UNIVERSIDAD TECNOLOGICA NACIONAL

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

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

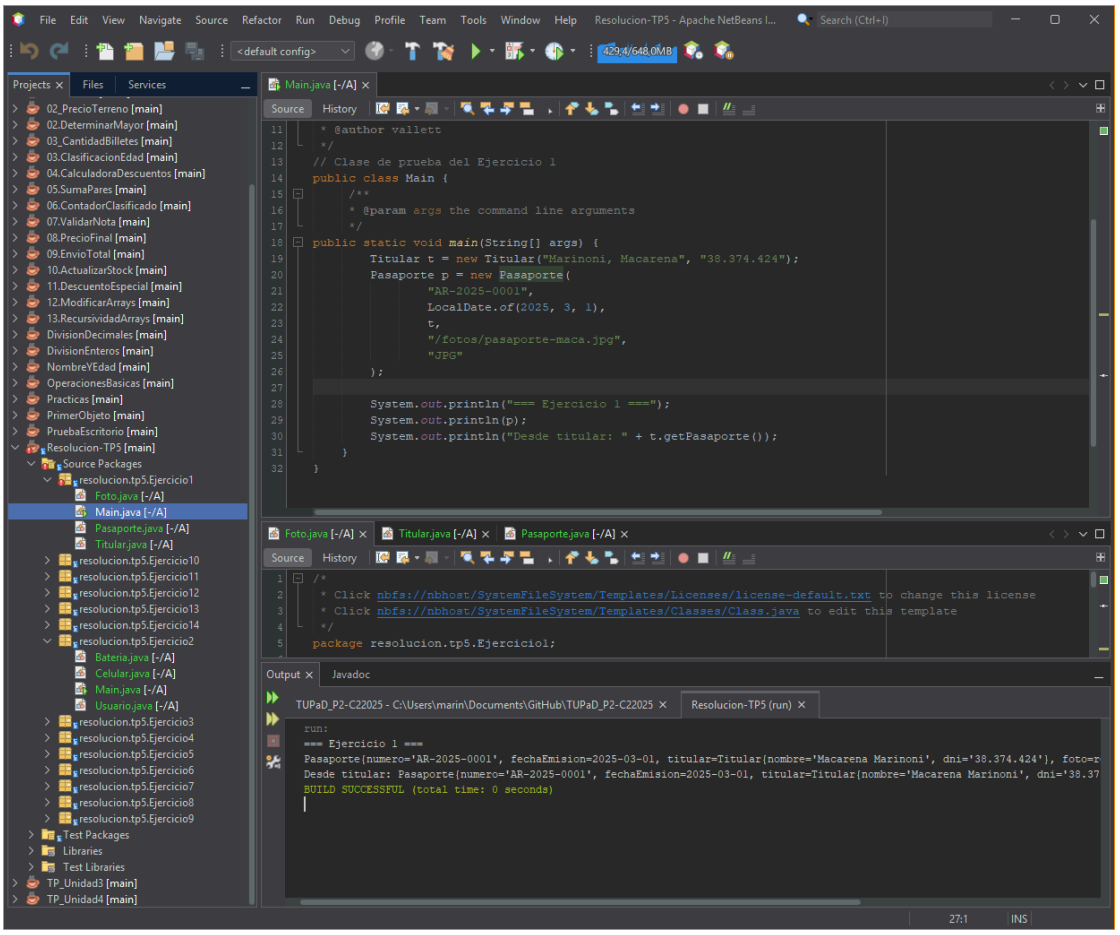
- Titular 1 <——————————> 1 Pasaporte ♦—— 1 Foto

  (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
    a. Agregación: Celular → Batería
    b. Asociación bidireccional: Celular ↔ Usuario

Clases y atributos:

i. Celular: imei, marca, modelo

ii. Batería: modelo, capacidad

iii. 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

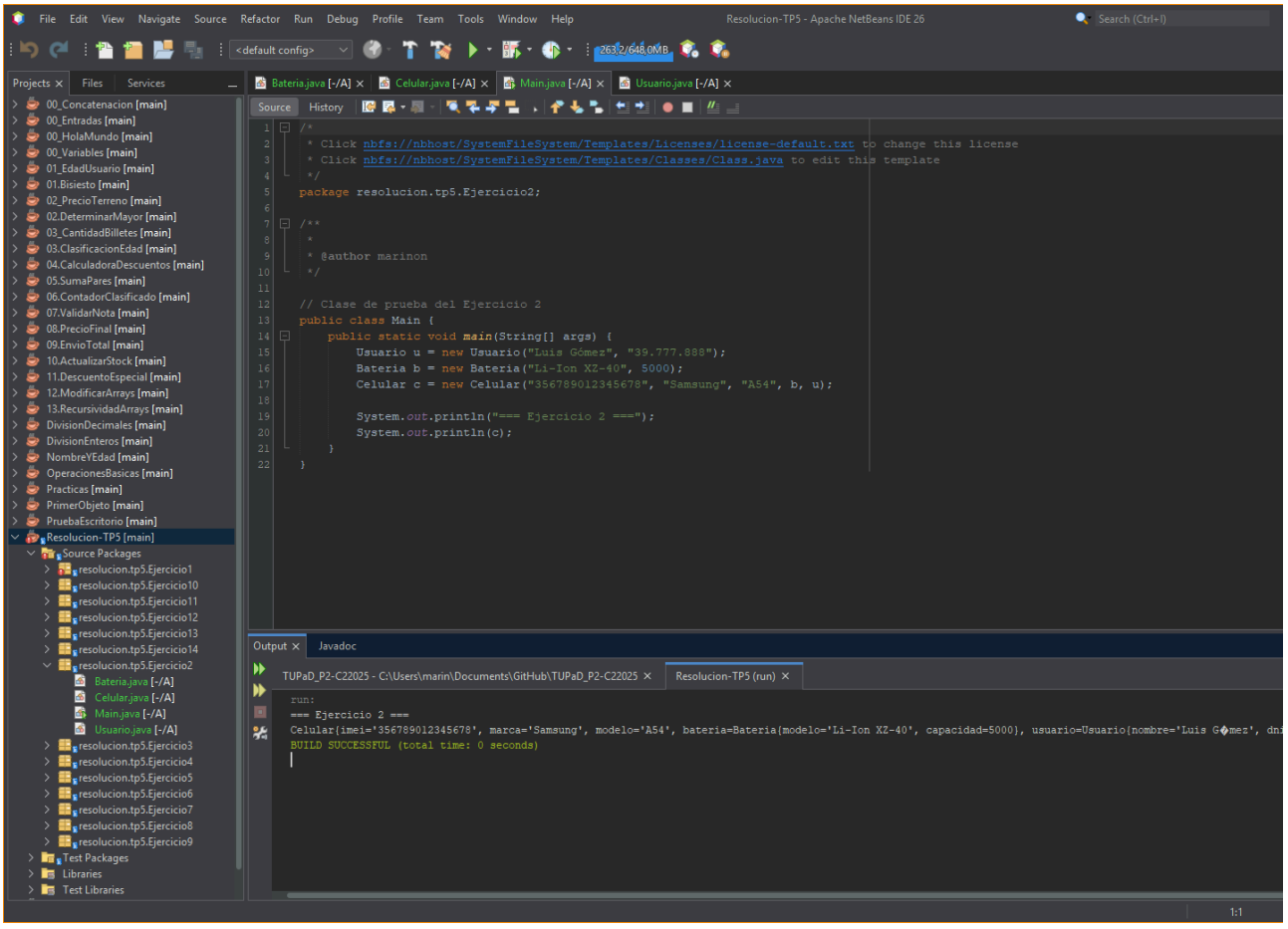
- Usuario 1 <————————> 1 Celular o——— 1 Bateria

  (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



3. Libro - Autor - Editorial
    a. Asociación unidireccional: Libro ⟶ Autor
    b. Agregación: Libro ⟶ Editorial

    Clases y atributos:

    i. Libro: titulo, isbn

    ii. Autor: nombre, nacionalidad

    iii. Editorial: nombre, direccion

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

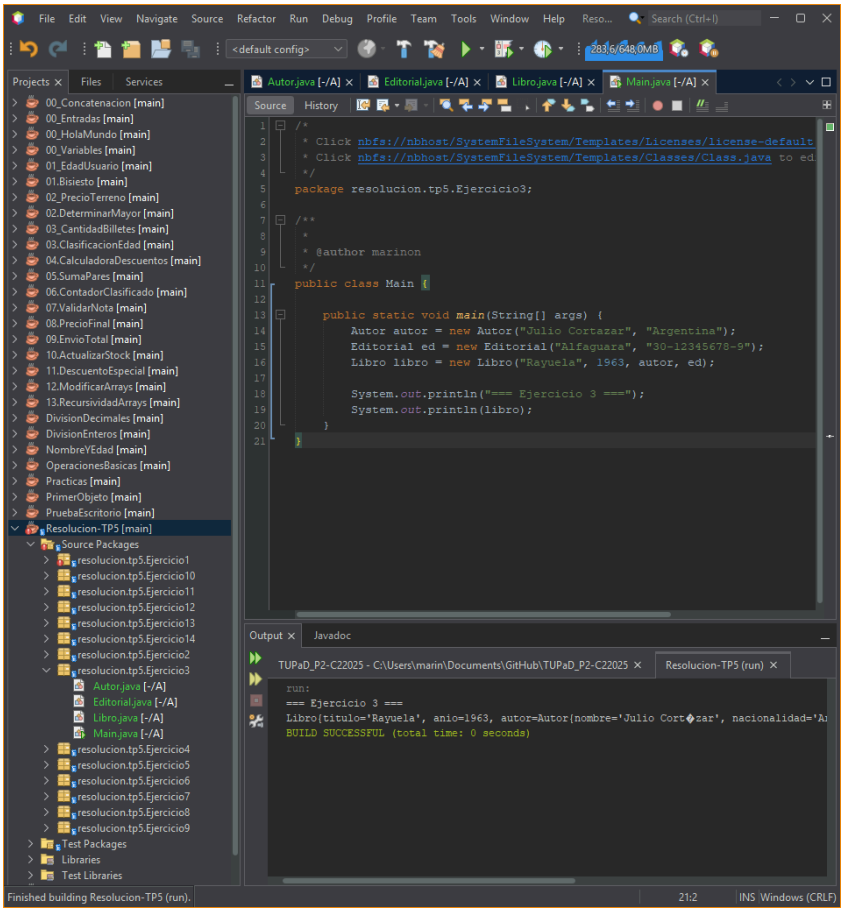
Diagrama UML

- 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



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

    Clases y atributos:

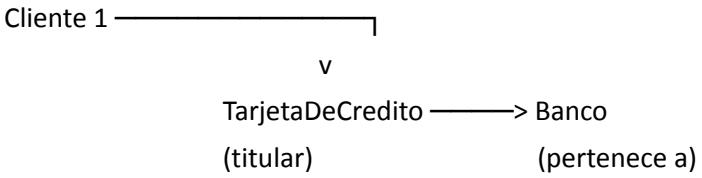
    i. TarjetaDeCrédito: numero, fechaVencimiento

    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.

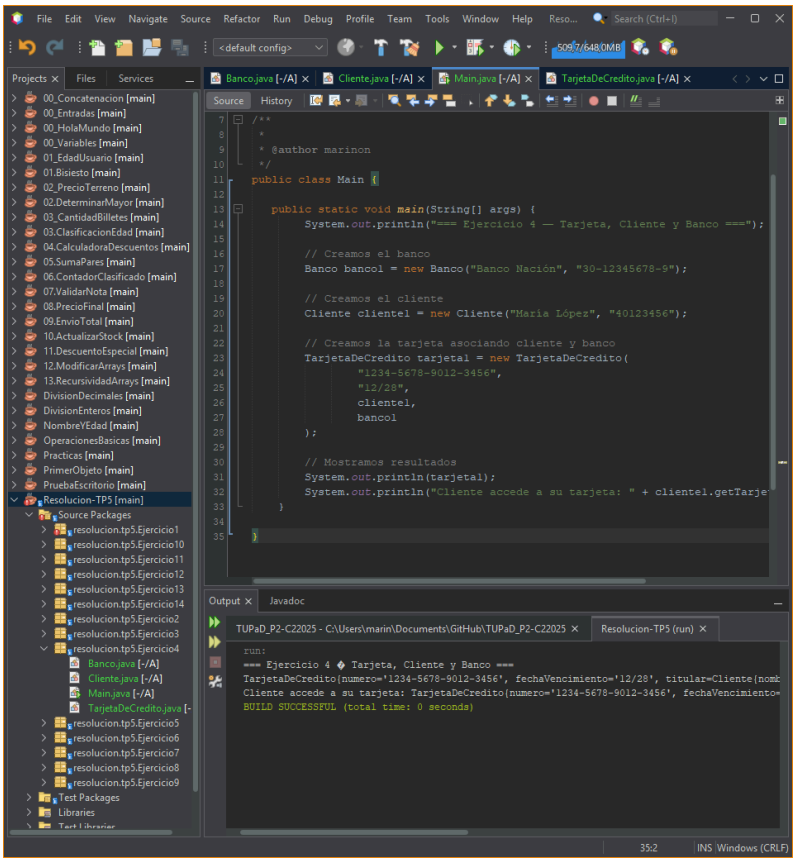
Diagrama UML



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



5. Computadora - PlacaMadre - Propietario
    a. Composición: Computadora ⟶ PlacaMadre
    b. Asociación bidireccional: Computadora ↔ Propietario

    Clases y atributos:

    i. Computadora: marca, numeroSerie

    ii. PlacaMadre: modelo, chipset

    iii. 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

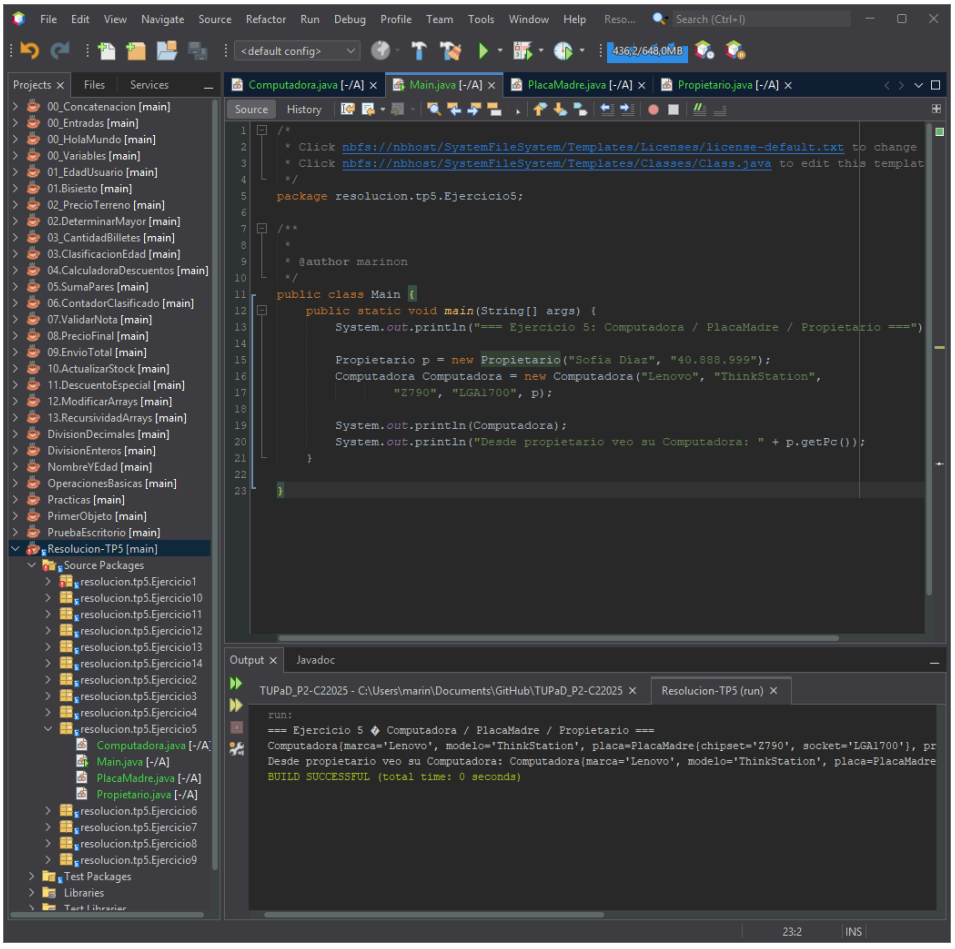
Propietario 1 <——————————> 1 PC ♦——— 1 PlacaMadre

(asociación bidi) (composición)

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



6. Reserva - Cliente - Mesa
    a. Asociación unidireccional: Reserva ⟶ Cliente
    b. Agregación: Reserva ⟶ Mesa

    Clases y atributos:

    i. Reserva: fecha, hora

    ii. Cliente: nombre, telefono

    iii. 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

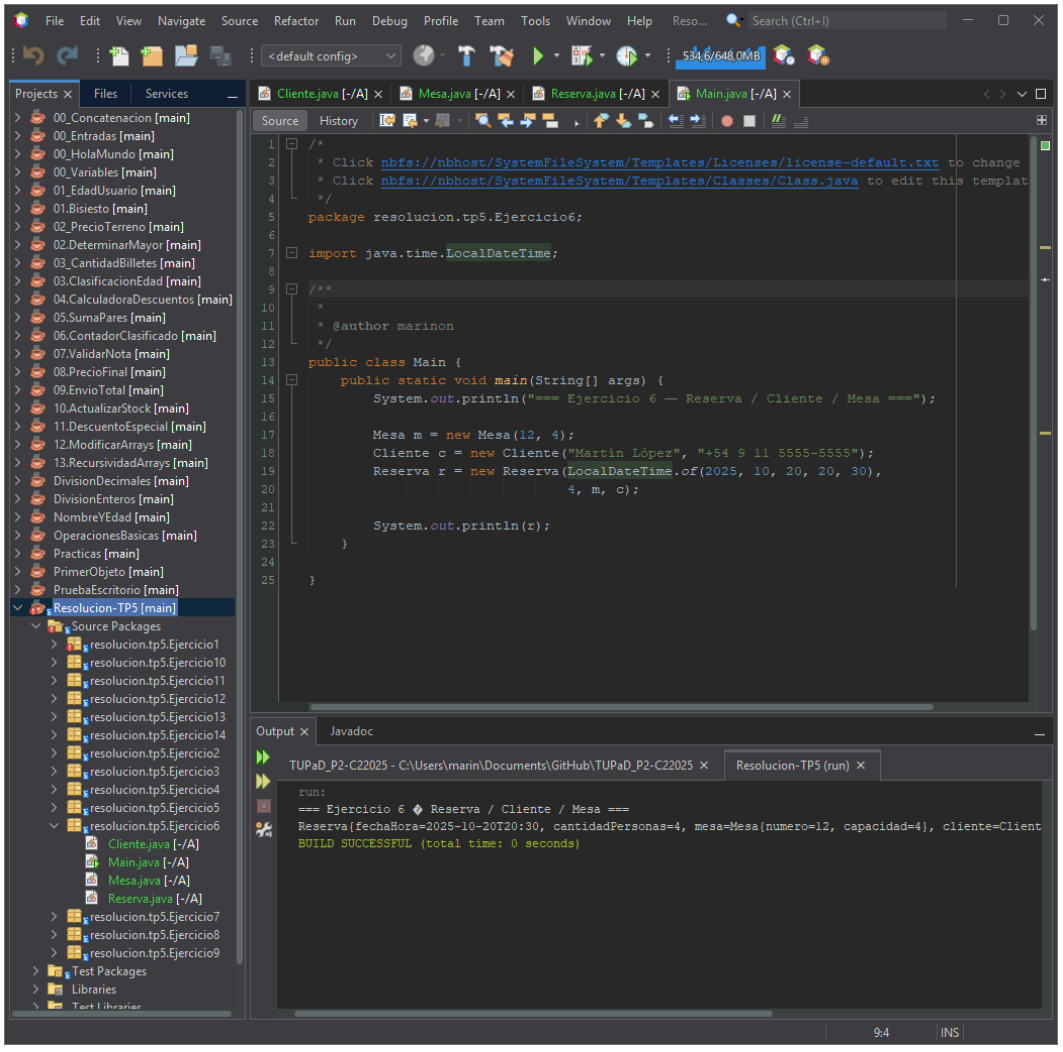
Reserva ———> Cliente

(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



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

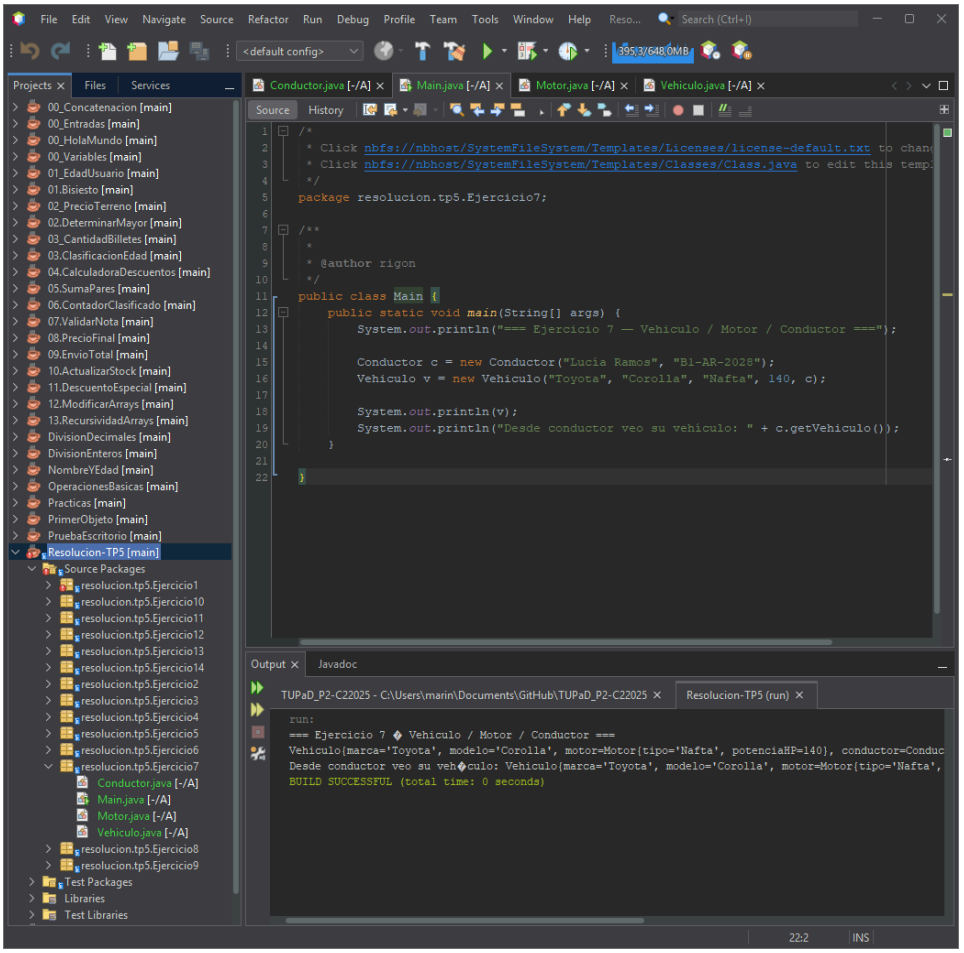
Conductor 1 <——————————> 1 Vehiculo ♦——— 1 Motor

(asociación bidi) (composición)

Relaciones y dirección

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

Implementación:



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

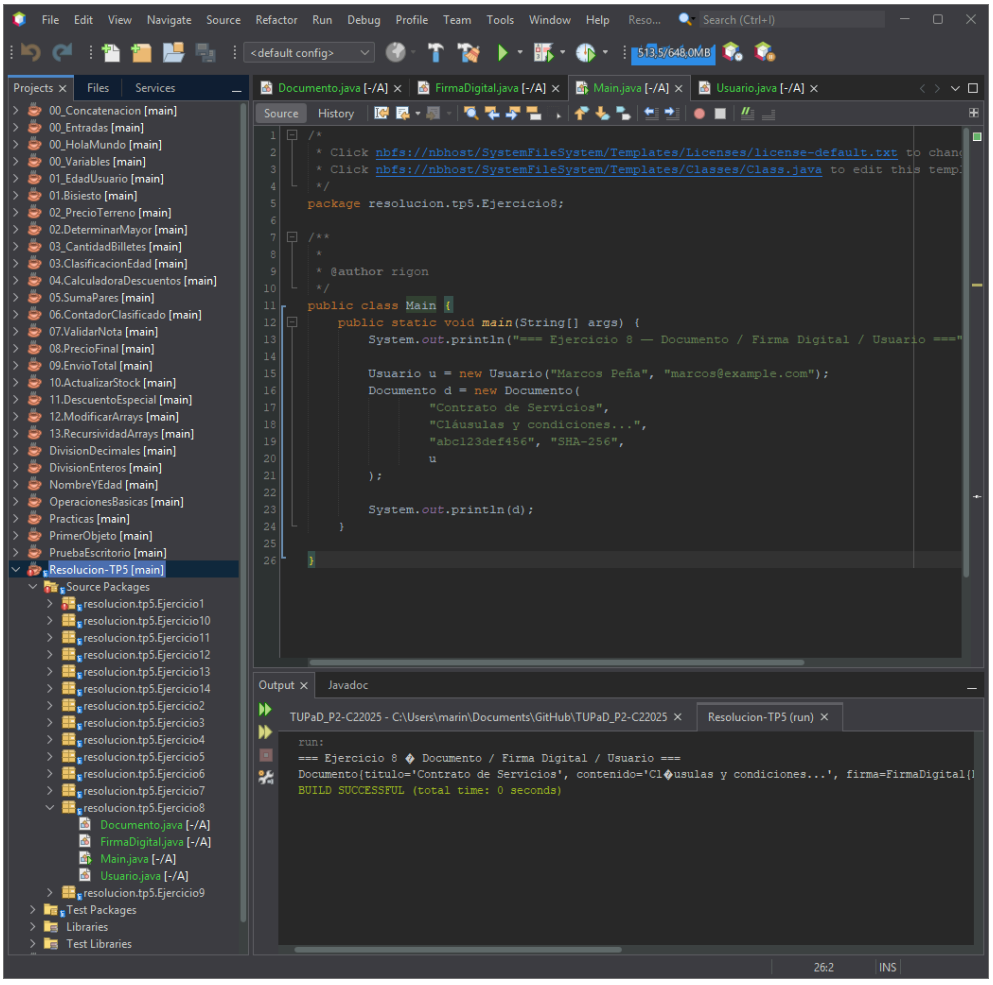
Usuario 1 ———> 1 Documento ♦——— 1 Firma

(asociación uni) (composición)

Relaciones y dirección

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

Implementación



9. CitaMédica - Paciente - Profesional
    a. Asociación unidireccional: CitaMédica ⟶ Paciente,
    b. Asociación unidirecciona: CitaMédica ⟶ Profesional

    Clases y atributos:

    i. CitaMédica: fecha, hora

    ii. Paciente: nombre, obraSocial

    iii. 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

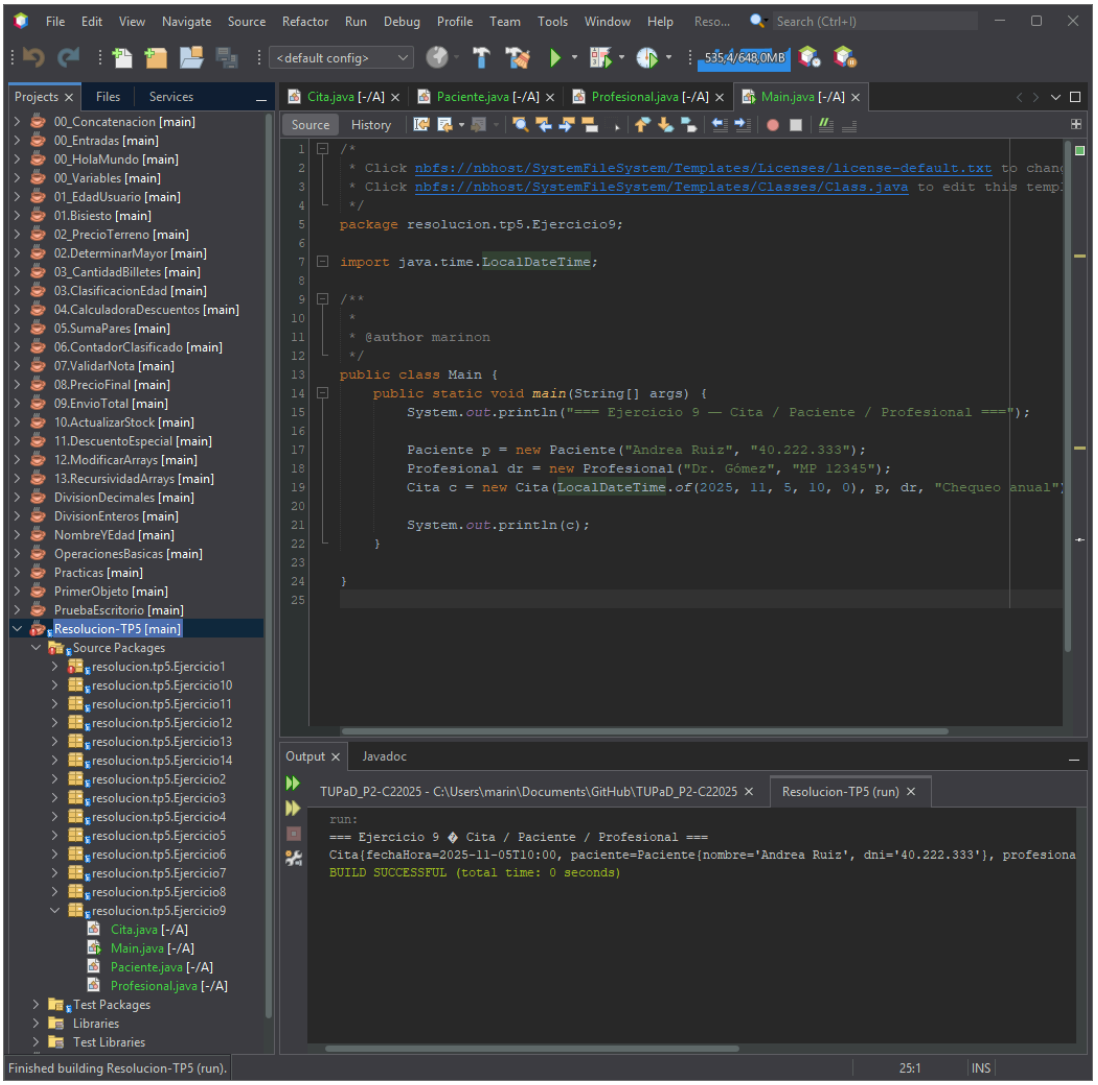
Cita ———> Profesional

(asociación uni) (asociación uni)

Relaciones y dirección

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

Implementación:



10. CuentaBancaria - ClaveSeguridad - Titular
    a. Composición: CuentaBancaria ⟶ ClaveSeguridad
    b. Asociación bidireccional: CuentaBancaria ↔ Titular

    Clases y atributos:

    i. CuentaBancaria: cbu, saldo

    ii. ClaveSeguridad: codigo, ultimaModificacion

    iii. 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

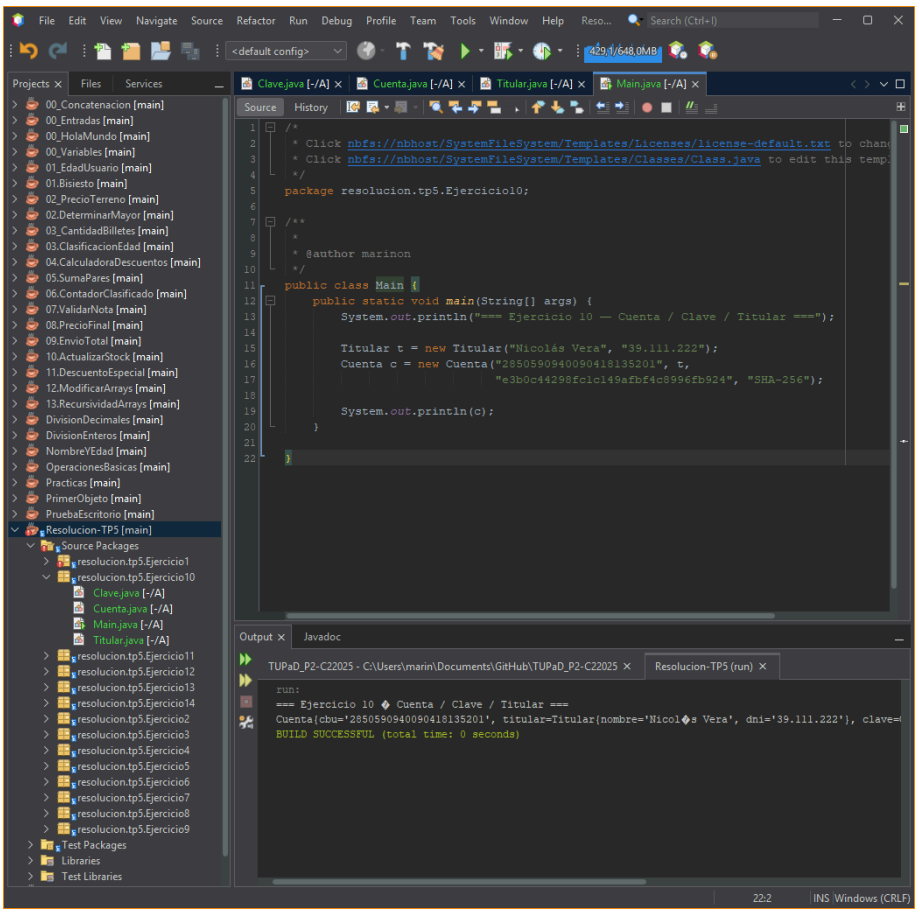
Titular 1 ————> 1 Cuenta ♦——— 1 Clave

(asociación uni) (composición)

Relaciones y dirección

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

Implementación:



Ejercicios de Dependencia de Uso

11. Reproductor - Canción - Artista
    a. Asociación unidireccional: Canción ⟶ Artista
    b. Dependencia de uso: Reproductor.reproducir(Cancion)

    Clases y atributos:

    i. Canción: titulo.

    ii. Artista: nombre, genero.

    iii. 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

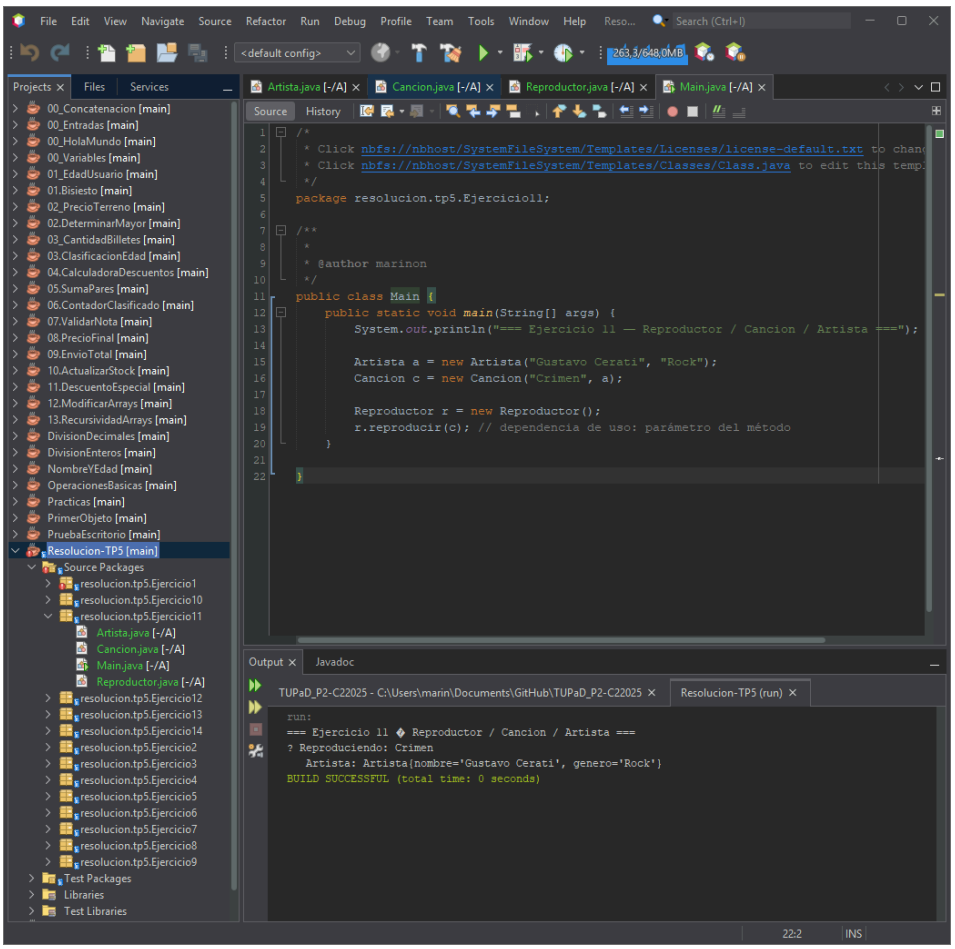
Cancion ————> Artista (asociación unidireccional 1:1)

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:



12. Impuesto - Contribuyente - Calculadora
    a. Asociación unidireccional: Impuesto ⟶ Contribuyente
    b. Dependencia de uso: Calculadora.calcular(Impuesto)

    Clases y atributos:

    i. Impuesto: monto.

    ii. Contribuyente: nombre, cuil.

    iii. 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

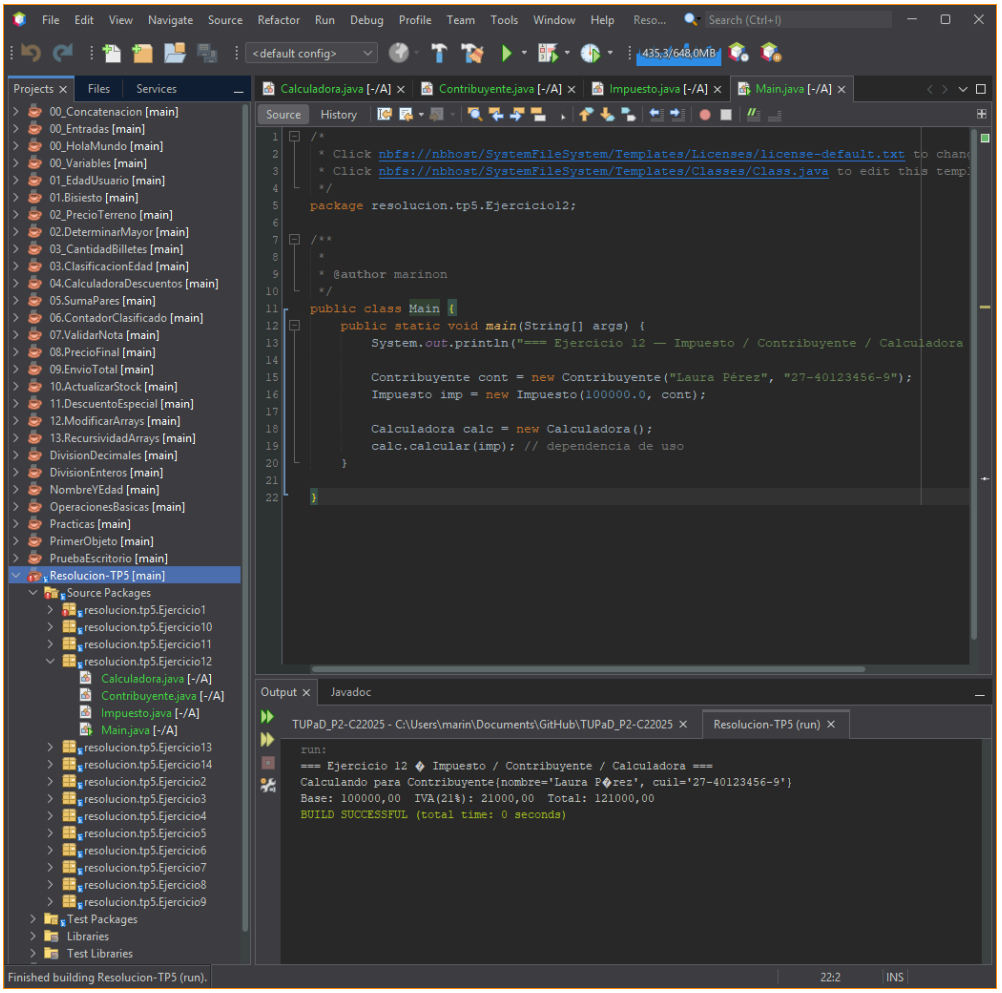
Impuesto ———> Contribuyente (asociación unidireccional 1:1)

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



Ejercicios de Dependencia de Creación

13. GeneradorQR - Usuario - CódigoQR
    a. Asociación unidireccional: CódigoQR ⟶ Usuario
    b. Dependencia de creación: GeneradorQR.generar(String, Usuario)

    Clases y atributos:

    i. CodigoQR: valor.

    ii. Usuario: nombre, email.

    iii. 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

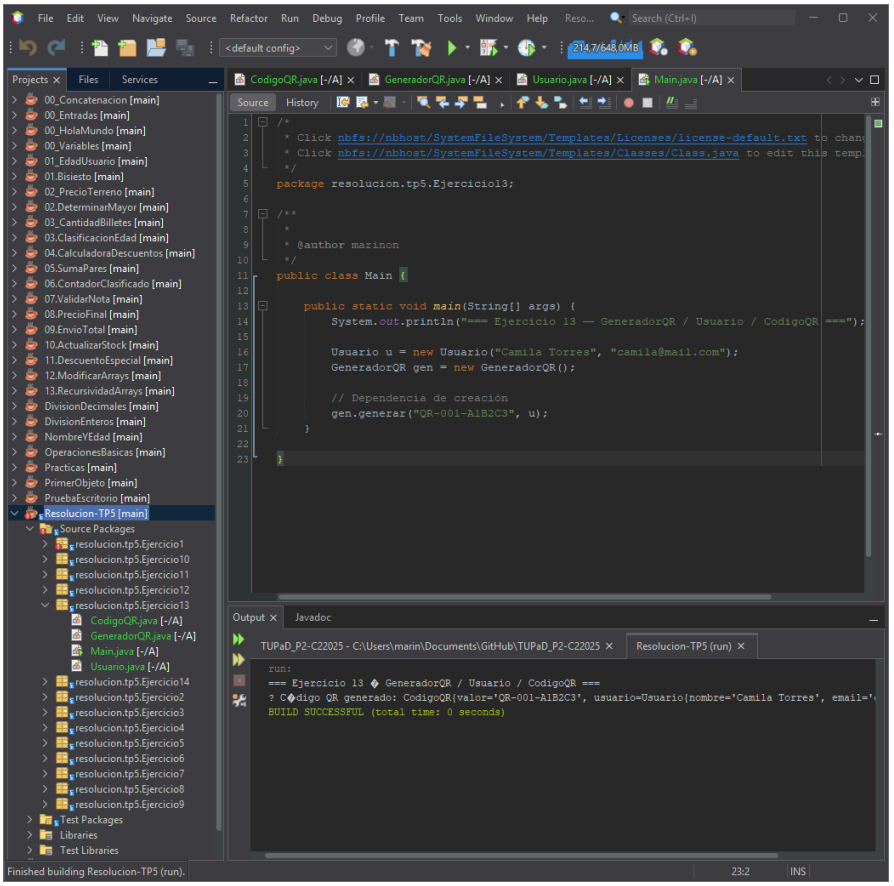
CodigoQR ———> Usuario (asociación unidireccional 1:1)

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



14. EditorVideo - Proyecto - Render
    a. Asociación unidireccional: Render ⟶ Proyecto
    b. Dependencia de creación: EditorVideo.exportar(String, Proyecto)

Clases y atributos:

i. Render: formato.

ii. Proyecto: nombre, duracionMin.

iii. 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

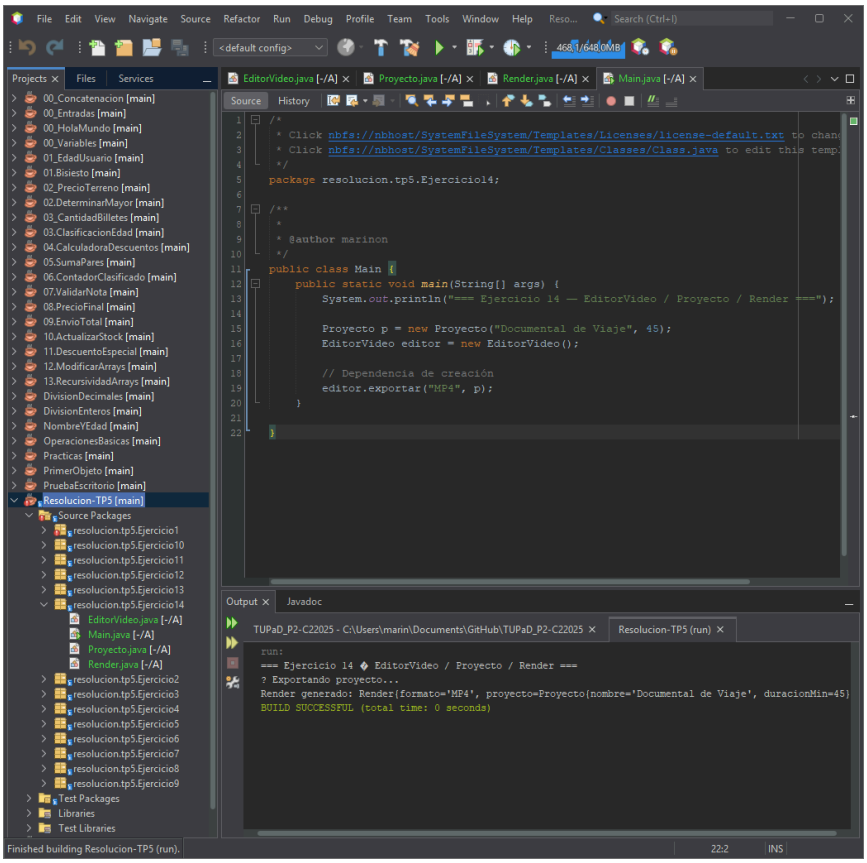
Render ————> Proyecto (asociación unidireccional 1:1)

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