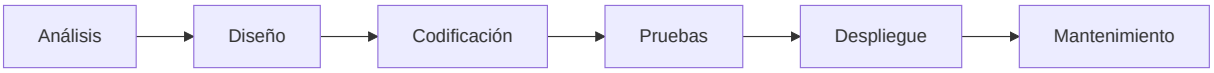
UD4.1. Introducción al diseño orientado a objetos y diagramas de clases UML

1. Introducción

Como hemos visto en las primeras unidades didácticas, el proceso de creación de aplicaciones y sistemas informáticas es complejo y consta de múltiples fases o etapas.



En cada una de estas etapas, se llevan a cabo diferentes actividades, y, por tanto, necesitamos diferentes herramientas para realizarlas.

Además, sabemos que la forma de organizar estas etapas, así como las formas de trabajar, son multiples y muy diversas, por lo que debemos adaptarlas al sistema a desarrollar.

Uno de los conceptos asociados a esta estrategia a la hora de desarrollar nuestro sistema es el de paradigma de desarrollo.

2. Paradigma de desarrollo de software

En el mundo del desarrollo de software, un **paradigma de programación** es un enfoque o estilo para **diseñar y escribir** código. Cada paradigma organiza y resuelve los problemas de manera distinta, lo que afecta directamente la forma en que entendemos, diseñamos y modelamos los sistemas, incluyendo el uso de UML. A continuación, describimos los paradigmas más importantes:

2.1. Programación Imperativa

La programación imperativa se basa en la idea de dar instrucciones paso a paso para resolver un problema.

- **Características clave**:
  - El programador especifica cómo debe ejecutarse cada operación.
  - Se centra en cambiar el estado del programa mediante variables y estructuras de control (como bucles y condicionales).
- **Ejemplo de lenguajes**: C, Pascal.

2.2. Programación Orientada a Objetos (POO)

Este paradigma organiza el código en torno a objetos que combinan datos (atributos) y comportamientos (métodos).

- **Características clave**:
    - Principios como **encapsulación**, **herencia** y **polimorfismo**.
    - Modela el sistema basándose en entidades del mundo real.
- **Ejemplo de lenguajes**: Java, C++, Python, C#.

2.3. Programación Funcional

La programación funcional se centra en resolver problemas utilizando funciones matemáticas puras, sin modificar el estado ni los datos.

- **Características clave**:
    - Evita efectos secundarios y cambios de estado.
    - Facilita la concurrencia y el paralelismo.
- **Ejemplo de lenguajes**: Haskell, Scala, JavaScript (con funciones de orden superior).

2.4. Programación Declarativa

En este paradigma, el programador especifica qué desea obtener, pero no cómo lograrlo.

- **Características clave**:
    - Los lenguajes declarativos se usan a menudo en bases de datos o sistemas de reglas.
    - Ejemplo: Escribir una consulta SQL para obtener datos sin especificar cómo buscarlos.
- **Ejemplo de lenguajes**: SQL, Prolog.

2.5. Paradigmas Combinados

Muchos lenguajes modernos combinan varios paradigmas para adaptarse mejor a diferentes tipos de problemas.

- Ejemplo: Python es tanto orientado a objetos como funcional, y JavaScript mezcla paradigmas funcionales, orientados a objetos y declarativos.

3. Importancia de los diagramas de clases en el desarrollo de aplicaciones

Los diagramas de clases son un pilar fundamental en la **programación orientada a objetos**, ya que permiten representar de manera visual las estructuras y relaciones de los componentes clave de un sistema. Su uso facilita la comprensión, el diseño y la comunicación en equipos de desarrollo, especialmente en proyectos complejos.

4. UML en el contexto de la ingeniería de software

El Lenguaje Unificado de Modelado (**UML, Unified Modeling Language**) es un estándar ampliamente utilizado en la ingeniería de software para **especificar, visualizar, construir y documentar** sistemas. UML no solo se limita a los diagramas de clases, sino que abarca una amplia variedad de diagramas que se adaptan a las distintas fases del ciclo de vida del desarrollo de software.

UML proporciona una base común que permite a desarrolladores, analistas y otros actores del proyecto comunicarse de manera eficiente. Al incluir diagramas estructurales, de comportamiento y de interacción, cubre todos los aspectos del diseño y la implementación de un sistema.

5. Tipos de diagramas en UML y su relación con el ciclo de vida

UML clasifica sus diagramas en tres grandes categorías:

1. **Diagramas estructurales**
    - Representan la arquitectura estática del sistema.
    - Ejemplos: **Diagramas de clases, diagramas de objetos, diagramas de componentes, diagramas de paquetes y diagramas de despliegue**.
    - **Fase del ciclo de vida**: Más utilizados en las fases de análisis y diseño.
2. **Diagramas de comportamiento**
    - Muestran la lógica dinámica y el comportamiento del sistema.
    - Ejemplos: **Diagramas de casos de uso, diagramas de actividades y diagramas de máquinas de estado**.
    - **Fase del ciclo de vida**: Orientados al análisis de requisitos y validación.
3. **Diagramas de interacción**
    - Profundizan en cómo los elementos del sistema interactúan entre sí.
    - Ejemplos: **Diagramas de secuencia, comunicación, tiempos e interacción general**.
    - **Fase del ciclo de vida**: Comunes en diseño detallado y pruebas.

6. Relación de los diagramas de clases con el ciclo de vida

El **diagrama de clases**, al ser un diagrama estructural, tiene un papel destacado en las siguientes fases:

- **Análisis de requisitos**: Representa los conceptos clave y sus relaciones.
- **Diseño**: Especifica la estructura detallada del sistema, incluyendo clases, atributos y métodos.
- **Implementación**: Sirve como base para generar código automáticamente.
- **Mantenimiento**: Permite interpretar, modificar y documentar sistemas existentes.

7. Historia y Evolución de UML

El Lenguaje Unificado de Modelado (**UML**) ha evolucionado desde sus inicios en la década de 1990 hasta convertirse en un estándar internacional ampliamente utilizado en el desarrollo de software. Aquí repasamos su historia, las versiones más relevantes y cómo ha llegado a ser una herramienta indispensable en la ingeniería de software.

7.1. Los Orígenes de UML

Antes de UML, no existía un estándar común para modelar sistemas orientados a objetos, lo que dificultaba la comunicación entre equipos y organizaciones. En la década de 1980 y principios de 1990, varios métodos competían por dominar el diseño orientado a objetos, como:

- **OMT (Object Modeling Technique)** de James Rumbaugh.
- **Booch Method** de Grady Booch.
- **OOSE (Object-Oriented Software Engineering)** de Ivar Jacobson.

Estos tres métodos tenían enfoques complementarios pero inconsistentes. En 1994, los creadores de estos métodos (Booch, Rumbaugh y Jacobson) unieron fuerzas para desarrollar un lenguaje unificado. Este esfuerzo culminó en la creación de UML.

7.2. La Primera Versión de UML

- En **1997**, la versión **UML 1.0** fue adoptada oficialmente por el **Object Management Group (OMG)**, una organización dedicada a establecer estándares de modelado en ingeniería de software.
- UML 1.0 combinaba elementos de los tres métodos principales y ofrecía un marco coherente para modelar sistemas.

Características clave de UML 1.0:

- Introducción de diagramas estructurales (como los diagramas de clases).
- Diagramas de comportamiento para modelar dinámicas del sistema.
- Enfoque en sistemas orientados a objetos.

7.3. Evolución de UML: Principales Versiones

Desde su lanzamiento inicial, UML ha evolucionado para incluir nuevas capacidades y adaptarse a las necesidades de la industria del software.

1. **UML 1.x (1997-2000)**
    - Versión inicial con mejoras iterativas.
        - Introducción de más tipos de diagramas, como diagramas de secuencia y casos de uso.
2. **UML 2.0 (2003)**
    - Publicado por el OMG en 2003. Supuso un salto significativo en funcionalidad y robustez.
    - Principales mejoras:
        - Ampliación de los diagramas estructurales y de interacción.
        - Diagramas de componentes y despliegue más detallados.
        - Inclusión del concepto de "fragmentos combinados" en diagramas de secuencia.
3. **UML 2.x (2005 - Actualidad)**
    - Series de actualizaciones menores para perfeccionar la versión 2.0.
    - Última versión oficial: **UML 2.5.1**, publicada en 2017.
    - Cambios clave:
        - Simplificación del estándar para facilitar su adopción.
        - Mejora en la interoperabilidad con herramientas CASE (Computer-Aided Software Engineering).

7.4. UML Hoy en Día

En la actualidad, UML es el estándar más utilizado para modelar sistemas en múltiples industrias, no solo en software, sino también en áreas como sistemas embebidos y gestión empresarial.

- **Usos comunes**: Análisis de requisitos, diseño de sistemas, comunicación entre equipos.

- **Herramientas populares**: StarUML, Enterprise Architect, Visual Paradigm, y módulos de UML en IDEs como Eclipse o IntelliJ IDEA.

UML sigue evolucionando en función de las necesidades de la industria y los avances tecnológicos. Aunque no todos los diagramas de UML se usan con la misma frecuencia, los diagramas de clases, casos de uso y secuencia siguen siendo los más populares.

Conceptos Básicos de la Programación Orientada a Objetos

La programación orientada a objetos (POO) es un paradigma de programación que organiza el código en torno a "objetos", los cuales representan elementos del mundo real o conceptos abstractos. Es fundamental en el desarrollo de aplicaciones modernas y está directamente relacionado con los diagramas de clases.

8. Clases y Objetos: Definición y Características

8.1. Clases

Una **clase** es una plantilla o modelo que define cómo serán los objetos.

- Incluye **atributos**, que son las características o propiedades del objeto (datos).
- Contiene **métodos**, que son las acciones o comportamientos que el objeto puede realizar (funciones).

Ejemplo:
Imagina que queremos crear una aplicación para gestionar libros en una biblioteca.
La clase podría llamarse `Libro` y definirse así:

```
class Libro {
    String titulo;  // Atributo: título del libro
    String autor;   // Atributo: autor del libro

    void leer() {   // Método: leer el libro
        System.out.println("Estás leyendo el libro: " + titulo);
    }
}
```

8.2. Objetos

Un **objeto** es una instancia de una clase, es decir, un ejemplar concreto basado en la plantilla de la clase.

- Si la clase `Libro` es el modelo, un objeto podría ser un libro específico, como "1984" de George Orwell.

Ejemplo de instanciación:

```
Libro libro1 = new Libro(); // Crear un objeto de la clase Libro
libro1.titulo = "1984";     // Asignar valores a los atributos
libro1.autor = "George Orwell";

libro1.leer(); // Llama al método: muestra "Estás leyendo el libro: 1984"`
```

9. Atributos, Métodos y Visibilidad

9.1. Atributos

Son las características o propiedades que describen a una clase.
En el ejemplo anterior, `titulo` y `autor` son atributos de la clase `Libro`.

9.2. Métodos

Son las acciones que un objeto puede realizar. En el ejemplo, `leer()` es un método que representa la acción de leer un libro.

9.3. Visibilidad

La **visibilidad** determina quién puede acceder a los atributos y métodos de una clase. Las palabras clave más comunes son:

- **`public`**: accesible desde cualquier parte del programa.
- **`private`**: accesible solo dentro de la propia clase.
- **`protected`**: accesible dentro de la clase, sus subclases y el mismo paquete (más avanzado).

Ejemplo de visibilidad:

```
class Libro {
    private String titulo; // Solo accesible dentro de la clase
    public String autor;   // Accesible desde cualquier parte

    public void setTitulo(String titulo) { // Método público para
asignar el título
        this.titulo = titulo;
    }

    public String getTitulo() { // Método público para obtener el
título
        return titulo;
    }
}`
```

10. Relaciones: Herencia o jerarquía, Composición y Agregación

Las relaciones entre clases son fundamentales en la programación orientada a objetos (POO) porque permiten organizar, estructurar y modelar el comportamiento y las interacciones entre diferentes entidades en un programa. Una correcta comprensión y uso de estas relaciones mejora la reutilización de código, la modularidad y la flexibilidad del diseño.

En este apartado, analizaremos cinco tipos de relaciones entre clases:

- Clientela
- Herencia
- Composición
- Agregación
- Anidamiento

10.1. Clientela

La clientela (o asociación de uso) es una relación donde una clase utiliza los servicios o métodos de otra clase. Es una relación muy flexible y frecuente.

Características:

- Relación temporal y no necesariamente fuerte.
- Una clase "cliente" depende de otra clase para realizar sus funciones.

Ejemplo: Tenemos una clase `Usuario` que es cliente de la clase `Impresora`. El `Usuario` utiliza el método `imprimir` sin ser "dueño" de la impresora.

```
class Impresora {
    public void imprimir(String texto) {
        System.out.println("Imprimiendo: " + texto);
    }
}

class Usuario {
    public void enviarAImprimir(Impresora impresora, String texto) {
        impresora.imprimir(texto);
    }
}

public class Main {
    public static void main(String[] args) {
        Usuario usuario = new Usuario();
        Impresora impresora = new Impresora();

        usuario.enviarAImprimir(impresora, "Hola, mundo.");
    }
}
```

10.2. Herencia

La herencia permite que una clase "hija" herede atributos y métodos de una clase "padre". Esto evita la duplicación de código.

Ejemplo:
Si tenemos una clase general `Publicacion`, podemos crear una clase `Libro` que herede de ella.

```
class Publicacion {
    String titulo;
    String autor;
}

class Libro extends Publicacion {
    int paginas;
}
```

La clase `Libro` hereda `titulo` y `autor` de `Publicacion`, y añade su propio atributo `paginas`.

10.3. Composición

Ocurre cuando una clase contiene objetos de otras clases como parte de sus atributos, indicando una relación "tiene un".

Ejemplo:
Un objeto de clase `Biblioteca` contiene varios objetos de clase `Libro`.

```
class Biblioteca {
    Libro[] libros; // Composición: la biblioteca tiene libros
}
```

10.4. Agregación

Es similar a la composición, pero la relación entre las clases es más débil. Los objetos relacionados pueden existir por separado.

Ejemplo:
Un `Autor` puede escribir varios `Libros`, pero puede existir sin ellos.

```
class Autor {
    String nombre;
}

class Libro {
    Autor autor; // Agregación: el libro tiene un autor
}
```

10.5. Anidamiento

El anidamiento es una relación en la que una clase está contenida dentro de otra clase. Se implementa usando clases anidadas o clases internas.

Características:

- Permite organizar y modularizar el código.
- Una clase interna tiene acceso a los miembros de la clase contenedora.

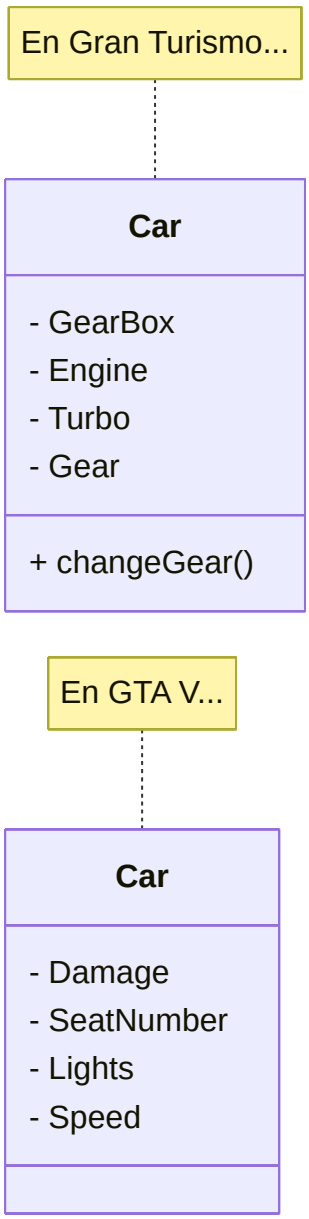
- Puede ser estática o no estática.

Ejemplo: Una clase `Procesador` que está anidada dentro de una clase `Ordenador`, y puede acceder a los atributos de la clase contenedora.

```
class Ordenador {
    private String marca;

    public Ordenador(String marca) {
        this.marca = marca;
    }

    class Procesador {
        public void mostrarMarca() {
            System.out.println("El procesador pertenece a un
ordenador de marca: " + marca);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Ordenador ordenador = new Ordenador("Lenovo");
        Ordenador.Procesador procesador = ordenador.new Procesador();
        procesador.mostrarMarca();
    }
}
```

11. Concepto de Abstracción y Encapsulación en Programación Orientada a Objetos

La **abstracción** es uno de los principios fundamentales de la programación orientada a objetos (POO) y se refiere al proceso de ocultar los detalles complejos de implementación, mostrando solo lo necesario para el uso de un componente o sistema. Este principio permite centrarse en "qué hace un objeto" en lugar de "cómo lo hace", simplificando el diseño y la comprensión del código. Por ejemplo, si queremos modelar la clase `Coche`, dependiendo del contexto, podríamos definir los siguientes atributos y métodos:



Por su parte, la **encapsulación** es el proceso de ocultar todos los detalles de un objeto que no son de interés desde fuera de dicho objeto, es decir, separar el aspecto externo accesible por otros objetos del interno, innaccesible para los demás. De esta manera, la encapsulación consiste en ocultar los atributos y métodos del objeto a otros objetos, pasando a denominarse **privados**.

11.1. Clases Abstractas

Una **clase abstracta** es una clase que no puede instanciarse directamente. Sirve como modelo o plantilla para otras clases y puede incluir:

- **Métodos abstractos**: Declarados sin implementación (por ejemplo, `metodoAbstracto()`), que deben ser implementados en las subclases concretas.

- **Métodos concretos**: Métodos con implementación, que pueden heredarse por las subclases.

Características clave:

- Se utiliza para definir comportamientos comunes entre clases relacionadas.
- Facilita la reutilización de código y el cumplimiento de una estructura.

Ejemplo (Java):

```
abstract class Animal {
    String nombre;

    // Método abstracto: las subclases deben implementarlo.
    abstract void hacerSonido();

    // Método concreto: las subclases pueden usarlo tal cual.
    void comer() {
        System.out.println(nombre + " está comiendo.");`
    }
}

class Perro extends Animal {
    @Override
    void hacerSonido() {
        System.out.println("Guau, guau!");
    }
}

class Gato extends Animal {
    @Override
    void hacerSonido() {
        System.out.println("Miau, miau!");
    }
}
```

En este ejemplo, la clase `Animal` es abstracta. Define el método abstracto `hacerSonido()` y un método concreto `comer()`. Las subclases `Perro` y `Gato` implementan `hacerSonido()` con su propio comportamiento.

11.2. Interfaces

Una **interfaz** define un conjunto de métodos que una clase debe implementar, sin proporcionar ninguna implementación concreta.

- A diferencia de las clases abstractas, una interfaz no puede tener atributos ni métodos concretos (en la mayoría de los lenguajes).
- Una clase puede implementar múltiples interfaces, lo que permite cierta flexibilidad que las clases abstractas no ofrecen.

Características clave:

- Promueven el uso de la abstracción al nivel más alto.
- Son ideales para definir contratos entre clases no relacionadas jerárquicamente.

Ejemplo (Java):

```java
interface Volador {
    void volar();
}

class Ave implements Volador {
    @Override
    public void volar() {
        System.out.println("El ave está volando.");
    }
}

class Avion implements Volador {
    @Override
    public void volar() {
        System.out.println("El avión está volando.");
    }
}
```

En este ejemplo, la interfaz `Volador` define el método `volar()`. Las clases `Ave` y `Avion` implementan esta interfaz, cada una con su propio comportamiento.

11.3. Diferencias entre Clase Abstracta e Interfaz

Aspecto	Clase Abstracta	Interfaz
Propósito	Representar un concepto genérico común.	Definir un contrato que debe cumplirse.

Aspecto	Clase Abstracta	Interfaz
Atributos	Puede tener atributos concretos y abstractos.	No permite atributos (salvo constantes).
Herencia	Una clase solo puede heredar de una clase abstracta.	Una clase puede implementar múltiples interfaces.
Métodos	Puede tener métodos concretos y abstractos.	Solo métodos abstractos (hasta versiones recientes de algunos lenguajes).

11.4. Abstracción y Diagramas de Clases

La abstracción es clave para diseñar sistemas escalables y fáciles de mantener. En UML:

- Se utiliza para modelar comportamientos genéricos con clases abstractas y para definir contratos con interfaces.
- Ayuda a visualizar jerarquías complejas y relaciones entre clases de forma clara.

12. Polimorfismo en Programación Orientada a Objetos

El **polimorfismo** es un principio fundamental de la programación orientada a objetos que permite a una entidad, como un método o un objeto, comportarse de múltiples formas dependiendo del contexto. En términos simples, el polimorfismo permite utilizar una misma interfaz para representar diferentes comportamientos o implementaciones.

12.1. Tipos de Polimorfismo

Existen dos tipos principales de polimorfismo en la programación orientada a objetos:

12.1.1. Polimorfismo en Tiempo de Compilación (Sobrecarga de Métodos)

- Ocurre cuando un método en una clase tiene el mismo nombre pero diferentes firmas (diferente número o tipo de parámetros).
- El compilador selecciona cuál método invocar en función de los argumentos proporcionados.
- Esto se conoce como *early binding* o vinculación temprana.

Ejemplo en Java:

```java
class Calculadora {
    // Suma de dos números enteros
    int sumar(int a, int b) {
        return a + b;
    }

    // Suma de tres números enteros
    int sumar(int a, int b, int c) {
        return a + b + c;
    }

    // Suma de dos números de punto flotante
    double sumar(double a, double b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        Calculadora calc = new Calculadora();
        System.out.println(calc.sumar(2, 3));        // Llama al
método con dos enteros
        System.out.println(calc.sumar(2, 3, 4));     // Llama al
método con tres enteros
        System.out.println(calc.sumar(2.5, 3.5));    // Llama al
método con dos dobles
    }
}
```

Resultado del programa:

```
5
9
6.0
```

En este ejemplo, el método `sumar` tiene diferentes implementaciones según los parámetros, y el compilador selecciona automáticamente la versión apropiada.

12.1.2. Polimorfismo en Tiempo de Ejecución (Sobrescritura de Métodos)

- Ocurre cuando una subclase proporciona su propia implementación de un método definido en una clase base.
- La invocación del método se decide en tiempo de ejecución en función del tipo del objeto real, no del tipo de referencia.
- Esto se conoce como *late binding* o vinculación tardía.

Ejemplo en Java:

```java
class Animal {
    void hacerSonido() {
        System.out.println("El animal hace un sonido.");
    }
}

class Perro extends Animal {
    @Override
    void hacerSonido() {
        System.out.println("Guau, guau!");
    }
}

class Gato extends Animal {
    @Override
    void hacerSonido() {
        System.out.println("Miau, miau!");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal miAnimal;

        miAnimal = new Perro();
        miAnimal.hacerSonido();  // Salida: "Guau, guau!"

        miAnimal = new Gato();
        miAnimal.hacerSonido();  // Salida: "Miau, miau!"
    }
}
```

Resultado del programa:

```
Guau, guau! Miau, miau!
```

En este ejemplo, el método `hacerSonido` es sobrescrito en las clases `Perro` y `Gato`. Aunque la referencia es de tipo `Animal`, el método invocado depende del tipo real del objeto (`Perro` o `Gato`) en tiempo de ejecución.

12.2. Importancia del Polimorfismo

El polimorfismo permite:

- Diseñar sistemas extensibles y modulares, ya que el comportamiento puede cambiar sin alterar el código existente.
- Reducir la duplicación de código al reutilizar métodos comunes en clases base y personalizar el comportamiento en las subclases.
- Implementar patrones de diseño, como el **patrón de estrategia**, que dependen del uso de polimorfismo.

12.3. Polimorfismo y Principio de Sustitución de Liskov

El polimorfismo respeta el **Principio de Sustitución de Liskov (LSP)**, que establece que un objeto de una clase base debe poder ser sustituido por un objeto de una clase derivada sin alterar la funcionalidad del programa. Esto garantiza que el comportamiento esperado en tiempo de ejecución sea coherente, independientemente del tipo específico del objeto.

```
public class Main {
 static void reproducirSonido(Animal animal) {
     animal.hacerSonido();
 }

 public static void main(String[] args) {
     reproducirSonido(new Perro());  // Salida: "Guau, guau!"
     reproducirSonido(new Gato());  // Salida: "Miau, miau!"
 }
}
```

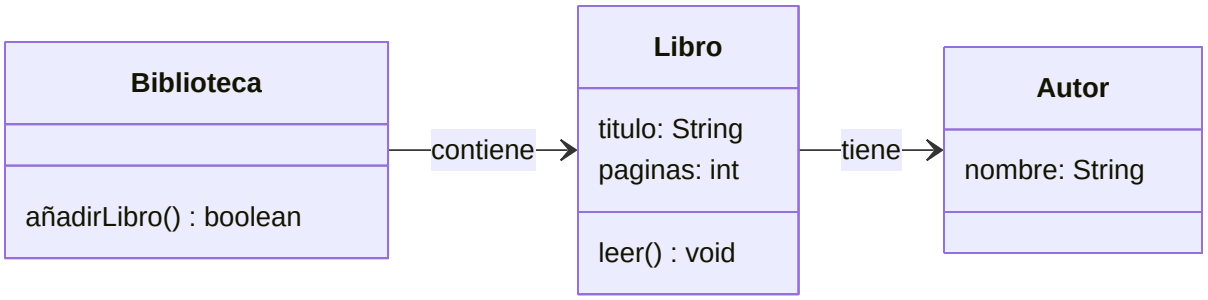
13. Ejemplo Práctico de un Diagrama de Clases

Antes de entrar en el detalle de la creación de diagramas de clases, y para consolidar los conceptos vistos anteriormente, vamos a representar las clases y sus relaciones vistas anteriormente mediante diagramas de clases.

- **Clases**: `Libro`, `Autor`, `Biblioteca`.
- **Relaciones**:

- Composición: `Biblioteca` tiene varios `Libro`.
- Agregación: `Libro` tiene un `Autor`.

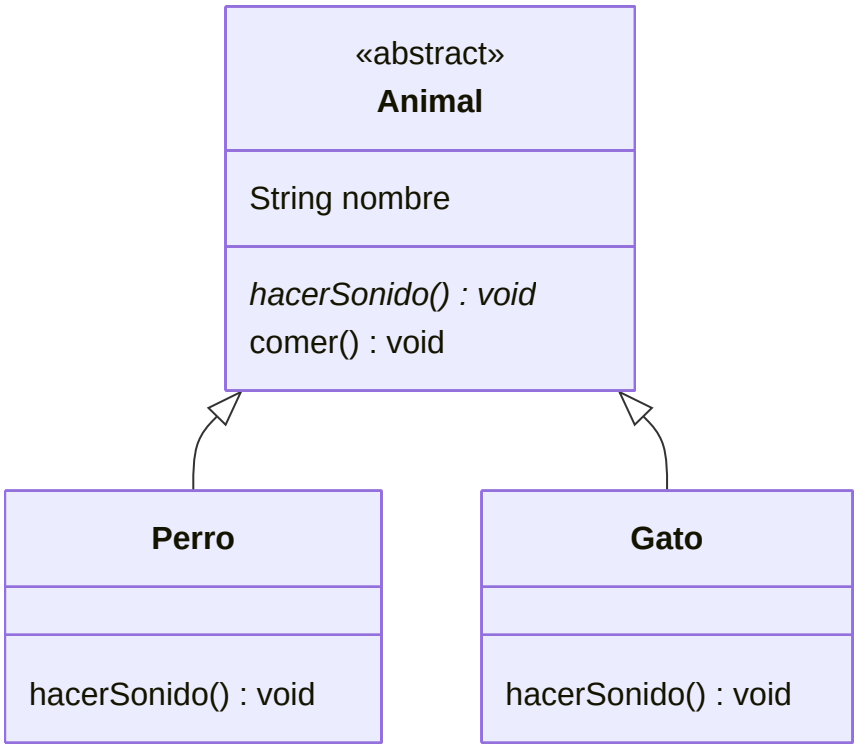
Diagrama UML para clases con relaciones:



Este ejemplo muestra cómo representar las relaciones entre clases y sus atributos/métodos.

Representación de clases y métodos abstractos en UML:

- Las **clases abstractas** se representan con el nombre en cursiva o con el estereotipo `<<abstract>>`.
- Los **métodos abstractos** también aparecen en cursiva.



Representación de interfaces en UML:

- Las **interfaces** se representan con el estereotipo `<<interface>>`.
- Las clases que implementan una interfaz tienen una línea discontinua con un triángulo apuntando hacia la interfaz.

Siguiendo el ejemplo del apartado 2.4, este sería el diagrama de clases UML con la interfaz `Volador` y las clases `Ave` y `Avion`