UML(Unified Modeling Language)

UML (Unified Modeling Language) es un lenguaje de modelado estándar utilizado para visualizar, especificar, construir y documentar los componentes de un sistema de software. Permite representar gráficamente la estructura y comportamiento de un sistema a través de diagramas como diagramas de clases, casos de uso, secuencia, entre otros, facilitando el diseño y la comunicación entre desarrolladores y otras partes interesadas.

Tipos de diagramas UML
Un diagrama es la representación gráfica de un conjunto de elementos con sus relaciones, ofreciendo una vista del sistema a modelar. Para poder representar correctamente un sistema, UML ofrece varios tipos de diagramas que permiten visualizar un sistema desde varias perspectivas. UML incluye los siguientes diagramas:

Diagrama de clases: Es el diagrama más importante a la hora de describir el diseño de los sistemas orientados a objetos. El diagrama de clases muestra un conjunto de clases, atributos, operaciones, interfaces y sus relaciones.

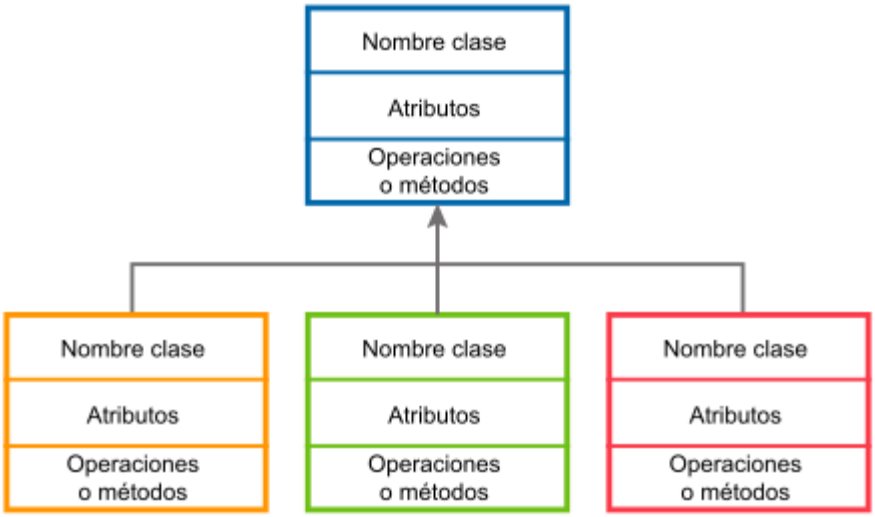


Diagrama de actividades: El diagrama de actividades permite mostrar la secuencia de las acciones de varios objetos y seleccionar el orden en que se harán las cosas. Representa los procesos de negocios de alto nivel, incluidos el flujo de datos. También puede utilizarse para modelar lógica compleja y/o paralela dentro de un sistema. Un diagrama de actividades es la versión UML de un diagrama de flujo y se usan para analizar procesos.

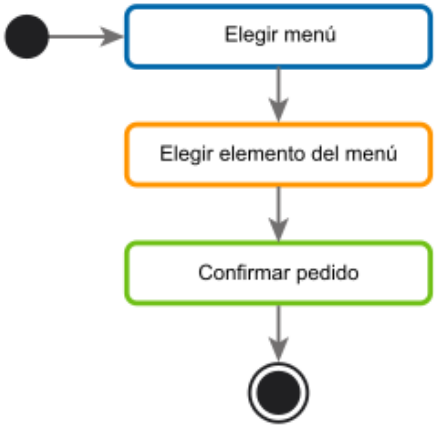


Diagrama de secuencias:
En el diagrama de secuencia se muestran las clases a lo largo de la parte superior y los mensajes enviados entre esas clases, modelando un solo flujo a través de los objetos del sistema.
Un diagrama que representa una interacción, poniendo el foco en la secuencia de los mensajes que se intercambian, junto con sus correspondientes ocurrencias de eventos en las líneas de vida.

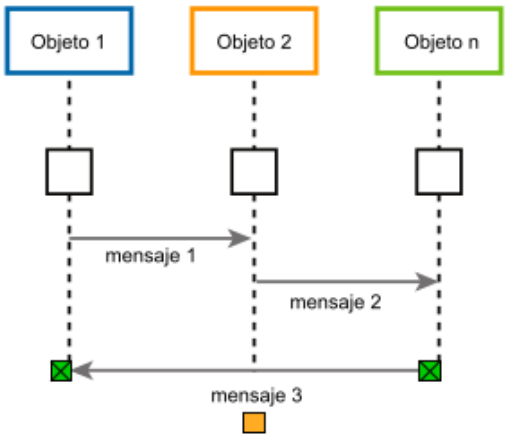


Diagrama de casos de uso: Un diagrama que muestra las relaciones entre los actores y el sujeto (sistema), y los casos de uso. Modela la funcionalidad del sistema según la percepción del usuario externo (actor) y las transacciones entre los actores y el sistema. Los diagramas de casos de uso son responsables principalmente de documentar los macrorrequisitos del sistema. Piense en los diagramas de casos de uso como la lista de las funcionalidades que debe proporcionar el sistema.

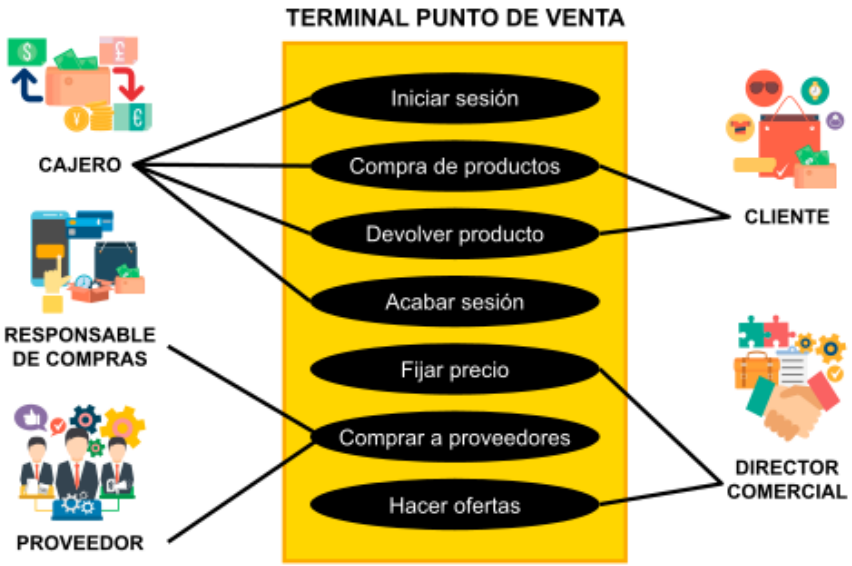
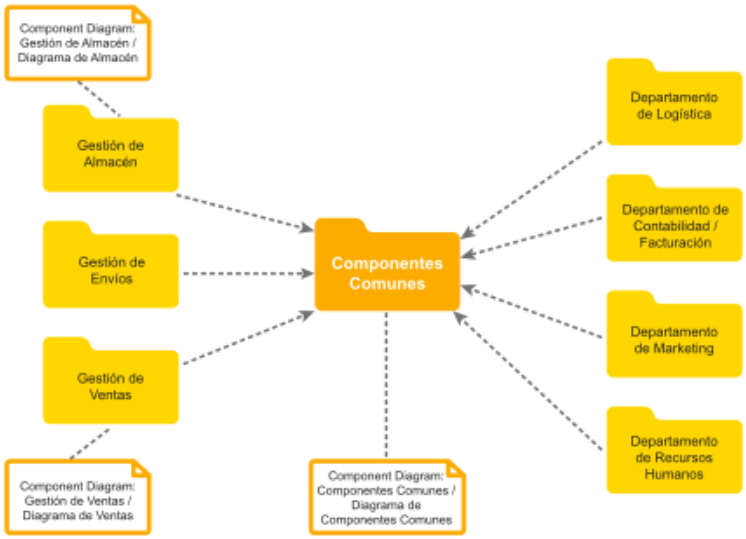


Diagrama de componentes: Representa los componentes que conforman una aplicación, sistema o empresa. Los componentes, sus relaciones, interacciones y sus interfaces públicas. En esta vista se modelan los componentes del sistema, a partir del cual se construye la aplicación. Son importantes para construir sistemas más grandes a partir de partes pequeñas.



Otros diagramas:

Diagrama	Descripción
Diagrama de estado	**Un diagrama de estados muestra el estado cambiante o dinámica de un objeto. Son importantes en el modelado de comportamiento de una interfaz, clase ante los eventos disparadores del objeto. Cada estado modela un periodo de tiempo, durante la vida de un objeto. Cuando ocurre un evento, se puede desencadenar una transición que lleve el objeto a un nuevo estado.**
Diagrama de despliegue físico	**Un diagrama de despliegue físico muestra cómo y dónde se desplegará el sistema. Las máquinas físicas y los procesadores se representan como nodos y la construcción interna puede ser representada por nodos o artefactos embebidos. Como los artefactos se ubican en los nodos para modelar el despliegue del sistema, la ubicación es guiada por el uso de las especificaciones de despliegue. En este tipo de diagrama se muestran los tipos de nodos del sistema y los tipos de componentes que contienen. Aquí el nodo se visualiza como un cubo.**
Diagrama de objetos	**Un diagrama que presenta los objetos y sus relaciones en un punto en el tiempo. Un**

	diagrama de objetos se puede considerar como un caso especial de un diagrama de clases o un diagrama de comunicaciones.
Diagrama de estructura de composición	**Representa la estructura interna de un clasificador (tal como una clase, un componente o un caso de uso), incluyendo los puntos de interacción de clasificador con otras partes del sistema. Un uso adicional que se puede dar a los diagramas de estructura compuesta es para mostrar las partes que colaboran, por ejemplo, en un caso de uso.**
Diagrama de paquetes	**Diagrama que permite la organización de elementos de modelado en paquetes y las dependencias entre ellos. Los usos más comunes para los diagramas de paquete son para organizar diagramas de casos de uso y diagramas de clase, a pesar de que el uso de los diagramas de paquete no es limitado a estos elementos UML.**
Diagrama de comunicaciones	**Diagrama que describen los mensajes que transmiten los objetos y muestran las asociaciones que existen entre las clases. Modela las interacciones entre objetos en términos de mensajes en secuencia.**
Diagrama de tiempos	**Diagrama utilizado para mostrar el cambio en el estado o valor de uno o más objetos en el tiempo o en su línea de vida, en respuesta a los eventos o estímulos recibidos.** **Los eventos que se reciben se anotan, a medida que muestran cuándo se desea mostrar el evento que causa el cambio en la condición o en el estado.**

Caso de estudio práctico

El propietario de un centro médico de su ciudad, requiere con urgencia la construcción de un sistema de información que le permita administrar los datos básicos de sus pacientes, tratamientos, citas y gestión de reportes de tal manera que al ejemplificar el sistema se pueda fijar claramente el límite de este.

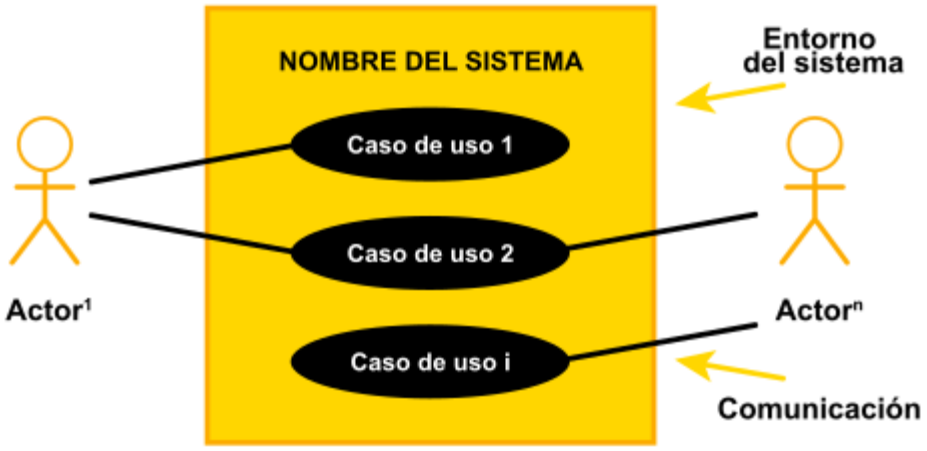
Diagramas del sistema

Al momento de desarrollar un proyecto se debe pensar en cuáles serán las principales funcionalidades que el software debe permitir llevar a cabo y quiénes serán los que podrán ejecutar dichas funcionalidades. La identificación de estos elementos se puede visualizar de manera efectiva a través de la elaboración de diagramas de Casos de Uso. Estos diagramas que son elaborados durante las etapas iniciales de un proyecto se convierten en un referente para cada una de las etapas siguientes del desarrollo del proyecto.

Componentes:

Sistema: se representa mediante un rectángulo. Este se emplea para delimitar gráficamente lo que se encuentra por dentro del sistema (los casos de uso) y lo que está por fuera del sistema (los actores).	
Actor: se representa mediante un “hombre de palo”. Este se emplea para indicar el tipo de usuario del sistema que podrá ejecutar alguna función (caso de uso) en el mismo.	
Caso de uso: se representa mediante un óvalo e indica una función que el sistema debe realizar.	

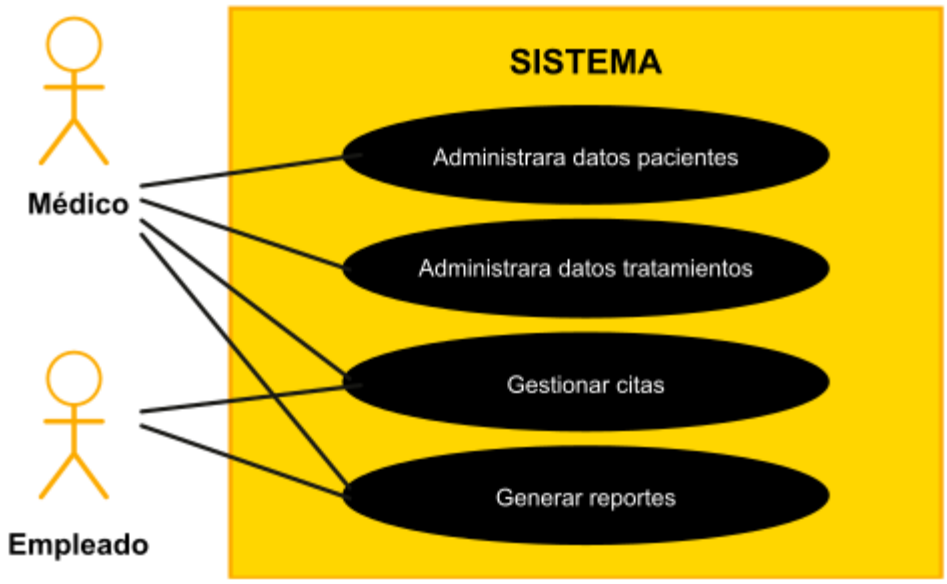
Ejemplo de la representación gráfica de un Diagrama de Caso de Uso:



Para nombrar un proceso se puede emplear un verbo conjugado en infinitivo y que represente la función a realizar (Administrar, Gestionar, Registrar, entre otros). Por ejemplo:

- **Administrar los datos de pacientes.**
- **Gestionar citas médicas.**
- **Administrar datos de historias clínicas o tratamientos.**
- **Generar reportes.**

Representación gráfica



Identificación de casos de uso: en el ejemplo anterior se observan los casos de uso identificados en el sistema, es decir, las funcionalidades que el sistema va a proveer (Administrar datos pacientes, Administrar datos tratamientos, Gestionar citas, Generar reportes).

Identificación de actores: los actores son los usuarios externos que podrán ejecutar los casos de uso, en el ejemplo anterior, se identificaron dos actores (Médico y Empleado).

Las líneas que van del actor al caso de uso se denominan Asociación y sirven para determinar cuáles Casos de uso lleva a cabo un determinado Actor.

Tipos de relaciones

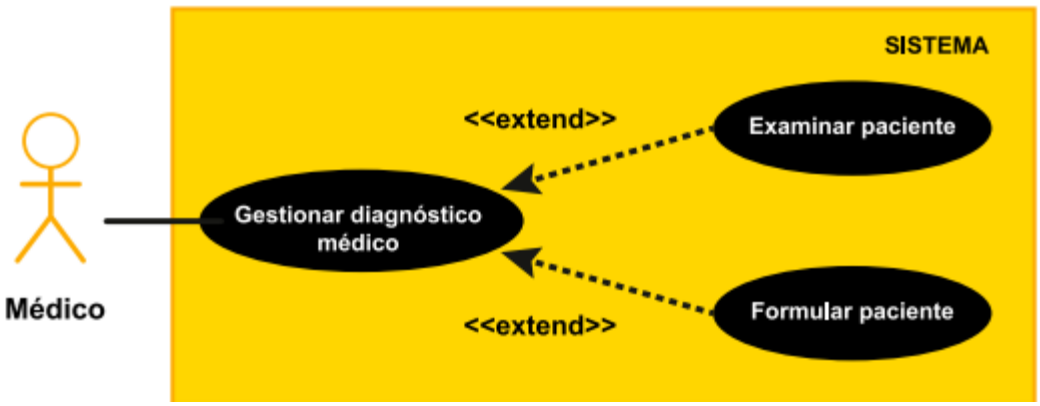
Asociación: es la relación que se da entre los actores y el caso de uso. Esta relación indica que el actor lleva a cabo el caso de uso.

El ejemplo anterior indica que el Médico dentro del sistema Sismédico puede llevar a cabo la funcionalidad de Administrar Datos Pacientes.

Incluido ó <<uses>> ó <<include>>: simboliza la relación que se da entre casos de uso donde uno termina incluyendo al otro (describe el comportamiento de uno en otro).

El ejemplo anterior indica que para Gestionar una cita se debe validar datos, es decir, el caso de uso Gestionar Cita "usa" al caso de uso Validar Datos.

Extendido ó <<extend>>: relación que se da entre casos de uso donde se evidencia que un caso de uso es la generalización o especialización de otro.



El ejemplo anterior indica que, al gestionar un diagnóstico médico, se puede examinar un paciente o formular un paciente, es decir, los casos de uso FORMULAR A PACIENTE y EXAMINAR A PACIENTE se extienden del caso de uso GESTIONAR DIAGNÓSTICO MÉDICO.

Documentación

La técnica de casos de uso requiere además de construir el diagrama de Casos de Uso, la descripción de los mismos. Esta descripción permite detallar el flujo de eventos que se da entre el Sistema y el Actor para llevar a cabo el Caso de Uso. A continuación, se presenta el formato que describe el Caso de Uso del centro médico.

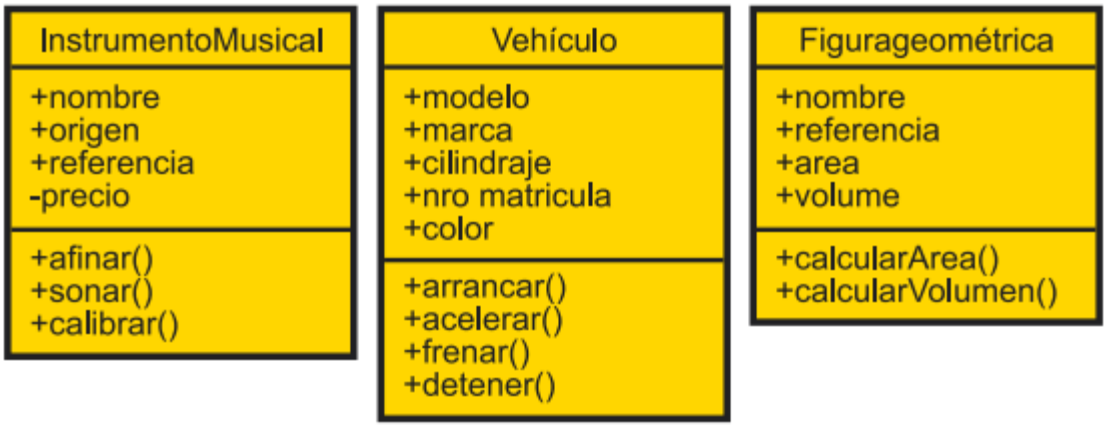
CASO DE USO	ADMINISTRAR DATOS PACIENTES	
Descripción	El comportamiento del sistema deberá describir el paso a paso del caso de uso cuando el personal encargado de gestionar datos del paciente inicie el ingreso de estos.	
Precondición	El paciente no se encuentra ingresado al sistema y tiene la documentación necesaria para poder ser ingresado al sistema.	
Secuencia Normal	Paso	Acción
	1	El personal médico ingresa al sistema para registrar el nuevo paciente.
	2	El sistema carga formulario para registro de datos del paciente así: identificación, nombre(s), apellido(s), dirección, teléfono, estrato, tipo de RH, género, acudiente.
	3	El personal médico ingresa los datos suministrados por el paciente y ejecuta la acción en el sistema.
	4	El sistema almacena los datos suministrados por el personal médico, imprime el carnet del Sistema General de Seguridad Social en Salud, el sistema comunica al personal médico que el proceso ha terminado de manera exitosa.
	5	El personal médico genera reporte del sistema de registro al nuevo paciente, mediante la expedición del carnet o constancia de inscripción al sistema de Seguridad Social en Salud.
PostCondición	El paciente se encuentra registrado en el Sistema General de Seguridad Social en Salud, su historial clínico es nuevo.	
Excepciones	Paso	Acción
	1	Si el sistema detecta la duplicación de un paciente registrado con la identificación que se registra, procede a informar al personal médico, estos deben modificar y/o actualizar la información que sea necesaria y continuar el caso de uso.
	2	Si el personal médico cancela el registro del paciente se termina el caso de uso.

Clases

Una clase es la unidad básica que agrupa una colección de objetos que poseen un tipo de comportamiento. Toda clases se compone de los siguientes elementos:

- **Nombre de la clase.**
- **Atributos, datos o propiedades también denominados miembros de la clase.**
- **Métodos (operaciones) o acciones propias de la clase. Estas acciones se identifican con verbos en infinitivo**

Ejemplos de clases:



ATRIBUTOS: los atributos o características de una Clase pueden ser de tres tipos, que definen su visibilidad:

- **Public (+): indica que el atributo será visible tanto dentro como fuera de la clase, es decir, es accesible desde todos lados. Disponible para consumo externo.**
- **Private(-): indica que el atributo sólo será accesible desde dentro de la clase (sólo sus métodos lo pueden acceder). Disponible para consumo interno.**
- **Protected (#): indica que el atributo no será accesible desde fuera de la clase, pero si podrá ser accedido por métodos de la clase además de las subclases que se deriven (ver herencia). Disponible para consumo interno o consumo por parte de las clases hijos.**

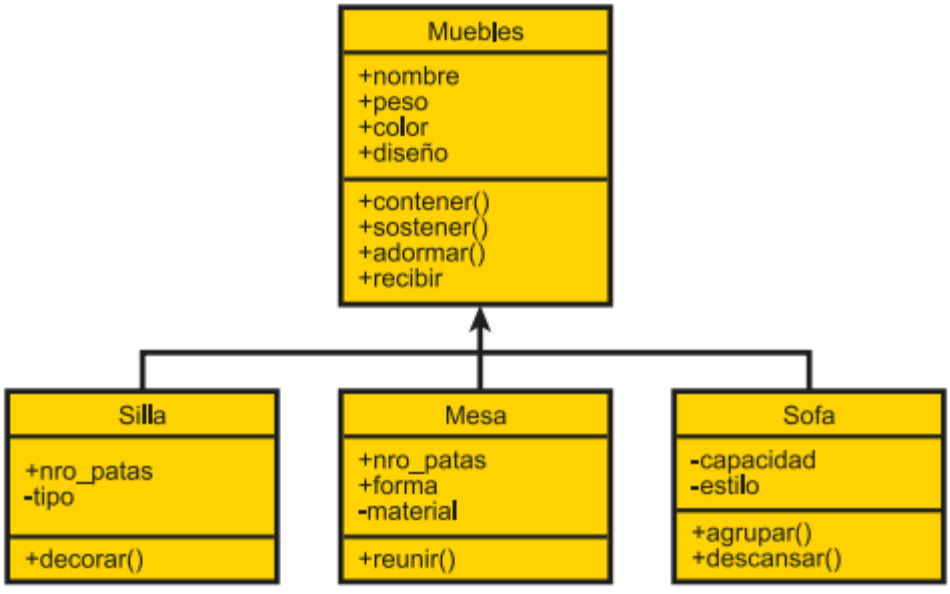
MÉTODOS: los métodos u operaciones de una clase son la forma en cómo ésta interactúa con su entorno, éstos pueden tener las características

- **Public (+): indica que el método será visible tanto dentro como fuera de la clase, es decir, es accesible desde todos lados. Disponible para consumo externo.**
- **Private (-): indica que el método sólo será accesible desde dentro de la clase (sólo otros métodos de la clase lo pueden acceder). Disponible para consumo interno.**
- **Protected (#): indica que el método no será accesible desde fuera de la clase, pero si podrá ser accedido por métodos de la clase además de métodos de las subclases que se deriven (ver herencia)**

Relaciones

Las relaciones entre las clases se dan por Herencia, Asociación, Agregación, Composición y Dependencia (uso).

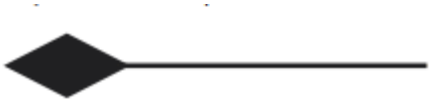
Herencia (Especialización / Generalización)



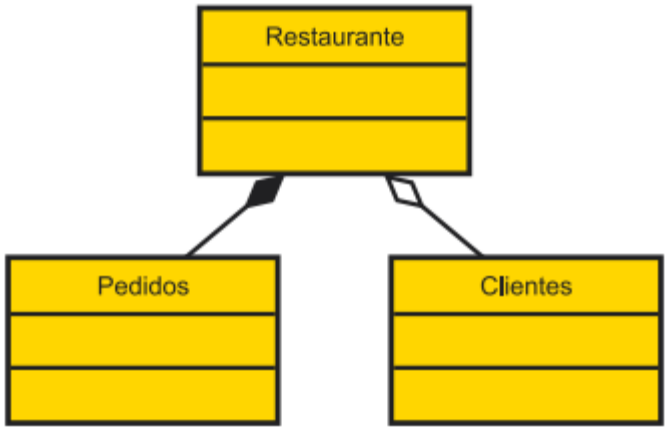
En el ejemplo anterior se ilustra que un Mueble puede ser una Silla, una Mesa o un Sofá.

Agregación y composición

• Composición por valor: relación estática, donde el tiempo de vida del objeto incluido está condicionado por el tiempo de vida del que lo incluye. Este tipo de relación es comúnmente llamada Composición (el Objeto base se construye a partir del objeto incluido, es decir, es "parte/todo"). Se representa de forma gráfica:



• Agregación por referencia: tipo de relación dinámica, en donde el tiempo de vida del objeto incluido es independiente del que lo incluye. Este tipo de relación es comúnmente llamada Agregación (el objeto base utiliza al incluido para su funcionamiento). Se representa de forma gráfica:



En el ejemplo anterior indica que el Restaurante tiene Pedidos y Clientes, sin embargo, los Pedidos requieren del Restaurante para poder existir (Composición), mientras que los Clientes no (Agregación).

Multiplicidades

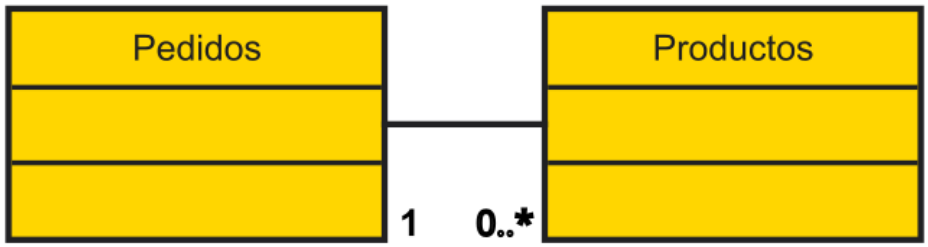
Las multiplicidades en un diagrama de clases UML indican cuántas instancias de una clase pueden estar asociadas con instancias de otra clase. Los valores comunes de multiplicidad son:

1. **1: Exactamente una instancia. Se utiliza para indicar que una instancia de una clase está asociada con una y solo una instancia de la otra clase. Ejemplo: un `Empleado` pertenece a 1 `Empresa`.**
2. **0..1: Cero o una instancia. Representa que una instancia de una clase puede o no estar asociada con una instancia de la otra clase. Ejemplo: un `Usuario` puede tener 0 o 1 `Perfil`.**
3. **0..* o simplemente ***: Cero o muchas instancias. Indica que una instancia de una clase puede estar asociada con cero o más instancias de la otra clase. Ejemplo: un `Cliente` puede tener 0 o muchos `Pedidos`.**
4. **1..*: Una o más instancias. Señala que una instancia de una clase debe estar asociada con al menos una instancia de la otra clase, pero podría tener muchas. Ejemplo: un `Pedido` tiene al menos 1 o más `Productos`.**
5. **n..m: Un rango específico de instancias. Define que una instancia de una clase debe estar asociada con al**

menos n y como máximo `m` instancias de la otra clase.

Asociación

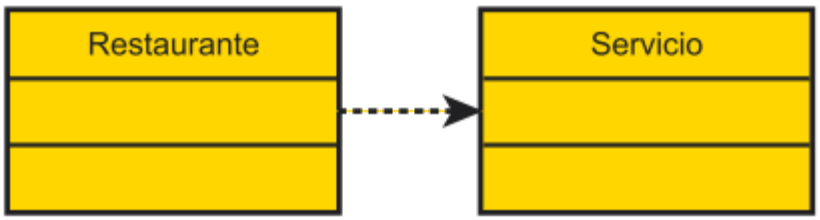
La relación entre clases, permite asociar objetos que colaboran entre sí. Cabe destacar que no es una relación fuerte, es decir, el tiempo de vida de un objeto no depende del otro.



El ejemplo anterior indica que un Pedido está relacionado con cero o muchos Productos.

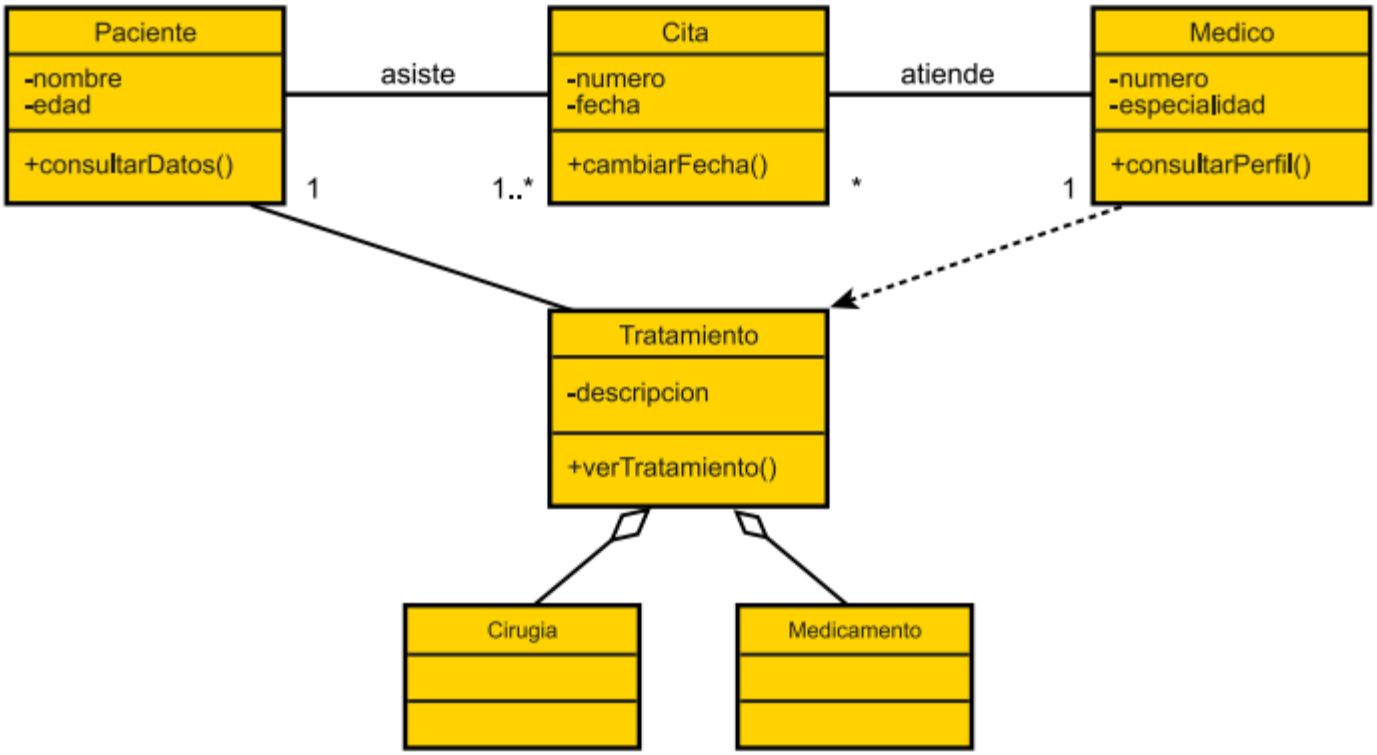
Dependencia

Representa un tipo de relación muy particular, en la que una clase es instanciada (su instanciación es dependiente de otro objeto/clase).



En el ejemplo anterior se indica que la clase Restaurante instancia o crea objetos de la clase Servicio.

Volviendo al ejemplo del caso de uso “Gestionar Citas”, el respectivo diagrama de clases es:



Ejemplo con código JAVA de un diagrama de clases

El siguiente es un programa de Administración académica de una Universidad:

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

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

    public void mostrarInformacion() {
        System.out.println("Nombre: " + nombre + ", Edad: " + edad);
    }
}

// Herencia: Estudiante es una Persona
class Estudiante extends Persona {
    private String matricula;

    public Estudiante(String nombre, int edad, String matricula) {
```

```
        super(nombre, edad);
        this.matricula = matricula;
    }

    public void mostrarInformacion() {
        super.mostrarInformacion();
        System.out.println("Matrícula: " + matricula);
    }
}

// Composición: Una Universidad tiene múltiples Departamentos
class Departamento {
    private String nombre;

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

    public String getNombre() {
        return nombre;
    }
}

class Universidad {
    private String nombre;
    private Departamento[] departamentos;

    // Composición: Universidad crea los Departamentos en su constructor
    public Universidad(String nombre, Departamento[] departamentos) {
        this.nombre = nombre;
        this.departamentos = departamentos;
    }

    public void mostrarDepartamentos() {
        System.out.println("Departamentos en " + nombre + ":");
        for (Departamento dept : departamentos) {
            System.out.println(dept.getNombre());
        }
    }
}

// Agregación: Un Curso puede tener varios Estudiantes, pero los Estudiantes existen
//independientemente del Curso
class Curso {
    private String nombre;
    private Estudiante[] estudiantes;

    public Curso(String nombre, Estudiante[] estudiantes) {
        this.nombre = nombre;
        this.estudiantes = estudiantes;
```

```
    }

    public void mostrarEstudiantes() {
        System.out.println("Estudiantes en el curso " + nombre + ":");
        for (Estudiante estudiante : estudiantes) {
            estudiante.mostrarInformacion();
        }
    }
}

// Asociación: Un Profesor enseña en uno o más cursos
class Profesor {
    private String nombre;

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

    public void asignarCurso(Curso curso) {
        System.out.println(nombre + " enseña el curso: " + curso.nombre);
    }
}

// Dependencia: Un ServicioBiblioteca presta libros a un Estudiante, ServicioBiblioteca necesita
//crear instancias de Estudiante
class ServicioBiblioteca {
    public void prestarLibro(Estudiante estudiante) {
        System.out.println("Prestando libro a: " + estudiante.nombre);
    }
}

// Programa principal
public class Main {
    public static void main(String[] args) {
        // Composición: Universidad y Departamentos
        Departamento[] departamentos = {new Departamento("Ciencias"), new
Departamento("Humanidades")};
        Universidad universidad = new Universidad("Universidad Nacional", departamentos);
        universidad.mostrarDepartamentos();

        // Herencia: Estudiante hereda de Persona
        Estudiante estudiante1 = new Estudiante("Ana", 20, "M1234");
        Estudiante estudiante2 = new Estudiante("Luis", 22, "M5678");

        // Agregación: Curso y Estudiantes
        Estudiante[] estudiantes = {estudiante1, estudiante2};
        Curso curso = new Curso("Matemáticas", estudiantes);
        curso.mostrarEstudiantes();

        // Asociación: Profesor y Curso
```

```
        Profesor profesor = new Profesor("Dr. García");
        profesor.asignarCurso(curso);

        // Dependencia: ServicioBiblioteca y Estudiante
        ServicioBiblioteca biblioteca = new ServicioBiblioteca();
        biblioteca.prestarLibro(estudiante1);
    }
}
```

Identificando las relaciones: Identificamos las relaciones existentes entre las clases del programa

Herencia:

- **La clase `Estudiante` hereda de la clase `Persona`.**
- **Esto se ve en la declaración: `class Estudiante extends Persona`**

Composición:

- **La clase `Universidad` tiene una composición con `Departamento`.**
- **La `Universidad` crea y es responsable del ciclo de vida de los `Departamentos`.**
- **Esto se ve en el constructor de `Universidad` que recibe y almacena un array de `Departamentos`.**

Agregación:

- **La clase `Curso` tiene una agregación con `Estudiante`.**
- **Los `Estudiantes` existen independientemente del `Curso`, pero un `Curso` puede tener varios `Estudiantes`.**
- **Esto se ve en el constructor de `Curso` que recibe un array de `Estudiantes` ya existentes.**

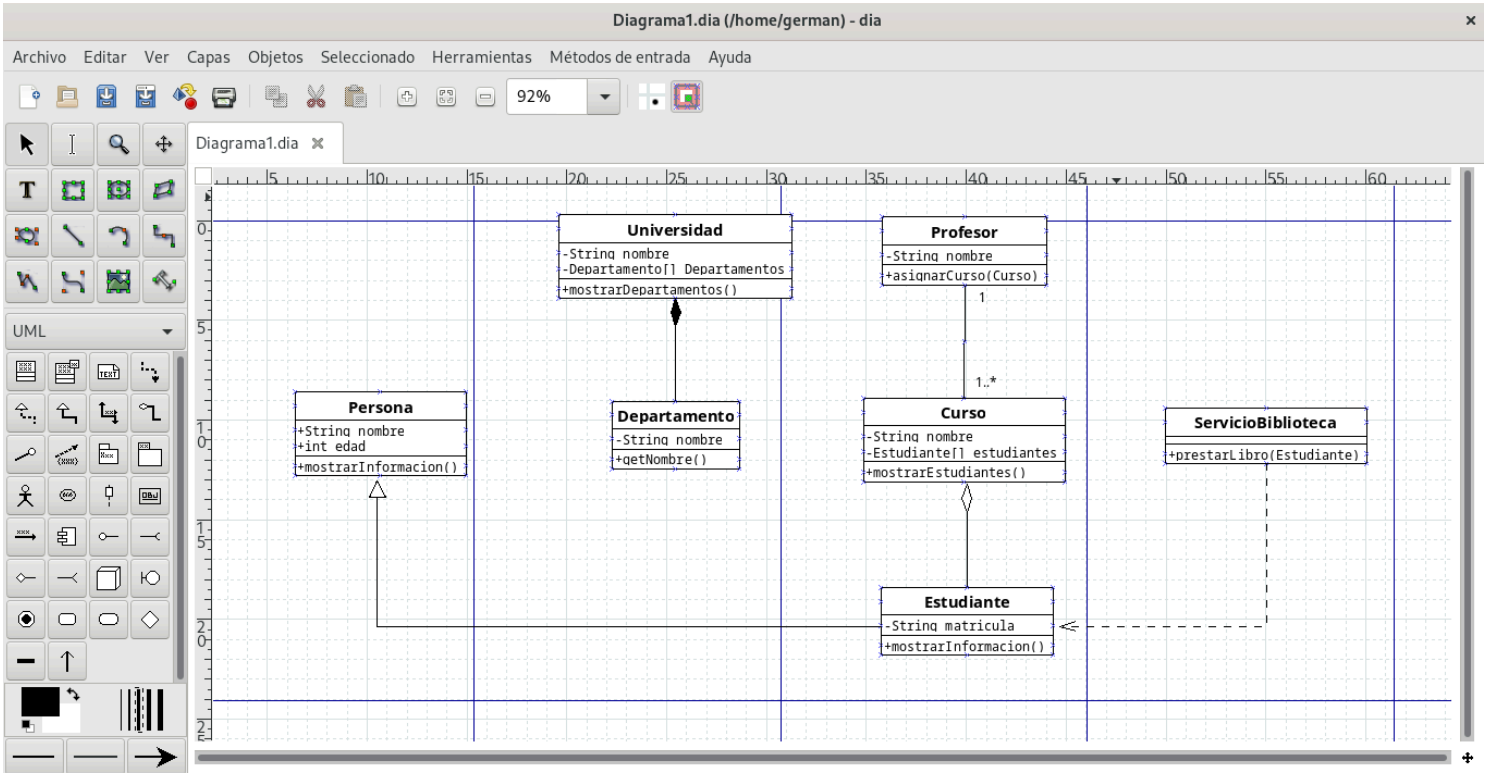
Asociación:

- **La clase `Profesor` tiene una asociación con `Curso`.**
- **Un `Profesor` puede enseñar en uno o más `Cursos`.**
- **Esto se ve en el método `asignarCurso` de la clase `Profesor`.**

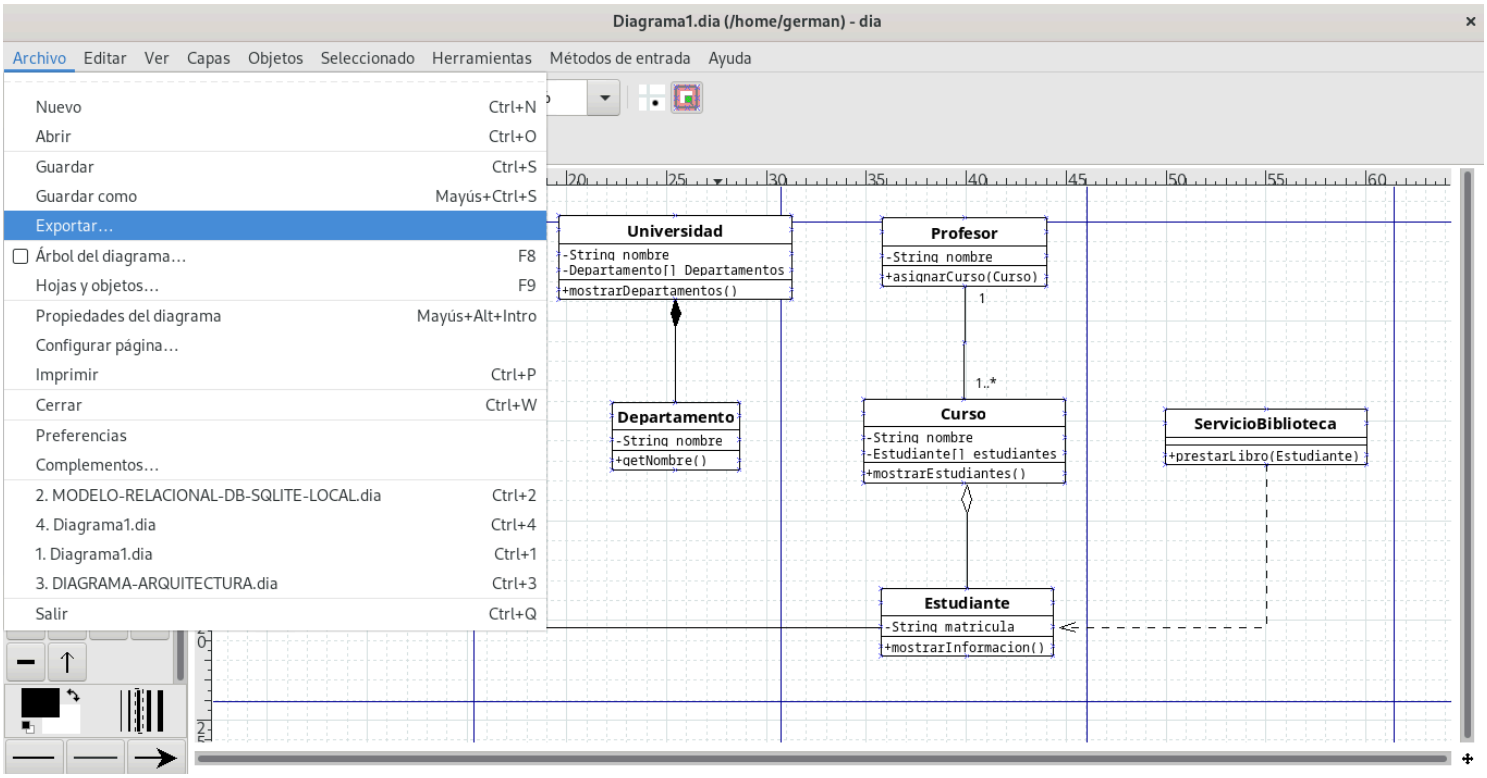
Dependencia:

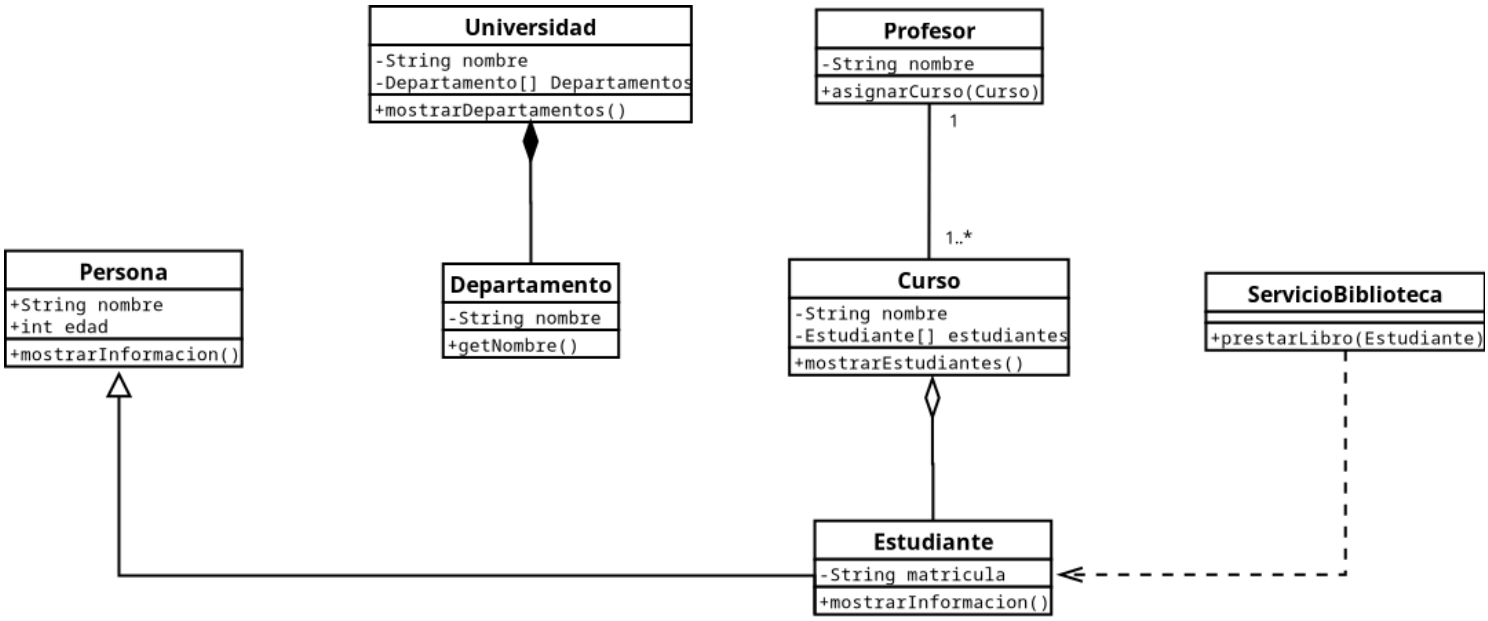
- **La clase `ServicioBiblioteca` tiene una dependencia con `Estudiante`.**
- **`ServicioBiblioteca` usa `Estudiante` en su método `prestarLibro`, pero no mantiene una referencia permanente a `Estudiante`.**
- **Esto se ve en el método `prestarLibro` que recibe un `Estudiante` como parámetro.**

Ahora que hemos identificado las relaciones, procedemos a hacer nuestro diagrama de clases, para esto usaremos el programa DIA el cual podemos descargar desde http://dia-installer.de/index.html.es
También, se puede usar otro editor de diagramas , el que sea de su preferencia.



Podemos exportar nuestro diagrama como .png





Taller:

a) **Realizar los diagramas de clases de**

1. **Sistema de Biblioteca**

```
class Persona {
    protected String nombre;

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

class Empleado extends Persona {
    private String puesto;

    public Empleado(String nombre, String puesto) {
        super(nombre);
        this.puesto = puesto;
    }

    public void mostrarInformacion() {
        System.out.println("Nombre: " + nombre + ", Puesto: " + puesto);
    }
}


class Libro {
```

```
    private String titulo;

    public Libro(String titulo) {
        this.titulo = titulo;
    }

    public String getTitulo() {
        return titulo;
    }
}

class Biblioteca {
    private String nombre;
    private Libro[] libros;

    public Biblioteca(String nombre, Libro[] libros) {
        this.nombre = nombre;
        this.libros = libros;
    }

    public void mostrarLibros() {
        System.out.println("Libros en la biblioteca " + nombre + ":");
        for (Libro libro : libros) {
            System.out.println(libro.getTitulo());
        }
    }
}


class Socio {
    private String nombre;

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

    public String getNombre() {
        return nombre;
    }
}


class ServicioBiblioteca {
    public void prestarLibro(Socio socio) {
        System.out.println("Prestando libro a: " + socio.getNombre());
    }
}


public class Main {
```

```
    public static void main(String[] args) {
        Libro[] libros = {new Libro("El Quijote"), new Libro("1984")};
        Biblioteca biblioteca = new Biblioteca("Biblioteca Central", libros);
        biblioteca.mostrarLibros();

        Empleado empleado = new Empleado("Ana", "Bibliotecaria");
        empleado.mostrarInformacion();

        Socio socio = new Socio("Luis");
        ServicioBiblioteca servicio = new ServicioBiblioteca();
        servicio.prestarLibro(socio);
    }
}
```

2. **Sistema de Gestión de Hospital**

```
class Persona {
    protected String nombre;

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

class Paciente extends Persona {
    private String enfermedad;

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

    public void mostrarInformacion() {
        System.out.println("Paciente: " + nombre + ", Enfermedad: " + enfermedad);
    }
}

class Doctor extends Persona {
    private String especialidad;

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

    public void tratarPaciente(Paciente paciente) {
        System.out.println("Doctor " + nombre + " trata a " + paciente.nombre);
    }
```

```
}


class Sala {
    private String nombre;

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

    public String getNombre() {
        return nombre;
    }
}

class Hospital {
    private String nombre;
    private Sala[] salas;

    public Hospital(String nombre, Sala[] salas) {
        this.nombre = nombre;
        this.salas = salas;
    }

    public void mostrarSalas() {
        System.out.println("Salas en el hospital " + nombre + ":");
        for (Sala sala : salas) {
            System.out.println(sala.getNombre());
        }
    }
}


public class Main {
    public static void main(String[] args) {
        Sala[] salas = {new Sala("Cirugía"), new Sala("Urgencias")};
        Hospital hospital = new Hospital("Hospital General", salas);
        hospital.mostrarSalas();

        Paciente paciente = new Paciente("Carlos", "Fiebre");
        paciente.mostrarInformacion();

        Doctor doctor = new Doctor("Dr. Pérez", "Pediatría");
        doctor.tratarPaciente(paciente);
    }
}
```

3. Sistema de Compras Online

```
class Usuario {
    protected String nombre;

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

class Cliente extends Usuario {
    public Cliente(String nombre) {
        super(nombre);
    }

    public void realizarCompra() {
        System.out.println("Cliente " + nombre + " ha realizado una compra.");
    }
}

class Administrador extends Usuario {
    public Administrador(String nombre) {
        super(nombre);
    }

    public void gestionarSistema() {
        System.out.println("Administrador " + nombre + " está gestionando el sistema.");
    }
}


class Producto {
    private String nombre;

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

    public String getNombre() {
        return nombre;
    }
}

class Carrito {
    private Producto[] productos;

    public Carrito(Producto[] productos) {
        this.productos = productos;
    }

    public void mostrarProductos() {
```

```
        System.out.println("Productos en el carrito:");
        for (Producto producto : productos) {
            System.out.println(producto.getNombre());
        }
    }
}


class Pedido {
    public void procesarPedido(Cliente cliente) {
        System.out.println("Procesando pedido para: " + cliente.nombre);
    }
}


public class Main {
    public static void main(String[] args) {
        Producto[] productos = {new Producto("Laptop"), new Producto("Teléfono")};
        Carrito carrito = new Carrito(productos);
        carrito.mostrarProductos();

        Cliente cliente = new Cliente("Juan");
        cliente.realizarCompra();

        Administrador admin = new Administrador("Laura");
        admin.gestionarSistema();

        Pedido pedido = new Pedido();
        pedido.procesarPedido(cliente);
    }
}
```

4. Sistema de Gestión de Cursos

```
class Persona {
    protected String nombre;
    protected String identificacion;

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

    public String getNombre() {
        return nombre;
    }

    public String getIdentificacion() {
        return identificacion;
```

```
    }
}

class Profesor extends Persona {
    private String especialidad;

    public Profesor(String nombre, String identificacion, String especialidad) {
        super(nombre, identificacion);
        this.especialidad = especialidad;
    }

    public String getEspecialidad() {
        return especialidad;
    }

    public void enseñar() {
        System.out.println("Profesor " + nombre + " está enseñando.");
    }
}

class Estudiante extends Persona {
    private int grado;

    public Estudiante(String nombre, String identificacion, int grado) {
        super(nombre, identificacion);
        this.grado = grado;
    }

    public int getGrado() {
        return grado;
    }

    public void estudiar() {
        System.out.println("Estudiante " + nombre + " está estudiando.");
    }
}


class Curso {
    private String nombreCurso;
    private Profesor profesor;
    private Estudiante[] estudiantes;

    public Curso(String nombreCurso, Profesor profesor, Estudiante[] estudiantes) {
        this.nombreCurso = nombreCurso;
        this.profesor = profesor;
        this.estudiantes = estudiantes;
    }

    public void mostrarInformacion() {
```

```
        System.out.println("Curso: " + nombreCurso);
        System.out.println("Profesor: " + profesor.getNombre());
        System.out.println("Estudiantes en el curso:");
        for (Estudiante estudiante : estudiantes) {
            System.out.println(estudiante.getNombre());
        }
    }

    public Profesor getProfesor() {
        return profesor;
    }
}


class Escuela {
    private String nombreEscuela;
    private Curso[] cursos;

    public Escuela(String nombreEscuela, Curso[] cursos) {
        this.nombreEscuela = nombreEscuela;
        this.cursos = cursos;
    }

    public void mostrarCursos() {
        System.out.println("Cursos en la escuela " + nombreEscuela + ":");
        for (Curso curso : cursos) {
            System.out.println(curso.getProfesor().getNombre() + " enseña " + curso.nombreCurso);
        }
    }
}


class SistemaGestion {
    public void asignarProfesor(Curso curso, Profesor profesor) {
        System.out.println("Asignando al profesor " + profesor.getNombre() + " al curso " +
curso.nombreCurso);
    }
}


public class Main {
    public static void main(String[] args) {
        // Creación de estudiantes
        Estudiante[] estudiantes = {
            new Estudiante("Carlos", "123", 10),
            new Estudiante("María", "456", 10),
            new Estudiante("Luis", "789", 11)
        };
```

```
        Profesor profesor = new Profesor("Sr. Gómez", "001", "Matemáticas");


        Curso curso = new Curso("Álgebra", profesor, estudiantes);
        curso.mostrarInformacion();


        Curso[] cursos = {curso};
        Escuela escuela = new Escuela("Escuela Nacional", cursos);
        escuela.mostrarCursos();


        SistemaGestion gestion = new SistemaGestion();
        gestion.asignarProfesor(curso, profesor);
    }
}
```

b) Diseñar los siguientes programas(diagrama de clases y casos de uso)

1.

Descripción:

Crear un sistema simple para gestionar vehículos. El sistema debe permitir el registro de diferentes tipos de vehículos (coches y motocicletas), sus propietarios y el historial de mantenimiento.

Requisitos:

- **Una clase Vehiculo que tenga atributos como marca, modelo y año.**
- **Clases que hereden de Vehiculo, como Coche y Motocicleta.**
- **Una clase Propietario que esté asociada a un Vehiculo.**
- **Un historial de mantenimiento que registre los servicios realizados al vehículo.**

2. Descripción:

Diseñar un sistema para la gestión de reservas de habitaciones de hotel. El sistema debe manejar la información de los clientes, las habitaciones disponibles y las reservas realizadas.

Requisitos:

- **Clase Habitacion que contenga atributos como número, tipo de habitación y disponibilidad.**
- **Clase Cliente que maneje la información del cliente.**
- **Clase Reserva que registre la fecha de la reserva, la habitación asignada y el cliente asociado.**
- **Clase Hotel que contenga las habitaciones disponibles y gestione las reservas.**

3. Descripción:

Crea un sistema de gestión de universidad que maneje estudiantes, profesores, cursos, y matrículas.

Requisitos:

- **Clase Persona con atributos comunes a Estudiante y Profesor.**
- **Clases Estudiante y Profesor que heredan de Persona.**

- **Clase Curso que tenga un profesor asignado y una lista de estudiantes matriculados.**
- **Clase Matricula que asocie un estudiante con un curso y almacene el estado de la matrícula.**
- **Clase Universidad que gestione los cursos, profesores, y estudiantes.**