obtener y mostrar el producto con mayor stock.
9. Filtrar productos con precios entre $1000 y $3000.
10. Mostrar las categorías disponibles con sus descripciones.

CONCLUSIONES ESPERADAS

- Comprender el uso de **this** para acceder a atributos de instancia.
- Aplicar constructores sobrecargados para flexibilizar la creación de objetos.
- Implementar métodos con el mismo nombre y distintos parámetros.
- Representar objetos con **toString()** para mejorar la depuración.
- Diferenciar y aplicar atributos y métodos estáticos en Java.

- Reforzar el diseño modular y reutilizable mediante el paradigma orientado a objetos.

📝 Nuevo Ejercicio Propuesto 2: Biblioteca y Libros

1. Descripción general

Se debe desarrollar un sistema para gestionar una **biblioteca**, en la cual se registren los libros disponibles y sus autores. La relación central es de **composición 1 a N**: una Biblioteca contiene múltiples Libros, y cada Libro pertenece obligatoriamente a una Biblioteca. Si la Biblioteca se elimina, también se eliminan sus Libros.

2. Clases a implementar

Clase Autor

Atributos:

- **id (String)** → Identificador único del autor.
- **nombre (String)** → Nombre del autor.
- **nacionalidad (String)** → Nacionalidad del autor.

Métodos:

- **mostrarInfo()** → Muestra la información del autor en consola.

Clase Libro

Atributos:

- **isbn (String)** → Identificador único del libro.
- **titulo (String)** → Título del libro.
- **anioPublicacion (int)** → Año de publicación.
- **autor (Autor)** → Autor del libro.

Métodos:

- **mostrarInfo()** → Muestra título, ISBN, año y autor.

Clase Biblioteca

Atributo:

- **String nombre**
- **List<Libro> libros** → Colección de libros de la biblioteca.

Métodos requeridos:

- **agregarLibro(String isbn, String titulo,int anioPublicacion, Autor autor)**
- **listarLibros()**
- **buscarLibroPorIsbn(String isbn)**
- **eliminarLibro(String isbn)**
- **obtenerCantidadLibros()**
- **filtrarLibrosPorAnio(int anio)**
- **mostrarAutoresDisponibles()**

3. Tareas a realizar

1. Creamos una biblioteca.
2. Crear al menos tres autores
3. Agregar 5 libros asociados a alguno de los Autores a la biblioteca.
4. Listar todos los libros con su información y la del autor.
5. Buscar un libro por su ISBN y mostrar su información.
6. Filtrar y mostrar los libros publicados en un año específico.
7. Eliminar un libro por su ISBN y listar los libros restantes.
8. Mostrar la cantidad total de libros en la biblioteca.
9. Listar todos los autores de los libros disponibles en la biblioteca.

Conclusiones esperadas

- Comprender la **composición 1 a N** entre Biblioteca y Libro.
- Reforzar el manejo de **colecciones dinámicas** (ArrayList).
- Practicar el uso de **métodos de búsqueda, filtrado y eliminación**.
- Mejorar la modularidad aplicando el paradigma de **programación orientada a objetos.**

📝 Ejercicio: Universidad, Profesor y Curso (bidireccional 1 a N)

1. Descripción general

Se debe modelar un sistema académico donde **un Profesor dicta muchos Cursos** y cada **Curso** tiene exactamente **un Profesor responsable**. La relación **Profesor–Curso** es **bidireccional**:

- Desde **Curso** se accede a su **Profesor**.
- Desde **Profesor** se accede a la **lista de Cursos** que dicta.
  Además, existe la clase **Universidad** que administra el alta/baja y consulta de profesores y cursos.

Invariante de asociación: cada vez que se asigne o cambie el profesor de un curso, **debe actualizarse en los dos lados** (agregar/quitar en la lista del profesor correspondiente).

2. Clases a implementar

Clase Profesor

Atributos:

- **id (String)** → Identificador único.
- **nombre (String)** → Nombre completo.
- **especialidad (String)** → Área principal.
- **List<Curso> cursos** → Cursos que dicta.

Métodos sugeridos:

- **agregarCurso(Curso c)** → Agrega el curso a su lista si no está y sincroniza el lado del curso.
- **eliminarCurso(Curso c)** → Quita el curso y sincroniza el lado del curso (dejar `profesor` en `null` si corresponde).
- **listarCursos()** → Muestra códigos y nombres.
- **mostrarInfo()** → Imprime datos del profesor y cantidad de cursos.

Clase Curso

Atributos:

- **codigo (String)** → Código único.
- **nombre (String)** → Nombre del curso.
- **profesor (Profesor)** → Profesor responsable.

Métodos sugeridos:

- **setProfesor(Profesor p)** → Asigna/cambia el profesor **sincronizando ambos lados**:
  - Si tenía profesor previo, quitarse de su lista.
- **mostrarInfo()** → Muestra código, nombre y nombre del profesor (si tiene).

Clase Universidad

Atributos:

- **String nombre**

- **List<Profesor> profesores**
- **List<Curso> cursos**

Métodos requeridos:

- **agregarProfesor(Profesor p)**
- **agregarCurso(Curso c)**
- **asignarProfesorACurso(String codigoCurso, String idProfesor)** → Usa setProfesor del curso.
- **listarProfesores() / listarCursos()**
- **buscarProfesorPorId(String id)**
- **buscarCursoPorCodigo(String codigo)**
- **eliminarCurso(String codigo)** → Debe **romper la relación** con su profesor si la hubiera.
- **eliminarProfesor(String id)** → Antes de remover, dejar null los cursos que dictaba.

Tareas a realizar

1. Crear **al menos 3 profesores** y **5 cursos**.
2. Agregar profesores y cursos a la universidad.
3. Asignar profesores a cursos usando asignarProfesorACurso(...).
4. Listar cursos con su profesor y profesores con sus cursos.
5. Cambiar el profesor de un curso y verificar que ambos lados quedan sincronizados.
6. Remover un curso y confirmar que ya **no** aparece en la lista del profesor.
7. Remover un profesor y dejar profesor = null,
8. Mostrar un reporte: cantidad de cursos por profesor.

Conclusiones esperadas

- Diferenciar **bidireccionalidad** de una relación unidireccional (navegación desde ambos extremos).
- Mantener **invariantes de asociación** (coherencia de referencias) al **agregar, quitar o reasignar**.
- Practicar colecciones (ArrayList), búsquedas y operaciones de alta/baja.
- Diseñar métodos “seguros” que **sincronicen** los dos lados siempre.

RESOLUCIÓN

CASO 1: Sistema de Stock

El Caso Práctico 1 se centra en el uso de ArrayList y enum en Java para implementar un sistema básico de gestión de inventario, reforzando la relación 1 a N entre el Inventario y los Productos.

1. Clases y Enumeración Implementadas

A. Enum CategoriaProducto

Representa los valores fijos para las categorías de productos e incluye una descripción asociada, cumpliendo con la necesidad de tener métodos en el enum para mejorar la legibilidad.

```
package CASO1.modelo;

public enum CategoriaProducto {
    ALIMENTOS("Productos comestibles"),
    ELECTRONICA("Dispositivos electronicos"),
    ROPA("Prendas de vestir"),
    HOGAR("Articulos para el hogar");

    private final String descripcion;

    CategoriaProducto(String descripcion) {
        this.descripcion = descripcion;
    }

    public String getDescripcion() {
        return descripcion;
    }
}
```

B. Clase Producto

Modela el objeto a gestionar, incluyendo atributos como id, nombre, precio, cantidad y la relación con la categoría. Se implementan los getters, setters para cantidad y el método mostrarInfo().

```
package CASO1.modelo;

public class Producto {
    private String id;
    private String nombre;
    private double precio;
    private int cantidad;
    private CategoriaProducto categoria;

    public Producto(String id, String nombre, double precio, int cantidad, CategoriaProducto
categoria) {
        this.id = id;
        this.nombre = nombre;
        this.precio = precio;
        this.cantidad = cantidad;
        this.categoria = categoria;
    }

    // Métodos Getters
```

```
    public String getId() {
        return id;
    }

    public int getCantidad() {
        return cantidad;
    }

    public double getPrecio() {
        return precio;
    }

    public String getNombre() {
    return nombre;
    }

    public CategoriaProducto getCategoria() {
        return categoria;
    }

    // Método Setters
    public void setCantidad(int cantidad) {
        this.cantidad = cantidad;
    }

    // Método requerido
    public void mostrarInfo() {
        System.out.printf("  [ID: %s] %s - Precio: $%.2f - Stock: %d - Categoria: %s (%s)\n",
                id, nombre, precio, cantidad, categoria.name(), categoria.getDescripcion());
    }

    @Override
    public String toString() {
        return String.format("Producto{ID='%s', Nombre='%s', Precio=%.2f, Cantidad=%d,
Categoria=%s}",
                id, nombre, precio, cantidad, categoria.name());
    }
}
```

C. Clase Inventario

Clase gestora que utiliza un ArrayList<Producto> para almacenar los productos. Contiene todos los métodos de negocio CRUD, búsqueda, reportes requeridos por el sistema.

```
package CASO1.modelo;

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

public class Inventario {
```

```
    private List<Producto> productos;

    public Inventario() {
        this.productos = new ArrayList<>();
    }

    // Métodos CRUD y Búsqueda

    public void agregarProducto(Producto p) {
        productos.add(p);
        System.out.println("Producto '" + p.getNombre() + "' agregado al inventario.");
    }

    public Producto buscarProductoPorId(String id) {
        for (Producto p : productos) {
            if (p.getId().equalsIgnoreCase(id)) {
                return p;
            }
        }
        return null;
    }

    public void listarProductos() {
        System.out.println("\n--- LISTADO COMPLETO DE PRODUCTOS ---");
        if (productos.isEmpty()) {
            System.out.println("El inventario está vacío.");
            return;
        }
        for (Producto p : productos) {
            p.mostrarInfo();
        }
    }

    public boolean eliminarProducto(String id) {
        Producto p = buscarProductoPorId(id);
        if (p != null) {
            productos.remove(p);
            System.out.println("Producto con ID '" + id + "' eliminado.");
            return true;
        }
        System.out.println("Error: Producto con ID '" + id + "' no encontrado para eliminar.");
        return false;
    }

    public boolean actualizarStock(String id, int nuevaCantidad) {
        Producto p = buscarProductoPorId(id);
        if (p != null) {
            p.setCantidad(nuevaCantidad);
            System.out.println("Stock actualizado para el producto '" + p.getNombre() + "'.
Nuevo stock: " + nuevaCantidad);
```

```
            return true;
        }
        System.out.println("Error: Producto con ID '" + id + "' no encontrado para actualizar stock.");
        return false;
    }

    // Métodos de Reporte y Filtrado

    public void filtrarPorCategoria(CategoriaProducto categoria) {
        System.out.println("\n--- PRODUCTOS EN LA CATEGORIA: " + categoria.name() + " ---");
        boolean encontrado = false;
        for (Producto p : productos) {
            if (p.getCategoria() == categoria) {
                p.mostrarInfo();
                encontrado = true;
            }
        }
        if (!encontrado) {
            System.out.println("No hay productos en la categoria " + categoria.name() + ".");
        }
    }

    public int obtenerTotalStock() {
        int total = 0;
        for (Producto p : productos) {
            total += p.getCantidad();
        }
        return total;
    }

    public Producto obtenerProductoConMayorStock() {
        if (productos.isEmpty()) {
            return null;
        }
        Producto mayorStock = productos.get(0);
        for (Producto p : productos) {
            if (p.getCantidad() > mayorStock.getCantidad()) {
                mayorStock = p;
            }
        }
        return mayorStock;
    }

    public void filtrarProductosPorPrecio(double min, double max) {
        System.out.printf("\n--- PRODUCTOS CON PRECIOS ENTRE $%.2f y $%.2f ---\n", min, max);
        boolean encontrado = false;
        for (Producto p : productos) {
```

```
            if (p.getPrecio() >= min && p.getPrecio() <= max) {
                p.mostrarInfo();
                encontrado = true;
            }
        }
        if (!encontrado) {
            System.out.println("No se encontraron productos en ese rango de precios.");
        }
    }

    public void mostrarCategoriasDisponibles() {
        System.out.println("\n--- CATEGORIAS DISPONIBLES ---");
        for (CategoriaProducto categoria : CategoriaProducto.values()) {
            System.out.println("- " + categoria.name() + ": " + categoria.getDescripcion());
        }
    }
}
```

2. Ejecución de Tareas

El siguiente código muestra la instanciación de objetos y la llamada secuencial a los métodos del Inventario para cumplir con las 10 tareas solicitadas:

```
package CASO1.principal;

import CASO1.modelo.CategoriaProducto;
import CASO1.modelo.Inventario;
import CASO1.modelo.Producto;

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

        Inventario tienda = new Inventario();

        // 1. Crear al menos cinco productos y agregarlos al inventario.
        System.out.println("--- INICIO DE CARGA ---");
        tienda.agregarProducto(new Producto("A101", "Smartphone X", 2500.00, 15,
CategoriaProducto.ELECTRONICA));
        tienda.agregarProducto(new Producto("A102", "Camiseta Algodon", 850.50, 40,
CategoriaProducto.ROPA));
        tienda.agregarProducto(new Producto("A103", "Arroz Integral 1kg", 350.00, 100,
CategoriaProducto.ALIMENTOS));
        tienda.agregarProducto(new Producto("A104", "Licuadora Pro", 4500.00, 8,
CategoriaProducto.HOGAR));
        tienda.agregarProducto(new Producto("A105", "Zapatillas Running", 1800.75, 25,
CategoriaProducto.ROPA));
        tienda.agregarProducto(new Producto("A106", "Tablet Y", 1200.00, 12,
CategoriaProducto.ELECTRONICA));

        // 2. Listar todos los productos.
```

```
        tienda.listarProductos();

        // 3. Buscar un producto por ID y mostrar su información.
        System.out.println("\n--- Busqueda por ID (A104) ---");
        Producto buscado = tienda.buscarProductoPorId("A104");
        if (buscado != null) {
            System.out.print("Producto encontrado: ");
            buscado.mostrarInfo();
        } else {
            System.out.println("Producto no encontrado.");
        }

        // 4. Filtrar y mostrar productos que pertenezcan a una categoría específica (ROPA).
        tienda.filtrarPorCategoria(CategoriaProducto.ROPA);

        // 5. Eliminar un producto por su ID (A102) y listar los productos restantes.
        System.out.println("\n--- Eliminacion de producto (A102) ---");
        tienda.eliminarProducto("A102");
        tienda.listarProductos();

        // 6. Actualizar el stock de un producto existente (A101).
        System.out.println("\n--- Actualizacion de Stock (A101) ---");
        tienda.actualizarStock("A101", 30);

        // 7. Mostrar el total de stock disponible.
        System.out.println("\n--- Stock Total ---");
        int stockTotal = tienda.obtenerTotalStock();
        System.out.println("Stock total en el inventario: " + stockTotal);

        // 8. Obtener y mostrar el producto con mayor stock.
        System.out.println("\n--- Producto con Mayor Stock ---");
        Producto mayorStock = tienda.obtenerProductoConMayorStock();
        if (mayorStock != null) {
            System.out.print("Producto con mayor stock: ");
            mayorStock.mostrarInfo();
        }

        // 9. Filtrar productos con precios entre $1000 y $3000.
        tienda.filtrarProductosPorPrecio(1000.00, 3000.00);

        // 10. Mostrar las categorías disponibles con sus descripciones.
        tienda.mostrarCategoriasDisponibles();
    }
}
```

3. Conclusiones Esperadas

Se logró implementar estructuras de datos dinámicas utilizando ArrayList como contenedor principal para los productos.

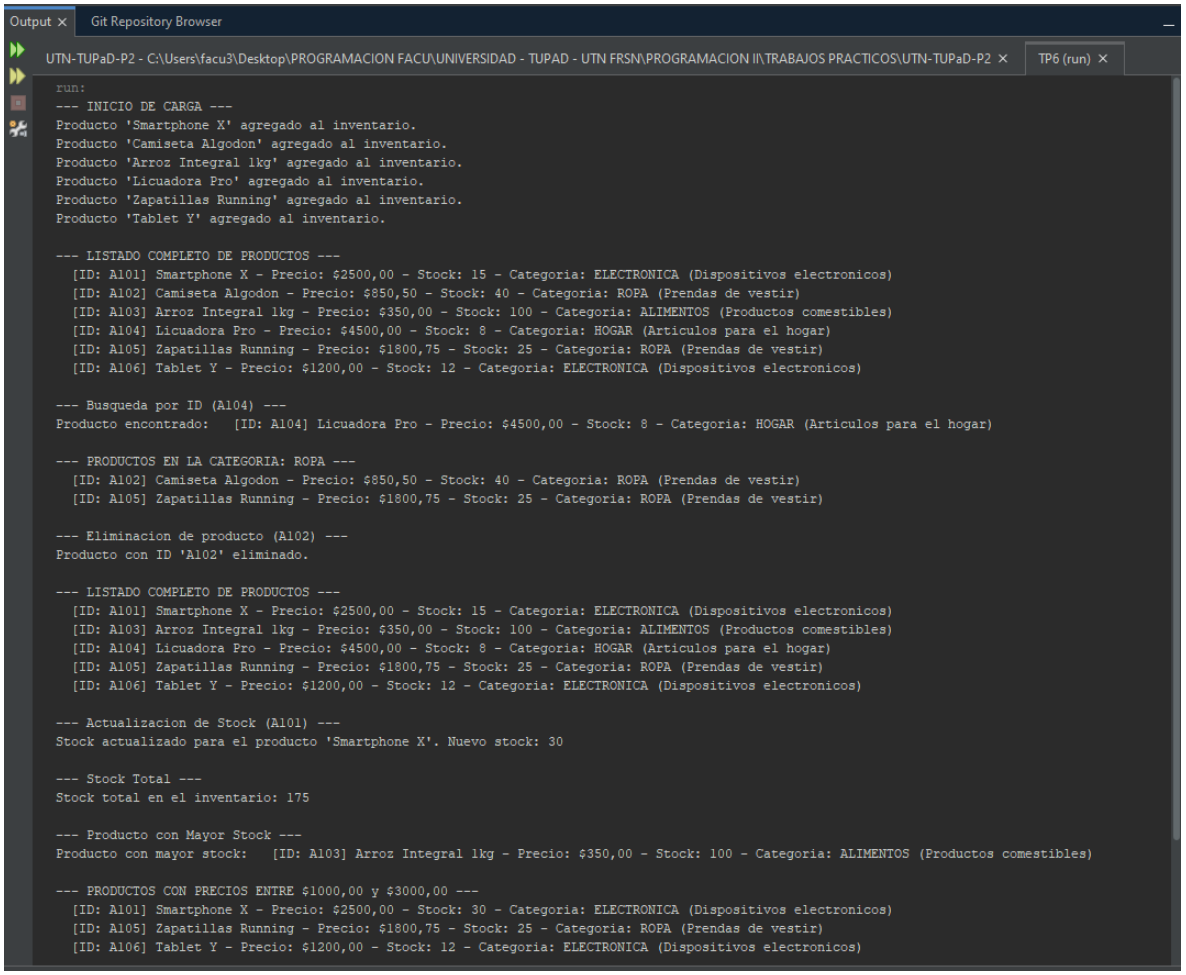
El encapsulamiento se respetó al restringir el acceso directo a los atributos de Producto.

El uso de this y constructores permitió inicializar correctamente los objetos de forma flexible.

Las enumeraciones enriquecieron el modelo de datos al dar valores predefinidos y asociar descripciones útiles.

Se aplicaron métodos de búsqueda y filtrado mediante el ciclo for-each, demostrando el manejo de colecciones.

4. Resultados output



CASO 2: Biblioteca y Libros (Composición 1 a N)

El Caso Práctico 2 modela una relación de Composición 1 a N entre Biblioteca y Libro, donde la existencia de un Libro depende de la Biblioteca. Se utilizan colecciones para gestionar los Libros y referencias de objeto para enlazar cada Libro con un Autor.

1. Clases y Enumeración Implementadas

A. Clase Autor: Clase sencilla para modelar al escritor.

```
package CASO2.modelo;

public class Autor {
    private String id;
    private String nombre;
    private String nacionalidad;

    public Autor(String id, String nombre, String nacionalidad) {
        this.id = id;
        this.nombre = nombre;
        this.nacionalidad = nacionalidad;
    }

    // Getters necesarios
    public String getId() {
        return id;
    }

    public String getNombre() {
        return nombre;
    }

    // Método requerido para mostrar la información
    public void mostrarInfo() {
        System.out.printf("   > Autor: %s (ID: %s, Nacionalidad: %s)\n", nombre, id,
nacionalidad);
    }
}
```

B. Clase Libro: Contiene atributos propios y una referencia a un objeto Autor (composición).

```
package CASO2.modelo;

public class Libro {
    private String isbn;
    private String titulo;
    private int anioPublicacion;
    private Autor autor; // Relación con la clase Autor

    public Libro(String isbn, String titulo, int anioPublicacion, Autor autor) {
        this.isbn = isbn;
        this.titulo = titulo;
        this.anioPublicacion = anioPublicacion;
        this.autor = autor;
    }

    // Getters necesarios para la búsqueda y filtrado
    public String getIsbn() {
        return isbn;
    }
```

```
    public int getAnioPublicacion() {
        return anioPublicacion;
    }

    public Autor getAutor() {
        return autor;
    }

    // Método requerido para mostrar la información (incluye la del Autor)
    public void mostrarInfo() {
        System.out.printf("  [ISBN: %s] Titulo: %s (Anio: %d)\n",
                isbn, titulo, anioPublicacion);
        // Llama al método del objeto Autor
        autor.mostrarInfo();
    }
}
```

C. Clase Biblioteca: Clase gestora que contiene la colección dinámica (List<Libro>) y toda la lógica de negocio.

```
package CASO2.modelo;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class Biblioteca {
    private String nombre;
    private List<Libro> libros; // Colección principal (Composición 1 a N)

    public Biblioteca(String nombre) {
        this.nombre = nombre;
        this.libros = new ArrayList<>();
    }

    // Tarea 3: Agregar Libro
    public void agregarLibro(String isbn, String titulo, int anioPublicacion, Autor autor) {
        Libro nuevoLibro = new Libro(isbn, titulo, anioPublicacion, autor);
        libros.add(nuevoLibro);
    }

    // Tarea 5: Buscar libro por ISBN
    public Libro buscarLibroPorIsbn(String isbn) {
        for (Libro l : libros) {
            if (l.getIsbn().equalsIgnoreCase(isbn)) {
                return l;
            }
        }
```

```
        return null;
    }

    // Tarea 4: Listar todos los libros
    public void listarLibros() {
        System.out.println("\n--- LISTADO COMPLETO DE LIBROS EN " + nombre + " ---
");
        if (libros.isEmpty()) {
            System.out.println("La biblioteca está vacia.");
            return;
        }
        for (Libro l : libros) {
            l.mostrarInfo();
        }
    }

    // Tarea 7: Eliminar libro por ISBN
    public boolean eliminarLibro(String isbn) {
        Libro l = buscarLibroPorIsbn(isbn);
        if (l != null) {
            libros.remove(l);
            System.out.println("🗑️ Libro con ISBN '" + isbn + "' eliminado.");
            return true;
        }
        return false;
    }

    // Tarea 8: Obtener cantidad total de libros
    public int obtenerCantidadLibros() {
        return libros.size();
    }

    // Tarea 6: Filtrar libros por año
    public void filtrarLibrosPorAnio(int anio) {
        System.out.println("\n--- LIBROS PUBLICADOS EN EL ANIO: " + anio + " ---");
        boolean encontrado = false;
        for (Libro l : libros) {
            if (l.getAnioPublicacion() == anio) {
                l.mostrarInfo();
                encontrado = true;
            }
        }
        if (!encontrado) {
            System.out.println("No se encontraron libros publicados en el anio " + anio + ".");
        }
    }

    // Tarea 9: Mostrar autores disponibles (usando HashSet para unicidad)
    public void mostrarAutoresDisponibles() {
        System.out.println("\n--- AUTORES DISPONIBLES EN LA BIBLIOTECA ---");
```

```
        // Usamos Set para garantizar que cada autor aparezca solo una vez.
        Set<Autor> autoresUnicos = new HashSet<>();

        for (Libro l : libros) {
            autoresUnicos.add(l.getAutor());
        }

        for (Autor a : autoresUnicos) {
            a.mostrarInfo();
        }
    }
}
```

2. Ejecución de Tareas

El siguiente código demuestra la correcta instanciación de la Biblioteca y la ejecución de las 9 tareas solicitadas.

```
package CASO2.principal;

import CASO2.modelo.Autor;
import CASO2.modelo.Biblioteca;
import CASO2.modelo.Libro;

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

        // --- INICIO CASO 2: BIBLIOTECA Y LIBROS ---

        // 1. Crear una biblioteca.
        Biblioteca utnBiblioteca = new Biblioteca("Biblioteca Central UTN");

        // 2. Crear al menos tres autores.
        System.out.println("--- CREACION DE AUTORES ---");
        Autor autor1 = new Autor("A001", "Gabriel Garcia Marquez", "Colombiana");
        Autor autor2 = new Autor("A002", "Jorge Luis Borges", "Argentina");
        Autor autor3 = new Autor("A003", "Isabel Allende", "Chilena");

        // 3. Agregar 5 libros asociados a alguno de los Autores a la biblioteca.
        System.out.println("\n--- AGREGANDO 5 LIBROS ---");
        utnBiblioteca.agregarLibro("978-3-16-148410-0", "Cien anios de soledad", 1967,
autor1);
        utnBiblioteca.agregarLibro("978-1-23-456789-7", "El Aleph", 1949, autor2);
        utnBiblioteca.agregarLibro("978-3-16-148411-1", "Cronica de una muerte
anunciada", 1981, autor1);
        utnBiblioteca.agregarLibro("978-0-12-345678-9", "La casa de los espiritus", 1982,
autor3);
        utnBiblioteca.agregarLibro("978-9-87-654321-0", "Ficciones", 1944, autor2);

        // 4. Listar todos los libros con su información y la del autor.
```

```
        utnBiblioteca.listarLibros();

        // 5. Buscar un libro por su ISBN y mostrar su información.
        System.out.println("\n--- TAREA 5: Busqueda por ISBN ---");
        Libro buscado = utnBiblioteca.buscarLibroPorIsbn("978-0-12-345678-9");
        if (buscado != null) {
            System.out.println("Libro encontrado:");
            buscado.mostrarInfo();
        }

        // 6. Filtrar y mostrar los libros publicados en un año específico (1949).
        utnBiblioteca.filtrarLibrosPorAnio(1949);

        // 7. Eliminar un libro por su ISBN y listar los libros restantes.
        System.out.println("\n--- TAREA 7: Eliminacion de libro ---");
        utnBiblioteca.eliminarLibro("978-3-16-148411-1");
        utnBiblioteca.listarLibros();

        // 8. Mostrar la cantidad total de libros en la biblioteca.
        System.out.println("\n--- TAREA 8: Cantidad Total de Libros ---");
        System.out.println("Total de libros disponibles: " +
utnBiblioteca.obtenerCantidadLibros());

        // 9. Listar todos los autores de los libros disponibles en la biblioteca.
        utnBiblioteca.mostrarAutoresDisponibles();
    }
}
```

3. Conclusiones Esperadas

Se implementó la Composición 1 a N modelando la dependencia del Libro respecto a la Biblioteca como su clase contenedora.

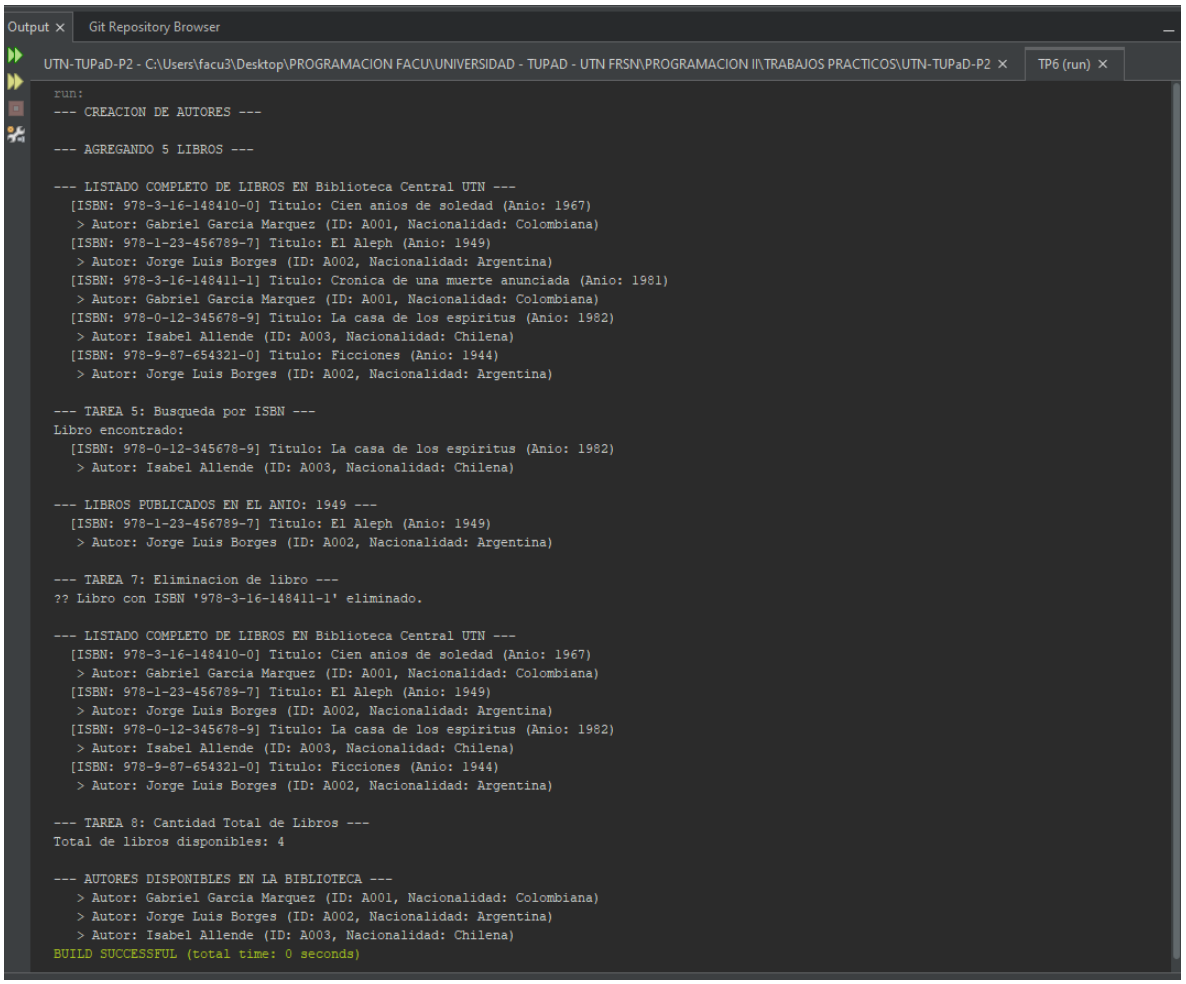
Se reforzó el manejo de colecciones dinámicas (ArrayList) para el almacenamiento de los objetos Libro.

Se practicó el uso de referencias de objeto, donde la clase Libro contiene un objeto de la clase Autor.

Se implementaron con éxito métodos de búsqueda, filtrado y eliminación, demostrando la capacidad de interactuar con objetos dentro de una colección.

Para la Tarea 9, se utilizó la colección HashSet para asegurar la unicidad de los autores en el reporte final.

4. Resultados output



CASO 3: Universidad, Profesor y Curso

El sistema requiere tres clases principales: Profesor, Curso, y la clase gestora Universidad. La clave está en los métodos setProfesor en Curso y agregarCurso/eliminarCurso en Profesor, que deben sincronizar la relación en ambos extremos para evitar incoherencias.

1. Clases y Enumeración Implementadas

A. Clase Profesor: Implementa la colección List<Curso> y métodos internos (agregarCursoInterno, eliminarCursoInterno) para manipular su lista sin provocar ciclos de llamadas. Los métodos públicos agregarCurso y eliminarCurso inician el proceso de sincronización llamando a Curso.setProfesor().

```
package CASO3.modelo;

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

public class Profesor {
    private String id;
    private String nombre;
    private String especialidad;
    private List<Curso> cursos; // Relación 1 a N: Profesor dicta muchos Cursos
```

```
  public Profesor(String id, String nombre, String especialidad) {
     this.id = id;
     this.nombre = nombre;
     this.especialidad = especialidad;
     this.cursos = new ArrayList<>();
  }

  // Getters
  public String getId() { return id; }
  public String getNombre() { return nombre; }
  public List<Curso> getCursos() { return cursos; }

  public String getEspecialidad() {
  return especialidad;
  }

  // Métodos para la gestión INTERNA de la lista de cursos (SIN SINCRONIZACIÓN)
  // Se usan desde Curso.setProfesor() para evitar ciclos infinitos.
  public void agregarCursoInterno(Curso c) {
     if (!cursos.contains(c)) {
        cursos.add(c);
     }
  }

  public void eliminarCursoInterno(Curso c) {
     cursos.remove(c);
  }

  // Métodos de SINCRONIZACIÓN y PÚBLICOS

  // Tarea: Sincroniza la adición de un curso
  public void agregarCurso(Curso c) {
     if (c != null && !cursos.contains(c)) {
        // 1. Agrega el curso a su lista (lado 1 a N)
        cursos.add(c);
        // 2. Sincroniza el lado N a 1 (Llama a setProfesor para que rompa lazo previo si
existe)
        c.setProfesor(this);
     }
  }

  // Tarea: Sincroniza la eliminación de un curso
  public void eliminarCurso(Curso c) {
     if (cursos.remove(c)) {
        // 1. Quita el curso de la lista.
        // 2. Sincroniza el lado N a 1 (Quita el profesor del curso)
        if (c.getProfesor() == this) {
           c.setProfesor(null);
        }
```

```
        }
    }

    // Método requerido: Muestra datos del profesor y cantidad de cursos (Tarea 4)
    public void mostrarInfo() {
        System.out.printf("  [ID: %s] %s - Especialidad: %s (Cursos que dicta: %d)\n",
                id, nombre, especialidad, cursos.size());
    }

    // Método requerido: Listar cursos que dicta (Tarea 4)
    public void listarCursos() {
        if (cursos.isEmpty()) {
            System.out.println("    - No dicta cursos actualmente.");
            return;
        }
        for (Curso c : cursos) {
            System.out.printf("    - [%s] %s\n", c.getCodigo(), c.getNombre());
        }
    }
}
```

B. Clase Curso: La clave es el método setProfesor(Profesor p), que aplica la lógica del invariante de asociación: si ya tenía profesor, lo quita de su lista; luego establece el nuevo profesor y lo agrega a la nueva lista (utilizando los métodos internos del Profesor).

```
package CASO3.modelo;

public class Curso {
    private String codigo;
    private String nombre;
    private Profesor profesor; // Relación N a 1: Curso tiene un Profesor

    public Curso(String codigo, String nombre) {
        this.codigo = codigo;
        this.nombre = nombre;
        this.profesor = null;
    }

    // Getters
    public String getCodigo() { return codigo; }
    public String getNombre() { return nombre; }
    public Profesor getProfesor() { return profesor; }

    // Método requerido y CLAVE para la sincronización (Invariante de Asociación)
    public void setProfesor(Profesor nuevoProfesor) {
        // --- 1. Gestionar la ruptura de la relación con el profesor anterior ---
        if (this.profesor != null && this.profesor != nuevoProfesor) {
            // Solo remueve si el curso estaba asociado al viejo profesor
            this.profesor.eliminarCursoInterno(this);
```

```
        }

        // --- 2. Establecer el nuevo profesor ---
        this.profesor = nuevoProfesor;

        // --- 3. Gestionar el establecimiento de la relación con el nuevo profesor ---
        if (nuevoProfesor != null) {
            // Agrega el curso a la lista del nuevo profesor (si aún no está)
            nuevoProfesor.agregarCursoInterno(this);
        }
    }

    // Método requerido: Muestra información del curso y su profesor (Tarea 4)
    public void mostrarInfo() {
        String nombreProfesor = (profesor != null) ? profesor.getNombre() : "N/A";
        System.out.printf("  [CÓDIGO: %s] Curso: %s | Profesor: %s\n",
                codigo, nombre, nombreProfesor);
    }
}
```

C. Clase Universidad: Orquesta las operaciones maestras (asignarProfesorACurso, eliminarCurso, eliminarProfesor), asegurando que, al eliminar un elemento, se rompan correctamente todas las referencias asociadas en ambos lados.

```
package CASO3.modelo;

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

public class Universidad {
    private String nombre;
    private List<Profesor> profesores;
    private List<Curso> cursos;

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

    // Métodos de Administración de la Universidad
    public void agregarProfesor(Profesor p) {
        profesores.add(p);
    }

    public void agregarCurso(Curso c) {
        cursos.add(c);
    }

    public Profesor buscarProfesorPorId(String id) {
```

```
    for (Profesor p : profesores) {
        if (p.getId().equalsIgnoreCase(id)) return p;
    }
    return null;
}

public Curso buscarCursoPorCodigo(String codigo) {
    for (Curso c : cursos) {
        if (c.getCodigo().equalsIgnoreCase(codigo)) return c;
    }
    return null;
}

// Método requerido: Asignar profesor a curso (Tarea 3)
public boolean asignarProfesorACurso(String codigoCurso, String idProfesor) {
    Curso curso = buscarCursoPorCodigo(codigoCurso);
    Profesor profesor = buscarProfesorPorId(idProfesor);

    if (curso != null && profesor != null) {
        // La magia de la sincronización ocurre dentro de setProfesor()
        curso.setProfesor(profesor);
        System.out.printf("✅ Asignación: Profesor %s asignado al Curso %s.\n",
            profesor.getNombre(), curso.getNombre());
        return true;
    }
    System.out.println("❌ Error: Curso o Profesor no encontrado para la asignación.");
    return false;
}

// Método requerido: Listar Profesores (Tarea 4)
public void listarProfesores() {
    System.out.println("\n--- PROFESORES DE " + nombre + " ---");
    for (Profesor p : profesores) {
        p.mostrarInfo();
        p.listarCursos();
    }
}

// Método requerido: Listar Cursos (Tarea 4)
public void listarCursos() {
    System.out.println("\n--- CURSOS EN " + nombre + " ---");
    for (Curso c : cursos) {
        c.mostrarInfo();
    }
}

// Método requerido: Eliminar Curso (Tarea 6)
public boolean eliminarCurso(String codigo) {
    Curso c = buscarCursoPorCodigo(codigo);
    if (c != null) {
```

```
        // 1. Romper la relación con su profesor si la tiene.
        if (c.getProfesor() != null) {
            c.setProfesor(null); // Esto sincroniza eliminando el curso de la lista del profesor.
        }
        // 2. Eliminar el curso de la lista de la Universidad.
        cursos.remove(c);
        System.out.println("🗑️ Curso '" + c.getNombre() + "' eliminado.");
        return true;
    }
    System.out.println("❌ Error: Curso no encontrado para eliminar.");
    return false;
  }

  // Método requerido: Eliminar Profesor (Tarea 7)
  public boolean eliminarProfesor(String id) {
    Profesor p = buscarProfesorPorId(id);
    if (p != null) {
        // 1. Dejar null en los cursos que dictaba (romper la relación N a 1)
        // Es CRUCIAL iterar sobre una copia para evitar
ConcurrentModificationException
        List<Curso> cursosDictados = new ArrayList<>(p.getCursos());
        for (Curso c : cursosDictados) {
            c.setProfesor(null); // Esto sincroniza eliminando el curso de la lista del profesor
(copia).
        }
        // 2. Eliminar el profesor de la lista de la Universidad.
        profesores.remove(p);
        System.out.println("🗑️ Profesor '" + p.getNombre() + "' eliminado. Sus cursos
quedaron sin asignar.");
        return true;
    }
    System.out.println("❌ Error: Profesor no encontrado para eliminar.");
    return false;
  }

  // Método requerido: Reporte de cursos por profesor (Tarea 8)
  public void generarReporteCursosPorProfesor() {
    System.out.println("\n--- REPORTE: Cantidad de Cursos por Profesor ---");
    for (Profesor p : profesores) {
        System.out.printf("  %s (%s): %d cursos\n",
            p.getNombre(), p.getEspecialidad(), p.getCursos().size());
    }
  }
}
```

2. Ejecución de Tareas

El siguiente código verifica la funcionalidad de la sincronización bidireccional mediante operaciones de alta, baja y reasignación.

```
package CASO3.principal;

import CASO3.modelo.Curso;
import CASO3.modelo.Profesor;
import CASO3.modelo.Universidad;

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

        // --- INICIO CASO 3: UNIVERSIDAD, PROFESOR Y CURSO ---

        Universidad utn = new Universidad("Universidad Tecnológica Nacional");

        // 1. Crear al menos 3 profesores y 5 cursos.
        Profesor p1 = new Profesor("P001", "Dr. Andrés Gómez", "Programación");
        Profesor p2 = new Profesor("P002", "Ing. Laura Pérez", "Bases de Datos");
        Profesor p3 = new Profesor("P003", "Lic. Mónica Ruiz", "Diseño Web");

        Curso c1 = new Curso("C101", "Programación Orientada a Objetos");
        Curso c2 = new Curso("C102", "Estructuras de Datos Avanzadas");
        Curso c3 = new Curso("C201", "Introducción a SQL");
        Curso c4 = new Curso("C301", "Desarrollo Frontend con HTML/CSS");
        Curso c5 = new Curso("C302", "JavaScript Moderno");

        // 2. Agregar profesores y cursos a la universidad.
        utn.agregarProfesor(p1);
        utn.agregarProfesor(p2);
        utn.agregarProfesor(p3);

        utn.agregarCurso(c1);
        utn.agregarCurso(c2);
        utn.agregarCurso(c3);
        utn.agregarCurso(c4);
        utn.agregarCurso(c5);

        System.out.println("--- 3. ASIGNACIÓN INICIAL DE CURSOS ---");
        // 3. Asignar profesores a cursos.
        utn.asignarProfesorACurso("C101", "P001"); // Gómez: POO
        utn.asignarProfesorACurso("C102", "P001"); // Gómez: Estructuras
        utn.asignarProfesorACurso("C201", "P002"); // Pérez: SQL
        utn.asignarProfesorACurso("C301", "P003"); // Ruiz: HTML/CSS

        // 4. Listar cursos con su profesor y profesores con sus cursos.
        utn.listarCursos();
        utn.listarProfesores();

        System.out.println("\n--- 5. CAMBIO DE PROFESOR Y VERIFICACIÓN DE
SINCRONIZACIÓN ---");
        // 5. Cambiar el profesor del C102 (Estructuras) de Gómez (P001) a Pérez (P002).
```

```
        utn.asignarProfesorACurso("C102", "P002");

        // Verificar que Gómez (P001) perdió el curso y Pérez (P002) lo ganó.
        utn.listarProfesores();

        System.out.println("\n--- 6. REMOVER UN CURSO Y VERIFICAR RUPTURA DE
RELACIÓN ---");
        // 6. Remover el curso C301 (Diseño Web).
        utn.eliminarCurso("C301");

        // Verificar que Ruiz (P003) no dicta C301.
        utn.listarProfesores();

        System.out.println("\n--- 7. REMOVER UN PROFESOR Y DEJAR CURSOS EN
NULL ---");
        // 7. Remover al Profesor Pérez (P002).
        utn.eliminarProfesor("P002");

        // Verificar que C102 y C201 quedaron sin profesor.
        utn.listarCursos();
        utn.listarProfesores(); // Pérez ya no aparece

        // 8. Mostrar un reporte: cantidad de cursos por profesor.
        utn.generarReporteCursosPorProfesor();
    }
}
```

3. Conclusiones Esperadas

Se implementó una relación bidireccional 1 a N entre Profesor y Curso, permitiendo la navegación desde ambos extremos.

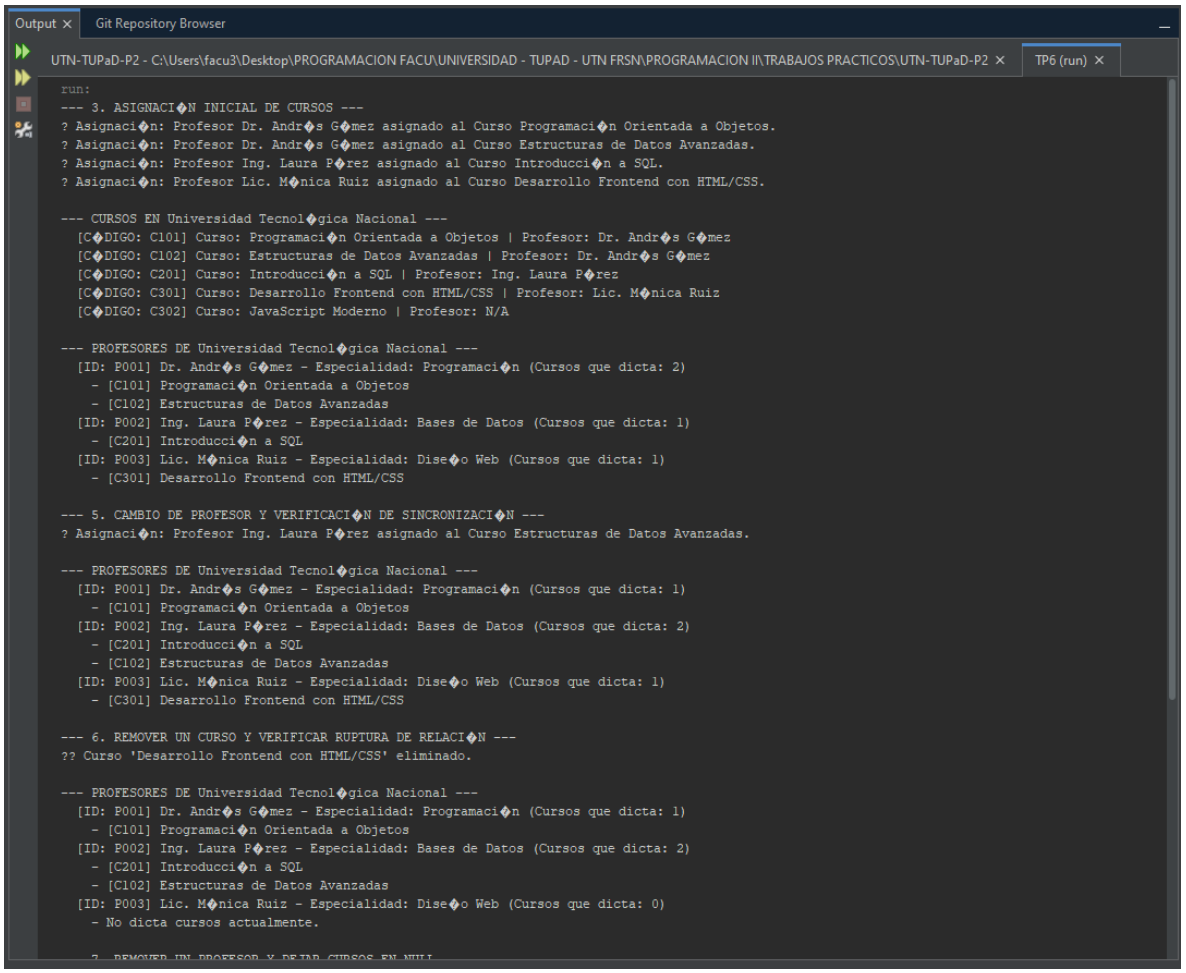
Se logró mantener la invariante de asociación (coherencia) mediante el diseño de métodos "seguros" en Curso.setProfesor y en los métodos de eliminación de Universidad, evitando ciclos recursivos y asegurando que las listas de los profesores se mantengan sincronizadas con los cursos asignados.

Se demostró la funcionalidad al reasignar un curso (Tarea 5), donde el profesor anterior perdió el curso y el nuevo lo ganó automáticamente.

Las funciones de baja (eliminarCurso y eliminarProfesor) demostraron la capacidad de romper las referencias correctamente, dejando profesor = null en los cursos afectados.

Se reforzó el uso de colecciones (ArrayList) para gestionar tanto la lista de cursos como la lista de profesores.

4. Resultados output



Link de repo: https://github.com/Dario-Cabrera/UTN-TUPaD-P2.git

Alumno: Cabrera Dario Ezequiel

DNI: 41375492