Libro

Luis Jaramillo

2024-08-11

Table of contents

Programación Orientada a Objetos

La programación orientada a objetos (POO) es un paradigma de programación que se basa en el concepto de "objetos", los cuales son instancias de clases. En POO, un "objeto" puede entenderse como una entidad que encapsula datos y funciones que operan sobre esos datos. Este enfoque permite modelar el mundo real de manera más efectiva, ya que se pueden representar entidades y sus interacciones de manera más natural.

En POO, una "clase" es un plano o plantilla para crear objetos. Define la estructura y el comportamiento de los objetos que se crearán a partir de ella. Los objetos son instancias concretas de una clase, y cada objeto puede tener sus propios datos (llamados atributos) y funciones (llamadas métodos).

Los cuatro principios fundamentales de la POO son la encapsulación, la herencia, el polimorfismo y la abstracción. La "**encapsulación**" implica la ocultación de los detalles internos de un objeto y la exposición solo de lo necesario. La "**herencia**" permite la creación de nuevas clases basadas en clases existentes, heredando sus atributos y métodos. El "**polimorfismo**" permite que un objeto pueda tomar varias formas, es decir, un mismo método puede comportarse de manera diferente según el contexto. La "**abstracción**" consiste en simplificar la representación de un objeto centrándose en los aspectos esenciales y omitiendo los detalles innecesarios. Para el siguiente documento revisaremos las siguientes temáticas:

Primer Parcial

- Control de versionamiento
- Entorno de Desarrollo
- Paradigmas de programación
- Conceptos Generales de la POO
- Modelamiento de clases y objetos
- Código limpio
- Estructura General de un Programa - ————————————————————-
- Lectura y escritura de datos por consola
- Manejo de Excepciones
- Encapsulamiento
- Constructores
- Métodos Getter y Setters
- Persistencia de datos
- Arreglos y Colecciones

2 Parcial - Relaciones entre clases (Generalización/Especialización) - Revisiones de Código - Polimorfismo - Interfaces de Programación - Modelo Vista Controlador - Bases de Datos no SQL - Componentes y Objetos Gráficos

3er Parcial - Principios SOLID - Modularidad - Introducción a Patrones de diseño - Patrones de creación - Patrones de Estructura - Patrones de Comportamiento

Part I

Unidad 1: Nombre de la Unidad

1 Introduction

This is a book created from markdown and executable code.

See Knuth (1984) for additional discussion of literate programming.

Part II

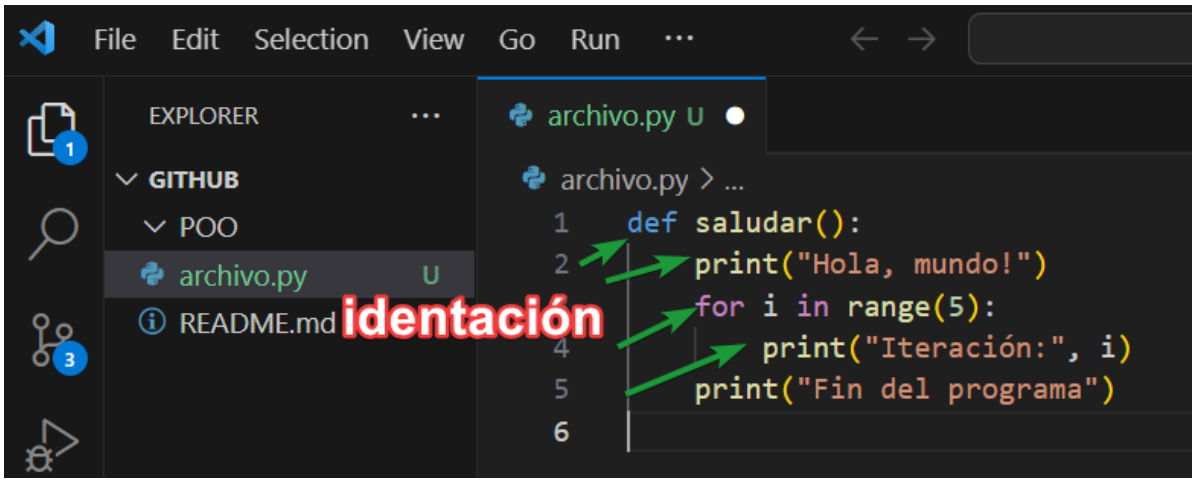
Unidad 2: Entorno de Desarrollo

2 ENTORNO DE DESARROLLO.

Un entorno de desarrollo, comúnmente conocido como IDE por sus siglas en inglés (Integrated Development Environment), es una herramienta de software integral que se utiliza para facilitar la programación y el desarrollo de software. Un IDE proporciona un conjunto de características y herramientas que ayudan a los programadores a escribir, depurar, compilar y administrar código de manera más eficiente, en otras palabras es un paquete de software que combina todas las herramientas de desarrollo en una única interfaz gráfica de usuario (GUI), Algunos ejemplos de IDE más utilizados son NetBeans, Microsoft Visual Studio, Adobe Flex Builder y Eclipse. Algunos componentes comunes de un entorno de desarrollo incluyen:

Editor de código: Ofrece funciones de edición de texto avanzadas, como resaltado de sintaxis, autoindentación y autocompletado, lo que facilita la escritura de código.

La Identación se refiere al formato visual utilizado para estructurar y organizar el código fuente en un lenguaje de programación.



Depurador: Permite a los programadores identificar y corregir errores en su código al proporcionar herramientas para establecer puntos de interrupción, inspeccionar variables y seguir la ejecución del programa paso a paso.

Compilador/Intérprete: Proporciona la capacidad de compilar y ejecutar el código directamente desde el IDE. Algunos IDE también son específicos para un lenguaje y pueden incluir un intérprete para ejecutar el código directamente.

Gestión de proyectos: Permite organizar y administrar proyectos de desarrollo, lo que incluye la creación, apertura y gestión de archivos y directorios.

Herramientas de versionado: Puede integrar sistemas de control de versiones, como Git, para realizar un seguimiento de los cambios en el código y colaborar con otros desarrolladores.

Asistencia en el desarrollo: Ofrece sugerencias y advertencias en tiempo real mientras se escribe código, lo que ayuda a detectar posibles errores y mejorar la calidad del código.

Herramientas de construcción: Facilita la construcción y automatización de tareas, como la compilación, la generación de documentación y la ejecución de pruebas.

Herramientas de depuración visual: Algunos IDE proporcionan una interfaz gráfica para depurar aplicaciones de manera más visual, lo que puede incluir representaciones gráficas de estructuras de datos y flujos de programa.

Integración con SDK y bibliotecas: Puede integrar SDK (kits de desarrollo de software) y bibliotecas relevantes para el lenguaje o plataforma de desarrollo específico.

Finalización de código: conocida como prompt de código, esta función se utiliza para analizar todo el código y para identificar e insertar los componentes de código que faltan. Su uso permite ahorrar tiempo y minimizar los fallos o errores

Por ejemplo, un SDK de Android proporciona las herramientas y recursos necesarios para desarrollar aplicaciones para dispositivos Android, incluyendo bibliotecas para interactuar con el hardware y el sistema operativo, herramientas de depuración, un emulador de Android para probar aplicaciones y documentación detallada.

Los IDE varían según el lenguaje de programación y la plataforma de desarrollo. Ejemplos populares de IDE incluyen Visual Studio (para desarrollo en C# y .NET), Eclipse (para Java), IntelliJ IDEA (para Java y otros lenguajes), PyCharm (para Python) y muchos otros. La elección de un IDE suele depender de las preferencias del desarrollador y del entorno de desarrollo específico en el que trabaje.

CARACTERISTICAS E INTSALACIÓN.

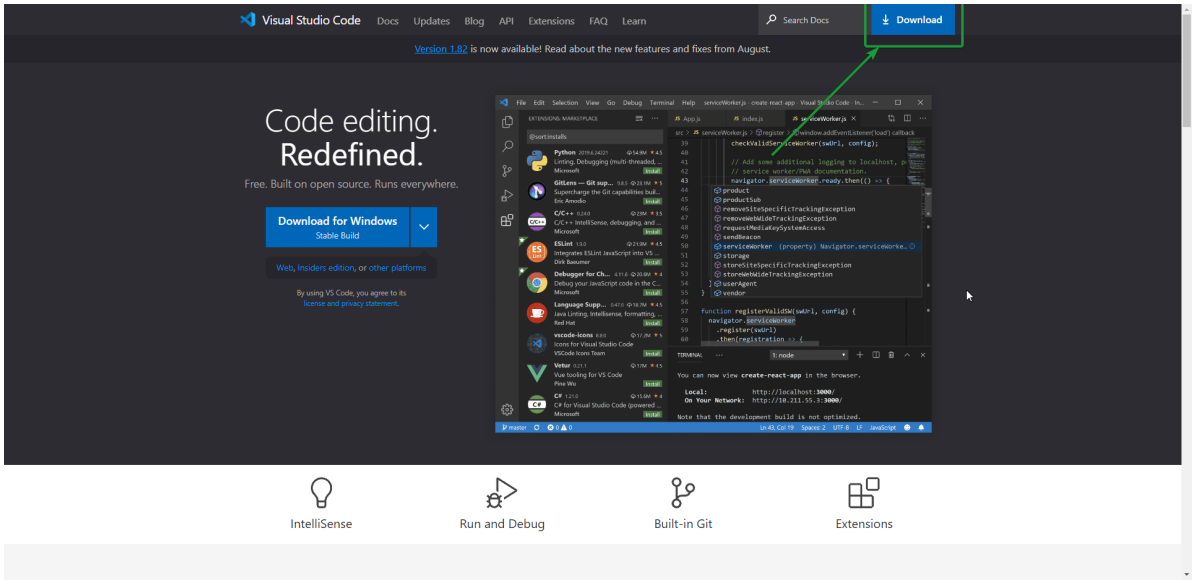
Para este curso se utilizará los lenguajes de programación de Python y Java, por lo que instalaremos dos entornos virtuales, para sacar el mayor provecho en el caso del lenguaje de programación Java.

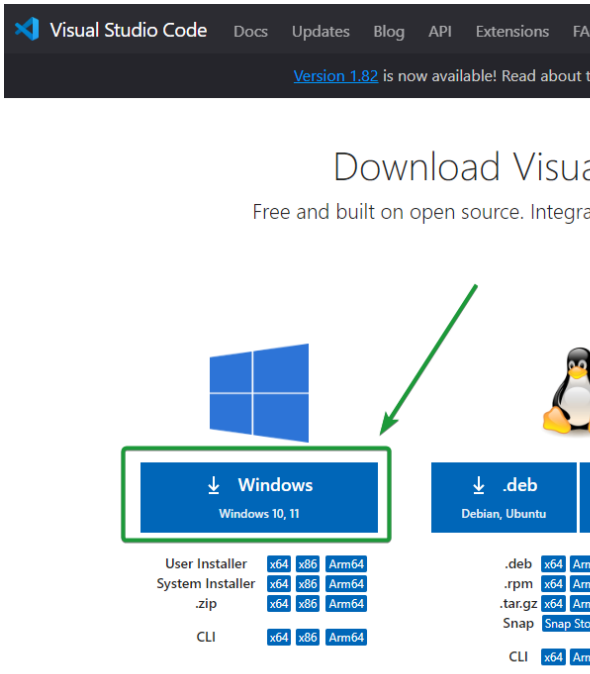
El primer entorno virtual será Visual Studio Code  y el IDE para el lenguaje de programación será Netbeans IDE 16  .

Para la instalación del IDE Visual Studio Code hay que seguir los siguientes pasos.

Paso 1: Descarga de VS Code

Dirígete al sitio web oficial de Visual Studio Code en https://code.visualstudio.com/ y busca un botón de descarga, generalmente ubicado en la página principal. Aquí deberías encontrar una imagen que representa el botón de descarga.



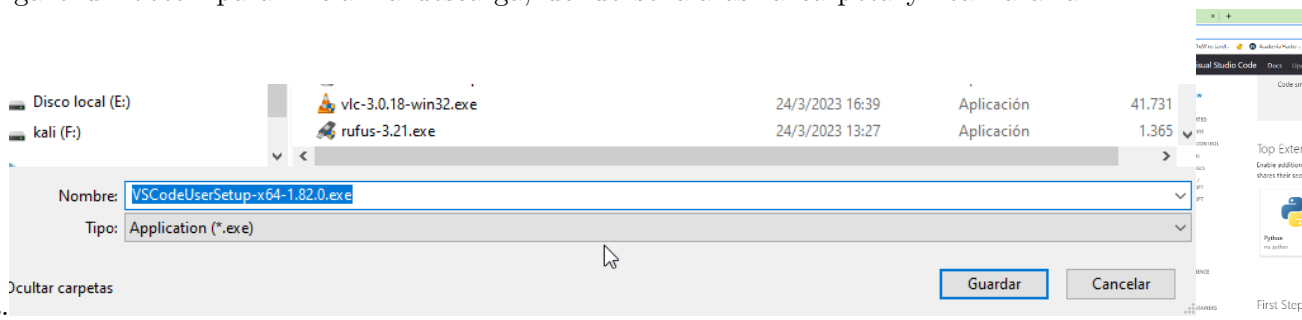


Paso 2: Selecciona tu Sistema Operativo, en mi caso Windows 10.

Paso 3: Descarga del Instalador

Después de seleccionar tu sistema operativo, deberías ser redirigido a una página que te

permitirá descargar el instalador específico para tu plataforma. Aquí encontrarás un enlace de descarga o un botón para iniciar la descarga, donde señalarás la carpeta y realizará la descarga.



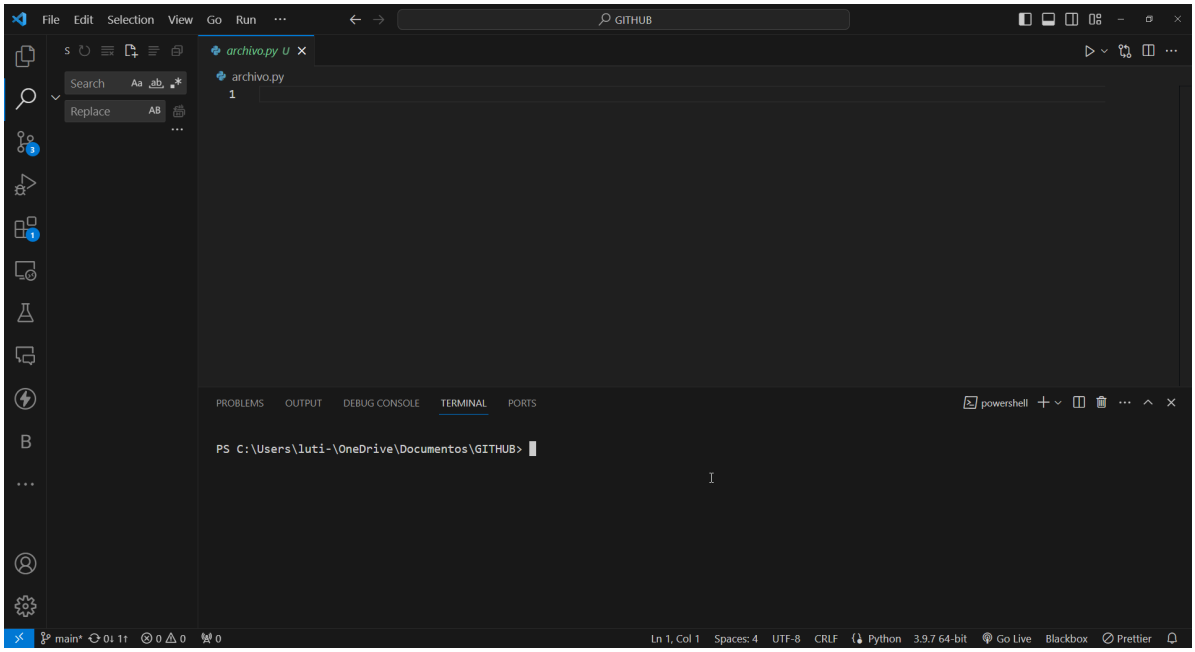
Paso 4: Ejecuta el Instalador

Una vez que se complete la descarga, ejecuta el instalador haciendo doble clic en él (en Windows) o siguiendo las instrucciones específicas para tu sistema operativo.

Durante la instalación, se te pueden presentar opciones de configuración. Estas opciones pueden variar según la versión y la plataforma. Generalmente, se te preguntará si deseas crear accesos directos en el escritorio o en el menú de inicio (en el caso de Windows), así como otras preferencias.

Paso 5: Finaliza la Instalación

Finalmente, cuando se complete la instalación, verás una pantalla que confirma que Visual Studio Code se ha instalado correctamente.



Paso 6: Configuración de los lenguajes.

La configuración de los lenguajes es muy importante ya que este nos permite al momento de ejecutar los archivos con sus diferentes extensiones que corran de manera correcta, para este curso instalaremos Python y luego netbeans por lo que los pasos a seguir son los siguientes:

Paso 1: Descargar Python: Ve al sitio web oficial de Python en https://www.python.org/downloads/.

En la página de descargas, verás las últimas versiones de Python disponibles. Elige la versión más reciente de Python 3 (por ejemplo, Python 3.9.6).

Desplázate hacia abajo hasta encontrar la sección de descargas de archivos ejecutables. Dependiendo de tu sistema operativo (Windows, macOS o Linux), selecciona el enlace de descarga adecuado. Para Windows, es posible que veas dos opciones: una para 32 bits y otra para 64 bits. Si tienes un sistema operativo de 64 bits, selecciona la versión de 64 bits.

Paso 2: Ejecutar el Instalador de Python: Una vez que se complete la descarga, haz doble clic en el archivo ejecutable que descargaste para iniciar el instalador de Python.

Asegúrate de marcar la casilla que dice "Add Python X.Y to PATH" (donde "X.Y" representa la versión de Python que estás instalando). Esto añadirá Python al PATH de tu sistema, lo que facilitará su uso desde la línea de comandos.

Haz clic en "Install Now" para comenzar la instalación.

Paso 3: Verificar la Instalación de Python: Abre una terminal o línea de comandos, Escribe el siguiente comando y presiona Enter:

bash

Copy code

python –version

Esto mostrará la versión de Python que has instalado, como "Python 3.9.6" o similar.

Instalación del Netbeans.

Part III

Unidad 3: Paradigmas de Programación

3 Introducción a los Paradigmas de Programación

Los paradigmas de programación son enfoques o estilos para resolver problemas mediante código. Cada paradigma tiene sus principios, ventajas y desventajas, y está diseñado para abordar problemas de diferentes maneras. En este documento, exploraremos cuatro paradigmas principales: imperativo, declarativo, orientado a objetos y funcional.

-

3.1 Paradigma Imperativo

El paradigma imperativo es uno de los enfoques más tradicionales. En él, el programador indica paso a paso cómo se debe realizar cada tarea.

- Características:
  - Enfoque en **cambiar el estado** mediante secuencias de instrucciones.
  - Uso intensivo de **bucles** y **estructuras de control de flujo** (if, else, for, while).
- Ejemplo en Java

En Java, este paradigma se utiliza ampliamente debido a su sintaxis estructurada.

```java
public class SumaImperativa {
    public static void main(String[] args) {
        int sum = 0;
        for (int i = 1; i <= 5; i++) {
            sum += i;
        }
        System.out.println("La suma es: " + sum);
    }
}
```

-

3.2 Paradigma Declarativo

En el paradigma declarativo, el programador especifica **qué** se quiere lograr, no **cómo** hacerlo. Esto permite centrarse en el resultado y no en los pasos intermedios.

- Características:
  - Describe el resultado deseado en lugar de los pasos detallados.
  - Más conciso y fácil de leer para tareas específicas como consultas de bases de datos.
  - Ejemplo con Java y SQL

En Java, el uso de SQL en aplicaciones es un ejemplo de un enfoque declarativo.

```
SELECT nombre, edad FROM estudiantes WHERE edad > 18;
```

-

3.3 Paradigma Orientado a Objetos

La programación orientada a objetos (POO) es ampliamente utilizada en Java. Este paradigma organiza el código en "objetos", que representan entidades del mundo real.

- Características:
  - Usa conceptos de **clases** y **objetos**.
  - Principios de **abstracción, encapsulación, herencia y polimorfismo**.
- Ejemplo en Java

Aquí un ejemplo básico de POO en Java:

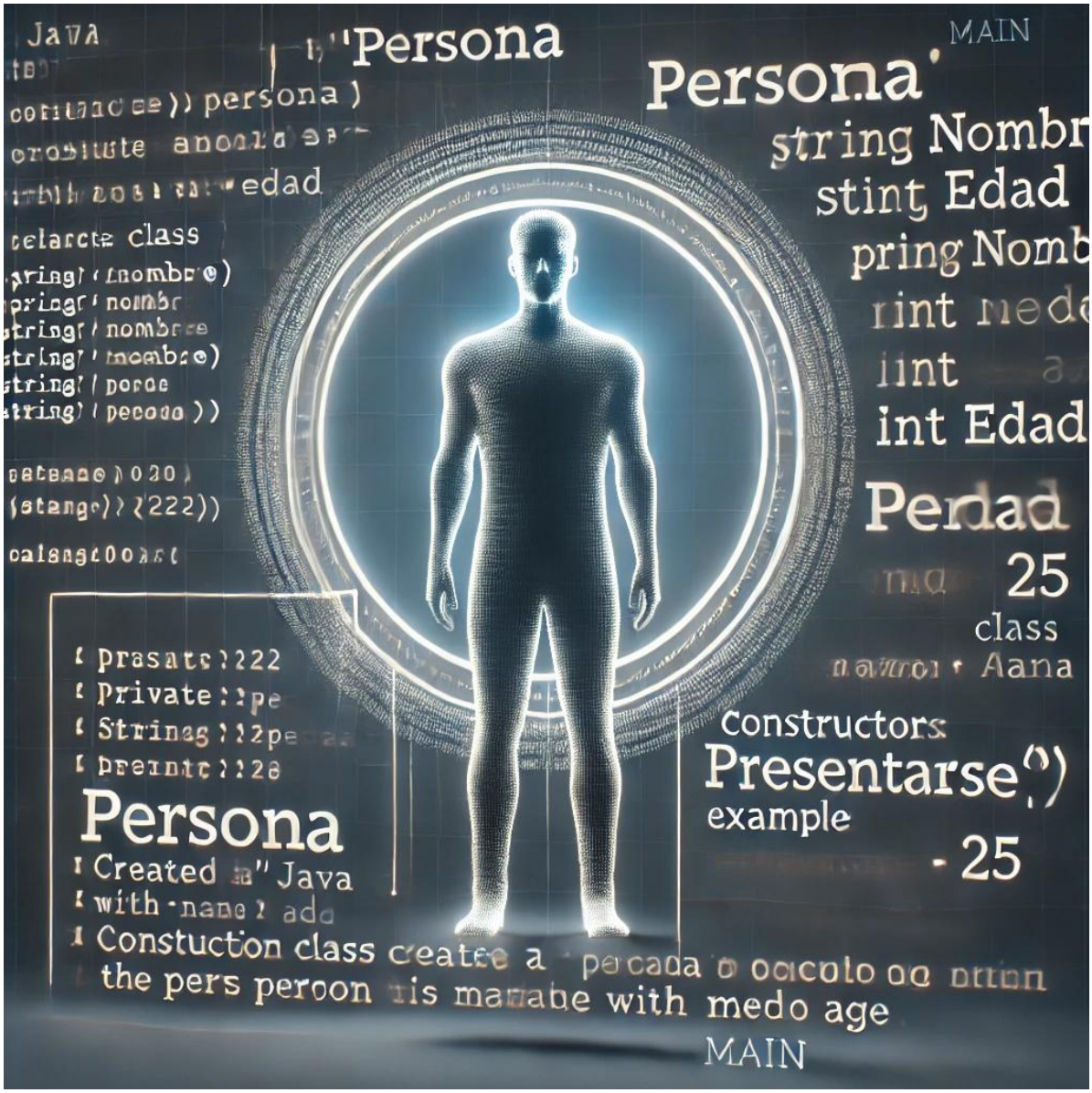
```
class Persona {
    String nombre;
    int edad;

    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    public void presentarse() {
        System.out.println("Hola, soy " + nombre + " y tengo " + edad + " años.");
```

```
    }
}

public class Main {
    public static void main(String[] args) {
        Persona persona = new Persona("Ana", 25);
        persona.presentarse();
    }
}
```



-

3.4 Paradigma Funcional.

El paradigma funcional trata las funciones como ciudadanos de primera clase. En este estilo, se evita el cambio de estado y las variables mutables.

- Características:

  - Uso de **funciones puras** (sin efectos secundarios).
  - Uso de **funciones de orden superior** y técnicas como el **mapeo** y el **filtrado**.

- Ejemplo en Java.

magina que tienes una lista de números y quieres obtener solo los números pares y calcular el doble de cada uno de ellos. En el **paradigma funcional**, no describimos cada paso de cómo lograr esto con bucles o variables que cambien, sino que aplicamos funciones que transforman la lista de manera declarativa. 1. Primero, aplicamos una **función de filtro** para seleccionar solo los números pares. 2. Luego, aplicamos una **función de mapeo** para multiplicar cada número par por dos. 3. Finalmente, **obtenemos el resultado** como una nueva lista.

Test

Part IV

Unidad 4: Conceptos Generales de la POO

4 Conceptos Generales de la POO

La **Programación Orientada a Objetos (POO)** es un paradigma de programación que organiza el código en torno a "objetos" que representan entidades del mundo real. La POO permite una estructura modular, reutilizable y escalable, haciendo el código más fácil de entender y mantener.

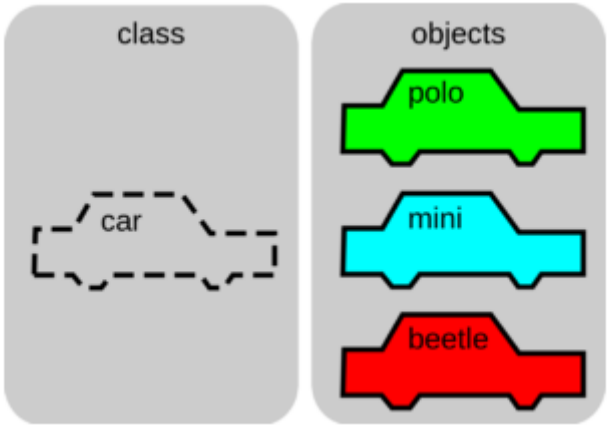
4.1 Principios Fundamentales de la POO

La POO se basa en cuatro principios clave: **Clases y Objetos**, **Encapsulación**, **Herencia** y **Polimorfismo**. A continuación, veremos cada uno en detalle.

4.2 1 Clases y Objetos

4.2.1 Clases

Una **clase** es una plantilla o modelo que define las propiedades y comportamientos de un tipo de objeto. Es como un plano que describe cómo debe ser un objeto.



```java
public class Persona { // NOMBRE DE LA CLASE
    String nombre;     // ATRIBUTO
    int edad;

    public void presentarse() { // METODO
        System.out.println("Hola, soy " + nombre + " y tengo " + edad + " años.");
    }
}
```

4.2.2 Objetos

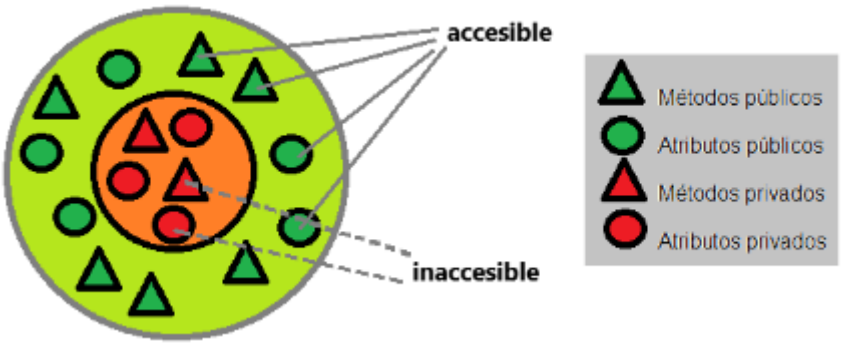
Un **objeto** es una instancia concreta de una clase. Cuando se crea un objeto, se asigna un espacio en memoria y se puede interactuar con sus atributos y métodos.

```java
public class Main {
    public static void main(String[] args) {
        Persona persona = new Persona();
        persona.nombre = "Ana";
        persona.edad = 25;
        persona.presentarse();
    }
}
```

En este ejemplo, `persona` es un objeto de la clase `Persona` con atributos `nombre` y `edad`.

4.3 2. Encapsulación

La **encapsulación** consiste en restringir el acceso directo a ciertos datos de un objeto, protegiendo sus atributos para que solo se modifiquen a través de métodos específicos.



4.3.1 Ejemplo de Encapsulación

```
class Persona {
    public String nombre;
    private int edad;

    public Persona(String nombre) {
        this.nombre = nombre;
    }

    public String getEdad() {
        return edad;
    }

    public void setEdad(int edad) {
        this.edad = edad;
    }
}
```

En este caso, el atributos `nombre` es público y `edad` es privados, y solo se pueden acceder mediante los métodos `getEdad()` y `setEdad()`.

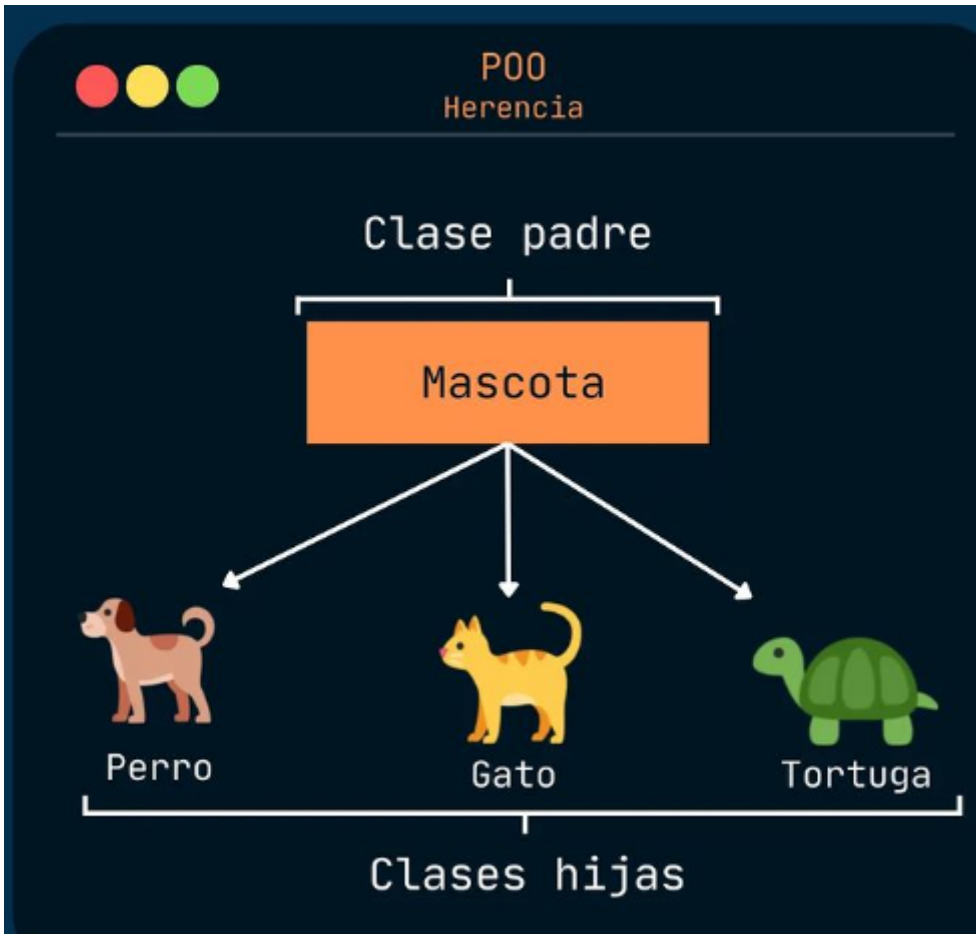
4.4 3 Herencia

La **herencia** permite que una clase (subclase) herede atributos y métodos de otra clase (superclase), permitiendo reutilizar código y extender funcionalidades.

Ejemplo de Herencia

```
class Animal {
  public String nombre;
  public Animal (String nombre){
    this.nombre = nombre;
  }
  public void sonido() {
        System.out.println("El animal hace un sonido");
  }
}

class Perro extends Animal {
  @Override
```

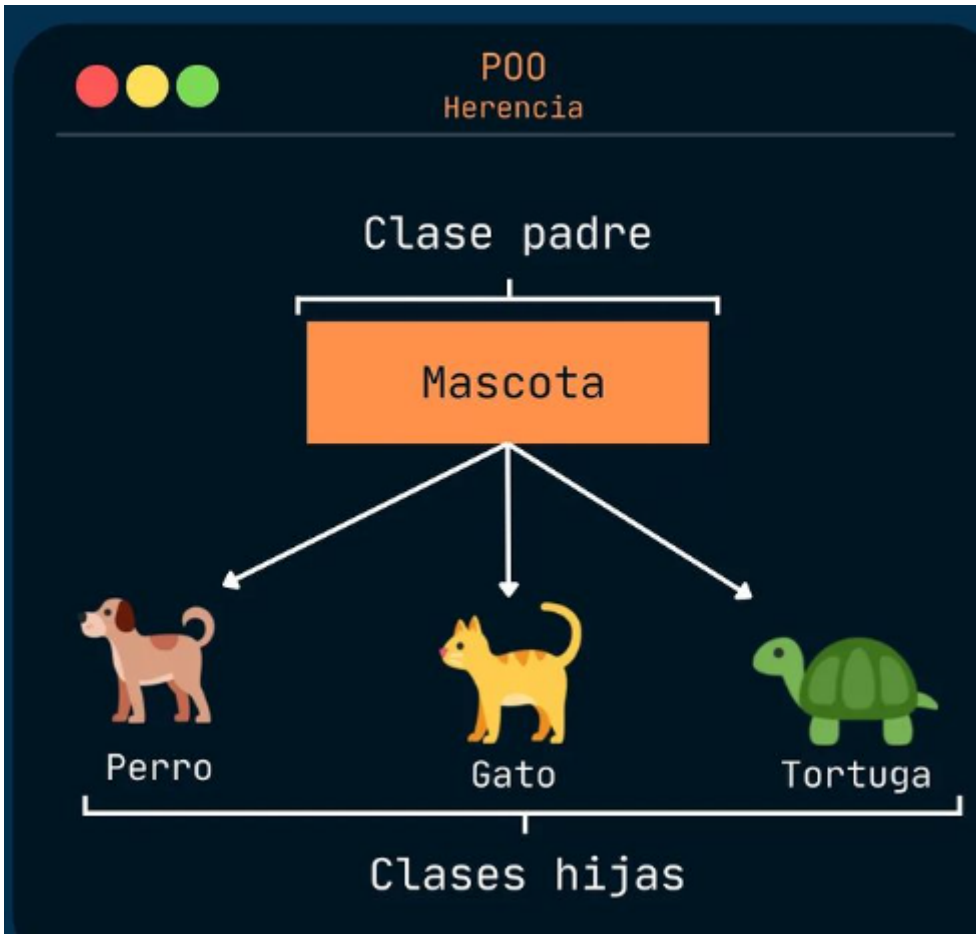

POO
Herencia
Clase padre
Mascota
Perro
Gato
Tortuga
Clases hijas

```
    public void sonido() {
        System.out.println("El perro ladra");
    }
}
```

Aquí, `Perro` es una subclase de `Animal` y hereda el atributo nombre y su método `sonido()`, que luego sobrescribimos (@verride) para proporcionar un comportamiento específico.

5 Polimorfismo

El **polimorfismo** permite que objetos de diferentes clases respondan al mismo método de formas distintas, aumentando la flexibilidad del código.



5.0.1 Ejemplo de Polimorfismo

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

class Perro extends Animal {
    public void sonido() {
        System.out.println("El perro ladra");
    }
}

class Gato extends Animal {
    public void sonido() {
        System.out.println("El gato maúlla");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal miAnimal = new Perro();
        miAnimal.sonido();
    }
}
```

En este ejemplo, el método `sonido()` se comporta de manera distinta dependiendo del tipo de objeto (`Perro` o `Gato`) que lo invoque.

TEST

Part V

Unidad 5: Modelamiento de clases y objetos

6 Modelamiento de clases y objetos.

ntro

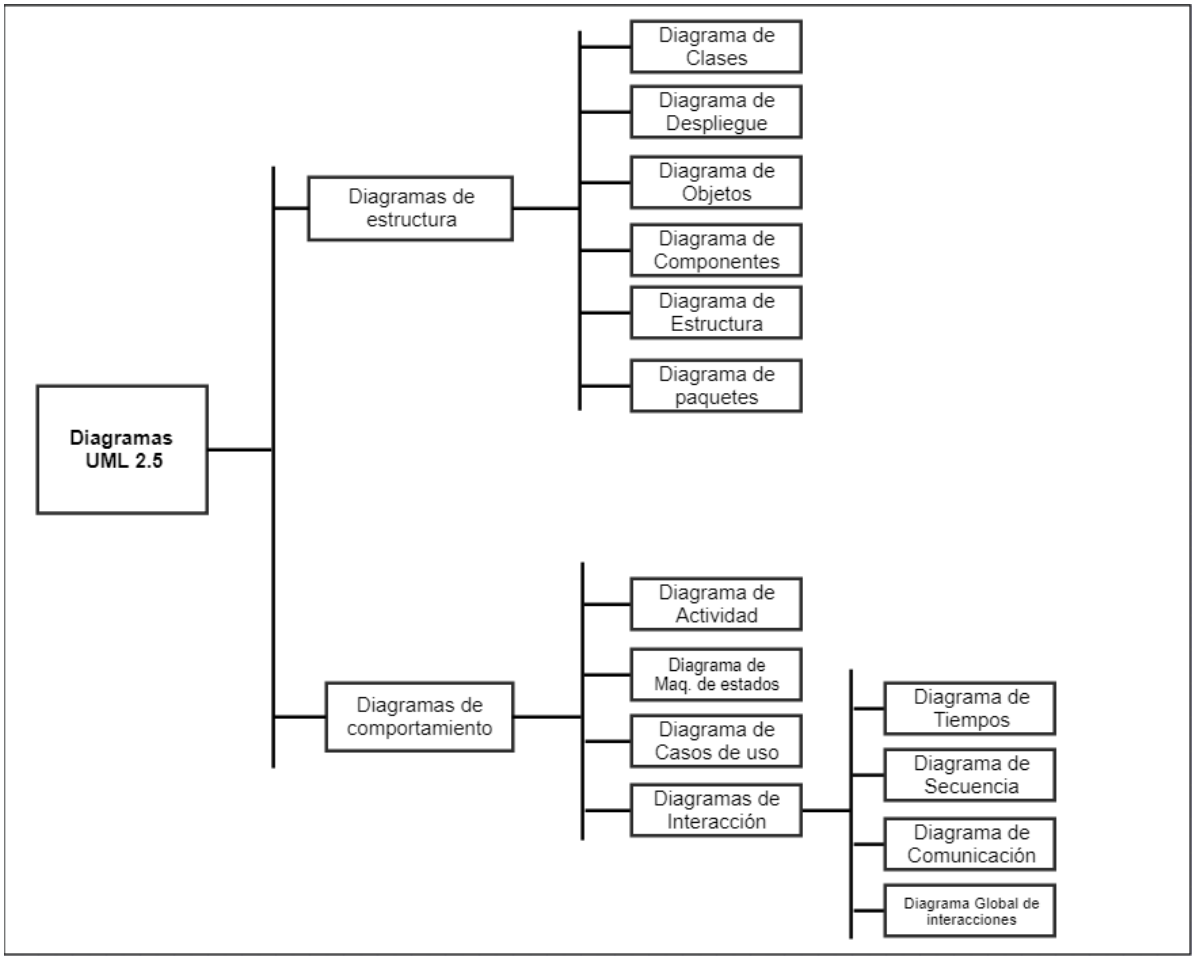
El modelamiento de clases y objetos es un tema fundamental en la programación orientada a objetos, un paradigma de programación que se basa en la idea de que los programas pueden estar formados por entidades que representan conceptos o cosas del mundo real. Estas entidades se llaman objetos, y tienen características (atributos) y comportamientos (métodos) que los definen y los diferencian de otros objetos. Los objetos se agrupan en clases, que son modelos o plantillas que definen las propiedades y las operaciones comunes de un conjunto de objetos. Por ejemplo, una clase puede ser Persona, y un objeto puede ser Juan, que es una instancia de la clase Persona.

El modelamiento de clases y objetos consiste en diseñar y representar gráficamente las clases y los objetos que forman parte de un sistema de software, así como sus relaciones y sus interacciones. Para ello, se utiliza un lenguaje estándar llamado UML (Lenguaje Unificado de Modelado), que tiene una notación gráfica y textual para describir los diferentes aspectos de un sistema. UML tiene varios tipos de diagramas, pero el más importante para el modelamiento de clases y objetos es el diagrama de clases, que muestra las clases, sus atributos, sus métodos y sus relaciones con otras clases.

Otros diagramas que se puede realizar es aquellos que muestran diferentes aspectos de un sistema. Por ejemplo, un diagrama de clases muestra las clases de un sistema y sus atributos y operaciones, un diagrama de secuencia muestra cómo se comunican los objetos entre sí a lo largo del tiempo, un diagrama de casos de uso muestra los requisitos funcionales de un sistema y los actores involucrados, etc.

El modelamiento de clases y objetos tiene muchos beneficios para el desarrollo de software, como facilitar la comprensión del problema, mejorar la comunicación entre los desarrolladores y los clientes, aumentar la reutilización y la modularidad del código, y reducir los errores y los costos de mantenimiento. El modelamiento de clases y objetos es una habilidad esencial para cualquier programador que quiera crear sistemas de software de calidad, eficientes y adaptables. En este tema, aprenderás los conceptos básicos del modelamiento de clases y objetos, cómo usar UML para representarlos, y cómo implementarlos en un lenguaje de programación como Java.

Los tipos de diagramas UML se pueden observar en la siguiente imagen.


Diagramas UML 2.5
Diagramas de estructura
Diagrama de Clases
Diagrama de Despliegue
Diagrama de Objetos
Diagrama de Componentes
Diagrama de Estructura
Diagrama de paquetes
Diagramas de comportamiento
Diagrama de Actividad
Diagrama de Maq. de estados
Diagrama de Casos de uso
Diagramas de Interacción
Diagrama de Tiempos
Diagrama de Secuencia
Diagrama de Comunicación
Diagrama Global de interacciones

6.1 TIPOS DE RELACIONES | DIAGRAMAS DE CASOS DE USOS.

Los diagramas de casos de uso muestran los casos de uso, los actores y las relaciones entre ellos.

Es, con total seguridad, el diagrama más conocido y es utilizado para representar los actores externos que interactúan con el sistema de información y a través de que funcionalidades (casos de uso o requisitos funcionales) se relacionan. Dicho de otra manera, muestra de manera visual las distintas funciones que puede realizar un usuario (más bien un tipo de usuario) de un Sistema de Información.

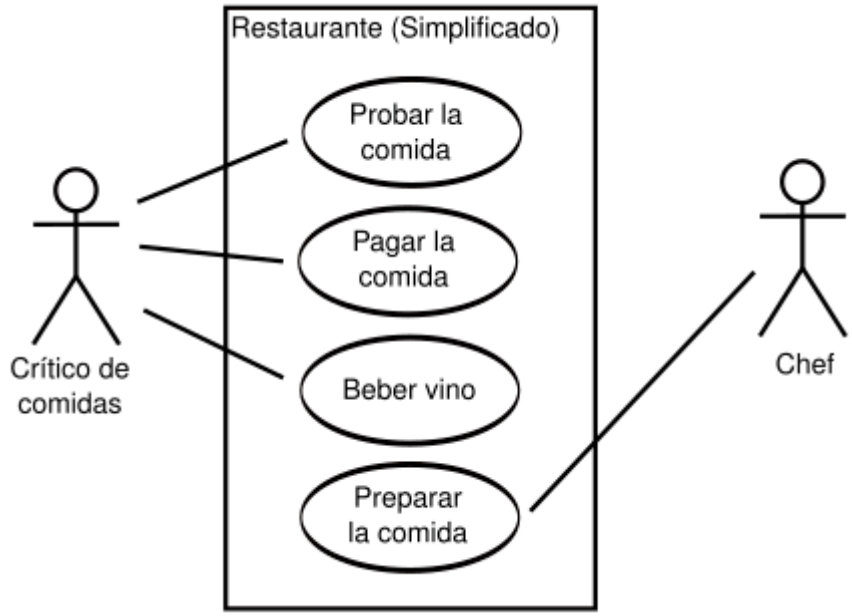
Lo primero es saber cual es su finalidad. El diagrama de casos de uso, dependiendo de la profundidad que le demos, puede ser utilizado para muchos fines, entre ellos podemos encontrar los siguientes:

- **Representar los requisitos funcionales**.
- **Representar los actores** que se comunican con el sistema. Normalmente los actores del sistema son los usuarios y otros sistemas externos que se relacionan con el sistema. En el caso de los usuarios hay que entender el actor como un "perfil", pudiendo existir varios usuarios que actúan como el mismo actor.
- **Representar las relaciones** entre requisitos funcionales y actores.
- **Guiar el desarrollo** del sistema. Crear un punto de partida sobre el que empezar a desarrollar el sistema.
- **Comunicarse de forma precisa entre cliente y desarrollador**. Simplifica la forma en que todos los participes del desarrollo, incluyendo el cliente, perciben como el sistema funcionará y ofrecerá una visión general común del mismo.

6.2 Elementos de un diagrama de casos de uso

Un diagrama de casos de uso está compuesto, principalmente, de 3 elementos: **Actores, Casos de uso y Relaciones**.

La interacción entre actores no se ve en el **diagrama de casos de uso**. Si esta interacción es esencial para una descripción coherente del comportamiento deseado, quizás los límites del sistema o del caso de uso deban de ser re-examinados. Alternativamente, la interacción entre actores puede ser parte de suposiciones usadas en el caso de uso. Sin embargo, los actores son una especie de rol, un usuario humano u otra entidad externa puede jugar varios papeles o roles. Así el Chef y el Cajero podrían ser realmente la misma persona.



6.2.1 Actores

Como ya hemos comentado en la presentación, un actor es algo o alguien externo al sistema que interactúa de forma directa con el sistema. Cuando decimos que interactúa nos referimos a que aporta información, recibe información, inicia una acción…

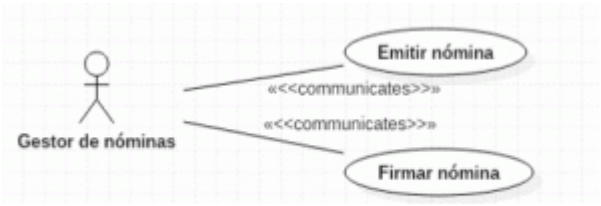
Se representan con una imagen de un "muñeco de palo" con el nombre del actor debajo

Representación de un actor

Existen dos tipos de actores: Los usuarios y los sistemas.

No hay que entender los usuarios como personas singulares, sino como "perfiles o roles" que identifican a un tipo de usuario, pero no al usuario en sí. Por ejemplo, en una aplicación de gestión de nóminas, un actor de este tipo podría ser "gestor de nóminas" que se encarga de emitir y firmar nóminas. Este rol podría ser tomado, por ejemplo, por cualquier individuo del

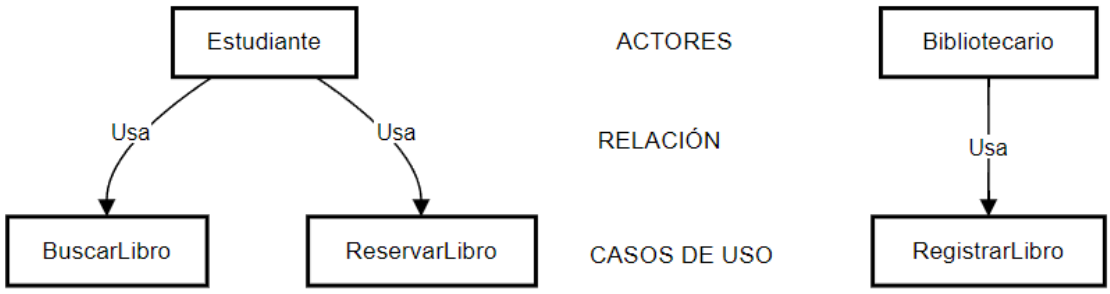
personal de recursos humanos y, además, por el jefe de la empresa. Es un ejemplo muy sencillo, pero como puedes ver, un actor no representa a una única persona o a un único usuario.



Ejemplo de actor

Por otro lado, los actores pueden ser otros sistemas que también interactúan con nuestro propio sistema. Un ejemplo podría ser, en nuestra aplicación de nóminas, un sistema que almacene las nóminas firmadas a modo de archivo. En este caso cuando se firma la nómina se recibe la misma por el sistema de archivo, por tanto el caso de uso se relaciona con el actor.

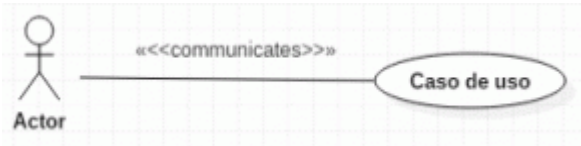
En ocasiones este tipo de actores no se representa con un "hombre de palo" porque puede dar la sensación de que es un usuario y queda poco intuitivo.



6.3 Relaciones

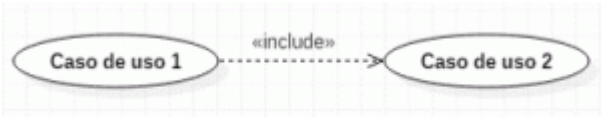
Las relaciones **conectan los casos de uso** con los actores o los casos de uso entre sí.

Cuando conectan un actor con un caso de uso representa que ese actor **interactúa** de alguna manera con ese caso de uso y se representa con una linea continua con la identificación *«communicates»*.



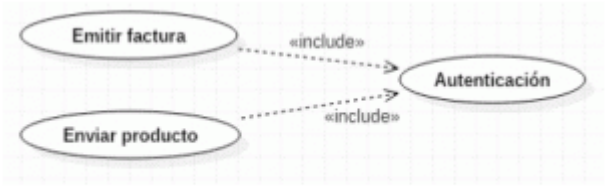
Cuando conectan casos de uso entre sí se pueden diferenciar dos tipos de relaciones: **«include» y «extends»**. En español a veces se usa la nomenclatura «usa» y «extiende»:

- **«include»:** Se utiliza para representar que un caso de uso **utiliza siempre** a otro caso de uso. Es decir, un caso de uso se ejecutará obligatoriamente (lo incluye, lo usa). Se representa con una flecha discontinua que va desde el caso de uso de origen al caso de uso que se incluye.



Relación include entre dos casos de uso

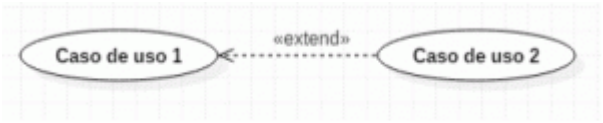
Un uso típico de este tipo de relaciones se produce cuando dos casos de uso **comparten una funcionalidad**. Esa funcionalidad es extraida de los dos y se crea un caso de uso nuevo que se relaciona con los anteriores con un include.



Ejemplo de uso de include

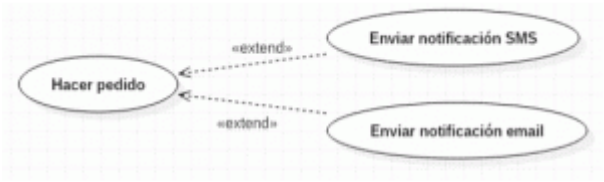
En este ejemplo, los casos de uso emitir factura y enviar producto ejecutarán ambos el caso de uso autenticación.

- **«extend»:** Este tipo de relaciones se utilizan cuando un caso de uso tiene un comportamiento **opcional**, reflejado en otro caso de uso. Es decir, un caso de uso puede ejecutar, normalmente dependiendo de alguna condición o flujo del programa, otro caso de uso. Se representa con una flecha discontinua que va desde el caso de uso opcional al original.



Relación extend entre dos casos de uso

Un ejemplo de esta relación podría ser la siguiente:

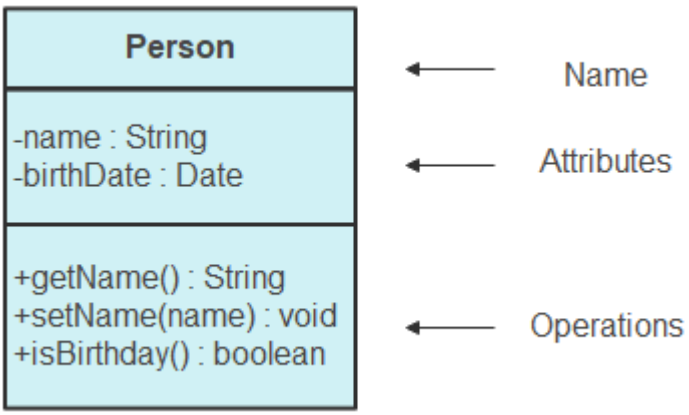


Ejemplo de relaciones extend

En este supuesto el caso de uso Hacer pedido puede dar lugar (o no) a otros dos casos de uso: Enviar notificación SMS y Enviar notificación email. Se supone que, cuando un usuario hace un pedido, el sistema le permite elegir si quiere que se envíe una notificación de ese pedido por SMS o por email

6.4 Tipos de relaciones.

Para una mejor explicación del tipo de relaciones hagamos un ejemplo teniendo presente el siguiente diagrama.



En el grafico anterior podemos observar que consta de lo siguiente: nombre | atributos | métodos.

Con esta información realicemos ejemplos para un mejor entendimiento.

Si tenemos una clase denominada Animales, cuyos atributos son nombre, ID, edad y sus métodos son definirNombre y comer.

Para esto tener presente la información de la imagen siguiente.

Símbolo	Mnemónico	Resultado
-	Private	Atributos **privados** solo pueden ser vistos y utilizados por métodos de la misma clase. Por omisión los atributos son privados.
+	Public	Atributos **públicos** pueden ser vistos y utilizados por métodos de la misma clase y de otras clases.
#	Protected*	Atributos **protegidos** pueden ser vistos y utilizados por métodos de la misma clase y por métodos de clases derivadas o subclases.
_	Static*	Atributos **estáticos** son únicos para toda la clase, es decir, no pertenecen a ningún ejemplar u objeto de la clase, pero pueden ser vistos por todos los objetos de la clase.

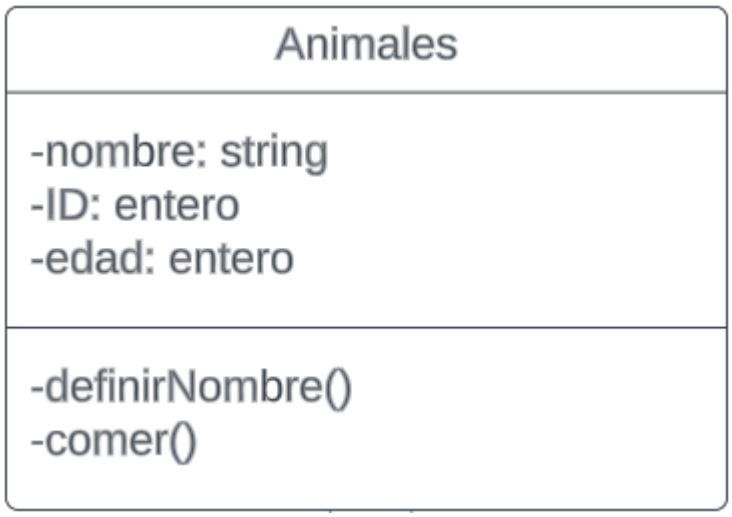
Es decir para representar atributos privados sería de la siguiente manera:

-nombre | Atributo privado

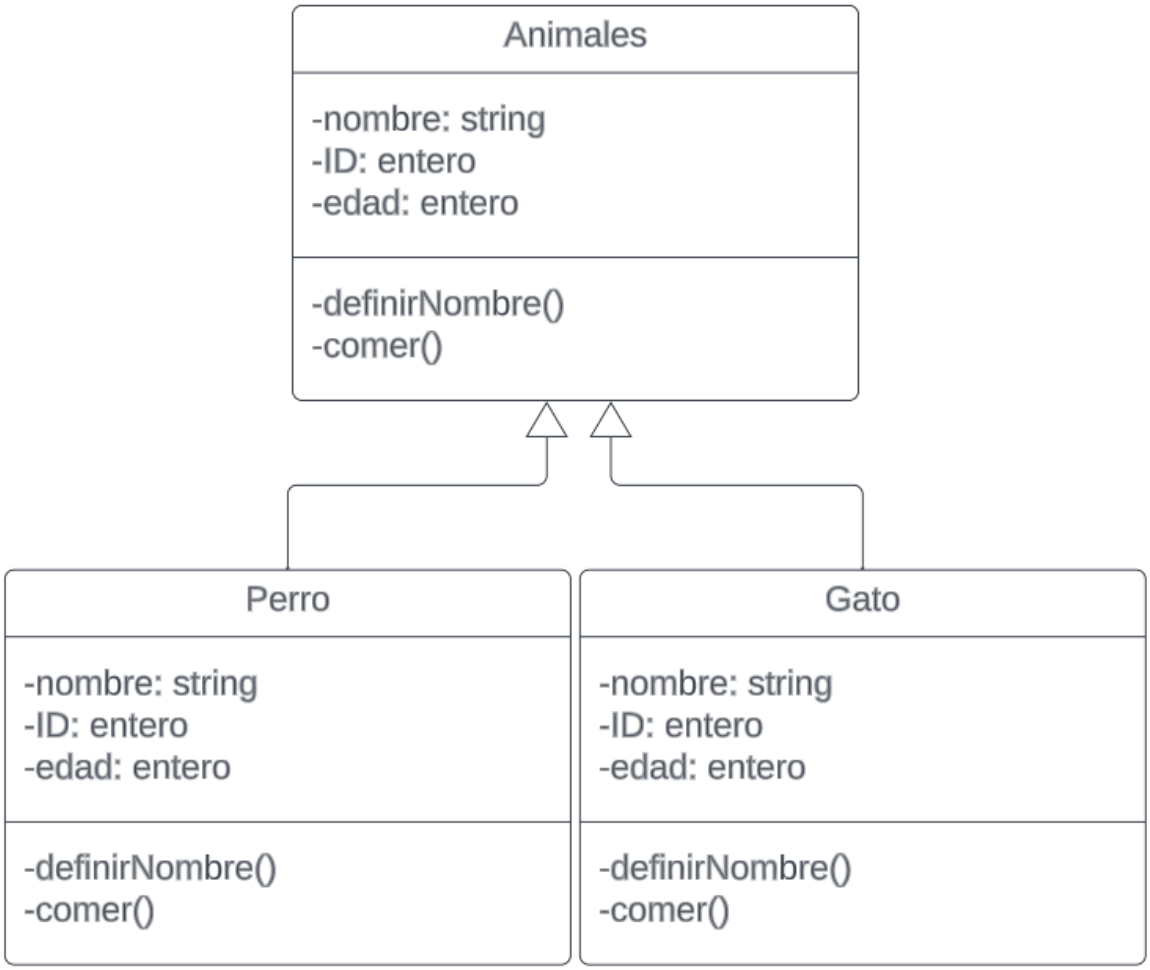
+ID | Atributo público

#edad | Atributo protegido.

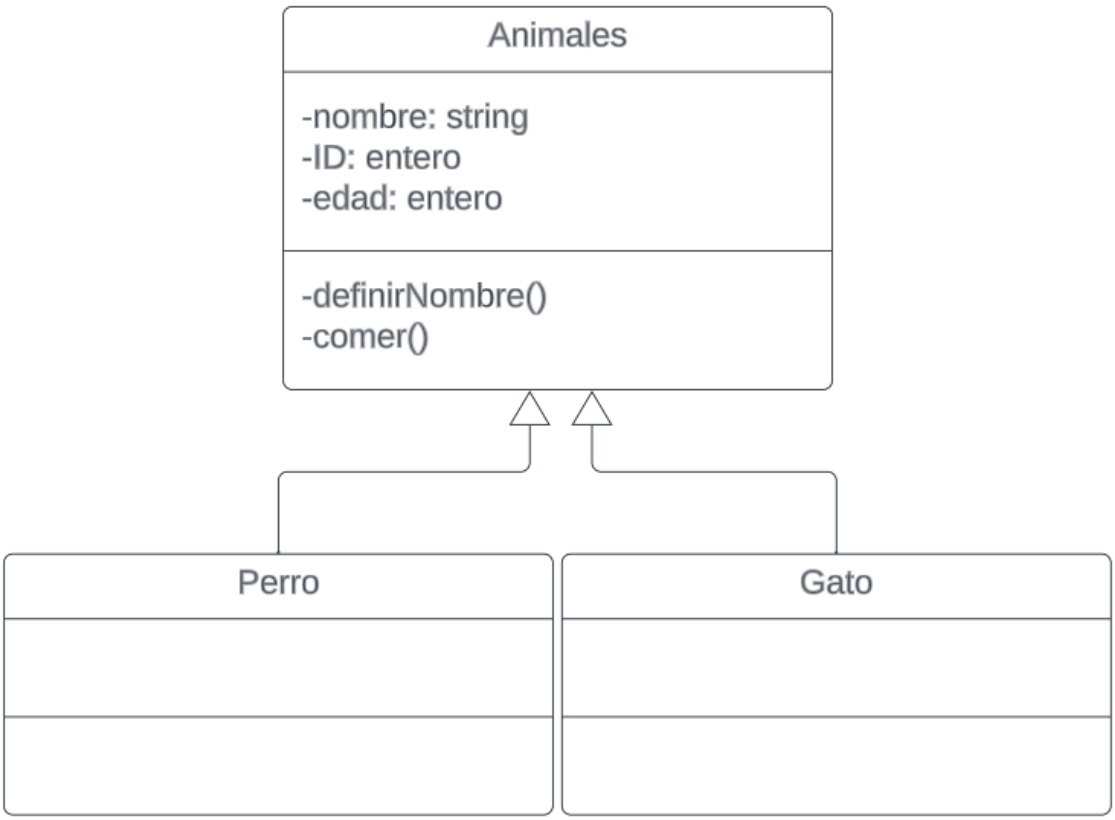
Teniendo la representación de la clase como se observa en la siguiente imagen.



Que sucedería si se quisiera crear 2 animales diferentes un perro y un gato.



Tendríamos el siguiente diagrama, sin embargo, aplicando el principio de herencia, podemos re utilizar el código, como se observa en la imagen siguiente.

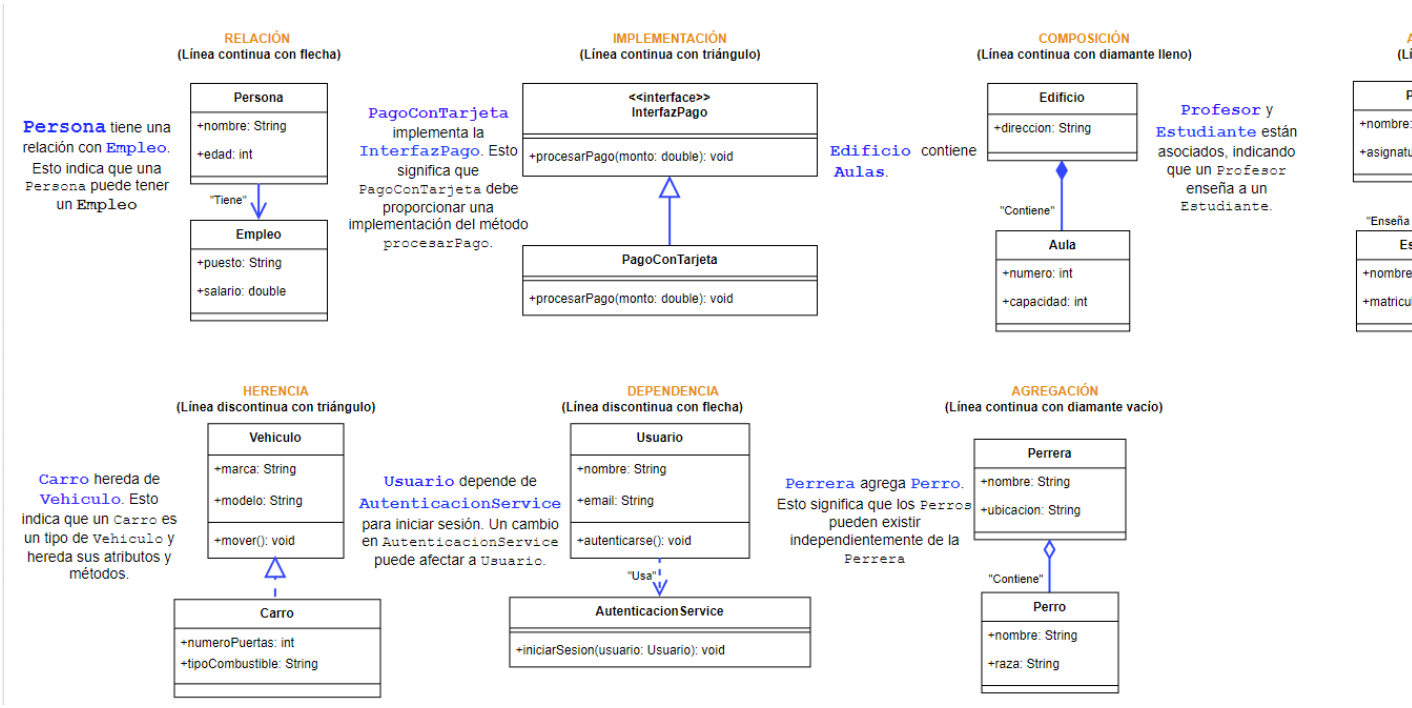


La clase animales en la imagen anterior toma el nombre de clase padre o superclase y las clases de perro y gato toman el nombre de subclase o clase hija.

6.5 Tipos de relaciones en diagramas de clases UML:

6.5.1 Relación (Línea continua con flecha)

- **Ejemplo**: `Persona` tiene una relación con `Empleo`.
- **Descripción**: Esta relación indica que una clase (`Persona`) está asociada a otra clase (`Empleo`). En este caso, una persona puede tener un empleo, lo que representa una asociación básica entre estas dos entidades.
- **Símbolo**: Línea continua con flecha.



6.5.2 Implementación (Línea continua con triángulo)

- **Ejemplo**: `PagoConTarjeta` implementa la `InterfazPago`.
- **Descripción**: En este caso, `PagoConTarjeta` debe proporcionar una implementación del método `procesarPago()` definido en la interfaz `InterfazPago`. Esto significa que `PagoConTarjeta` se compromete a cumplir con el contrato especificado por la interfaz.
- **Símbolo**: Línea continua con triángulo en la dirección de la interfaz.

6.5.3 Composición (Línea continua con diamante lleno)

- **Ejemplo**: `Edificio` contiene `Aulas`.
- **Descripción**: La composición representa una relación fuerte de "parte de". En este caso, las aulas forman parte del edificio, y si el edificio se destruye, las aulas también desaparecen. `Aula` no puede existir sin `Edificio`.
- **Símbolo**: Línea continua con diamante lleno en el lado del todo (`Edificio`).

6.5.4 Asociación (Línea continua)

- **Ejemplo**: `Profesor` y `Estudiante` están asociados.
- **Descripción**: Indica una relación entre dos clases en la que ambas pueden interactuar entre sí. En este caso, un `Profesor` enseña a un `Estudiante`. Es una relación básica, sin dependencia fuerte ni implicaciones de pertenencia.
- **Símbolo**: Línea continua sin ningún otro símbolo.

6.5.5 Herencia (Línea discontinua con triángulo)

- **Ejemplo**: `Carro` hereda de `Vehiculo`.
- **Descripción**: La herencia indica una relación "es un tipo de". `Carro` hereda de `Vehiculo`, lo que significa que `Carro` es una especialización de `Vehiculo` y hereda sus atributos y métodos. Este es un caso de polimorfismo donde `Carro` puede ser tratado como `Vehiculo`.
- **Símbolo**: Línea discontinua con triángulo apuntando hacia la superclase (`Vehiculo`).

6.5.6 Dependencia (Línea discontinua con flecha)

- **Ejemplo**: `Usuario` depende de `AutenticacionService`.
- **Descripción**: La dependencia indica que una clase (`Usuario`) utiliza otra clase (`AutenticacionService`) temporalmente para realizar una acción. En este ejemplo, `Usuario` depende de `AutenticacionService` para iniciar sesión. La relación es débil, lo que significa que `Usuario` solo necesita `AutenticacionService` temporalmente.
- **Símbolo**: Línea discontinua con flecha apuntando hacia la clase de la que depende (`AutenticacionService`).

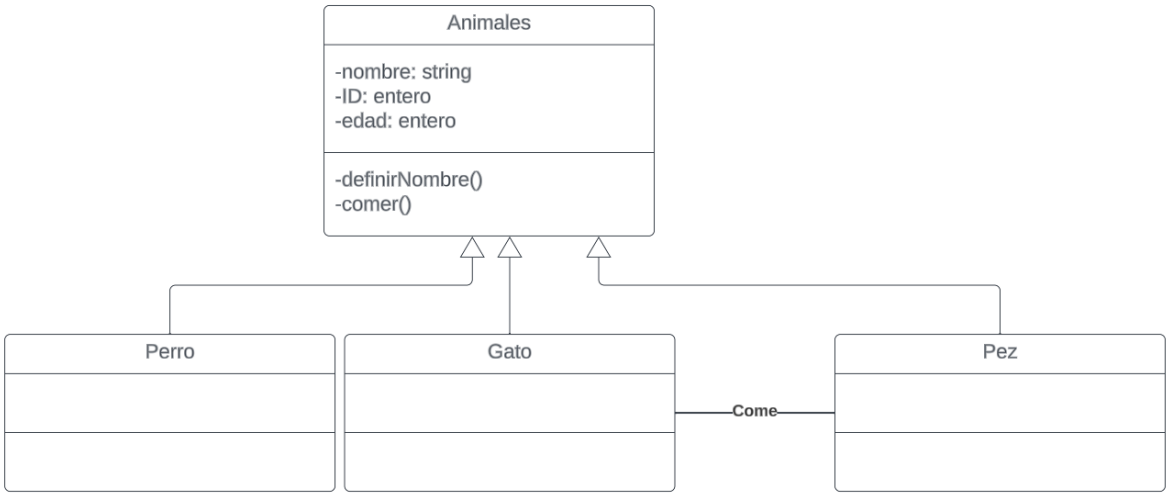
6.5.7 Agregación (Línea continua con diamante vacío)

- **Ejemplo**: `Perrera` agrega `Perro`.
- **Descripción**: La agregación representa una relación de "parte de" débil. `Perro` es una parte de `Perrera`, pero puede existir independientemente de ella. Es decir, si la `Perrera` deja de existir, los `Perros` pueden seguir existiendo.
- **Símbolo**: Línea continua con un diamante vacío en el lado del todo (`Perrera`).

6.6 Asociación:

Utilizando una linea simple la asociación indica una relación entre dos clases sin dependencia.

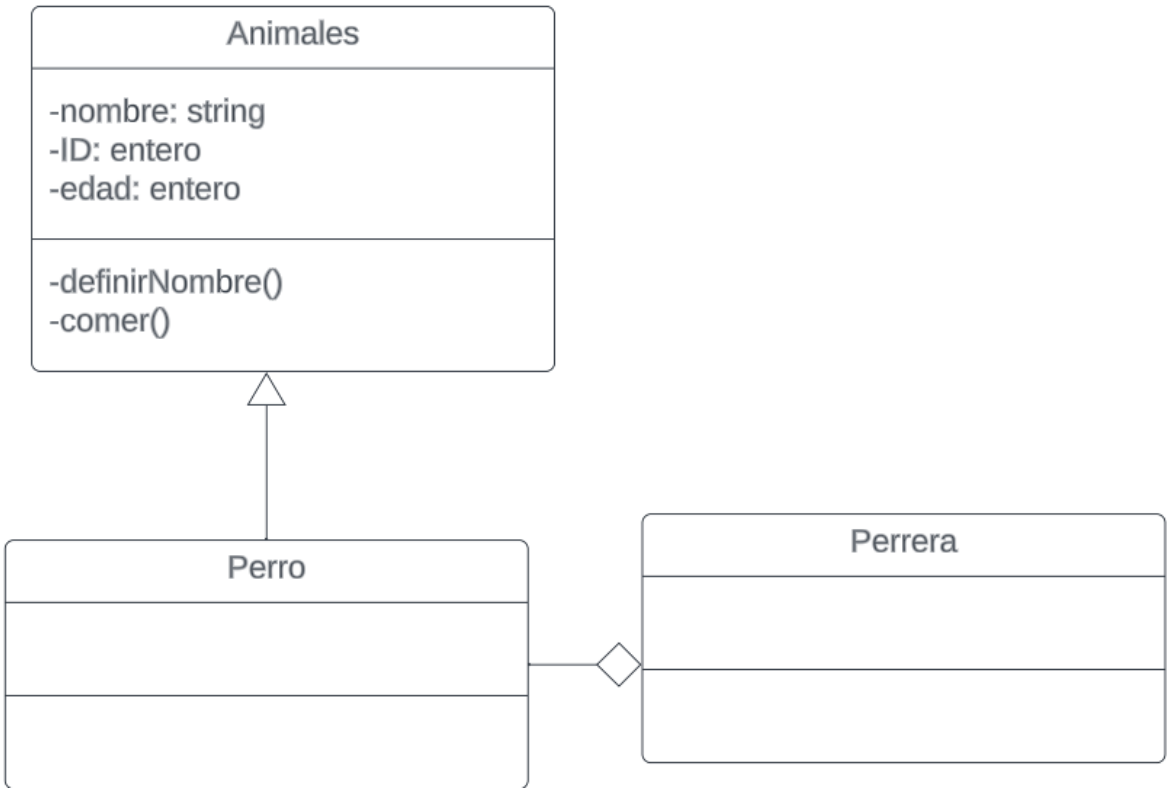
Ejemplo si ahora creamos un animal (PEZ), podemos decir que los gatos pueden comer peces utilizando el diagrama que se observa en la siguiente imagen.



6.7 Agregación:

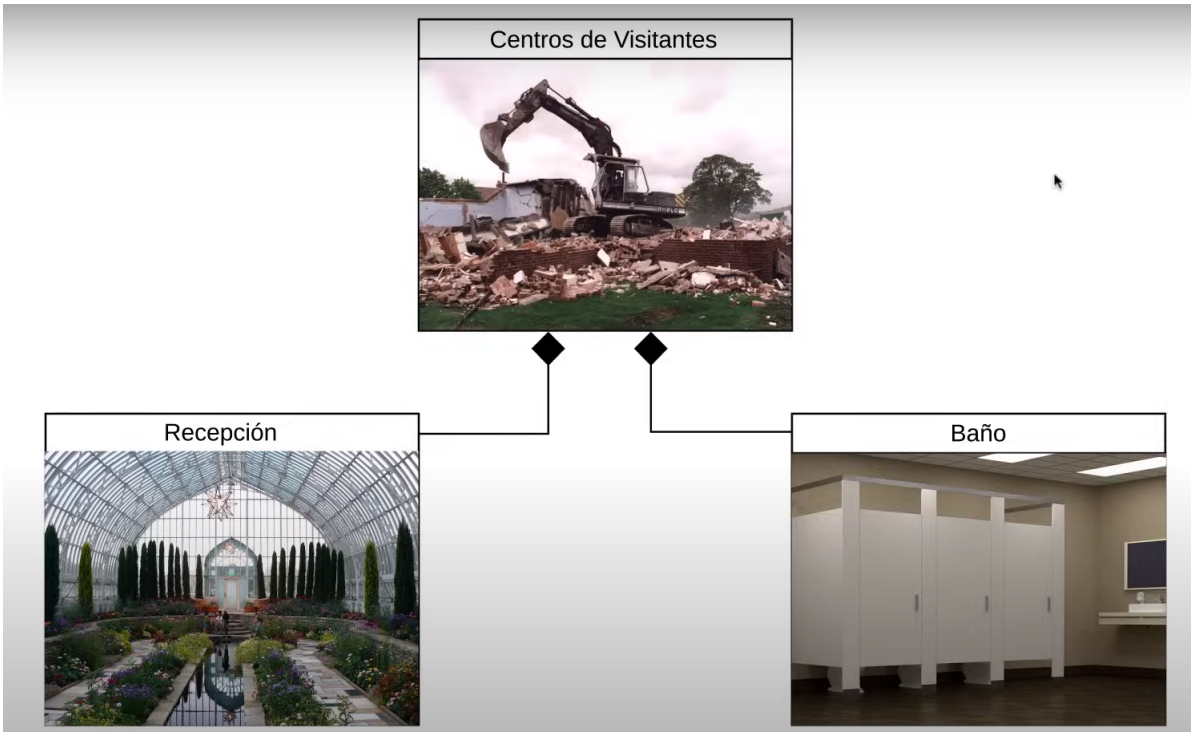
Es un tipo especial de asociación que especifíca un todo y sus partes, esta relación indica que los elementos pueden existir fuera del todo. Ejemplo:

Si creamos una clase Perro, y una clase Perrera, el perro podría pertenecer a la clase Perrera, pero no depende necesariamente la clase perro de la perrera y se representa como se observa en la siguiente imagen.



6.8 Composición:

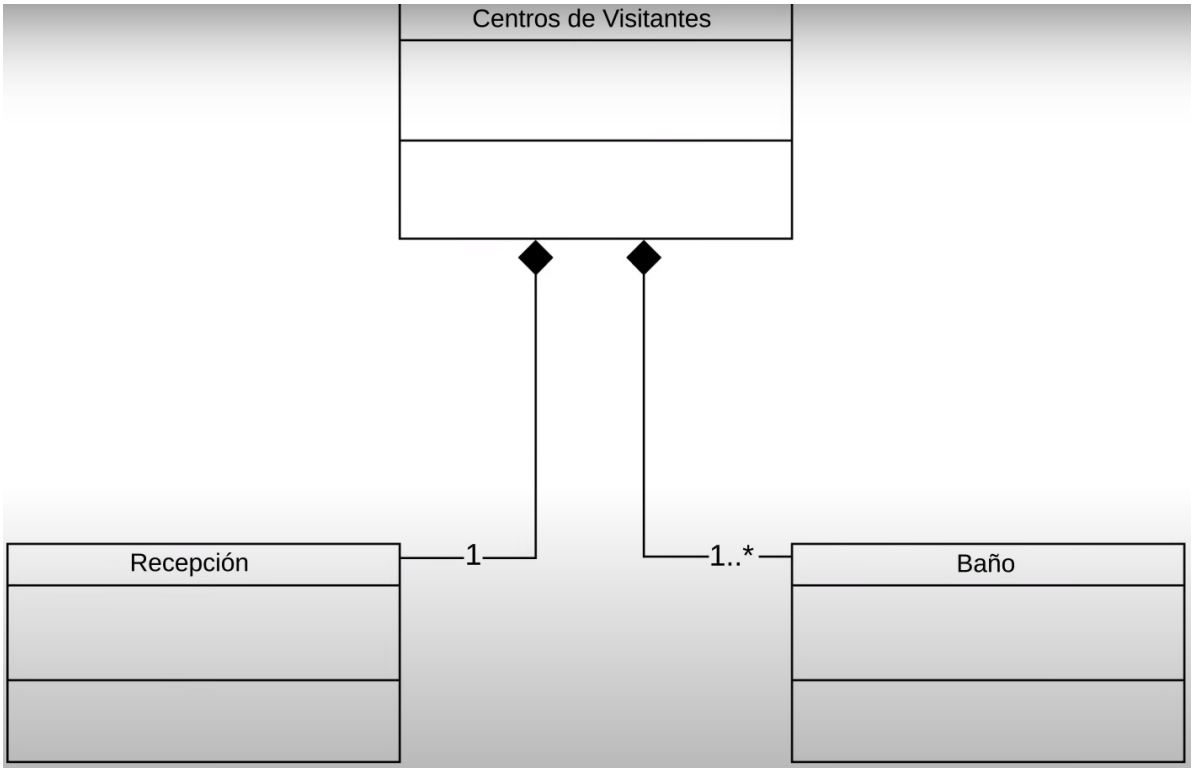
Es la relación en la que la parte no puede existir fuera del todo sin la otra clase. Ejemplo, si tenemos un centro de visitantes en la cual tenga varios espacios físicos como recepción y baños, Si el centro de visitantes se derrumbara sus espacios físicos como recepción y baño no podrían exisitir separados.



6.9 Multiplicidad.

Permite restringir las restricciones numéricas de dependencia.

Ejemplo si queremos especificar que el centro de visitantes tiene un solo centro de recepción, y que puede tener uno o varios baños, se puede realizar la siguiente manera.



Si tienes más dudas, te sugiero revisar el siguiente video:

6.10 PlantiumUML herramienta

```
@startuml
class Persona {
    + nombre : String
    + correr() : void
}
@enduml
```

```
@startuml
class Persona {
    + nombre : String
    + correr() : void
}

class Estudiante extends Persona {
```

```
    + matricula : String
    + estudiar() : void
}
@enduml
```

6.11 PlantiumUML Video

6.12 Video

6.13 TEST

7 Summary

In summary, this book has no content whatsoever.

References

Knuth, Donald E. 1984. “Literate Programming.” *Comput. J.* 27 (2): 97–111. https://doi.org/10.1093/comjnl/27.2.97.