Diagramas UML (1º DAM/DAW)

Casos de uso • Clases • Secuencia — del enunciado al código (Java)

Objetivo: dibujar antes de programar para pensar mejor y cometer menos errores.

ÍNDICE

Diagramas UML

¿Para qué sirven los diagramas?

Un diagrama es un “dibujo con reglas” para entender y explicar un sistema.

- Te obligan a pensar antes de programar.
- Reducen malentendidos (equipo / cliente).
- Muestran estructura (qué cosas hay) y comportamiento (qué pasa).
- Ayudan a detectar errores “de diseño” a tiempo.

Estructura
(qué cosas hay)

Comportamiento
(qué pasa y en qué orden)

Diagrama de Casos de Uso

Responde: ¿qué puede hacer el sistema y quién lo usa?

- Actor = quien usa el sistema.
- Caso de uso = óvalo con un VERBO.
- Rectángulo grande = límite del sistema.
- Línea = el actor usa esa función.
- Flecha discontínua de base a opción: <<include>> (siempre ocurre)
- Flehca disco´ntínua de opción a base: <<extend>> (opcional)

Ejemplo: Tienda Online

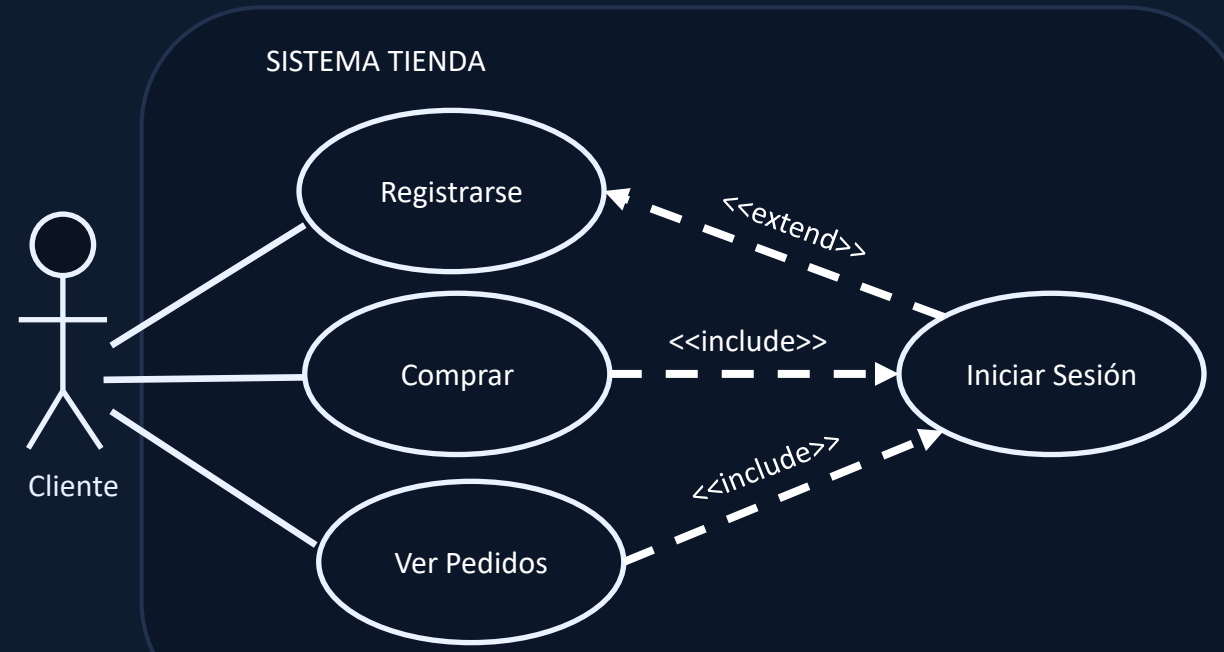


Diagrama de Casos de Uso

<<include>> vs <<extend>>

<<include>> (INCLUDE) — "siempre se hace"

- **Significa: este caso de uso SIEMPRE incluye otro.**
  - Es como "parte obligatoria" del proceso.
- Ejemplo biblioteca:
  - **Prestar libro** <<include>> **Registrar préstamo**
    Si prestas un libro, **sí o sí** hay que registrar el préstamo.
  - **Devolver libro** <<include>> **Registrar devolución**
    Si devuelves, **sí o sí** se registra la devolución.
- Regla fácil:
  include = "esto se hace siempre dentro de lo otro".
- Cómo se dibuja:
  - Flecha discontinua desde el caso principal al incluido
  - Etiqueta <<include>>

VS

<<extend>> (EXTEND) — "a veces se hace"

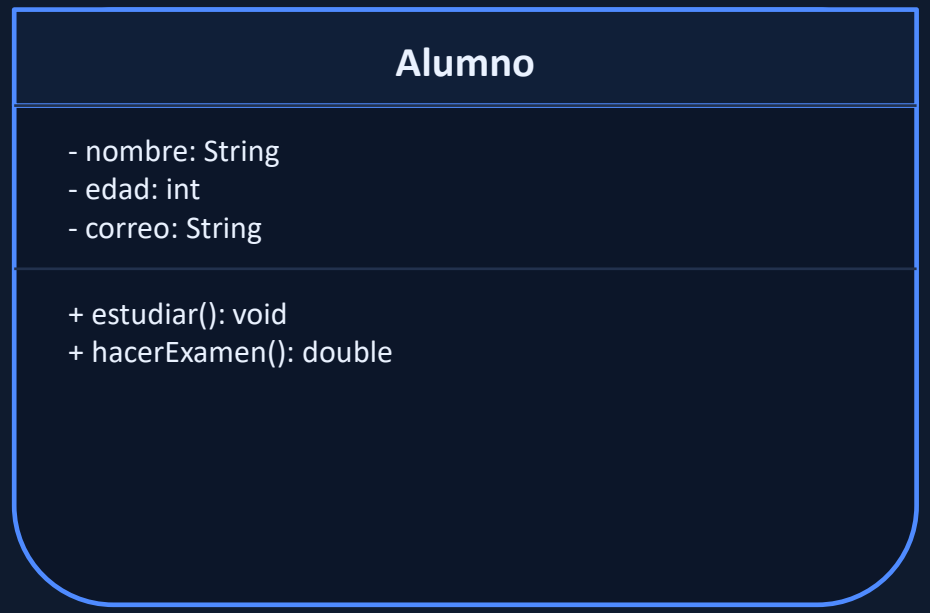
- **Significa: este caso de uso añade un comportamiento –**
  - **opcional** que ocurre **solo en algunas condiciones**.
- Ejemplos típicos:
  - **Comprar** <<extend>> **Aplicar descuento** (solo si tienes cupón)
  - **Iniciar sesión** <<extend>> **Recuperar contraseña** (solo si olvidaste)
  - **Prestar libro** <<extend>> **Avisar "no disponible"** (solo si el libro está prestado)
- Regla fácil:
  - extend = "esto pasa a veces, solo en ciertos casos".
- Cómo se dibuja:
  - Flecha discontinua desde el caso "extra" hacia el caso base
  - Etiqueta <<extend>>

Diagrama de Clases

Responde: ¿qué “cosas” hay y cómo se relacionan?

¿Cómo reconocerlo en un enunciado?

- SUSTANTIVOS → clases (Alumno, Libro, Pedido…).
- DATOS → atributos (nombre, edad, precio…).
- VERBOS → métodos (matricular(), prestar(), pagar()…).
- Relaciones: “tiene”, “contiene”, “pertenece a”…



Truco
Atributos = “tiene” (sustantivos)
Métodos = “hace” (verbos)

Relaciones y multiplicidades

La parte que más “cuesta”: cuántos hay en cada lado

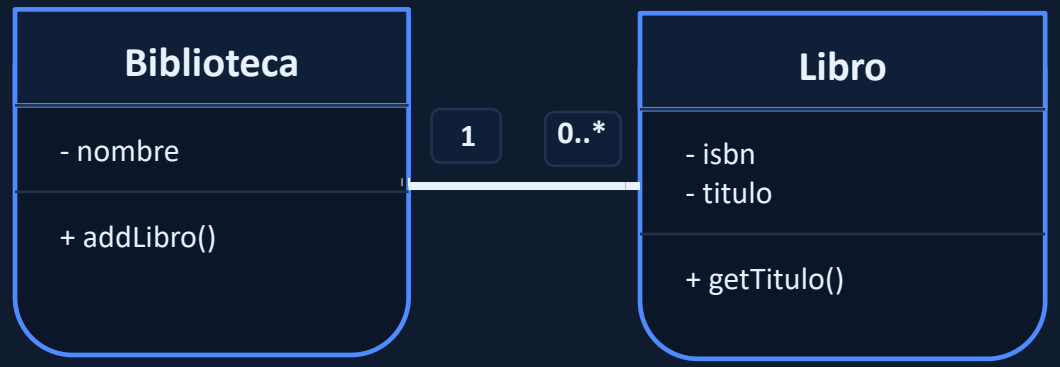
MULTIPLICIDADES TÍPICAS

- 1 = exactamente uno
- 0..1 = cero o uno
- * = muchos
- 0..* = cero o muchos
- 1..* = al menos uno

Cómo leerlo

Pon la multiplicidad cerca del extremo que “estás contando”.
Ej.: Biblioteca 1 — Libro 0..* (una biblioteca tiene muchos libros).

Ejemplo (líneas gruesas)



Lectura

Una Biblioteca (1) tiene cero o muchos Libros (0..*).
Cada Libro pertenece a una Biblioteca.

Relaciones: Unidireccionales vs Bidireccionales

Relaciones Unidireccionales y Bidireccionales

1) **Regla rápida con flechas (la más útil)**
   - **Unidireccional**: la línea tiene **flecha de navegación** hacia un lado
     👉 A → B (A conoce a B)
   - **Bidireccional**: la línea **no tiene flechas** (o tiene flechas en ambos lados)
     👉 A — B (ambos se conocen)

En muchos diagramas de clase "de examen" **no se dibujan flechas**, y entonces se suele asumir **bidireccional** *a nivel conceptual*, pero en Java puedes implementarlo uni o bi según lo que necesites.

2) Regla práctica (la que usan en programación)
- Si necesitas desde **Libro** saber su **Biblioteca** (libro.getBiblioteca()), entonces te hace falta **bidireccional** (y atributo Biblioteca biblioteca en Libro).
- Si solo vas a trabajar desde **Biblioteca** (listar libros, buscar libro, etc.), con **unidireccional** sobra (solo Biblioteca tiene la lista).

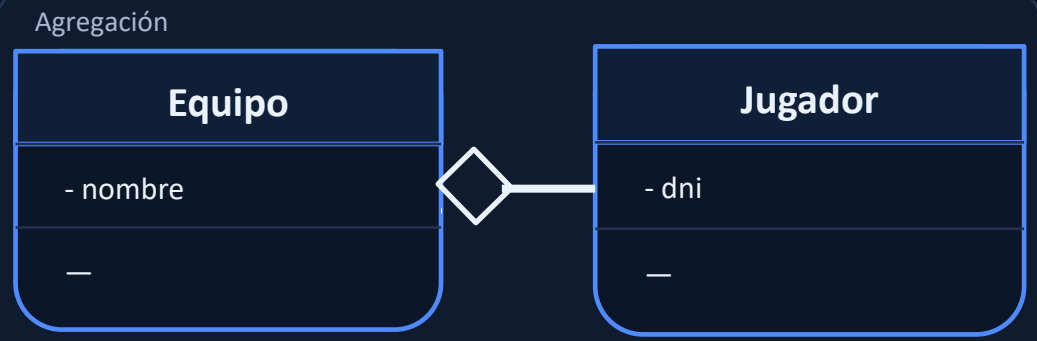
3) Pistas del enunciado
- Si dice: "La biblioteca **tiene/dispone de** libros y socios" → normalmente con unidireccional vale.
- Si dice: "Cada libro **pertenece a una biblioteca**" o "Un socio **está inscrito en una biblioteca**" → sugiere que Libro/Socio deben "conocer" la biblioteca (bidireccional).

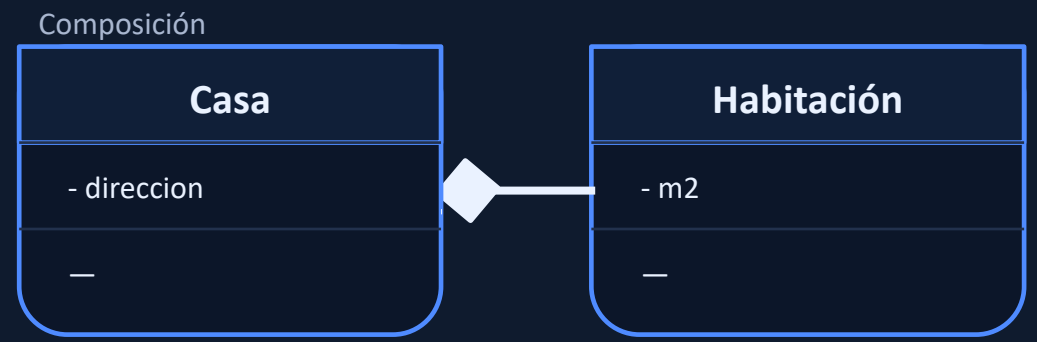
Agregación vs Composición

¿La “parte” puede existir sin el “todo”?

AGREGACIÓN vs COMPOSICIÓN

- Agregación: la parte puede vivir sola (rombo vacío).
- Composición: la parte depende del todo (rombo lleno).
- Si dudas: piensa si “se puede separar” en la vida real.





Clase Abstracta – Herencia (generalización)

¿Cuándo es una clase abstracta? <<abstract>>

Una **clase abstracta** se usa cuando tienes un concepto **general** que comparten varios tipos, pero **no tiene sentido** crear ese concepto "en bruto".

Ejemplo: Pagos en una tienda online
Imagina una tienda donde un cliente puede pagar de distintas formas:
- **PagoTarjeta / PagoPaypal / PagoBizum**

Todos son "pagos", así que comparten cosas comunes:
- Id / importe / fecha / EstadoPago

Pero cada tipo de pago tiene datos específicos:
- PagoTarjeta: numeroTarjeta, caducidad
- PagoPaypal: correoPaypal
- PagoBizum: teléfono

📌 Aquí tiene sentido crear una clase abstracta: **Pago** (abstracta) y que hereden:
- **PagoTarjeta / PagoPaypal / PagoBizum**

¿Por qué Pago sería abstracta?

- Porque en el sistema **nunca existe un pago "sin método"**. Siempre será tarjeta, PayPal o Bizum.
  Por tanto, "Pago" es solo el **concepto común**.

Qué ventaja te da

- Puedes tratar todos igual en el código y en UML:
- Guardar en una lista de Pago distintos tipos.
- Usar el mismo método general procesar() aunque cada tipo lo haga distinto.

Abstracta: cuando hay un "padre" común que no debe existir solo y sus hijos son los que se instancian.

En Java:

- extends

Dependencia

<<enum>>

¿Qué es una dependencia?

Una **dependencia** significa que **una clase "usa" a otra** para funcionar, pero **no la posee** como parte de su estructura principal.

Es una relación **débil**: si cambia la clase usada, puede afectar a la otra.

Ejemplo

- **Pago usa EstadoPago**, porque su atributo estado solo puede tener valores del enum.

Idea clave: ***Dependencia = "lo necesito / lo uso", no "lo tengo".***

¿Qué es un enum?

Un **enum** es un tipo de dato que solo puede tomar un **conjunto cerrado de valores**. Se usa cuando un atributo no puede ser "cualquier texto", sino una opción entre varias.

Ejemplo (Pagos)

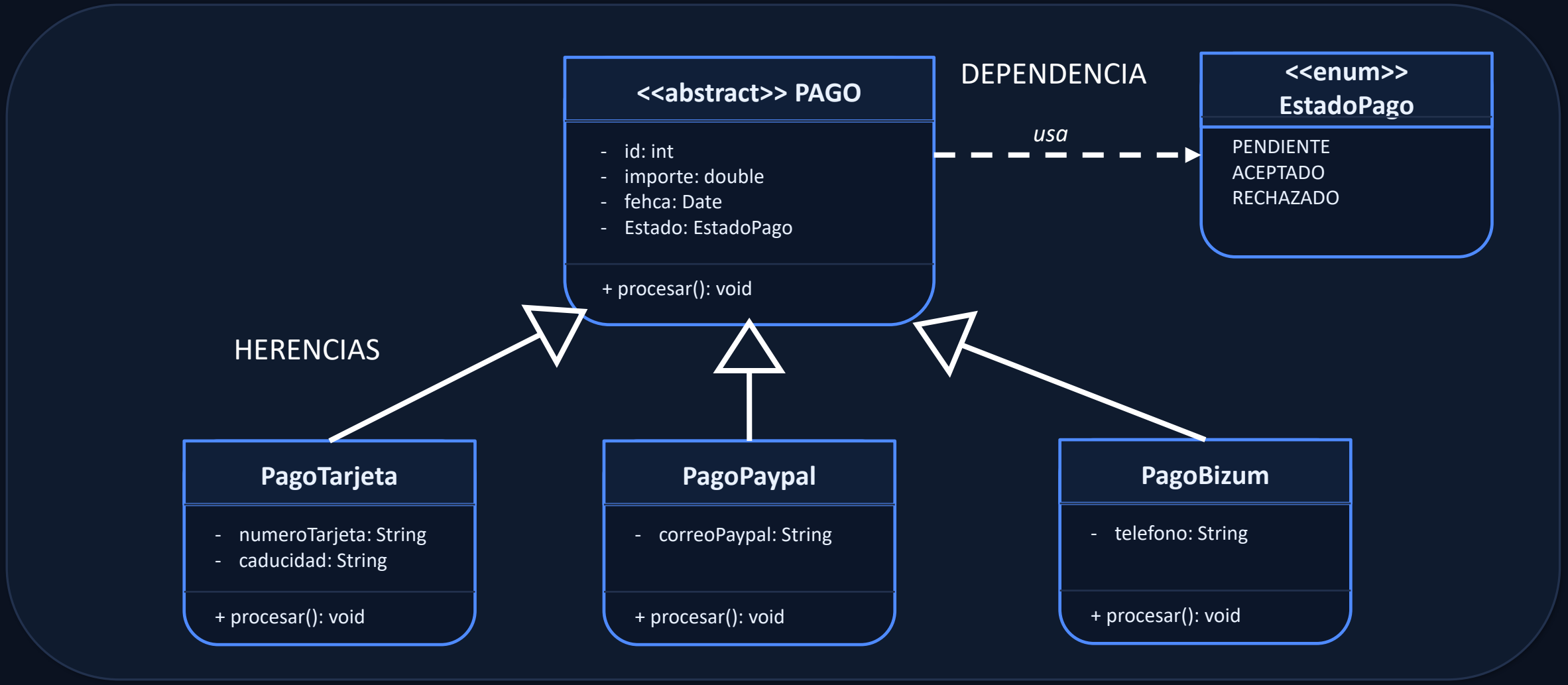
- Un pago puede estar solo en estos estados:
- **PENDIENTE / ACEPTADO / RECHAZADO**
- Eso se modela como un enum llamado **EstadoPago**.

¿Por qué se usa?

- Evita errores: no existen estados inventados.
- Aclara el diseño: todos conocen las opciones posibles.
- Facilita validaciones: el sistema sabe qué valores son válidos.

Clase Abstracta – Herencia (generalización)

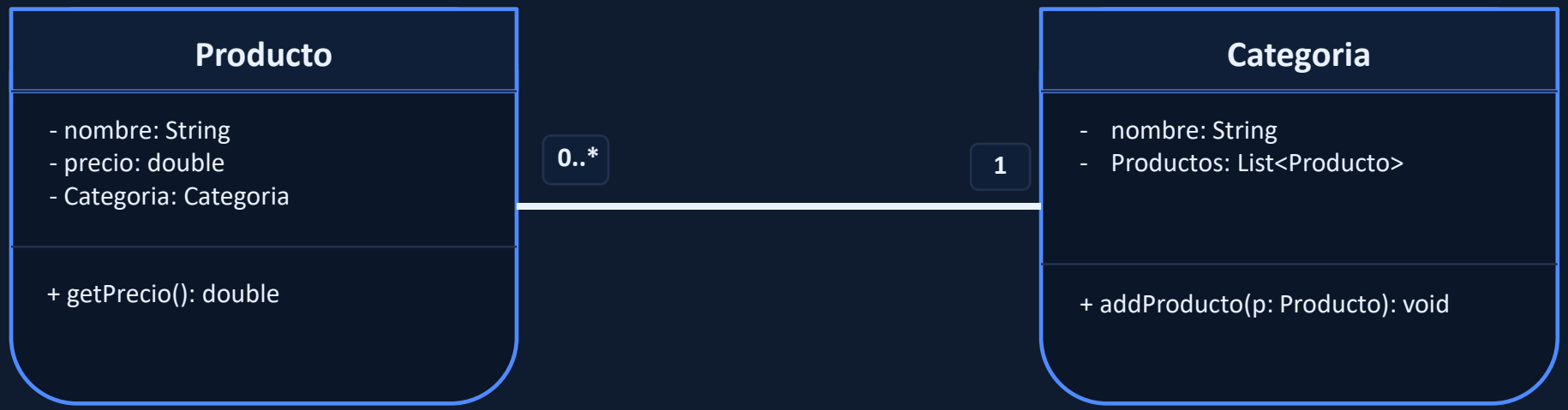
Diagrama



Ejemplo: del diagrama al código

Producto pertenece a una Categoría (1) y una Categoría tiene muchos Productos (0..*)

Diagrama



Qué significa

En Java: Categoria tendrá List<Producto> productos; y Producto tendrá Categoria categoria;

Ejemplo: del diagrama al código

Código (simplificado)

java

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

class Categoria {
    private String nombre;
    private List<Producto> productos = new ArrayList<>();
}

class Producto {
    private String nombre;
    private double precio;
    private Categoria categoria; // 1
}
```

Diagrama de Secuencia

Muestra mensajes en orden (el tiempo baja hacia abajo)

- Arriba: participantes (Usuario, App, BD…).
- Líneas verticales: “vida” del objeto.
- Flechas: mensajes (llamadas).
- Sirve para explicar un caso de uso paso a paso.

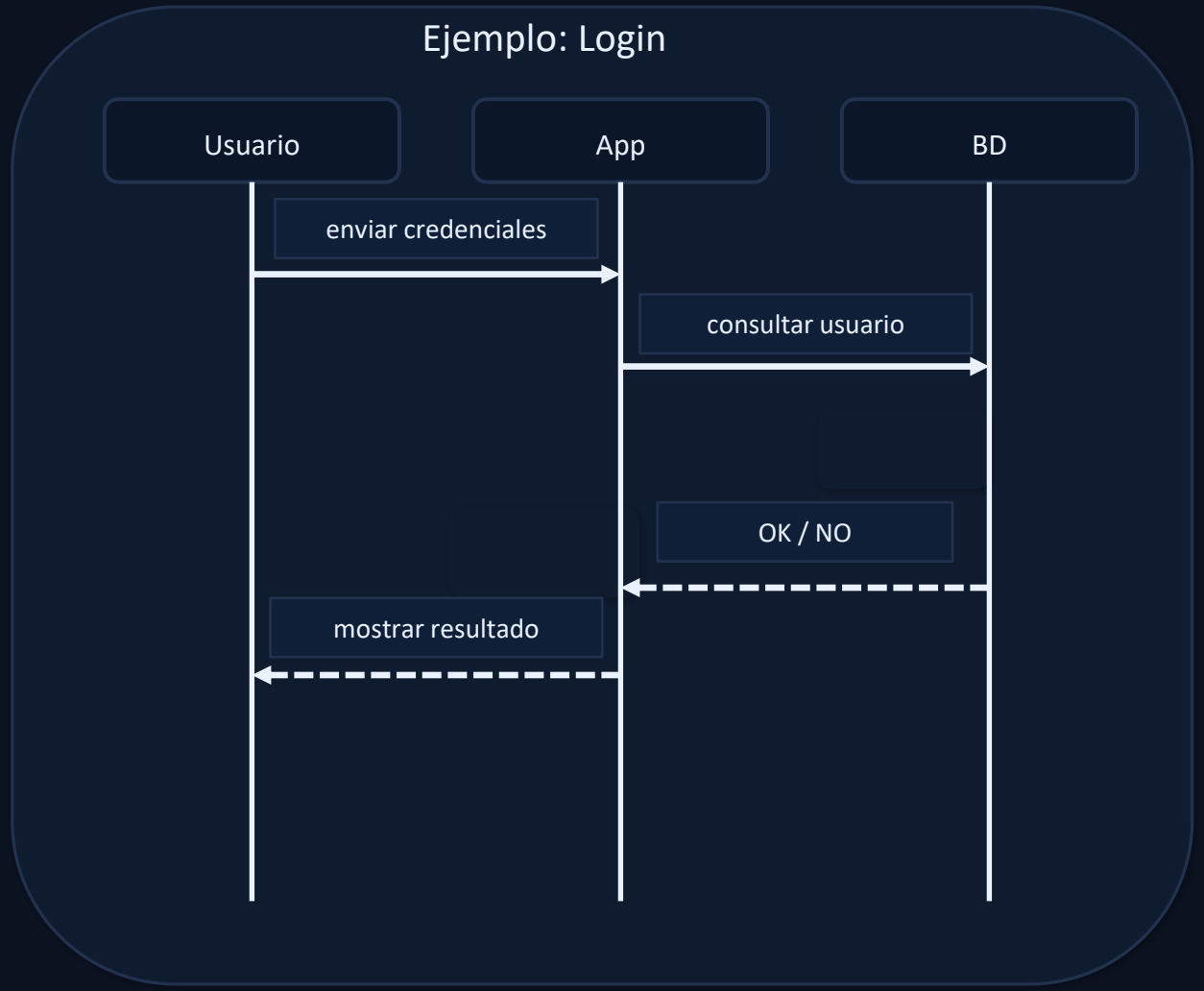


Diagrama de Secuencia: CAPAS

Resumen Capas

Idea clave: una petición siempre pasa por capas (ida) y vuelve con una respuesta (vuelta).

IDA (petición):

- UI → Controller → Service → Repository → DB

VUELTA (respuesta):

- DB → Repository → Service → Controller → UI

Resumen rápido de roles:

- **UI:** recoge datos y muestra resultados (no decide reglas).
- **Controller:** recibe la petición y llama al Service (no hace negocio).
- **Service:** decide reglas y coordina el proceso (validaciones).
- **Repository:** solo acceso a datos (buscar/guardar/actualizar/eliminar).
- **DB:** donde viven los datos (tablas).

Diagrama de Secuencia: CAPAS

Ejemplo: Admin crear Usuario

1) UI (Pantalla / Formulario)

Es **lo que toca el usuario**.

- Un formulario con "Nombre, Email, Password" y un botón **Crear**.
- La UI **recoge datos** y **muestra mensajes** ("Usuario creado", "Email inválido"…).

👉 La UI **no decide reglas importantes**. Solo muestra y recoge.

2) Controller (Entrada / Recepción)

Es como el **recepcionista**.

- Recibe lo que viene de la pantalla (la petición).
- Decide **a qué acción** se llama: crear, buscar, borrar…
- Llama al Service y luego devuelve la respuesta a la UI.

👉 El Controller **no hace reglas de negocio**, solo **recibe y deriva**.

3) Service (Reglas y flujo)

Es como el **encargado que sabe las normas**.

- Aquí van las **validaciones y decisiones**:
  - "¿El email tiene formato correcto?"
  - "¿La contraseña tiene mínimo 8 caracteres?"
  - "¿Ya existe ese email?"
- Decide si se puede crear o no y coordina el repositorio.

👉 El Service es el coordinador del proceso. Validaciones y comprobaciones.

4) Repository (Acceso a datos)

Es como el **archivero**.

- Su único trabajo es **hablar con la base de datos**:
  - guardar, buscar, actualizar, eliminar
- No valida reglas de negocio; solo hace operaciones de datos.

👉 Repository = "dame datos / guarda datos".

5) DB (Base de datos)

Es el **archivo/almacén real** (las tablas).

- Donde viven los usuarios guardados. Donde están los datos.

Diagrama de Secuencia: CAPAS

¿Cuándo usar CAPAS? – Crear Usuarios

¿Cuándo hay que usar capas (UI/Controller/Service/Repository/DB)?

Cuando el proyecto es una **aplicación web** (con **formularios**, **peticiones**, y una **BD real**), normalmente en **Entornos de Desarrollo** se espera el enfoque arquitectónico por capas:

- **UI**: formulario / pantalla de **registro de usuario**
- **Controller**: recibe la acción **"crear usuario"** (por ejemplo, al pulsar *Registrarse*)
- **Service**: aplica las **reglas de negocio** (¿datos válidos?, ¿email ya existe?, ¿contraseña cumple requisitos?)
- **Repository/DB**: acceso a datos (comprobar si existe el email, **guardar** el usuario, consultar/insertar)

Si el diagrama está hecho a nivel de dominio

En vez de capas, usas clases del **negocio** (dominio), por ejemplo:

Usuario, **RegistroUsuario** (o **GestorUsuarios**), **Credenciales**

Describe la lógica del negocio: **validar datos**, **comprobar email duplicado**, **crear el usuario**, etc.

Esto es totalmente válido si el objetivo es **modelar cómo funciona el sistema internamente** sin meterse en arquitectura (capas).

Diagrama de Secuencia: CAPAS

Ejemplo CREAR USUARIO con CAPAS

Ejemplo: “Crear usuario”:

- Imagina que el Admin rellena el formulario:
- **Nombre:** Ana; **Email:** ana@mail.com; **Pass:** 12345678

Flujo paso a paso:

1. **UI**: el Admin pulsa **Crear** → la pantalla envía los datos.
2. **Controller**: recibe “quiero crear usuario con esos datos” y llama al Service.
3. **Service**:
    1. comprueba formato del email
    2. comprueba que la contraseña cumple
    3. pide al repositorio: “¿existe ya ese email?”
4. **Repository**: consulta en la **DB** si existe ese email.
5. **DB**: responde “no existe”.
6. **Repository**: vuelve al Service con “no existe”.
7. **Service**: decide “OK, se puede crear” y pide al repositorio “guárdalo”.
8. **Repository**: hace INSERT en la **DB**.
9. **DB**: responde “guardado”.
10. **Controller**: devuelve a la UI “Usuario creado”.
11. **UI**: muestra “ ✅ Usuario creado correctamente”.

Diagrama de Secuencia: CAPAS

Ejemplo CREAR USUARIO con CAPAS

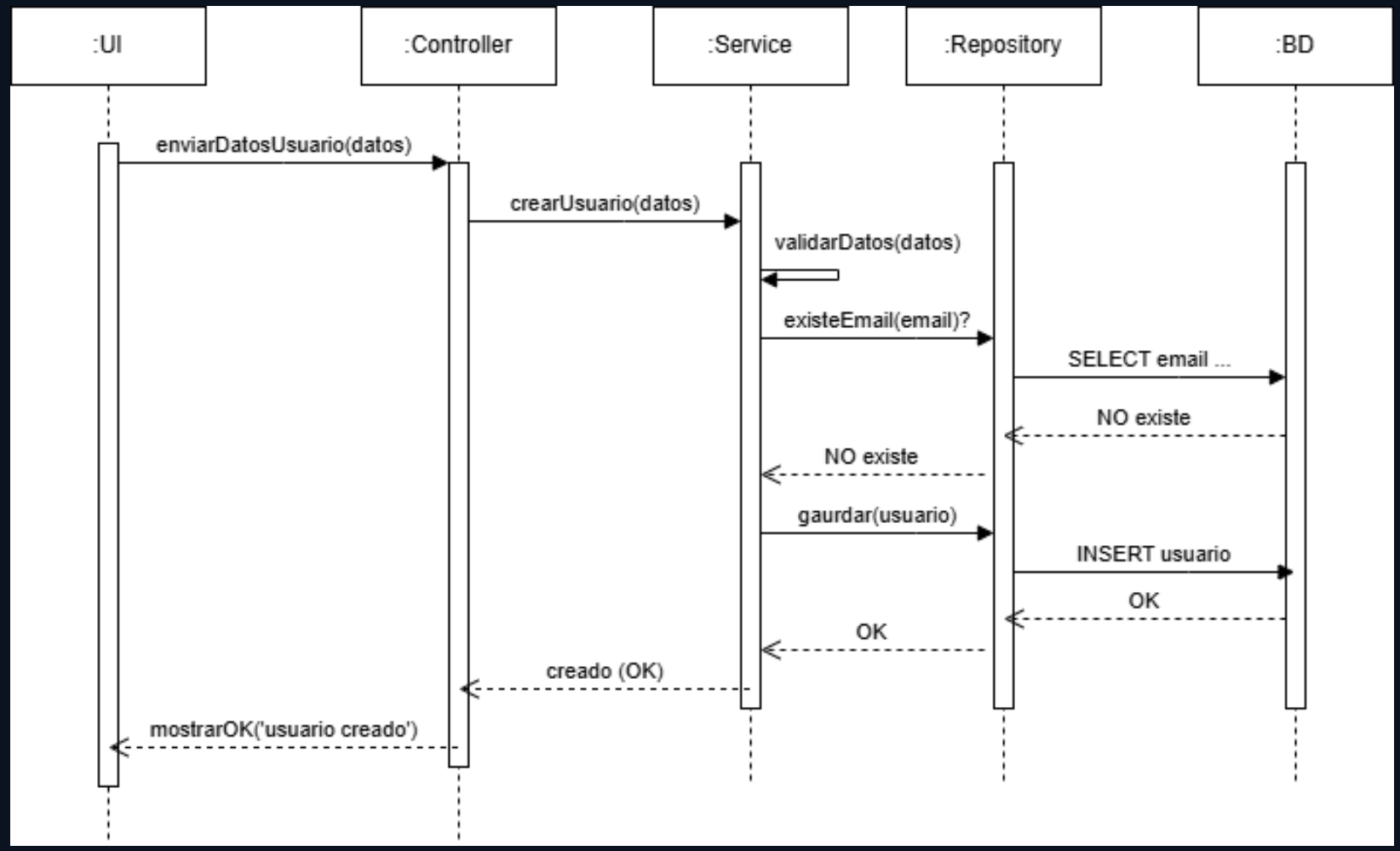


Diagrama de Secuencia: CAPAS - ALT

Ejemplo CREAR USUARIO con CAPAS - ALT

ALT (IF/ELSE) en “Crear usuario”

En este caso, el alt aparece cuando el **Service pregunta**:
“¿Existe ya ese email?”
A partir de ahí hay **dos caminos**.

alt [email NO existe] ✅ (caso OK)

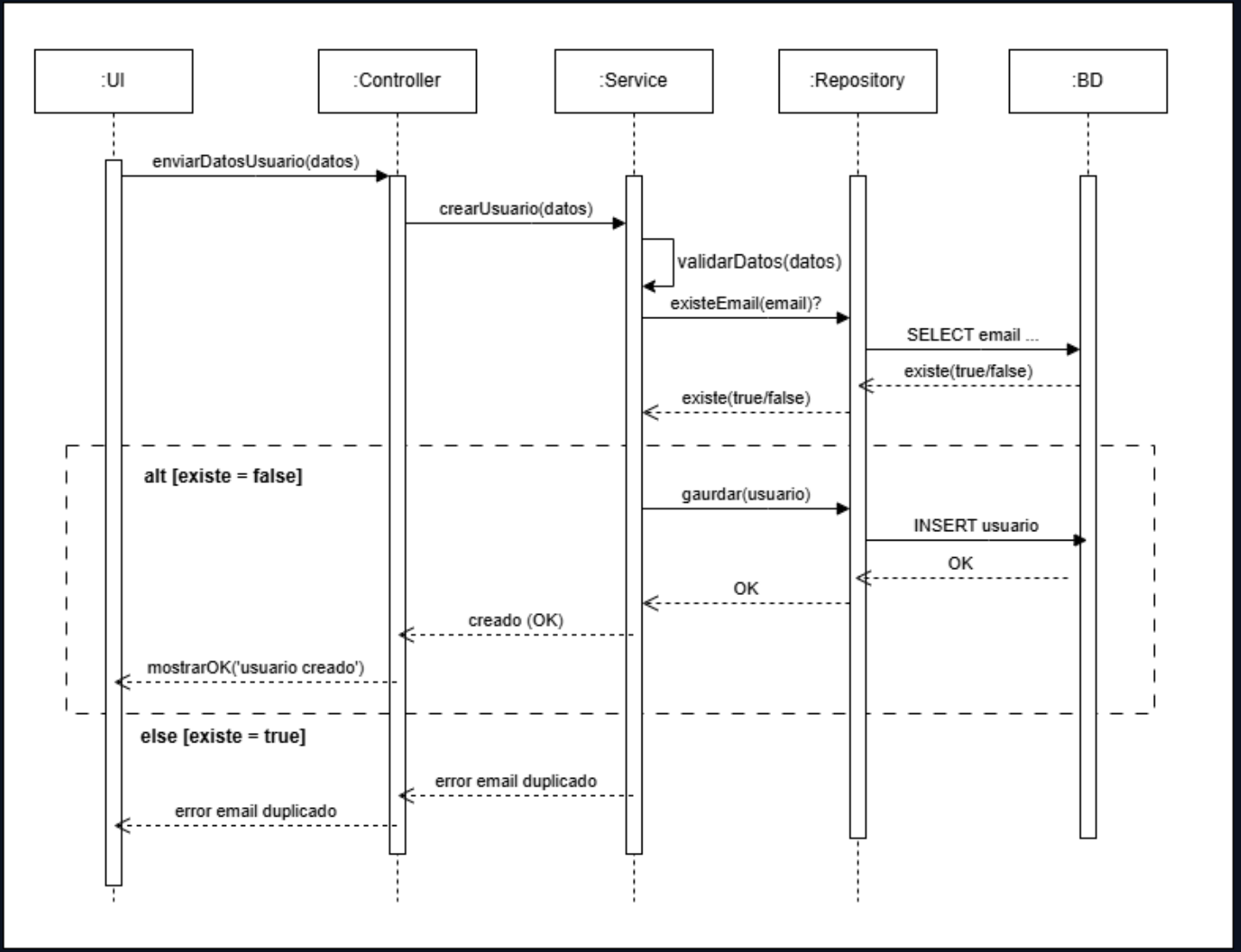
1. **UI:** Admin pulsa *Crear* → envía datos (Ana, ana@mail.com, 12345678).
2. **Controller:** recibe la petición y llama al **Service**.
3. **Service:** valida formato email y contraseña y pregunta al **Repository** si existe el email.
4. **Repository → DB:** consulta si existe ana@mail.com.
5. **DB:** responde **“no existe”**.
6. **Service:** decide **“OK, se puede crear”** y pide **guardar**.
7. **Repository → DB:** hace **INSERT** del usuario.
8. **DB:** responde **“guardado”**.
9. **Controller:** devuelve a la UI **“Usuario creado”**.
10. **UI:** muestra ✅ **Usuario creado correctamente**.

else [email YA existe] ❌ (caso ERROR)

1. **UI:** Admin pulsa *Crear* → envía datos.
2. **Controller:** llama al **Service**.
3. **Service:** valida datos y pregunta al repositorio si existe el email.
4. **Repository → DB:** consulta si existe ana@mail.com.
5. **DB:** responde **“sí existe”**.
6. **Service:** decide **“NO se puede crear”** (email duplicado).
7. **Controller:** devuelve a la UI **“Error: email ya registrado”**.
8. **UI:** muestra ❌ **Ese email ya existe. Usa otro.**

Diagrama de Secuencia: CAPAS

Ejemplo CREAR USUARIO con CAPAS y ALT



Ejercicio 1 — Biblioteca

Enunciado Completo

Se quiere modelar el funcionamiento básico de una **biblioteca**.

La biblioteca dispone de un conjunto de **libros** (identificados por *ISBN* y con *título*) y de una lista de **socios** (identificados por *DNI* y con *nombre*).

Un socio puede solicitar el **préstamo** de un libro. Para prestar un libro, la biblioteca debe **comprobar si el libro está disponible**.
Si está disponible, se crea un **préstamo** asociándolo a **un único socio** y a **un único libro**, registrando la **fecha de préstamo**. El libro pasa a estar **prestado** y el préstamo queda **registrado**.

Un socio también puede **devolver** un libro que tenga prestado. Al devolverlo, se debe **registrar la fecha de devolución** en el préstamo y el libro vuelve a estar **disponible**.
La biblioteca debe **mantener el historial de préstamos de cada socio** (préstamos activos y devueltos).

Se considera que un libro puede tener **como máximo un préstamo activo** al mismo tiempo, y un socio puede tener **varios préstamos** a lo largo del tiempo.

Ejercicio 1 — Biblioteca

Enunciado: Tareas

Tareas:

1.1. Dibuja el **diagrama de casos de uso**, indicando el actor y los casos de uso necesarios.

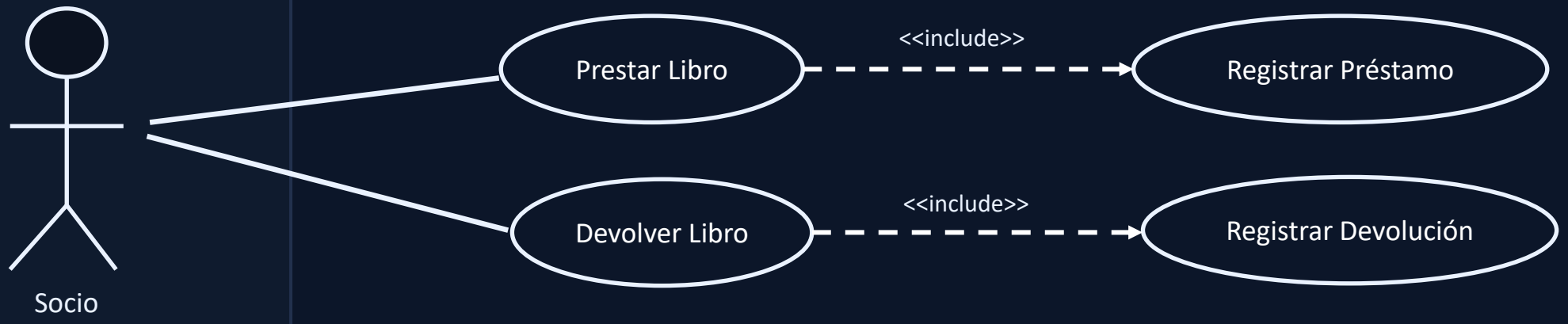
1.2. Dibuja el **diagrama de clases UML**, indicando **atributos**, **relaciones con multiplicidades** y los **métodos necesarios** para soportar el préstamo y la devolución.

1.3. Dibuja el **diagrama de secuencia** para:

- **1.3.1. Prestar un libro** (comprobación de disponibilidad, creación del préstamo y registro).
- **1.3.2. Devolver un libro** (registro de devolución y liberación del libro).

Solución Ej. 1 — Biblioteca

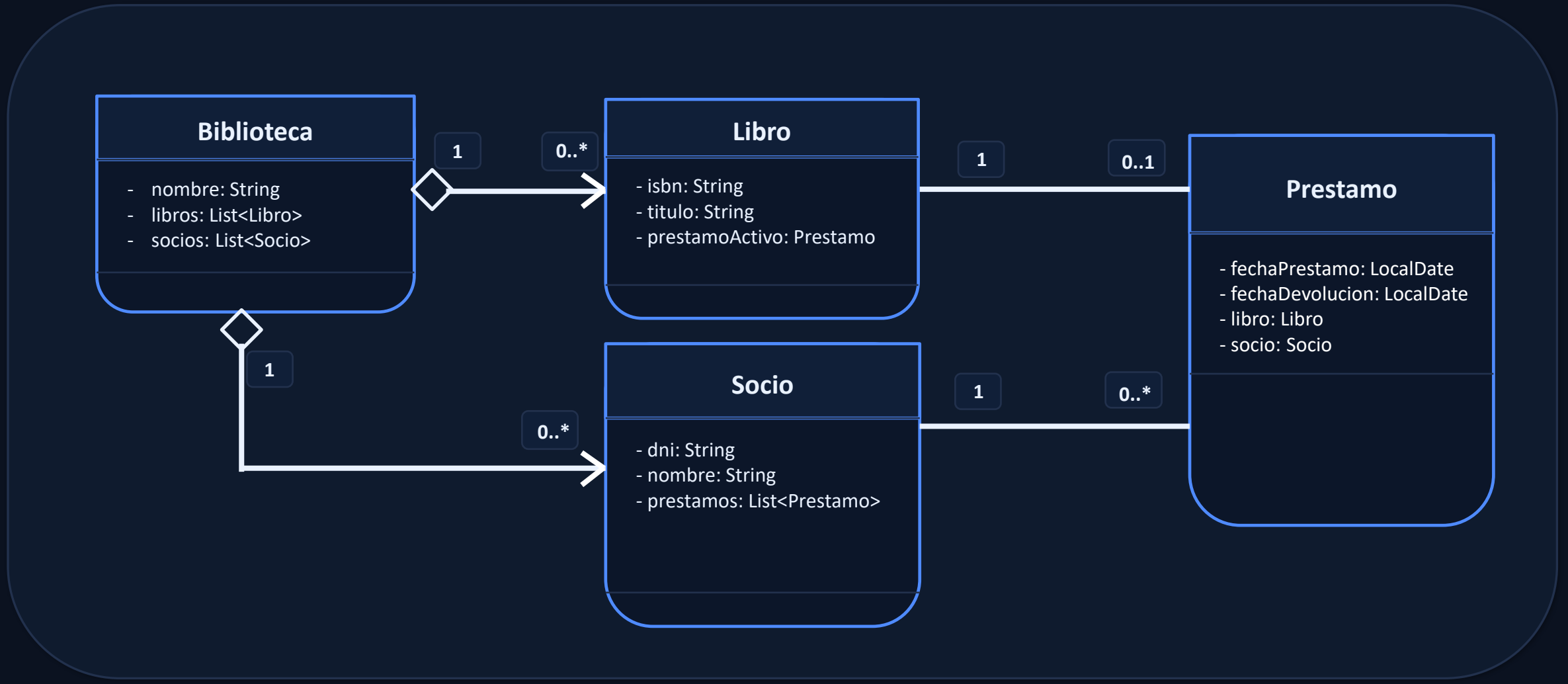
1.1. Diagrama de Casos de Uso



El actor **Socio** interactúa con el **Sistema Biblioteca** para realizar dos funcionalidades principales: **Prestar libro** y **Devolver libro**. En ambos casos se usa la relación <<include>> porque **siempre** que se presta un libro se debe **registrar el préstamo**, y **siempre** que se devuelve un libro se debe **registrar la devolución** como parte obligatoria del proceso.

Solución Ej. 1 — Biblioteca

1.2. Diagrama de clases: Relaciones y Atributos



Solución Ej. 1 — Biblioteca

Esqueleto Java (mínimo y legible)

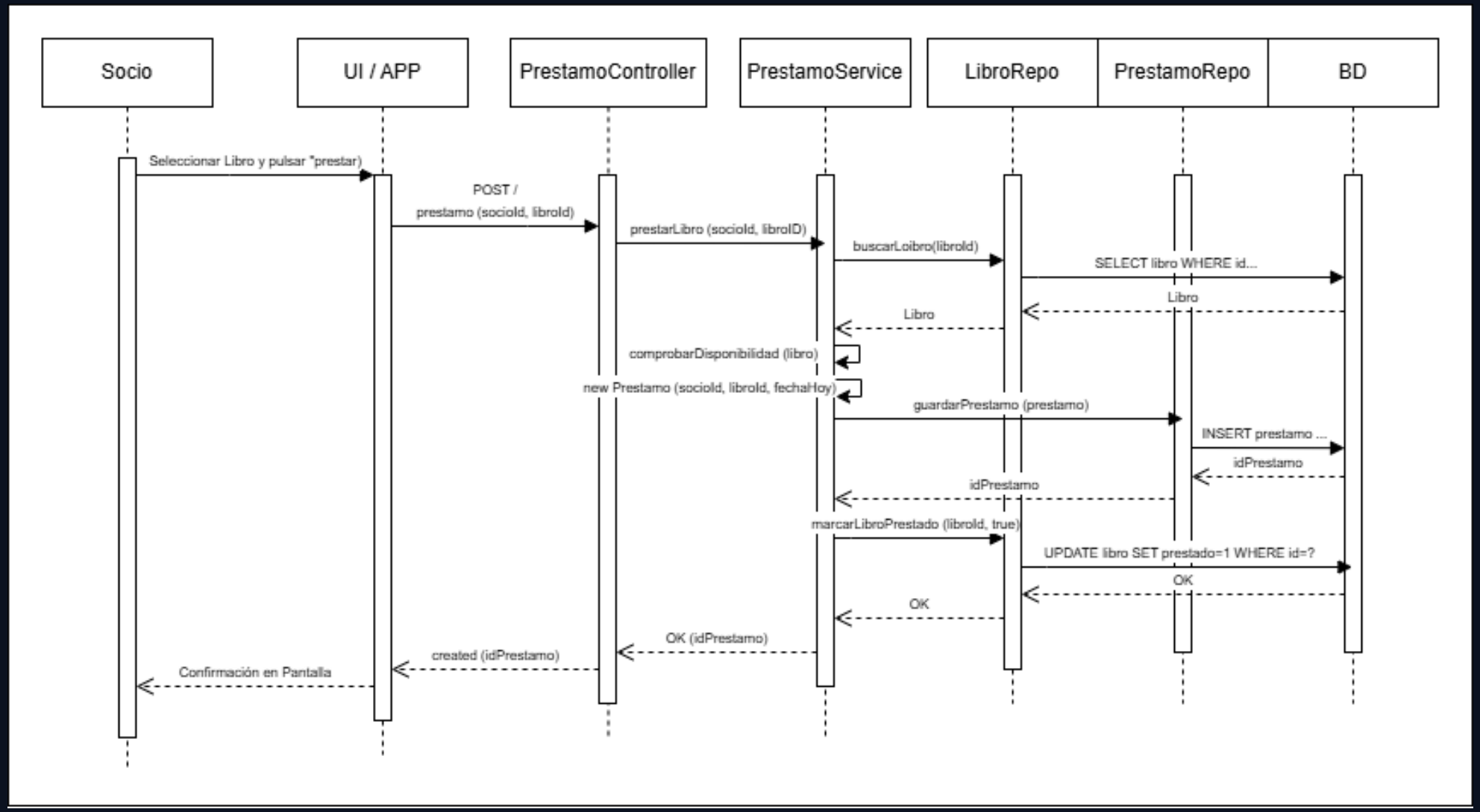
java

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

class Biblioteca {
    private String nombre;
    private List<Libro> libros = new ArrayList<>();
    private List<Socio> socios = new ArrayList<>();
}

class Libro {
    private String isbn;
    private String titulo;

    // 0..1 (null si está disponible)
    private Prestamo prestamoActivo;
}
```

```java
class Socio {
    private String dni;
    private String nombre;

    // 0..* -> histórico de préstamos del socio
    private List<Prestamo> prestamos = new ArrayList<>();
}

class Prestamo {
    private LocalDate fechaPrestamo;

    // 0..1 (null mientras el préstamo esté activo)
    private LocalDate fechaDevolucion;

    // 1 y 1
    private Libro libro;
    private Socio socio;
}
```

Solución Ej. 1 — Biblioteca

1.3.1. Diagrama de Secuencia: Prestar un Libro (CON CAPAS y SIN ALT)



Solución Ej. 1 — Biblioteca

1.3.2. Diagrama de Secuencia: Devolver un Libro (CON CAPAS y SIN ALT)