




Trabajo Práctico 5: Relaciones UML 1:1

Alumno: Calcatelli Renzo - rcalcatelli@gmail.com

Comisión: M2025-1

Materia: Programación II

Profesor: Ariel Enferrel

 [Enlace](#) al repositorio de GitHub

Trabajo Práctico 5: Relaciones UML 1:1

ÍNDICE

1. INTRODUCCIÓN

2. MARCO TEÓRICO

3. EJERCICIOS DE RELACIONES 1 A 1

- 3.1. Ejercicio 1: Pasaporte - Foto - Titular
- 3.2. Ejercicio 2: Celular - Batería - Usuario
- 3.3. Ejercicio 3: Libro - Autor - Editorial
- 3.4. Ejercicio 4: TarjetaDeCrédito - Cliente - Banco
- 3.5. Ejercicio 5: Computadora - PlacaMadre - Propietario
- 3.6. Ejercicio 6: Reserva - Cliente - Mesa
- 3.7. Ejercicio 7: Vehículo - Motor - Conductor
- 3.8. Ejercicio 8: Documento - FirmaDigital - Usuario
- 3.9. Ejercicio 9: CitaMédica - Paciente - Profesional
- 3.10. Ejercicio 10: CuentaBancaria - ClaveSeguridad - Titular

4. EJERCICIOS DE DEPENDENCIA DE USO

- 4.1. Ejercicio 11: Reproductor - Canción - Artista
- 4.2. Ejercicio 12: Impuesto - Contribuyente - Calculadora

5. EJERCICIOS DE DEPENDENCIA DE CREACIÓN

- 5.1. Ejercicio 13: GeneradorQR - Usuario - CódigoQR
- 5.2. Ejercicio 14: EditorVideo - Proyecto - Render

6. CONCLUSIONES

1. INTRODUCCIÓN

Contexto del Trabajo

El presente trabajo práctico tiene como objetivo principal profundizar en el modelado de relaciones entre clases utilizando el Lenguaje Unificado de Modelado (UML), específicamente enfocándose en relaciones uno a uno (1:1). Este tipo de modelado constituye uno de los pilares fundamentales del paradigma orientado a objetos, permitiendo representar de manera precisa las interacciones y dependencias entre diferentes entidades del sistema.

En el desarrollo de software moderno, la correcta identificación e implementación de relaciones entre clases es crucial para crear sistemas mantenibles, escalables y que reflejen fielmente la realidad del dominio del problema. Las relaciones UML no son meramente elementos gráficos, sino que tienen implicaciones directas en el diseño, la arquitectura y el comportamiento del código resultante.

Importancia del Modelado UML en la Programación

El **Unified Modeling Language** (UML) se ha establecido como el estándar de facto para el modelado de sistemas orientados a objetos. Su importancia radica en varios aspectos fundamentales:

- **Comunicación Clara:** UML proporciona un lenguaje visual común que permite a desarrolladores, analistas y stakeholders comunicarse de manera efectiva sobre la estructura y comportamiento del sistema, independientemente de su experiencia técnica específica.
- **Documentación Viva:** Los diagramas UML sirven como documentación que evoluciona junto con el código, facilitando el mantenimiento y la comprensión del sistema a largo plazo.
- **Análisis de Diseño:** Permite identificar problemas de diseño antes de la implementación, reduciendo costos y tiempo de desarrollo.
- **Estandarización:** Establece convenciones universales que facilitan el trabajo en equipos distribuidos y la transferencia de conocimiento entre proyectos.

Relaciones 1:1 en el Contexto Real

Las relaciones uno a uno son particularmente relevantes cuando modelamos entidades del mundo real donde existe una correspondencia única y específica entre objetos. Ejemplos cotidianos incluyen:

- Una persona y su pasaporte único
- Un vehículo y su número de patente
- Una cuenta bancaria y su clave de seguridad
- Un empleado y su número de legajo

Estas relaciones, aunque conceptualmente simples, presentan complejidades en su implementación que requieren un análisis cuidadoso del ciclo de vida de los objetos y las responsabilidades de cada clase.

Tipos de Relaciones Abordadas

En este trabajo práctico abordaremos los siguientes tipos de relaciones:

- **Asociación:** Representa una conexión semántica entre clases, indicando que los objetos de una clase están relacionados con objetos de otra clase. Puede ser unidireccional o bidireccional.
 - **Agregación:** Modela una relación "tiene-un" donde el objeto agregado puede existir independientemente del objeto contenedor. Representa una forma débil de contención.
 - **Composición:** Representa una forma fuerte de agregación donde el objeto contenido no puede existir sin el contenedor. El ciclo de vida del objeto contenido está íntimamente ligado al del contenedor.
 - **Dependencia:** Indica que una clase utiliza o depende de otra clase, pero sin mantener una referencia permanente. Se subdivide en dependencia de uso y dependencia de creación.
-

METODOLOGÍA Y CONSIDERACIONES TÉCNICAS

Criterios de Diseño Aplicados

Para cada ejercicio del trabajo práctico, se han aplicado los siguientes criterios de diseño:

- **Principio de Responsabilidad Única:** Cada clase tiene una única razón para cambiar y encapsula conceptos cohesivos del dominio del problema.
- **Encapsulamiento:** Los atributos se mantienen privados y se accede a ellos a través de métodos públicos apropiados, garantizando el control de acceso y validación.
- **Consistencia de Relaciones:** En las relaciones bidireccionales, se implementan mecanismos para mantener la integridad referencial automáticamente.
- **Gestión del Ciclo de Vida:** Se considera cuidadosamente cuándo y cómo se crean y destruyen los objetos, especialmente en composiciones.

2. MARCO TEÓRICO

Tipos de Relaciones UML

Tipo de Relación	Descripción	Notación UML	Ciclo de Vida
Asociación	Relación entre clases con referencia mutua o directa	Línea simple (→ o ↔)	Independiente
Agregación	Relación "tiene un" donde los objetos pueden vivir independientemente	Diamante vacío (◇→)	Independiente
Composición	Relación fuerte de contención, el ciclo de vida depende del contenedor	Diamante lleno (◆→)	Dependiente
Dependencia de Uso	Una clase usa otra como parámetro sin almacenarla	Línea punteada (-->)	Temporal
Dependencia de Creación	Una clase crea otra en tiempo de ejecución	Línea punteada con <<create>>	Temporal

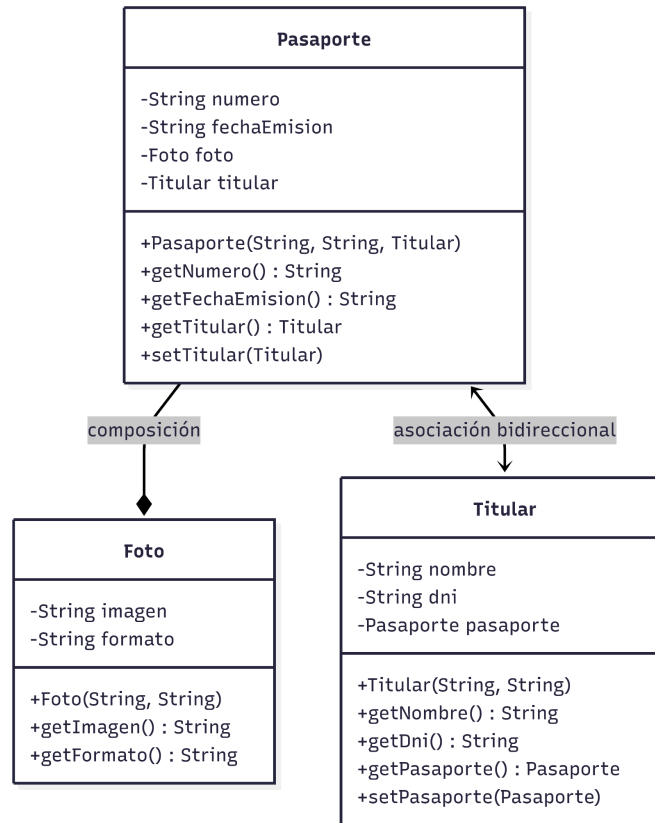
3. EJERCICIOS DE RELACIONES 1 A 1

Ejercicio 1: Pasaporte - Foto - Titular

Relaciones:

- **Composición:** Pasaporte → Foto (◆→)
- **Asociación bidireccional:** Pasaporte ↔ Titular (↔)

Diagrama UML:



Explicación:

- La **composición** indica que la Foto es parte integral del Pasaporte y no puede existir sin él.
- La **asociación bidireccional** permite que tanto el Pasaporte conozca a su Titular como viceversa.

Implementación en Java:

```

// Clase Foto
public class Foto {
    private String imagen;
    private String formato;

    public Foto(String imagen, String formato) {
        this.imagen = imagen;
        this.formato = formato;
    }

    public String getImagen() { return imagen; }
    public String getFormato() { return formato; }
}

```

```
// Clase Titular
public class Titular {
    private String nombre;
    private String dni;
    private Pasaporte pasaporte;

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

    public String getNombre() { return nombre; }
    public String getDni() { return dni; }
    public Pasaporte getPasaporte() { return pasaporte; }
    public void setPasaporte(Pasaporte pasaporte) { this.pasaporte =
pasaporte; }
}

// Clase Pasaporte
public class Pasaporte {
    private String numero;
    private String fechaEmision;
    private Foto foto; // Composición
    private Titular titular; // Asociación bidireccional

    public Pasaporte(String numero, String fechaEmision, Titular titular) {
        this.numero = numero;
        this.fechaEmision = fechaEmision;
        this.titular = titular;
        this.foto = new Foto("foto_pasaporte.jpg", "JPEG"); // Composición:
se crea automáticamente
        titular.setPasaporte(this); // Establece bidireccionalidad
    }

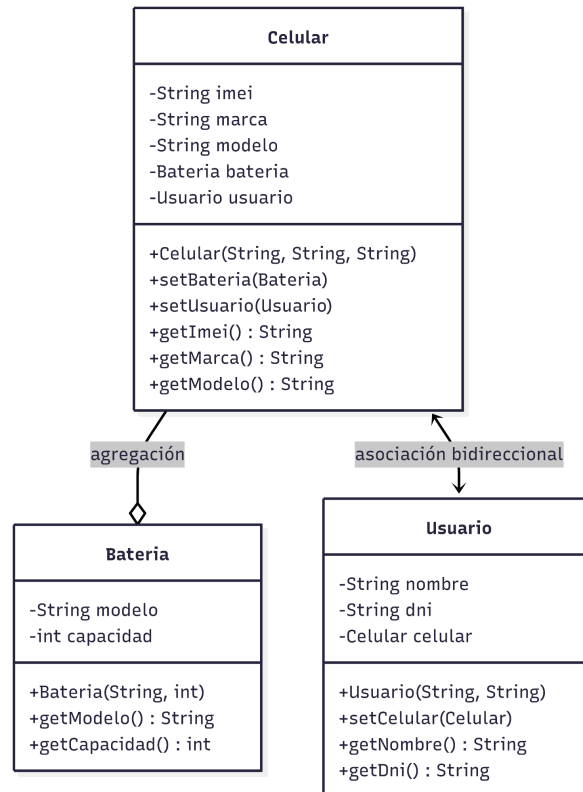
    public String getNumero() { return numero; }
    public String getFechaEmision() { return fechaEmision; }
    public Titular getTitular() { return titular; }
    public void setTitular(Titular titular) { this.titular = titular; }
    public Foto getFoto() { return foto; }
}
```

Ejercicio 2: Celular - Batería - Usuario

Relaciones:

- **Agregación:** Celular \rightarrow Batería ($\diamond \rightarrow$)
- **Asociación bidireccional:** Celular \leftrightarrow Usuario (\leftrightarrow)

Diagrama UML:



Explicación:

- La **agregación** indica que la Batería puede existir independientemente del Celular.
- La **asociación bidireccional** permite navegación en ambas direcciones entre Celular y Usuario.

Implementación Java:

```

// Clase Bateria
public class Bateria {
    private String modelo;
    private int capacidad;

    public Bateria(String modelo, int capacidad) {
        this.modelo = modelo;
    }
}
    
```



```
        this.capacidad = capacidad;
    }

    public String getModelo() { return modelo; }
    public int getCapacidad() { return capacidad; }
}

// Clase Usuario
public class Usuario {
    private String nombre;
    private String dni;
    private Celular celular;

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

    public String getNombre() { return nombre; }
    public String getDni() { return dni; }
    public Celular getCelular() { return celular; }
    public void setCelular(Celular celular) { this.celular = celular; }
}

// Clase Celular
public class Celular {
    private String imei;
    private String marca;
    private String modelo;
    private Bateria bateria; // Agregación
    private Usuario usuario; // Asociación bidireccional

    public Celular(String imei, String marca, String modelo) {
        this.imei = imei;
        this.marca = marca;
        this.modelo = modelo;
    }

    public void setBateria(Bateria bateria) { this.bateria = bateria; }
    public void setUsuario(Usuario usuario) {
        this.usuario = usuario;
        if (usuario != null) {
            usuario.setCelular(this); // Establece bidireccionalidad
        }
    }

    public String getImei() { return imei; }
    public String getMarca() { return marca; }
```

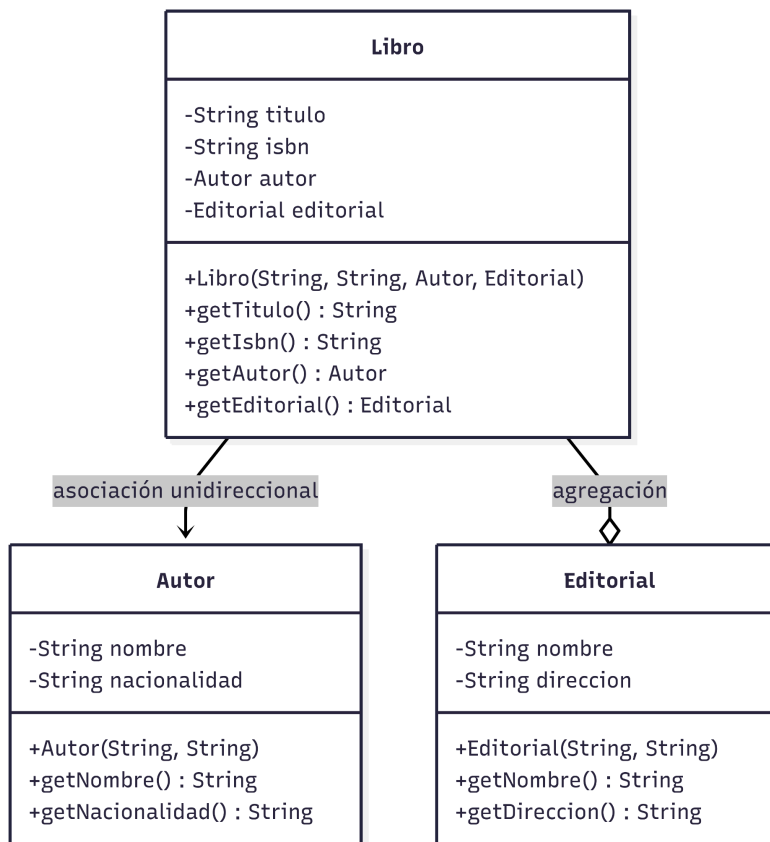
```
public String getModelo() { return modelo; }
public Bateria getBateria() { return bateria; }
public Usuario getUsuario() { return usuario; }
}
```

Ejercicio 3: Libro - Autor - Editorial

Relaciones:

- **Asociación unidireccional:** Libro → Autor (→)
- **Agregación:** Libro → Editorial (◊→)

Diagrama UML:



Implementación Java:

```
// Clase Autor
public class Autor {
    private String nombre;
    private String nacionalidad;

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

    public String getNombre() { return nombre; }
    public String getNacionalidad() { return nacionalidad; }
}

// Clase Editorial
public class Editorial {
    private String nombre;
    private String direccion;

    public Editorial(String nombre, String direccion) {
        this.nombre = nombre;
        this.direccion = direccion;
    }

    public String getNombre() { return nombre; }
    public String getDireccion() { return direccion; }
}

// Clase Libro
public class Libro {
    private String titulo;
    private String isbn;
    private Autor autor; // Asociación unidireccional
    private Editorial editorial; // Agregación

    public Libro(String titulo, String isbn, Autor autor, Editorial
editorial) {
        this.titulo = titulo;
        this.isbn = isbn;
        this.autor = autor;
    }
}
```

```

        this.editorial = editorial;
    }

    public String getTitulo() { return titulo; }
    public String getIsbn() { return isbn; }
    public Autor getAutor() { return autor; }
    public Editorial getEditorial() { return editorial; }
}

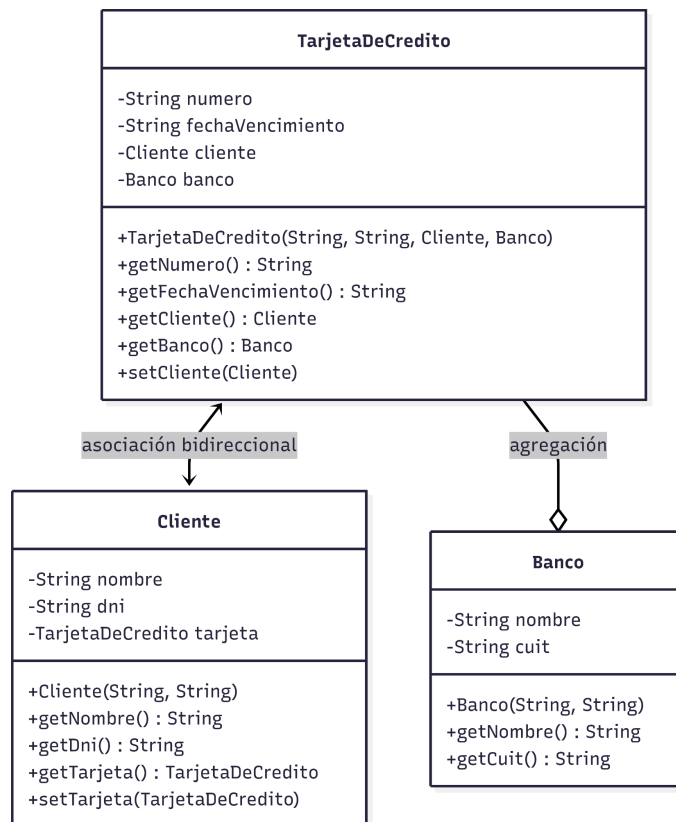
```

Ejercicio 4: TarjetaDeCrédito - Cliente - Banco

Relaciones:

- **Asociación bidireccional:** TarjetaDeCrédito ↔ Cliente
- **Agregación:** TarjetaDeCrédito → Banco

Diagrama UML:



Implementación Java:

```
// Clase Banco
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; }
}

// Clase Cliente
public class Cliente {
    private String nombre;
    private String dni;
    private TarjetaDeCredito tarjeta;

    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; }
    public void setTarjeta(TarjetaDeCredito tarjeta) { this.tarjeta =
tarjeta; }
}

// Clase TarjetaDeCredito
public class TarjetaDeCredito {
    private String numero;
    private String fechaVencimiento;
    private Cliente cliente; // Asociación bidireccional
    private Banco banco; // Agregación

    public TarjetaDeCredito(String numero, String fechaVencimiento, Cliente
cliente, Banco banco) {
```

```
this.numero = numero;
this.fechaVencimiento = fechaVencimiento;
this.cliente = cliente;
this.banco = banco;
cliente.setTarjeta(this); // Establece bidireccionalidad
}

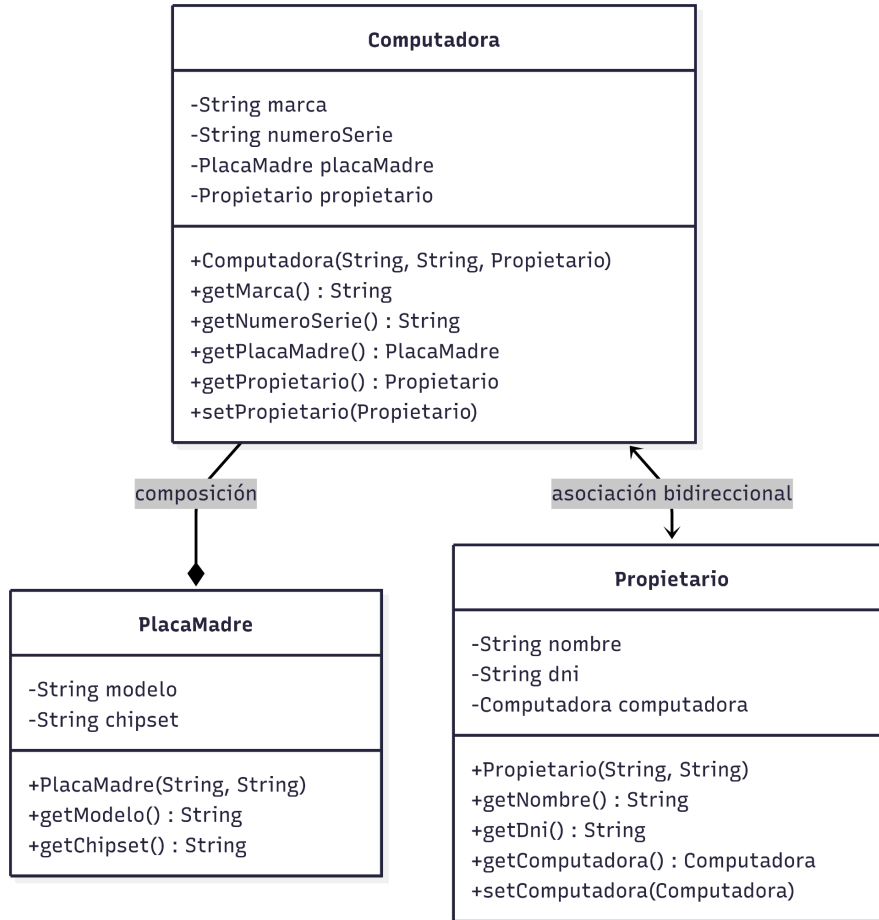
public String getNumero() { return numero; }
public String getFechaVencimiento() { return fechaVencimiento; }
public Cliente getCliente() { return cliente; }
public Banco getBanco() { return banco; }
public void setCliente(Cliente cliente) {
    this.cliente = cliente;
    if (cliente != null) {
        cliente.setTarjeta(this);
    }
}
}
```

Ejercicio 5: Computadora - PlacaMadre - Propietario

Relaciones:

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

Diagrama UML:



Implementación Java:

```

// Clase PlacaMadre
public class PlacaMadre {
    private String modelo;
    private String chipset;

    public PlacaMadre(String modelo, String chipset) {
        this.modelo = modelo;
        this.chipset = chipset;
    }

    public String getModelo() { return modelo; }
    public String getChipset() { return chipset; }
}

// Clase Propietario

```

```
public class Propietario {
    private String nombre;
    private String dni;
    private Computadora computadora;

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

    public String getNombre() { return nombre; }
    public String getDni() { return dni; }
    public Computadora getComputadora() { return computadora; }
    public void setComputadora(Computadora computadora) { this.computadora
= computadora; }
}

// Clase Computadora
public class Computadora {
    private String marca;
    private String numeroSerie;
    private PlacaMadre placaMadre; // Composición
    private Propietario propietario; // Asociación bidireccional

    public Computadora(String marca, String numeroSerie, Propietario
propietario) {
        this.marca = marca;
        this.numeroSerie = numeroSerie;
        this.propietario = propietario;
        this.placaMadre = new PlacaMadre("MSI B450", "AMD B450"); //
Composición: se crea automáticamente
        propietario.setComputadora(this); // Establece bidireccionalidad
    }

    public String getMarca() { return marca; }
    public String getNumeroSerie() { return numeroSerie; }
    public PlacaMadre getPlacaMadre() { return placaMadre; }
    public Propietario getPropietario() { return propietario; }
    public void setPropietario(Propietario propietario) {
        this.propietario = propietario;
        if (propietario != null) {
            propietario.setComputadora(this);
        }
    }
}
```



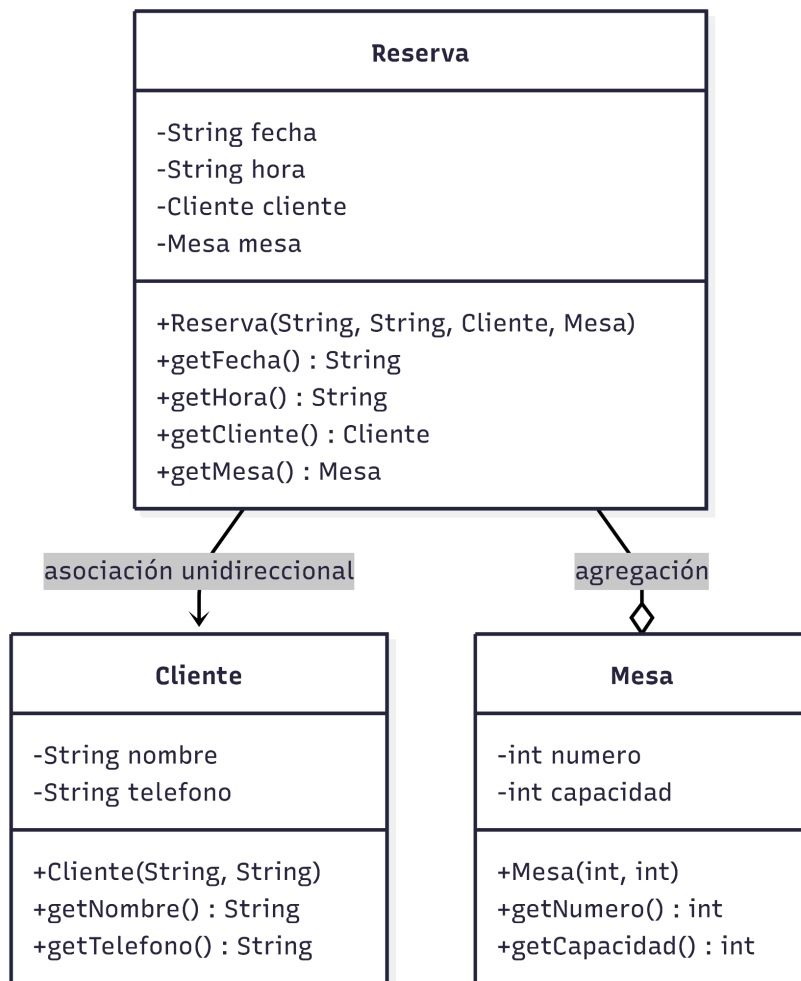
```
}
}
}
```

Ejercicio 6: Reserva - Cliente - Mesa

Relaciones:

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

Diagrama UML:



Implementación Java:

```
// Clase Cliente
public class Cliente {
    private String nombre;
    private String telefono;

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

    public String getNombre() { return nombre; }
    public String getTelefono() { return telefono; }
}

// Clase Mesa
public class Mesa {
    private int numero;
    private int capacidad;

    public Mesa(int numero, int capacidad) {
        this.numero = numero;
        this.capacidad = capacidad;
    }

    public int getNumero() { return numero; }
    public int getCapacidad() { return capacidad; }
}

// Clase Reserva
public class Reserva {
    private String fecha;
    private String hora;
    private Cliente cliente; // Asociación unidireccional
    private Mesa mesa; // Agregación

    public Reserva(String fecha, String hora, Cliente cliente, Mesa mesa) {
        this.fecha = fecha;
        this.hora = hora;
        this.cliente = cliente;
        this.mesa = mesa;
    }
}
```

```

}

public String getFecha() { return fecha; }
public String getHora() { return hora; }
public Cliente getCliente() { return cliente; }
public Mesa getMesa() { return mesa; }
}

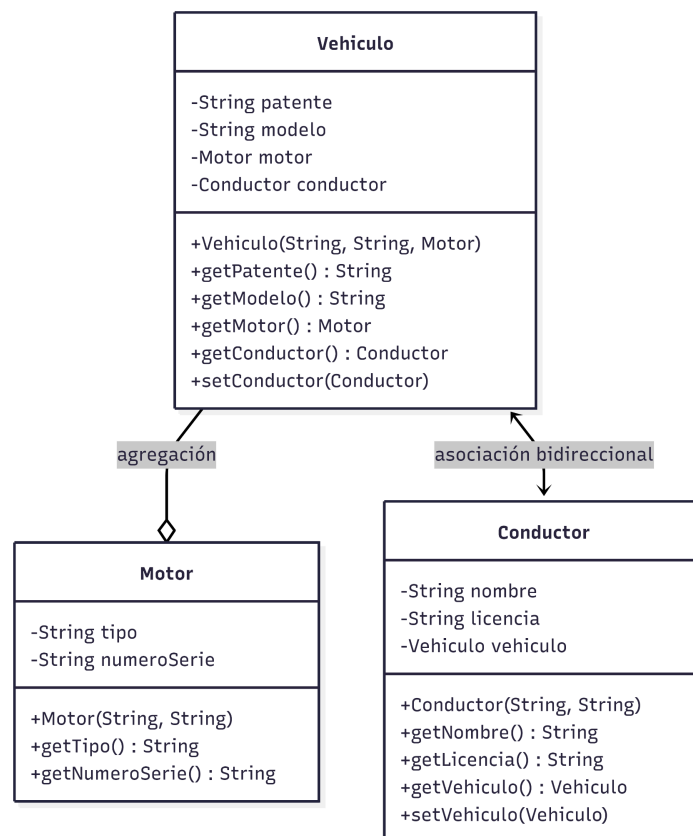
```

Ejercicio 7: Vehículo - Motor - Conductor

Relaciones:

- **Agregación:** Vehículo → Motor
- **Asociación bidireccional:** Vehículo ↔ Conductor

Diagrama UML:



Implementación Java:

```
// Clase Motor
public class Motor {
    private String tipo;
    private String numeroSerie;

    public Motor(String tipo, String numeroSerie) {
        this.tipo = tipo;
        this.numeroSerie = numeroSerie;
    }

    public String getTipo() { return tipo; }
    public String getNumeroSerie() { return numeroSerie; }
}

// Clase Conductor
public class Conductor {
    private String nombre;
    private String licencia;
    private Vehiculo vehiculo;

    public Conductor(String nombre, String licencia) {
        this.nombre = nombre;
        this.licencia = licencia;
    }

    public String getNombre() { return nombre; }
    public String getLicencia() { return licencia; }
    public Vehiculo getVehiculo() { return vehiculo; }
    public void setVehiculo(Vehiculo vehiculo) { this.vehiculo = vehiculo; }
}

// Clase Vehiculo
public class Vehiculo {
    private String patente;
    private String modelo;
    private Motor motor; // Agregación
    private Conductor conductor; // Asociación bidireccional

    public Vehiculo(String patente, String modelo, Motor motor) {
        this.patente = patente;
    }
}
```

```
        this.modelo = modelo;
        this.motor = motor;
    }

    public String getPatente() { return patente; }
    public String getModelo() { return modelo; }
    public Motor getMotor() { return motor; }
    public Conductor getConductor() { return conductor; }
    public void setConductor(Conductor conductor) {
        this.conductor = conductor;
        if (conductor != null) {
            conductor.setVehiculo(this); // Establece bidireccionalidad
        }
    }
}
```

Ejercicio 8: Documento - FirmaDigital - Usuario

Relaciones:

- **Composición:** Documento → FirmaDigital
- **Agregación:** FirmaDigital → Usuario

Diagrama UML:



Implementación Java:

```
// Clase Usuario
public class Usuario {
    private String nombre;
    private String email;

    public Usuario(String nombre, String email) {
        this.nombre = nombre;
        this.email = email;
    }

    public String getNombre() { return nombre; }
    public String getEmail() { return email; }
}

// Clase FirmaDigital
public class FirmaDigital {
    private String codigoHash;
    private String fecha;
    private Usuario usuario; // Agregación

    public FirmaDigital(String codigoHash, String fecha, Usuario usuario) {
        this.codigoHash = codigoHash;
        this.fecha = fecha;
        this.usuario = usuario;
    }

    public String getCodigoHash() { return codigoHash; }
    public String getFecha() { return fecha; }
    public Usuario getUsuario() { return usuario; }
}

// Clase Documento
public class Documento {
    private String titulo;
    private String contenido;
    private FirmaDigital firmaDigital; // Composición

    public Documento(String titulo, String contenido, Usuario usuario) {
        this.titulo = titulo;
        this.contenido = contenido;
        this.firmaDigital = new FirmaDigital("SHA256HASH123", "2024-01-15",
```

```

usuario); // Composición: se crea automáticamente
    }

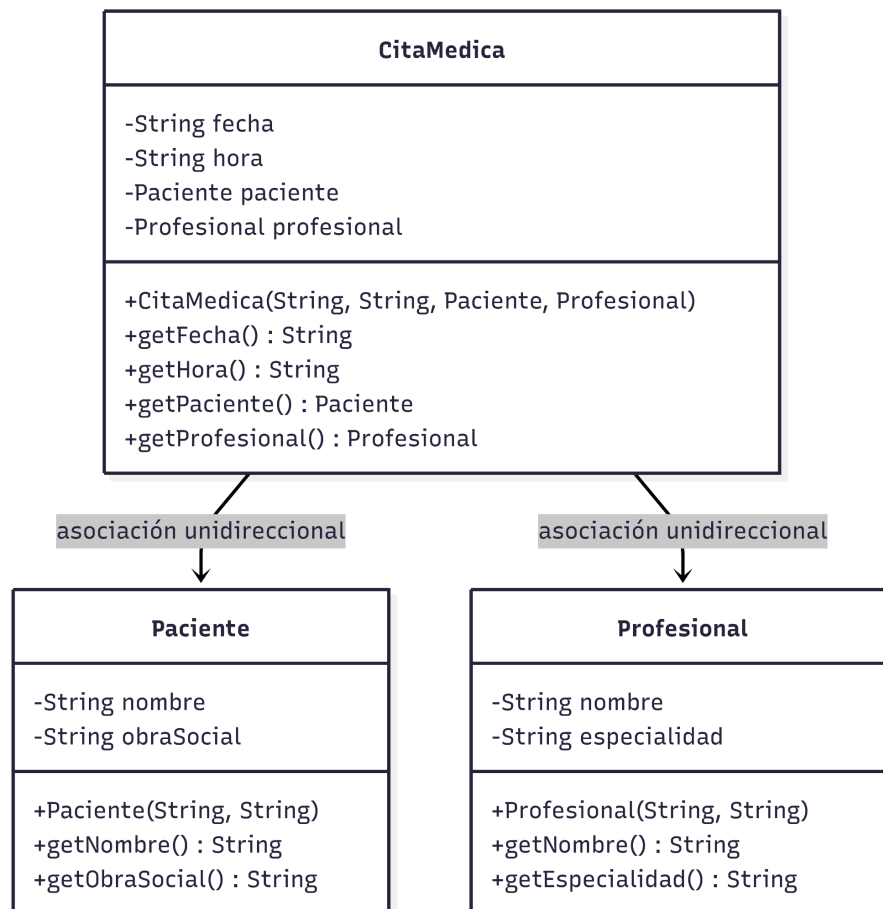
    public String getTitulo() { return titulo; }
    public String getContenido() { return contenido; }
    public FirmaDigital getFirmaDigital() { return firmaDigital; }
}
    
```

Ejercicio 9: CitaMédica - Paciente - Profesional

Relaciones:

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

Diagrama UML:



Implementación Java:

```
// Clase Paciente
public class Paciente {
    private String nombre;
    private String obraSocial;

    public Paciente(String nombre, String obraSocial) {
        this.nombre = nombre;
        this.obraSocial = obraSocial;
    }

    public String getNombre() { return nombre; }
    public String getObraSocial() { return obraSocial; }
}

// Clase Profesional
public class Profesional {
    private String nombre;
    private String especialidad;

    public Profesional(String nombre, String especialidad) {
        this.nombre = nombre;
        this.especialidad = especialidad;
    }

    public String getNombre() { return nombre; }
    public String getEspecialidad() { return especialidad; }
}

// Clase CitaMedica
public class CitaMedica {
    private String fecha;
    private String hora;
    private Paciente paciente; // Asociación unidireccional
    private Profesional profesional; // Asociación unidireccional

    public CitaMedica(String fecha, String hora, Paciente paciente,
        Profesional profesional) {
        this.fecha = fecha;
        this.hora = hora;
        this.paciente = paciente;
    }
}
```

```

        this.profesional = profesional;
    }

    public String getFecha() { return fecha; }
    public String getHora() { return hora; }
    public Paciente getPaciente() { return paciente; }
    public Profesional getProfesional() { return profesional; }
}

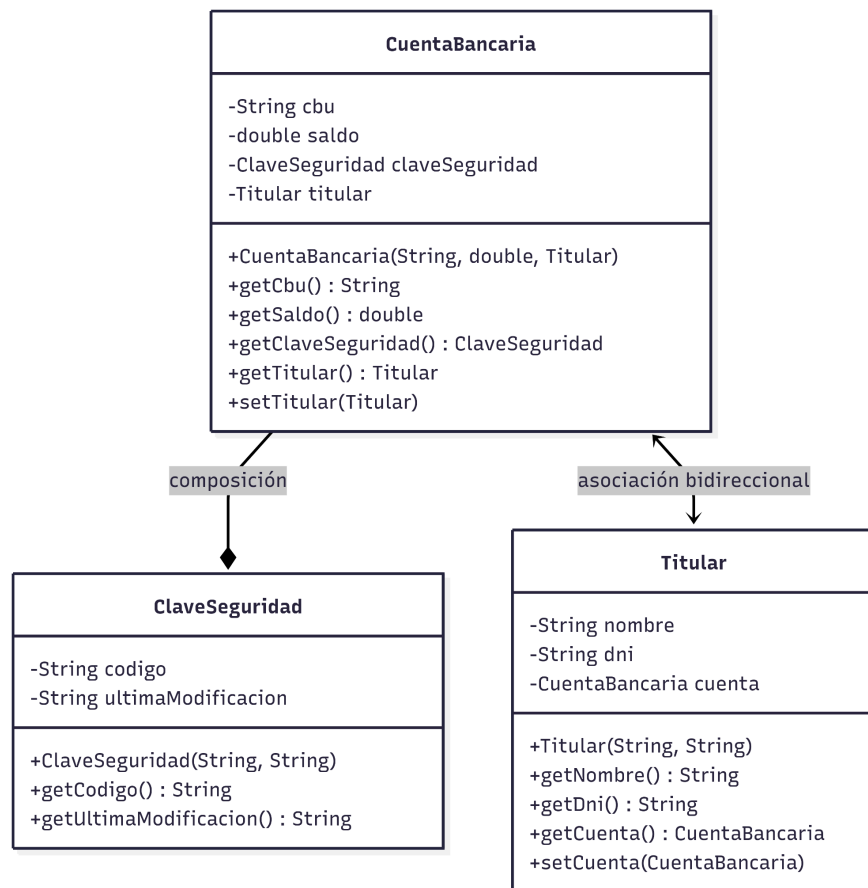
```

Ejercicio 10: CuentaBancaria - ClaveSeguridad - Titular

Relaciones:

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

Diagrama UML:



Implementación Java:

```
// Clase ClaveSeguridad
public class ClaveSeguridad {
    private String codigo;
    private String ultimaModificacion;

    public ClaveSeguridad(String codigo, String ultimaModificacion) {
        this.codigo = codigo;
        this.ultimaModificacion = ultimaModificacion;
    }

    public String getCodigo() { return codigo; }
    public String getUltimaModificacion() { return ultimaModificacion; }
}

// Clase Titular
public class Titular {
    private String nombre;
    private String dni;
    private CuentaBancaria cuenta;

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

    public String getNombre() { return nombre; }
    public String getDni() { return dni; }
    public CuentaBancaria getCuenta() { return cuenta; }
    public void setCuenta(CuentaBancaria cuenta) { this.cuenta = cuenta; }
}

// Clase CuentaBancaria
public class CuentaBancaria {
    private String cbu;
    private double saldo;
    private ClaveSeguridad claveSeguridad; // Composición
    private Titular titular; // Asociación bidireccional

    public CuentaBancaria(String cbu, double saldo, Titular titular) {
        this.cbu = cbu;
        this.saldo = saldo;
    }
}
```

```
        this.titular = titular;
        this.claveSeguridad = new ClaveSeguridad("1234", "2024-01-15"); //
Composición: se crea automáticamente
        titular.setCuenta(this); // Establece bidireccionalidad
    }

    public String getCbu() { return cbu; }
    public double getSaldo() { return saldo; }
    public ClaveSeguridad getClaveSeguridad() { return claveSeguridad; }
    public Titular getTitular() { return titular; }
    public void setTitular(Titular titular) {
        this.titular = titular;
        if (titular != null) {
            titular.setCuenta(this);
        }
    }
}
```

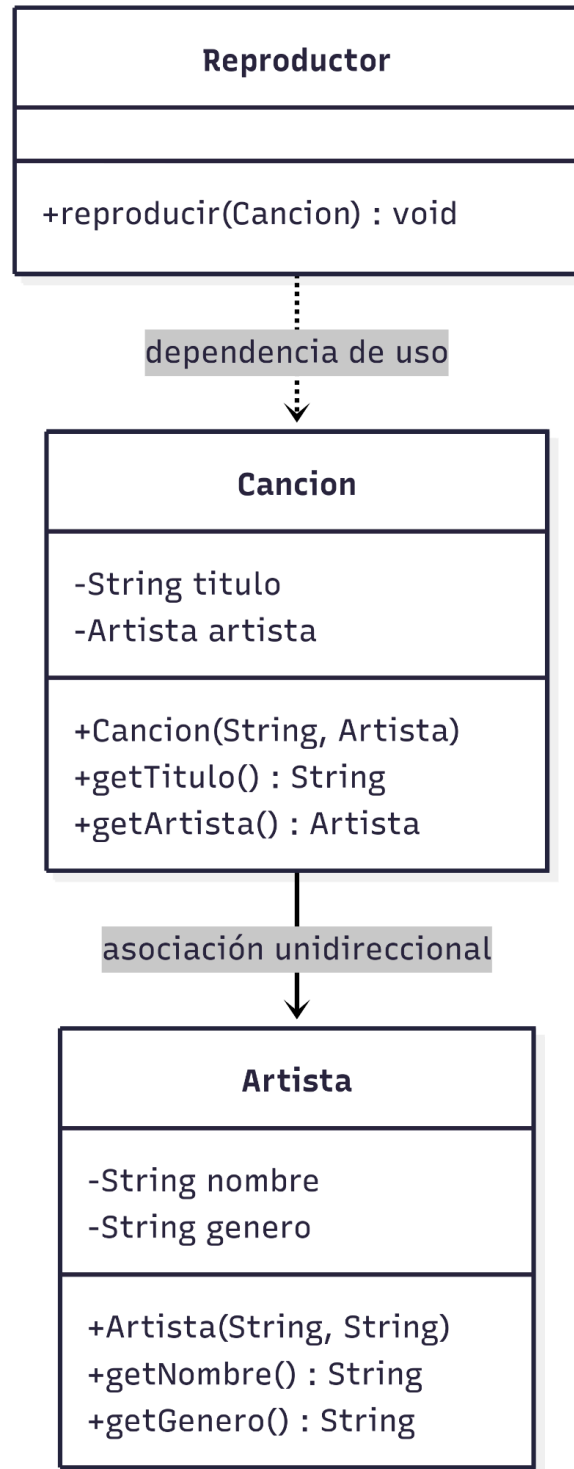
EJERCICIOS DE DEPENDENCIA DE USO

Ejercicio 11: Reproductor - Canción - Artista

Relaciones:

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

Diagrama UML:



Implementación Java:

```
// Clase Artista
public class Artista {
    private String nombre;
    private String genero;

    public Artista(String nombre, String genero) {
        this.nombre = nombre;
        this.genero = genero;
    }

    public String getNombre() { return nombre; }
    public String getGenero() { return genero; }
}

// Clase Cancion
public class Cancion {
    private String titulo;
    private Artista artista; // Asociación unidireccional

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

    public String getTitulo() { return titulo; }
    public Artista getArtista() { return artista; }
}

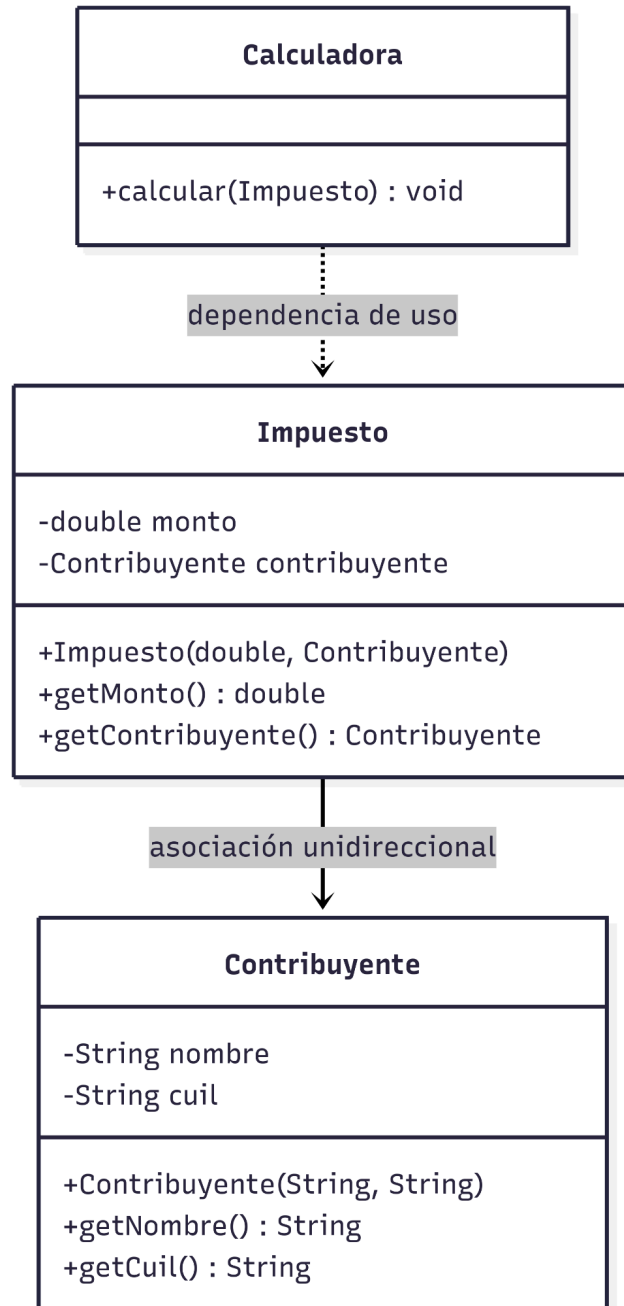
// Clase Reproductor
public class Reproductor {
    // Dependencia de uso: usa Cancion como parámetro pero no la almacena
    // como atributo
    public void reproducir(Cancion cancion) {
        System.out.println("Reproduciendo: " + cancion.getTitulo() +
                           " de " + cancion.getArtista().getNombre());
    }
}
```

Ejercicio 12: Impuesto - Contribuyente - Calculadora

Relaciones:

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

Diagrama UML:



Implementación Java:

```
// Clase Contribuyente
public class Contribuyente {
    private String nombre;
    private String cuil;

    public Contribuyente(String nombre, String cuil) {
        this.nombre = nombre;
        this.cuil = cuil;
    }

    public String getNombre() { return nombre; }
    public String getCuil() { return cuil; }
}

// Clase Impuesto
public class Impuesto {
    private double monto;
    private Contribuyente contribuyente; // Asociación unidireccional

    public Impuesto(double monto, Contribuyente contribuyente) {
        this.monto = monto;
        this.contribuyente = contribuyente;
    }

    public double getMonto() { return monto; }
    public Contribuyente getContribuyente() { return contribuyente; }
}

// Clase Calculadora
public class Calculadora {
    // Dependencia de uso: usa Impuesto como parámetro pero no la almacena
    // como atributo
    public void calcular(Impuesto impuesto) {
        double impuestoCalculado = impuesto.getMonto() * 1.21; // Ejemplo:
        IVA 21%
        System.out.println("Impuesto calculado para " +
            impuesto.getContribuyente().getNombre() +
            ": $" + impuestoCalculado);
    }
}
```

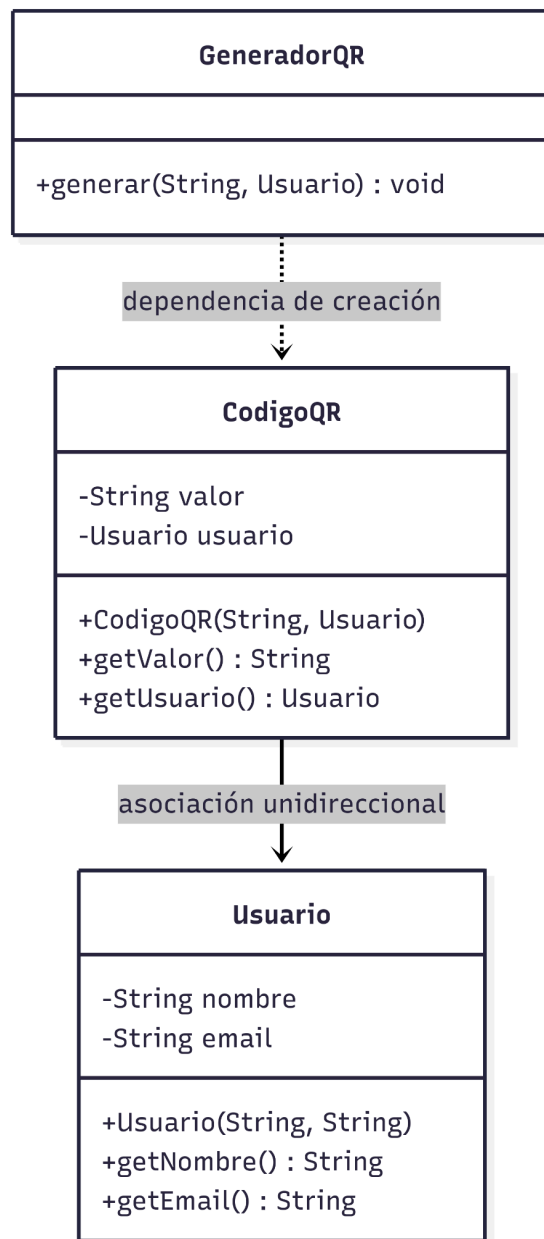

EJERCICIOS DE DEPENDENCIA DE CREACIÓN

Ejercicio 13: GeneradorQR - Usuario - CódigoQR

Relaciones:

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

Diagrama UML:



Implementación Java:

```
// Clase Usuario
public class Usuario {
    private String nombre;
    private String email;

    public Usuario(String nombre, String email) {
        this.nombre = nombre;
        this.email = email;
    }

    public String getNombre() { return nombre; }
    public String getEmail() { return email; }
}

// Clase CodigoQR
public class CodigoQR {
    private String valor;
    private Usuario usuario; // Asociación unidireccional

    public CodigoQR(String valor, Usuario usuario) {
        this.valor = valor;
        this.usuario = usuario;
    }

    public String getValor() { return valor; }
    public Usuario getUsuario() { return usuario; }
}

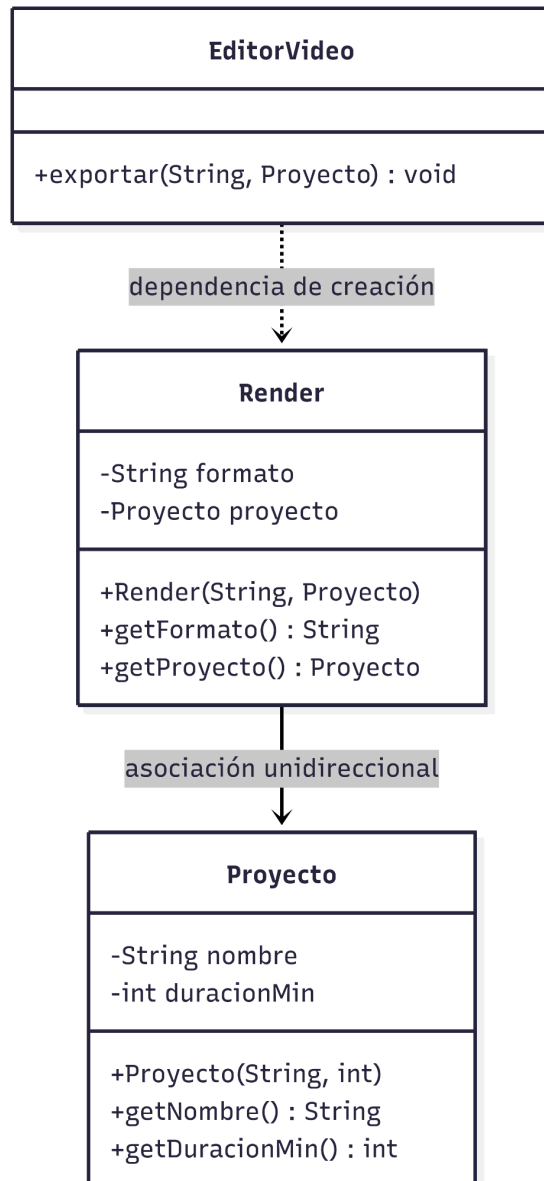
// Clase GeneradorQR
public class GeneradorQR {
    // Dependencia de creación: crea CodigoQR dentro del método pero no lo
    // conserva como atributo
    public void generar(String valor, Usuario usuario) {
        CodigoQR codigoQR = new CodigoQR(valor, usuario); // Crea el objeto
        System.out.println("Código QR generado: " + codigoQR.getValor() +
            " para usuario: " +
            codigoQR.getUsuario().getNombre());
        // El objeto CodigoQR se crea y se usa localmente, no se almacena
        // como atributo
    }
}
```

Ejercicio 14: EditorVideo - Proyecto - Render

Relaciones:

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

Diagrama UML:



Implementación Java:

```
// Clase Proyecto
public class Proyecto {
    private String nombre;
    private int duracionMin;

    public Proyecto(String nombre, int duracionMin) {
        this.nombre = nombre;
        this.duracionMin = duracionMin;
    }

    public String getNombre() { return nombre; }
    public int getDuracionMin() { return duracionMin; }
}

// Clase Render
public class Render {
    private String formato;
    private Proyecto proyecto; // Asociación unidireccional

    public Render(String formato, Proyecto proyecto) {
        this.formato = formato;
        this.proyecto = proyecto;
    }

    public String getFormato() { return formato; }
    public Proyecto getProyecto() { return proyecto; }
}

// Clase EditorVideo
public class EditorVideo {
    // Dependencia de creación: crea Render dentro del método pero no lo
    // conserva como atributo
    public void exportar(String formato, Proyecto proyecto) {
        Render render = new Render(formato, proyecto); // Crea el objeto
        System.out.println("Exportando proyecto: " +
            render.getProyecto().getNombre() +
                " en formato: " + render.getFormato() +
                " - Duración: " +
            render.getProyecto().getDuracionMin() + " min");
        // El objeto Render se crea y se usa localmente, no se almacena
    }
}
```

```
como atributo
    }
}
```

CONCLUSIONES

Análisis de Resultados Obtenidos

El desarrollo de este trabajo práctico ha permitido consolidar conocimientos fundamentales sobre el modelado UML y su implementación práctica en Java. A través de los 14 ejercicios desarrollados, se han abordado escenarios representativos del mundo real que ilustran la diversidad y complejidad de las relaciones entre objetos.

Dominio de Tipos de Relación: Se logró implementar correctamente cada tipo de relación UML:

- **Composición:** Se implementó el ciclo de vida dependiente en casos como Pasaporte-Foto, donde la foto no puede existir sin su pasaporte contenedor
- **Agregación:** Se modelaron relaciones de contención débil como Celular-Batería, donde los componentes mantienen independencia conceptual
- **Asociación:** Se establecieron conexiones semánticas tanto unidireccionales como bidireccionales, implementando mecanismos de consistencia apropiados
- **Dependencias:** Se distinguió claramente entre dependencias de uso y creación, evitando acoplamiento innecesario

Precisión en la Dirección de Relaciones: Se desarrolló la capacidad de determinar correctamente la direccionalidad:

- **Unidireccionales:** Implementadas cuando solo una clase necesita conocer a la otra (ej: Libro → Autor)
- **Bidireccionales:** Aplicadas cuando ambas clases requieren navegabilidad mutua, con métodos que garantizan consistencia automática

Este trabajo práctico demuestra la correcta implementación de todas las relaciones UML 1 a 1 en Java, respetando los principios de orientación a objetos y las características específicas de cada tipo de relación.