

Trabajo Práctico N° 5

Relaciones UML 1 a 1

Comision 16

Marinoni Macarena <marinonimacarena@gmail.com>

Tecnicatura Universitaria en Programación - Universidad Tecnológica Nacional

Programación II

Profesor: Cortez, Alberto

Tutor: Bianchi, Neyén

16/11/2025

ÍNDICE

Objetivo general.....	3
Marco teórico.....	3
Ejercicios de Relaciones 1 a 1.....	4
Ejercicios de Dependencia de Uso.....	14
Ejercicios de Dependencia de Creación.....	16
Conclusión.....	17
Link a Repositorio GitHub:.....	18

Objetivo general

Modelar clases con relaciones 1 a 1 utilizando diagramas UML. Identificar correctamente el tipo de relación (asociación, agregación, composición, dependencia) y su dirección, y llevarlas a implementación en Java.

Marco teórico

Concepto	Aplicación en el proyecto
Asociación	Relación entre clases con referencia mutua o directa, puede ser uni o bidireccional
Agregación	Relación de "tiene un" donde los objetos pueden vivir independientemente
Composición	Relación fuerte de contención, el ciclo de vida del objeto contenido depende del otro
Dependencia de uso	Una clase usa otra como parámetro en un método, sin almacenarla como atributo
Dependencia de creación	Una clase crea otra en tiempo de ejecución, sin mantenerla como atributo

Ejercicios de Relaciones 1 a 1

1. Pasaporte - Foto - Titular
 - a. Composición: Pasaporte → Foto
 - b. Asociación bidireccional: Pasaporte ↔ Titular

Clases y atributos:

- i. Pasaporte: numero, fechaEmision
- ii. Foto: imagen, formato
- iii. Titular: nombre, dni

Objetivo: Modelar un pasaporte que incluye una foto propia y está asociado a un titular. El pasaporte crea y administra su foto (parte), mientras que el titular es una entidad independiente que conoce su pasaporte.

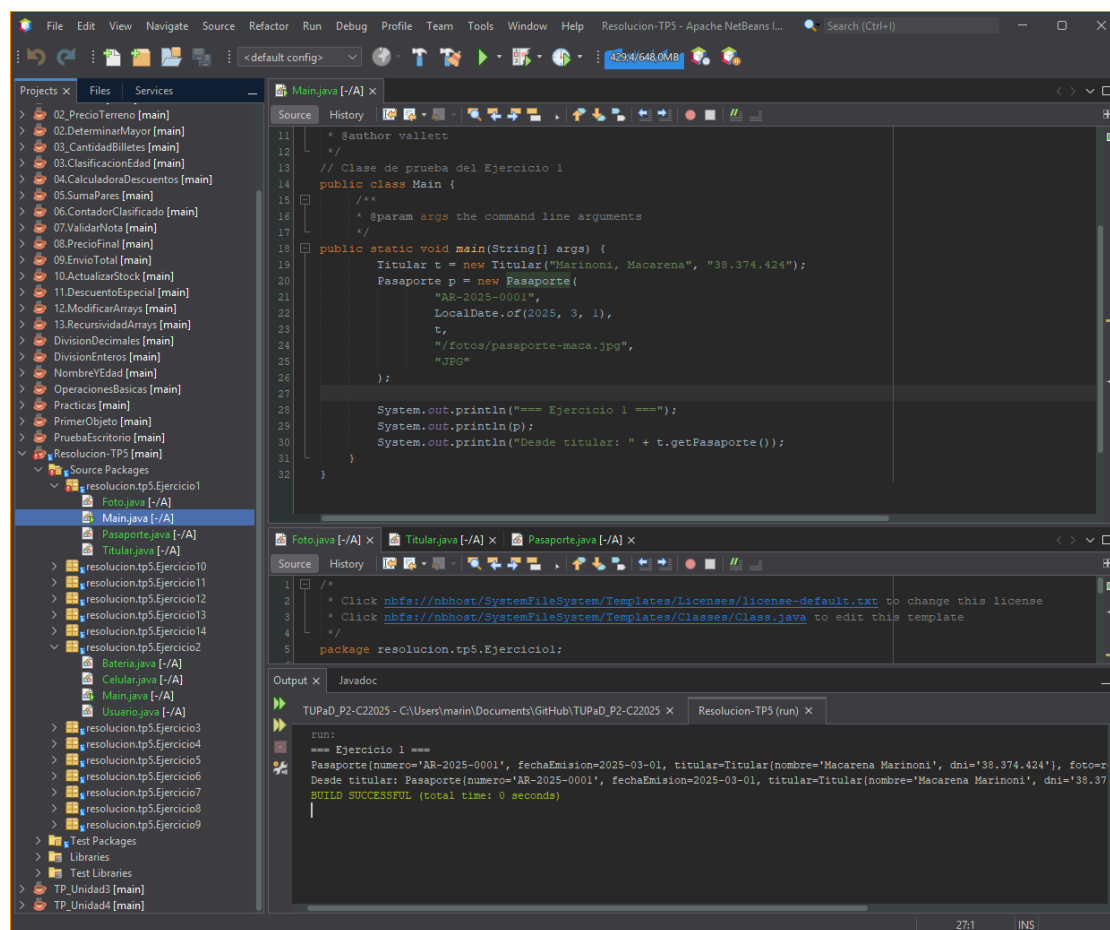
Diagrama UML

- Titular 1 \longleftrightarrow 1 Pasaporte \blacklozenge 1 Foto
(asociación bidi) (composición)

Tipo de relación y dirección

- Pasaporte–Foto: Composición (1:1). Unidireccional (el todo conoce a la parte)
- Pasaporte–Titular: Asociación (1:1). Bidireccional (ambos se conocen).

Implementación



2. Celular - Batería - Usuario

- Agregación: Celular → Batería
- Asociación bidireccional: Celular ↔ Usuario

Clases y atributos:

- Celular: imei, marca, modelo
- Batería: modelo, capacidad
- Usuario: nombre, dni

Objetivo: Representar un celular que usa una batería intercambiable y está asignado a un usuario. La batería no depende del ciclo de vida del celular.

Diagrama UML

- Usuario 1 \longleftrightarrow 1 Celular o --- 1 Batería
(asociación bidi) (agregación)

Tipo de relación y dirección

- Celular–Batería: Agregación (1:1). Unidireccional (el celular conoce su batería).
- Celular–Usuario: Asociación (1:1). Bidireccional (el usuario conoce su celular y viceversa).

Implementación

```

1  /*
2   * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
3   * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
4   */
5  package resolucion.tp5.Ejercicio2;
6
7  /**
8   *
9   * @author marlon
10  */
11
12  // Clase de prueba del Ejercicio 2
13  public class Main {
14      public static void main(String[] args) {
15          Usuario u = new Usuario("Luis Gómez", "39.777.888");
16          Bateria b = new Bateria("Li-Ion X2-40", 5000);
17          Celular c = new Celular("356789012345678", "Samsung", "A54", b, u);
18
19          System.out.println("=== Ejercicio 2 ===");
20          System.out.println(c);
21      }
22  }
  
```

Output x Javadoc

```

TUPaD_P2-C22025 - C:\Users\marin\Documents\GitHub\TUPaD_P2-C22025 x Resolucion-TP5 (run) x
run:
=== Ejercicio 2 ===
Celular(imei='356789012345678', marca='Samsung', modelo='A54', bateria=Bateria(modelo='Li-Ion X2-40', capacidad=5000), usuario=Usuario(nombre='Luis Gómez', dni=
BUILD SUCCESSFUL (total time: 0 seconds)
  
```

3. Libro - Autor - Editorial

- Asociación unidireccional: Libro → Autor
- Agregación: Libro → Editorial

Clases y atributos:

- Libro: titulo, isbn
- Autor: nombre, nacionalidad
- Editorial: nombre, direccion

Objetivo Modelar un libro que conoce a su autor y su editorial. El autor y la editorial existen independientemente del libro.

Diagrama UML

- Libro —→ Autor
- Libro o—→ Editorial

Tipo de relación y dirección

- Libro–Autor: Asociación (1:1). Unidireccional (el libro conoce al autor).
- Libro–Editorial: Agregación (1:1). Unidireccional (el libro conoce a la editorial).

Implementación

```

1  /*
2  * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default
3  * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to ed
4  */
5
6  package resolution.tp5.Ejercicio3;
7
8  /**
9   *
10  * @author Marinon
11  */
12
13  public class Main {
14
15      public static void main(String[] args) {
16          Autor autor = new Autor("Julio Cortezar", "Argentina");
17          Editorial ed = new Editorial("Alfaguara", "30-12345678-9");
18          Libro libro = new Libro("Rayuela", 1963, autor, ed);
19
20          System.out.println("=== Ejercicio 3 ===");
21          System.out.println(libro);
22      }
23  }

```

Output x Javadoc

```

TUPaD_P2-C22025 - C:\Users\marinon\Documents\Git-Hub\TUPaD_P2-C22025 x Resolucion-TP5 (run) x
run:
=== Ejercicio 3 ===
Libro[titulo='Rayuela', anio=1963, autor=Autor[nombre='Julio Cortezar', nacionalidad='A
BUILD SUCCESSFUL (total time: 0 seconds)

```

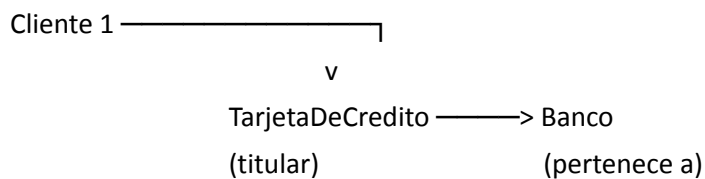
4. TarjetaDeCrédito - Cliente - Banco
 - a. Asociación bidireccional: TarjetaDeCrédito ↔ Cliente
 - b. Agregación: TarjetaDeCrédito → Banco

Clases y atributos:

- i. TarjetaDeCrédito: numero, fechaVencimiento
- ii. Cliente: nombre, dni
- iii. Banco: nombre, cuit

Objetivo: Representar una tarjeta con número y fecha de vencimiento que pertenece a un banco y está asociada a un cliente titular.

Diagrama UML



Tipo de relación y dirección

- TarjetaDeCredito–Cliente: Asociación (1:1), con referencia inversa opcional en Cliente. Bidireccional (recomendado) o uni si no se requiere inversa.
- TarjetaDeCredito–Banco: Asociación (1:1). Unidireccional (la tarjeta conoce al banco).

Implementación

```

// Main.java
public class Main {
    public static void main(String[] args) {
        System.out.println("=== Ejercicio 4 — Tarjeta, Cliente y Banco ===");

        // Creamos el banco
        Banco banco1 = new Banco("Banco Nación", "30-12345678-9");

        // Creamos el cliente
        Cliente cliente1 = new Cliente("Maria López", "40123456");

        // Creamos la tarjeta asociando cliente y banco
        TarjetaDeCredito tarjeta1 = new TarjetaDeCredito(
            "1234-5678-9012-3456",
            "12/28",
            cliente1,
            banco1
        );

        // Mostramos resultados
        System.out.println(tarjeta1);
        System.out.println("Cliente accede a su tarjeta: " + cliente1.getTarjeta());
    }
}

// Banco.java
public class Banco {
    private String nombre;
    private String cuit;

    public Banco(String nombre, String cuit) {
        this.nombre = nombre;
        this.cuit = cuit;
    }

    public String getNombre() {
        return nombre;
    }

    public String getCuit() {
        return cuit;
    }
}

// Cliente.java
public class Cliente {
    private String nombre;
    private String dni;

    public Cliente(String nombre, String dni) {
        this.nombre = nombre;
        this.dni = dni;
    }

    public String getNombre() {
        return nombre;
    }

    public String getDni() {
        return dni;
    }

    public TarjetaDeCredito getTarjeta() {
        return tarjeta;
    }
}

// TarjetaDeCredito.java
public class TarjetaDeCredito {
    private String numero;
    private String fechaVencimiento;
    private Cliente cliente;
    private Banco banco;

    public TarjetaDeCredito(String numero, String fechaVencimiento, Cliente cliente, Banco banco) {
        this.numero = numero;
        this.fechaVencimiento = fechaVencimiento;
        this.cliente = cliente;
        this.banco = banco;
    }

    public String getNumero() {
        return numero;
    }

    public String getFechaVencimiento() {
        return fechaVencimiento;
    }

    public Cliente getCliente() {
        return cliente;
    }

    public Banco getBanco() {
        return banco;
    }

    @Override
    public String toString() {
        return "TarjetaDeCredito[numero='" + numero + "', fechaVencimiento='" + fechaVencimiento + "', titular=" + cliente.getNombre() + "]\n";
    }
}
  
```

Output:

```

=== Ejercicio 4 — Tarjeta, Cliente y Banco ===
TarjetaDeCredito[numero='1234-5678-9012-3456', fechaVencimiento='12/28', titular=Cliente[nom
Cliente accede a su tarjeta: TarjetaDeCredito[numero='1234-5678-9012-3456', fechaVencimiento=
SUCCESSFUL (total time: 0 seconds)
  
```

5. Computadora - PlacaMadre - Propietario

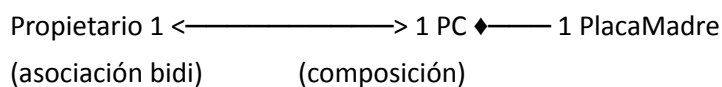
- Composición: Computadora → PlacaMadre
- Asociación bidireccional: Computadora ↔ Propietario

Clases y atributos:

- Computadora: marca, numeroSerie
- PlacaMadre: modelo, chipset
- Propietario: nombre, dni

Objetivo: Modelar una PC que contiene una placa madre (parte esencial) y está asociada a un propietario. La PC crea y administra su placa madre (composición). El propietario existe de forma independiente.

Diagrama UML



Tipo de relación y dirección

- PC-PlacaMadre: Composición (1:1). Unidireccional (la PC conoce su placa).
- PC-Propietario: Asociación (1:1). Bidireccional (opcional; aquí la implementamos).

Implementación

```

1  /**
2   * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change
3   * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
4   */
5  package resolucion.tp5.Ejercicio5;
6
7  /**
8   *
9   * @author marinoni
10  */
11
12  public class Main {
13
14      public static void main(String[] args) {
15          System.out.println("=== Ejercicio 5: Computadora / PlacaMadre / Propietario ===")
16
17          Propietario p = new Propietario("Sofia Diaz", "40.888.999");
18          Computadora computadora = new Computadora("Lenovo", "ThinkStation",
19              "Z790", "LGA1700", p);
20
21          System.out.println(computadora);
22          System.out.println("Desde propietario veo su Computadora: " + p.getPc());
23      }
24  }
  
```

Output x Javadoc

```

TUPaD_P2-C22025 - C:\Users\marin\Documents\GitHub\TUPaD_P2-C22025 x Resolucion-TP5 (run) x
run:
=== Ejercicio 5 ◆ Computadora / PlacaMadre / Propietario ===
Computadora[marca='Lenovo', modelo='ThinkStation', placa=PlacaMadre[chipset='Z790', socket='LGA1700'], pr
Desde propietario veo su Computadora: Computadora[marca='Lenovo', modelo='ThinkStation', placa=PlacaMadre
BUILD SUCCESSFUL (total time: 0 seconds)
  
```


6. Reserva - Cliente - Mesa

- Asociación unidireccional: Reserva → Cliente
- Agregación: Reserva → Mesa

Clases y atributos:

- Reserva: fecha, hora
- Cliente: nombre, telefono
- Mesa: numero, capacidad

Objetivo: Modelar una reserva de restaurante que usa una mesa existente y está asociada a un cliente. Ni el cliente ni la mesa dependen del ciclo de vida de la reserva.

Diagrama UML

Reserva o —> Mesa

Reserva —> Cliente

(agregación) (asociación)

Tipo de relación y dirección

- Reserva-Mesa: Agregación (1:1) — la mesa se inyecta; existe fuera.
- Reserva-Cliente: Asociación (1:1).
- Ambas unidireccionales desde Reserva (la reserva conoce a mesa y cliente).

Implementación

```

1  /*
2  * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change
3  * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
4  */
5  package resolucion.tp5.Ejercicio6;
6
7  import java.time.LocalDateTime;
8
9  /**
10   *
11   * @author marinoni
12   */
13  public class Main {
14      public static void main(String[] args) {
15          System.out.println("=== Ejercicio 6 — Reserva / Cliente / Mesa ===");
16
17          Mesa m = new Mesa(12, 4);
18          Cliente c = new Cliente("Martin López", "+54 9 11 5555-5555");
19          Reserva r = new Reserva(LocalDateTime.of(2025, 10, 20, 20, 30),
20                                  4, m, c);
21
22          System.out.println(r);
23      }
24  }
25

```

Output: run: === Ejercicio 6 — Reserva / Cliente / Mesa === Reserva[fechaHora=2025-10-20T20:30, cantidadPersonas=4, mesa=Mesa[numero=12, capacidad=4], cliente=Cliente[nombre=Martin López, telefono=+54 9 11 5555-5555] BUILD SUCCESSFUL (total time: 0 seconds)

7. Vehículo - Motor - Conductor

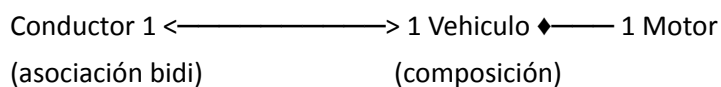
- a. Agregación: Vehículo → Motor
- b. Asociación bidireccional: Vehículo ↔ Conductor

Clases y atributos:

- i. Vehículo: patente, modelo
- ii. Motor: tipo, numeroSerie
- iii. Conductor: nombre, licencia

Objetivo: Modelar un vehículo que contiene su motor (parte esencial) y está asignado a un conductor. El vehículo crea su motor (composición). El conductor existe por separado y conoce su vehículo (asociación bidireccional).

Diagrama UML



Relaciones y dirección

- Vehículo–Motor: Composición (1:1), unidireccional desde Vehículo.
- Vehículo–Conductor: Asociación (1:1), bidireccional.

Implementación:

```

1  /*
2   * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change
3   * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
4   */
5  package resolucion.tp5.Ejercicio7;
6
7  /**
8   *
9   * @author rigon
10  */
11
12  public class Main {
13
14      public static void main(String[] args) {
15          System.out.println("=== Ejercicio 7 -- Vehiculo / Motor / Conductor ===");
16
17          Conductor c = new Conductor("Lucia Ramos", "B1-AR-2028");
18          Vehiculo v = new Vehiculo("Toyota", "Corolla", "Nafta", 140, c);
19
20          System.out.println(v);
21          System.out.println("Desde conductor veo su vehiculo: " + c.getVehiculo());
22      }
23  }
  
```

```

run:
=== Ejercicio 7 -- Vehiculo / Motor / Conductor ===
Vehiculo[marca='Toyota', modelo='Corolla', motor=Motor[tipo='Nafta', potenciaHP=140], conductor=Conduc
Desde conductor veo su vehiculo: Vehiculo[marca='Toyota', modelo='Corolla', motor=Motor[tipo='Nafta',
BUILD SUCCESSFUL (total time: 0 seconds)
  
```

8. Documento - FirmaDigital - Usuario

- a. Composición: Documento → FirmaDigital
- b. Agregación: FirmaDigital → Usuario

Clases y atributos:

- i. Documento: titulo, contenido
- ii. FirmaDigital: codigoHash, fecha
- iii. Usuario: nombre, email

Objetivo: Modelar un documento que incluye una firma única (parte) y está asociado a un usuario autor. El documento crea su firma (composición) a partir de datos crudos.

Diagrama UML

Usuario 1 ———> 1 Documento ◆—— 1 Firma
(asociación uni) (composición)

Relaciones y dirección

- Documento–Firma: Composición (1:1), unidireccional desde Documento.
- Documento–Usuario: Asociación (1:1), unidireccional desde Documento.

Implementación

```

1  /*
2  * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change
3  * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
4  */
5  package resolucion.tp5.Ejercicio8;
6
7  /**
8   *
9   * @author zigon
10  */
11
12  public class Main {
13
14      public static void main(String[] args) {
15          System.out.println("=== Ejercicio 8 — Documento / Firma Digital / Usuario ===");
16          Usuario u = new Usuario("Marcos Peña", "marcos@example.com");
17          Documento d = new Documento(
18              "Contrato de Servicios",
19              "Cláusulas y condiciones...",
20              "abc123def456", "SHA-256",
21              u
22          );
23          System.out.println(d);
24      }
25  }
26

```

Output: Javadoc

```

TUPaD_P2-C2025 - C:\Users\marin\Documents\GitHub\TUPaD_P2-C2025 x Resolucion-TP5 (run) x
runs
=== Ejercicio 8 ◆ Documento / Firma Digital / Usuario ===
Documento[titulo='Contrato de Servicios', contenido='Cláusulas y condiciones...', firma=FirmaDigital(
BUILD SUCCESSFUL (total time: 0 seconds)

```

9. CitaMédica - Paciente - Profesional

- Asociación unidireccional: CitaMédica → Paciente,
- Asociación unidireccional: CitaMédica → Profesional

Clases y atributos:

- CitaMédica: fecha, hora
- Paciente: nombre, obraSocial
- Profesional: nombre, especialidad

Objetivo: Modelar una cita médica que usa a un paciente y a un profesional existentes (inyectados). Ninguno depende de la vida de la cita; la cita solo agrega/relaciona a ambos.

Diagrama UML

Cita —> Paciente

Cita —> Profesional

(asociación uni) (asociación uni)

Relaciones y dirección

- Cita–Paciente: Asociación (1:1), unidireccional desde Cita.
- Cita–Profesional: Asociación (1:1), unidireccional desde Cita.

Implementación:

```

1  /*
2  * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change
3  * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
4  */
5  package resolucion.tp5.Ejercicio9;
6
7  import java.time.LocalDateTime;
8
9  /**
10   *
11   * @author marinoni
12   */
13  public class Main {
14      public static void main(String[] args) {
15          System.out.println("=== Ejercicio 9 — Cita / Paciente / Profesional ===");
16
17          Paciente p = new Paciente("Andrea Ruiz", "40.222.333");
18          Profesional dr = new Profesional("Dr. Gómez", "MP 12345");
19          Cita c = new Cita(LocalDateTime.of(2025, 11, 5, 10, 0), p, dr, "Chequeo anual");
20
21          System.out.println(c);
22      }
23  }
24
25

```

Output:

```

run:
=== Ejercicio 9 — Cita / Paciente / Profesional ===
Cita(fechaHora=2025-11-05T10:00, paciente=Paciente(nombre='Andrea Ruiz', dni='40.222.333'), profesiona
BUILD SUCCESSFUL (total time: 0 seconds)

```

10. CuentaBancaria - ClaveSeguridad - Titular

- Composición: CuentaBancaria → ClaveSeguridad
- Asociación bidireccional: CuentaBancaria ↔ Titular

Clases y atributos:

- CuentaBancaria: cbu, saldo
- ClaveSeguridad: codigo, ultimaModificacion
- Titular: nombre, dni.

Objetivo Modelar una cuenta bancaria que contiene su clave (dato interno, parte) y está asociada a un titular. La cuenta crea su clave (composición) y conoce a su titular (asociación).

Diagrama UML

Titular 1 ———> 1 Cuenta ◆—— 1 Clave
(asociación uni) (composición)

Relaciones y dirección

- Cuenta–Clave: Composición (1:1), unidireccional desde Cuenta.
- Cuenta–Titular: Asociación (1:1), unidireccional desde Cuenta.

Implementación:

```

1  /*
2  * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change
3  * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
4  */
5
6  package resolucion.tp5.Ejercicio10;
7
8  /**
9   *
10  * @author marinon
11  */
12
13  public class Main {
14
15      public static void main(String[] args) {
16          System.out.println("=== Ejercicio 10 — Cuenta / Clave / Titular ===");
17
18          Titular t = new Titular("Nicolás Vera", "39.111.222");
19          Cuenta c = new Cuenta("2850590940090418135201", t,
20                               "e3b0c44298fc1c149afbf4c8996fb924", "SHA-256");
21
22          System.out.println(c);
23      }
24  }
  
```

Output x Javadoc

```

TUPaD_P2-C22025 - C:\Users\marinon\Documents\GitHub\TUPaD_P2-C22025 x Resolucion-TP5 (run) x
run:
=== Ejercicio 10 ◆ Cuenta / Clave / Titular ===
Cuenta(cbu="2850590940090418135201", titular=Titular(nombre="Nicolás Vera", dni="39.111.222"), clave=
BUILD SUCCESSFUL (total time: 0 seconds)
  
```

Ejercicios de Dependencia de Uso

11. Reproductor - Canción - Artista

- Asociación unidireccional: Canción → Artista
- Dependencia de uso: Reproductor.reproducir(Cancion)

Clases y atributos:

- Canción: titulo.
- Artista: nombre, genero.
- Reproductor->método: void reproducir(Cancion cancion)

Objetivo: Modelar un reproductor que reproduce canciones sin almacenarlas como estado. Una canción conoce a su artista.

Diagrama UML

Cancion —> Artista (asociación unidireccional 1:1)
 Reproductor —> reproducir(Cancion) (dependencia de uso)

Tipo de relación y dirección

- Canción–Artista: Asociación 1:1. Unidireccional
- Reproductor–Canción: Dependencia de uso (parámetro de método).

Implementación:

```

1  /*
2   * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change
3   * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
4   */
5  package resolution.tp5.Ejercicio11;
6
7  /**
8   *
9   * @author marinon
10  */
11 public class Main {
12     public static void main(String[] args) {
13         System.out.println("=== Ejercicio 11 - Reproductor / Cancion / Artista ===");
14
15         Artista a = new Artista("Gustavo Cerati", "Rock");
16         Cancion c = new Cancion("Crimen", a);
17
18         Reproductor r = new Reproductor();
19         r.reproducir(c); // dependencia de uso: parámetro del método
20     }
21 }
  
```

Output x Javac

```

run:
TUPaD_P2-C22025 - C:\Users\marinon\Documents\GitHub\TUPaD_P2-C22025 x Resolution-TP5 (run) x
=== Ejercicio 11 - Reproductor / Cancion / Artista ===
? Reproduciendo: Crimen
Artista: Artista[nombre="Gustavo Cerati", genero="Rock"]
BUILD SUCCESSFUL (total time: 0 seconds)
  
```

12. Impuesto - Contribuyente - Calculadora

- Asociación unidireccional: Impuesto → Contribuyente
- Dependencia de uso: Calculadora.calcular(Impuesto)

Clases y atributos:

- Impuesto: monto.
- Contribuyente: nombre, cuil.
- Calculadora->método: void calcular(Impuesto impuesto)

Objetivo: Modelar una calculadora de impuestos que calcula sobre un impuesto recibido como parámetro (sin guardarlo). Un impuesto conoce al contribuyente al que pertenece.

Diagrama UML

Impuesto —→ Contribuyente (asociación unidireccional 1:1)
 Calculadora —→ calcular(Impuesto) (dependencia de uso)

Tipo de relación y dirección

- Impuesto–Contribuyente: Asociación 1:1. Unidireccional.
- Calculadora–Impuesto: Dependencia de uso (parámetro de método).

Implementación

```

1  /*
2   * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change
3   * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
4   */
5
6  package resolucion.tp5.Ejercicio12;
7
8  /**
9   *
10  * @author marinoni
11  */
12
13  public class Main {
14
15      public static void main(String[] args) {
16          System.out.println("=== Ejercicio 12 — Impuesto / Contribuyente / Calculadora ===");
17
18          Contribuyente cont = new Contribuyente("Laura Pérez", "27-40123456-9");
19          Impuesto imp = new Impuesto(100000.0, cont);
20
21          Calculadora calc = new Calculadora();
22          calc.calcular(imp); // dependencia de uso
23      }
24  }
  
```

Output: Javadoc

```

run:
=== Ejercicio 12 — Impuesto / Contribuyente / Calculadora ===
Calculando para Contribuyente(nombre='Laura Pérez', cuil='27-40123456-9')
Base: 100000,00 IVA(21%): 21000,00 Total: 121000,00
BUILD SUCCESSFUL (total time: 0 seconds)
  
```

Ejercicios de Dependencia de Creación

13. GeneradorQR - Usuario - CódigoQR

- Asociación unidireccional: CódigoQR → Usuario
- Dependencia de creación: GeneradorQR.generar(String, Usuario)

Clases y atributos:

- CódigoQR: valor.
- Usuario: nombre, email.
- GeneradorQR->método: void generar(String valor, Usuario usuario)

Objetivo: Modelar un generador de códigos QR que crea un código nuevo dentro del método generar() a partir de un valor y un usuario.

El QR conoce al usuario, pero el generador no conserva ninguna referencia (dependencia de creación).

Diagrama UML

CódigoQR —————> Usuario (asociación unidireccional 1:1)

GeneradorQR —————> generar(String, Usuario) (dependencia de creación)

Tipo de relación y dirección

- CódigoQR—Usuario: Asociación unidireccional 1:1.
- GeneradorQR—CódigoQR: Dependencia de creación (crea, pero no guarda).

Implementación

```

1  /**
2   * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change
3   * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
4   */
5  package resolucion.tp5.Ejercicio13;
6
7  /**
8   *
9   * @author marionon
10  */
11 public class Main {
12
13     public static void main(String[] args) {
14         System.out.println("=== Ejercicio 13 — GeneradorQR / Usuario / CódigoQR ===");
15
16         Usuario u = new Usuario("Camila Torres", "camila@mail.com");
17         GeneradorQR gen = new GeneradorQR();
18
19         // Dependencia de creación
20         gen.generar("QR-001-A1B2C3", u);
21     }
22 }
  
```

Output x Javadoc

```

TUPaD_P2-C22025 - C:\Users\marionon\Documents\GitHub\TUPaD_P2-C22025 x Resolución-TP5 (run) x
run:
=== Ejercicio 13 — GeneradorQR / Usuario / CódigoQR ===
? Código QR generado: CódigoQR(valor='QR-001-A1B2C3', usuario=Usuario[nombre='Camila Torres', email='
BUILD SUCCESSFUL (total time: 0 seconds)
  
```


14. EditorVideo - Proyecto - Render

- Asociación unidireccional: Render → Proyecto
- Dependencia de creación: EditorVideo.exportar(String, Proyecto)

Clases y atributos:

- Render: formato.
- Proyecto: nombre, duracionMin.
- EditorVideo->método: void exportar(String formato, Proyecto proyecto)

Objetivo: Simular un editor de video que exporta proyectos en distintos formatos.

El editor crea un objeto Render dentro del método exportar(), el cual está asociado a un Proyecto.

La dependencia es de creación, no se conserva el objeto generado.

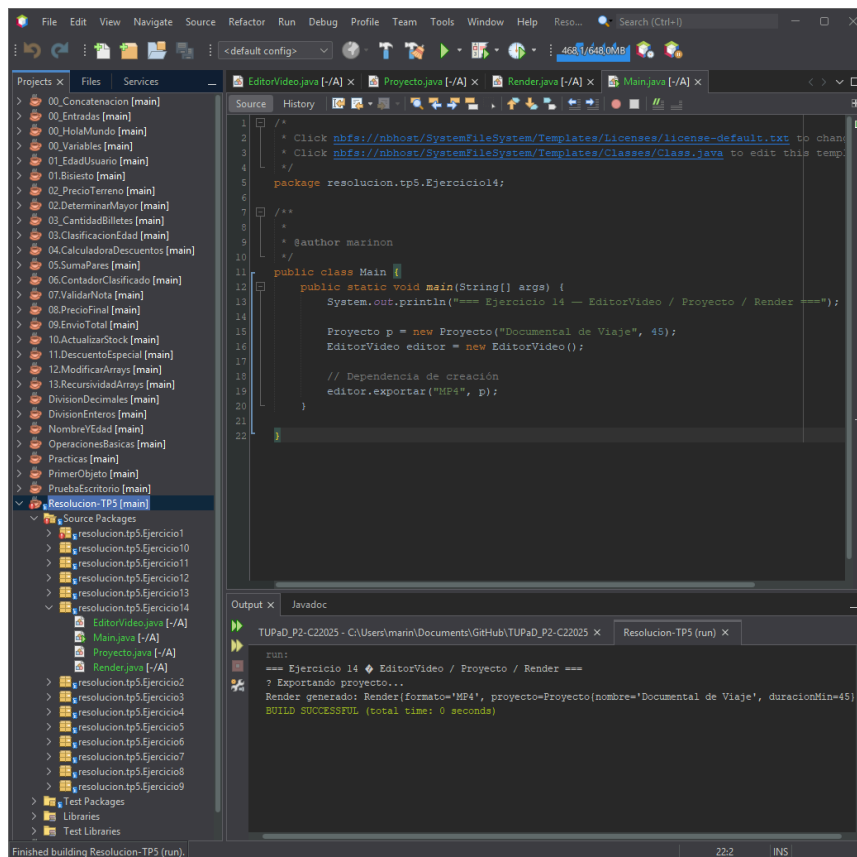
Diagrama UML

Render —→ Proyecto (asociación unidireccional 1:1)
 EditorVideo —→ exportar(String, Proyecto) (dependencia de creación)

Tipo de relación y dirección.

- Render–Proyecto: Asociación unidireccional 1:1.
- EditorVideo–Render: Dependencia de creación.

Implementación



Conclusión

A lo largo del presente trabajo práctico se desarrollaron 14 ejercicios que permitieron explorar y aplicar las distintas formas de vinculación entre clases en el paradigma de la Programación Orientada a Objetos, representadas mediante diagramas UML.

En las primeras consignas (Ejercicios 1 al 4), se abordaron las relaciones de asociación, agregación y composición, a partir de ejemplos concretos como el vínculo entre un pasaporte y su foto, un celular y su batería, o un cliente y su banco. Estas actividades ayudaron a comprender cómo varía el nivel de dependencia entre los objetos según el tipo de relación.

Por ejemplo, en la composición una clase se responsabiliza totalmente por la existencia de otra (como un pasaporte que crea su propia foto), mientras que en la agregación los objetos pueden existir por separado (como una batería que puede ser retirada o reemplazada de un celular). La asociación, por su parte, implica un conocimiento mutuo entre objetos, sin que haya control sobre el ciclo de vida de uno por parte del otro.

En los ejercicios intermedios (del 5 al 10), se trabajó con relaciones más complejas, incorporando vínculos bidireccionales y combinando distintos tipos de relaciones en un mismo modelo. Casos como el de una computadora con su placa madre y su propietario, o el de una reserva asociada a un cliente y a una mesa, reforzaron la importancia de manejar correctamente las referencias entre clases y respetar el principio de encapsulamiento para mantener la coherencia y claridad del código.

Más adelante, en los Ejercicios 11 y 12, se introdujo el concepto de dependencia de uso, en el cual una clase utiliza otra de forma puntual como parámetro de un método, sin conservarla como atributo. Ejemplos como un reproductor que reproduce una canción o una calculadora que procesa un impuesto mostraron cómo lograr bajo acoplamiento entre objetos, manteniendo independencia funcional.

Finalmente, los últimos ejercicios (13 y 14) abordaron la dependencia de creación, donde una clase instancia otra dentro de un método sin almacenarla. Modelos como el de un generador de códigos QR que crea un objeto asociado a un usuario, o un editor de video que exporta un render vinculado a un proyecto, permitieron reflexionar sobre el rol de ciertos objetos como creadores temporales o "fábricas" dentro del diseño.

En conjunto, este trabajo fue clave para afianzar la capacidad de traducir relaciones UML a código en Java, aplicando los distintos tipos de vínculos según el contexto, prestando atención a la direccionalidad de las relaciones, y organizando los atributos, métodos y constructores de manera clara y coherente.

Además, permitió ejercitar varios principios fundamentales de la programación orientada a objetos como el encapsulamiento, la modularidad, la reutilización y el bajo acoplamiento, fortaleciendo así una base sólida para futuros desarrollos en este paradigma.

Link a Repositorio GitHub:

https://github.com/MaquiMarinoni/TUPaD_P2-C22025.git