UNAH
UNIVERSIDAD NACIONAL
AUTÓNOMA DE HONDURAS

UNIVERSIDAD NACIONAL AUTÓNOMA DE HONDURAS

FACULTAD DE CIENCIAS ECONÓMICAS, ADMINISTRATIVAS Y CONTABLES

DEPARTAMENTO DE INFORMÁTICA ADMINISTRATIVA

ASIGNATURA:
Programación Intermedia

CATEDRÁTICO:
MARVIN JOSUE AGUILAR ROMERO

ACTIVIDAD:

Guía de ejercicios UML y POO

GRUPO

Erinson Josué Alvarez Garcia 20211023800

Cristina Lisbeth Cruz Aguilera 20231031044

Kevin Alejandro Laínez Salinas 20231031586

Jonatan Isai Varela Giron 20231000159

Dany Josue Noguera Tinoco 20212020372

FECHA:

13/11/2025

DESARROLLO DE EJERCICIOS

SECCIÓN 1 EJERCICIOS DE POO EN JAVA

EJERCICIO #1-A

Enunciado: A usted le toca colaborar con el desarrollo de una aplicación para el control de gastos e ingresos personales ¿Cuáles son las clases que el sistema requerirá? (haga una tabla donde la primera columna contenga el nombre de la clase y la segunda la descripción de esta).

Análisis de requerimientos

- **Objetivo**: Identificar y diseñar clases POO para un sistema de control de gastos e ingresos en LPS, con menú funcional y validaciones.
- **Entradas**:
  - Nombre de usuario (solo letras y espacios).
  - Monto numérico positivo.
  - Descripción limitada (1–50 caracteres).
  - Categoría seleccionada de lista predefinida.
  - Opciones de menú (1–6).
- **Salidas**:
  - Menú claro con opciones numeradas.
  - Balance calculado y formateado.
  - Historial y filtrado con detalles.
  - Mensajes de error amigables con ejemplos.
  - Sello y cierre elegante.
- **Procesos**:
  - Validación estricta en bucles (nombre, monto, descripción, categoría).
  - Bloqueo de funciones sin transacciones.
  - Registro automático de fecha.
  - Cálculo de balance (ingresos - gastos).
- **Restricciones**:
  - Uso de POO: encapsulamiento, abstracción, composición.
  - Consola, Java, tildes correctas.
  - Máximo 6 clases eficientes.
  - Preparado para extensión (polimorfismo en transacciones)

Especificaciones

Clase	Atributos / Métodos / Relaciones
`Main`	Atributos: scanner (Scanner), usuario (Usuario), gestor (GestorFinanzas), CATEGORIAS_VALIDAS (String[]). Métodos: main(), mostrarMenu(), ejecutarOpcion(), registrarIngreso(), registrarGasto(), calcularBalance(), mostrarHistorial(), filtrarPorTipo(), validadores (ingresarNombreValido(), ingresarDescripciónValida(), elegirCategoría(), etc.). Relaciones: Instancia Usuario, GestorFinanzas y Utilidades; coordina todas las operaciones del menú.
`Transacción`	Atributos: monto (double), tipo (String: "INGRESO"/"GASTO"), descripción (String), categoría (Categoría), fecha (String: fecha actual). Métodos: Constructor parametrizado, toString(), getMonto(), getTipo(), getCategoría(). Relaciones: Pertenece a Usuario (composición 1:*); usada por GestorFinanzas para cálculos y reportes.
`Categoría`	Atributos: nombre (String). Métodos: Constructor, toString(), getNombre(). Relaciones: Asociada a Transacción (1:*); seleccionada desde lista predefinida en Main.
`Usuario`	Atributos: nombre (String), transacciones (ArrayList<Transacción>). Métodos: Constructor, agregarTransacción(), getTransacciones(), getNombre(). Relaciones: Contiene múltiples Transacción (composición); gestionado por GestorFinanzas.
`GestorFinanzas`	Atributos: Ninguno (lógica pura). Métodos: registrarTransacción(Usuario, Transacción), calcularBalance(Usuario), mostrarHistorial(Usuario), filtrarPorTipo(Usuario, String). Relaciones: Administra transacciones de Usuario; usa Transacción para cálculos y filtros.
`Utilidades`	Atributos: ANCHO (int), H (char). Métodos: mostrarSello(), mostrarDespedida(), centrar(String) (privado). Relaciones: Llamada desde Main para salida visual; independiente del modelo de datos.

EJERCICIO #1-B

Enunciado: Su trabajo con el desarrollo de la APP fue notable por ello desean contratarle para el desarrollo de un sistema para generar planes de estudio. Una institución posee carreras, cada carrera posee un plan de estudios y cada plan contiene clases. ¿Cuáles son las clases que el sistema requerirá? (haga una tabla donde la primera columna contenga el nombre de la clase y la segunda la descripción de esta).

Análisis de requerimientos

- **Objetivo**: Desarrollar un sistema que permita *crear y administrar planes de estudio para distintas carreras*, incluyendo la gestión de clases pertenecientes a cada plan.
- **Entradas**:
    - Datos de la institución → Nombre, dirección, código.
    - Datos de la carrera → Nombre, código, duración, modalidad.
    - Datos del plan de estudio → Año, versión, carrera asociada.
    - Datos de cada clase → Código, nombre, créditos/UV, horas, requisitos.
- **Salidas**:
    - Lista de carreras de la institución → Visualización completa de carreras registradas.
    - Plan de estudios de una carrera → Estructura jerárquica del plan con todas las asignaturas.
    - Detalle de una clase → Información específica de una asignatura.
    - Reporte del plan de estudio → Versión imprimible o exportable del plan.
- **Procesos**:
    - Registrar institución → Crear una nueva institución en el sistema.
    - Registrar carrera → Asociar una carrera a una institución.
    - Crear un plan de estudios → Vincular un plan a una carrera.
    - Agregar clases a un plan → Insertar clases en el plan correspondiente.
    - Asignar requisitos entre clases → Establecer dependencias entre asignaturas (si aplica).
    - Consultar carreras / planes / clases → Generar reportes o listados del contenido del plan de estudio.
- **Restricciones**:
    - Una carrera solo puede tener un plan de estudio por versión/año. Evita duplicidad de planes.
    - Una clase debe tener un código único dentro del plan. Evita confusiones en los registros.

- No se puede asignar una clase como requisito de sí misma. Impide relaciones circulares.
- El sistema debe mantener integridad relacional (Institución → Carrera → Plan → Clases). Garantiza orden lógico y trazabilidad.

Especificaciones

Clase	Descripción
`Institución`	Representa a la institución educativa. Contiene el nombre, ubicación, código y la lista de carreras que administra.
`Carrera`	Representa una carrera ofrecida por la institución. Incluye código, nombre, duración y el plan de estudios asociado.
`PlanEstudio`	Representa el plan de estudios de una carrera. Contiene el año/versión del plan y una lista de clases.
`Asignatura`	Representa una clase dentro del plan. Incluye código, nombre, número de unidades valorativas (UV) o créditos, horas teóricas, horas prácticas, requisitos.
`Requisito`	Representa la relación de prerrequisitos entre clases (una clase puede depender de otra).

EJERCICIO #1-C

Enunciado:

Un familiar está estudiando informática y desea que le ayude con una tarea de lenguajes de programación, ésta persona necesita que usted le ayude identificando las clases que se necesitan para un sistema de matrícula de una universidad. ¿Cuáles son las clases que el sistema requerirá? (haga una tabla donde la primera columna contenga el nombre de la clase y la segunda la descripción de esta)

Análisis de requerimientos:

Objetivo: Diseñar y organizar clases POO para un sistema de matrícula universitaria en Java, asegurando un manejo eficiente de estudiantes, cursos y matrículas.

Entradas:

• Datos de estudiantes: nombre, ID, carrera.

• Datos de profesores: nombre, ID, asignaturas que imparten.

• Datos de cursos: código, nombre, créditos.

• Matrículas: asociación de estudiante a curso, fecha de inscripción.

Salidas:

• Listado de estudiantes, cursos y matrículas.

• Reportes de inscripciones por estudiante o por curso.

• Mensajes de error claros al intentar registrar cursos o estudiantes duplicados.

Procesos:

• Validación de datos de estudiantes, profesores y cursos.

• Registro y gestión de matrículas.

• Cálculo de total de estudiantes por curso.

• Gestión de relaciones entre estudiantes, cursos y profesores.

Restricciones:

• Uso de POO: encapsulamiento, herencia, composición y polimorfismo cuando sea necesario.

• Implementación en Java.

• Diseño escalable para agregar nuevas funcionalidades o tipos de usuarios.

Especificaciones del enunciado:

Clase	Atributos / Métodos / Relaciones
Main	**Atributos:** scanner (Scanner), universidad (Universidad). **Métodos:** main(), mostrarMenu(), ejecutarOpcion(), registrarEstudiante(), registrarProfesor(), registrarCurso(), matricularEstudiante(), mostrarReportes(). **Relaciones:** instancia Universidad; coordina todas las operaciones del sistema.
Estudiante	**Atributos:** nombre (String), id (String), carrera (String), matriculas (ArrayList). **Métodos:** Constructor, agregarMatricula(), getMatriculas(), getNombre(), getID(). **Relaciones:** Composición con Matricula (1:*).
Profesor	**Atributos:** nombre (String), id (String), cursos (ArrayList). **Métodos:** Constructor, agregarCurso(), getCursos(), getNombre(), getID(). **Relaciones:** Composición con Curso (1:*).
Curso	**Atributos:** codigo (String), nombre (String), creditos (int), profesor (Profesor), matriculas (ArrayList). **Métodos:** Constructor, agregarMatricula(), getMatriculas(), getProfesor(). **Relaciones:** Composición con Matricula (1:*); asociación con Profesor (1:1).
Matricula	**Atributos:** estudiante (Estudiante), curso (Curso), fecha (String), calificacion (double). **Métodos:** Constructor parametrizado, getEstudiante(), getCurso(), getFecha(), getCalificacion(). **Relaciones:** Asociación con Estudiante y Curso (muchos a uno).

Universidad	**Atributos:** estudiantes (ArrayList), profesores (ArrayList), cursos (ArrayList). **Métodos:** agregarEstudiante(), agregarProfesor(), agregarCurso(), listarEstudiantes(), listarCursos(), listarMatriculas(). **Relaciones:** Contiene listas de Estudiantes, Profesores y Cursos; gestiona todas las operaciones del sistema.

EJERCICIO #1-D

Enunciado

Considerar el sistema anterior y migrarlo para un colegio. ¿Cuáles son las clases que el sistema requerirá? (haga una tabla donde la primera columna contenga el nombre de la clase y la segunda la descripción de esta)

Análisis de Requerimientos

El sistema debe gestionar las inscripciones de Estudiantes a Asignaturas dentro de un Colegio. La estructura de POO se mantiene, pero la lógica se ajusta al ambiente escolar.

CLASE	ADAPTACIÓN
Estudiante	Asociado a un Grado o Nivel académico (ej. 7º Grado).
Profesor	Responsable de impartir una o varias asignaturas.
Curso	Cambia a Asignatura (ej. Matemáticas, Historia).
Matrícula	Cambia a Inscripción y almacena el Periodo o notas parciales.
Controlador	Cambia de Universidad a Colegio.

Especificaciones de Cada Enunciado (Clases del Colegio)

Estas son las clases que requiere el sistema de colegio, reemplazando los términos universitarios y definiendo su rol en el nuevo contexto:

CLASE	DESCRIPCIÓN
Main	Clase de inicio. Contiene el método main() y maneja la interfaz de usuario, coordinando las operaciones con la clase Colegio.
Colegio	La clase controladora central (Fascade). Contiene las colecciones (ArrayList) de todos los Estudiantes, Profesores, Asignaturas y Grados. Gestiona la lógica de negocio y las listas maestras.
Estudiante	Representa al alumno. Almacena nombre, id, el grado actual, y mantiene una colección de sus Inscripciones.

Profesor	Representa al maestro. Almacena nombre, id, y el listado de Asignaturas que tiene asignadas en el colegio.
Asignatura	Representa una materia escolar. Contiene código, nombre, el Profesor asignado, y la lista de Inscripciones asociadas a ella.
Inscripcion	La clase de asociación (antes Matrícula). Conecta un Estudiante con una Asignatura en un Periodo de tiempo, registrando el estado del alumno.
Grado	(Recomendada) Modela el nivel académico (ej. 7º Grado). Almacena el nombre y una lista de las asignaturas obligatorias para ese nivel.

EJERCICIO #1-E

Enunciado:

¿Cuáles son las clases que un sistema para venta de artículos anime en línea requerirá? (haga una tabla donde la primera columna contenga el nombre de la clase y la segunda la descripción de esta)

Análisis de requerimientos

Objetivo: Desarrollar un sistema que permita la venta en línea de artículos anime, gestionando productos, usuarios, compras y logística.

Entradas:

- Datos de usuario → Nombre, email, contraseña, dirección.
- Datos de producto → Nombre, precio, stock, categoría anime, imagen.
- Datos de pedido → Productos seleccionados, total, método de pago/envío.

Salidas:

- Catálogo de productos → Lista filtrable por serie, tipo, precio.
- Detalle de producto → Información completa con reseñas.
- Carrito de compras → Resumen de artículos a comprar.
- Confirmación de pedido → Recibo digital con tracking.

Procesos:

- Registrar usuario → Crear cuenta.
- Agregar al carrito → Seleccionar productos.
- Procesar pago → Validar transacción.
- Generar envío → Calcular costo y rastreo.
- Dejar reseña → Calificar producto tras entrega.

Restricciones:

- Un producto no puede venderse si está sin stock.
- Un usuario solo puede tener un carrito activo.
- No se permiten pagos sin validar dirección de envío.
- El sistema debe mantener integridad: Usuario → Pedido → Pago/Envío.

Especificaciones:

Nombre de la Clase	Descripción
Usuario	Cliente o administrador con datos personales, autenticación y dirección de envío.
Producto	Artículo anime (figura, manga, póster) con nombre, precio, stock, imagen y categoría.
Carrito	Colección temporal de productos seleccionados por un usuario antes de la compra.
CarritoItem	Ítem individual en el carrito que asocia un producto con su cantidad y subtotal.
Pedido	Orden de compra confirmada con fecha, estado, total y lista de artículos.
Pago	Transacción financiera asociada a un pedido (método, monto, estado).
Envío	Gestión logística: dirección, costo, método y número de rastreo.
Categoría	Clasificación de productos (ej. "Shonen", "Figuras", "One Piece").
Reseña	Opinión del usuario sobre un producto (calificación 1-5, comentario, fecha).
Administrador	Usuario con permisos extendidos para gestionar productos, pedidos y reportes.

EJERCICIO #1-F

Enunciado: ¿Cuáles son las clases que un sistema para venta de repuestos para vehículos terrestres requerirá? (haga una tabla donde la primera columna contenga el nombre de la clase y la segunda la descripción de esta)

Análisis de requerimientos

Objetivo Diseñar un sistema de venta de repuestos para vehículos terrestres usando POO, mostrando **solo las clases del dominio** en la tabla solicitada, con menú interactivo y registro de compras.

Entradas

- Nombre del cliente (solo letras, espacios y tildes; ≥2 caracteres).
- Saldo inicial (≥0 LPS).
- Selección de tipo de vehículo (1-3).
- Selección de subtipo (1-N).
- Selección de categoría (1-4).
- Selección de producto (1-N).
- Confirmación de compra (s / sí).

Salidas

- **Sello personalizado** al inicio.
- **Tabla de clases del dominio** (solo modelo POO real).
- Menú principal con 5 opciones (incluye **Ver historial**).
- Submenús anidados (tipo → subtipo → categoría → producto).
- Historial formateado con total gastado:

```
┌──────────────────────────────────────────────────┐
│                HISTORIAL DE COMPRAS              │
├──────────────────────────────────────────────────┤
│ - Aceite sintético (L. 350)                      │
│ - Limpiador de motor (L. 250)                    │
│ Total gastado: L. 600                            │
└──────────────────────────────────────────────────┘
```

- Mensajes de error amigables + ejemplos.

Procesos

- Validación estricta en bucles hasta entrada correcta.
- Compra solo si saldo suficiente y confirmación.
- Registro automático en Historial.
- Cálculo del total gastado al mostrar historial.
- Menú cíclico hasta opción **5. Salir**.

Restricciones

- **6 clases del dominio** (no incluir Main, Inventario, Utilidades).
- Encapsulamiento (saldo privado).
- Composición (Historial contiene Compra).
- Sobrecarga de constructores (Producto).
- Consola, tildes correctas, sin errores de compilación.

Especificaciones

Clase	Atributos	Métodos principales	Relaciones
`Cliente`	`nombre` (String), `saldo` (double)	`restarSaldo(double)`, `getSaldo()`, `getNombre()`	**1 : * Compra** (un cliente puede tener varias compras)
`Vehículo`	`tipo` (String), `subTipos` (String[])	`getSubTipos()`	**Usado** en la selección de menú (no persiste)
`Producto`	`nombre` (String), `precio` (double), `categoría` (Categoría)	Constructor (sobrecargado)	**1 : 1 Categoría**, **1 : * Compra**
`Categoría`	`nombre` (String)	Constructor	**1 : * Producto**
`Compra`	`cliente` (Cliente), `producto` (Producto), `costo` (double)	Constructor	**1 : 1 Cliente**, **1 : 1 Producto**
`Historial`	`compras` (ArrayList<Compra>)	`agregarCompra(Compra)`, `mostrarHistorial(Cliente)`	**1 : * Compra** (composición)

EJERCICIO #1-F

Enunciado: ¿Cuáles son las clases que un sistema para venta de repuestos para vehículos terrestres requerirá? (haga una tabla donde la primera columna contenga el nombre de la clase y la segunda la descripción de esta)

Análisis de requerimientos

- **Objetivo** Diseñar un sistema de venta de repuestos para vehículos terrestres usando POO, mostrando **solo las clases del dominio** en la tabla solicitada, con menú interactivo y registro de compras.
- **Entradas**
  - Nombre del cliente (solo letras, espacios y tildes; ≥2 caracteres).
  - Saldo inicial (≥0 LPS).
  - Selección de tipo de vehículo (1-3).
  - Selección de subtipo (1-N).
  - Selección de categoría (1-4).
  - Selección de producto (1-N).
  - Confirmación de compra (s / sí).
- **Salidas**
  - **Sello personalizado** al inicio.
  - **Tabla de clases del dominio** (solo modelo POO real).
  - Menú principal con 5 opciones (incluye **Ver historial**).
  - Submenús anidados (tipo → subtipo → categoría → producto).
  - Historial formateado con total gastado:

```
┌──────────────────────────────────────────────────┐
│          HISTORIAL DE COMPRAS                    │
├──────────────────────────────────────────────────┤
│ - Aceite sintético (L. 350)                      │
│ - Limpiador de motor (L. 250)                    │
│ Total gastado: L. 600                            │
└──────────────────────────────────────────────────┘
```

- Mensajes de error amigables + ejemplos.

Procesos

- Validación estricta en bucles hasta entrada correcta.
- Compra solo si saldo suficiente y confirmación.
- Registro automático en Historial.
- Cálculo del total gastado al mostrar historial.
- Menú cíclico hasta opción **5. Salir**.

Restricciones

- **6 clases del dominio** (no incluir Main, Inventario, Utilidades).
- Encapsulamiento (saldo privado).
- Composición (Historial contiene Compra).
- Sobrecarga de constructores (Producto).
- Consola, tildes correctas, sin errores de compilación.

Especificaciones de cada enunciado

Clase	Atributos	Métodos principales	Relaciones
`Cliente`	`nombre` (String), `saldo` (double)	`restarSaldo(double)`, `getSaldo()`, `getNombre()`	**1 : * Compra** (un cliente puede tener varias compras)
`Vehículo`	`tipo` (String), `subTipos` (String[])	`getSubTipos()`	**Usado** en la selección de menú (no persiste)
`Producto`	`nombre` (String), `precio` (double), `categoría` (Categoría)	Constructor (sobrecargado)	**1 : 1 Categoría**, **1 : * Compra**
`Categoría`	`nombre` (String)	Constructor	**1 : * Producto**
`Compra`	`cliente` (Cliente), `producto` (Producto), `costo` (double)	Constructor	**1 : 1 Cliente**, **1 : 1 Producto**
`Historial`	`compras` (ArrayList<Compra>)	`agregarCompra(Compra)`, `mostrarHistorial(Cliente)`	**1 : * Compra** (composición)

EJERCICIO #1-G

Enunciado: ¿Cuáles son las clases que un juego (copia de Mario Bros para Nintendo) requerirá? (haga una tabla donde la primera columna contenga el nombre de la clase y la segunda la descripción de esta).

Análisis de requerimientos

- **Objetivo**:
  - Se requiere desarrollar un videojuego estilo plataformas 2D inspirado en Mario Bros para Nintendo.
  - El juego debe permitir controlar un personaje principal que se desplaza por niveles, recolecta objetos, derrota enemigos y llega a una meta final.
  - El sistema debe gestionar niveles, enemigos, muros/plataformas, objetos recolectables y las reglas del juego.
- **Entradas**:
  - Teclas presionadas por el usuario → Movimiento (izquierda, derecha, salto).
  - Interacción con objetos → Recolección de monedas, power-ups.
  - Colisiones → Detectar contacto con enemigos, plataformas o meta.
- **Salidas**:
  - Animación del jugador y enemigos → Movimiento en pantalla.
  - Puntaje y número de vidas → Se muestran en la interfaz del juego.
  - Transición entre niveles → Cambio de escenario al completar uno.
  - Estado del jugador → Puede estar normal, grande o invencible.
- **Procesos**:
  - Movimiento del jugador → Aplicar físicas (gravedad, velocidad, salto).
  - Detección de colisiones → Entre jugador, plataformas, enemigos y objetos.
  - Gestión de puntaje y vidas → Recolección de monedas, pérdida por daño.
  - Gestión de enemigos → Movimiento automático y comportamiento.
  - Cambio de nivel → Cuando el jugador llega a la meta.
  - Activación de power-ups → Aplicar efectos temporales o permanentes.
- **Restricciones**:
  - El jugador no puede atravesar plataformas o paredes. → Mantiene reglas físicas coherentes.
  - Si el jugador pierde todas las vidas → Game Over. → Define fin del juego.
  - No puede haber final del nivel sin una meta definida. → Impide niveles incompletos.
  - Power-ups no son acumulables simultáneamente (ej: no invencible + fuego + mega). → Evita estados incompatibles.

Especificaciones

Clase	Descripción
Juego	Controla el flujo principal del juego: iniciar, finalizar, gestionar niveles, puntaje y vidas.
Jugador (Mario)	Representa al personaje principal controlado por el usuario. Tiene movimiento, salto, estado (vivo, invencible), y estadísticas (vidas, puntaje).
Enemigo	Representa enemigos del juego (como Goombas o Koopas). Contiene posición, comportamiento y daño que inflige al jugador.
Nivel	Representa un escenario del juego. Contiene plataformas, enemigos, objetos y una meta final.
Plataforma / Bloque	Elemento fijo del escenario con el que el jugador puede interactuar (pisar, romper, rebotar).
Objeto	Representa los ítems que el jugador puede recolectar (monedas, hongos, estrellas, etc.).
PowerUp	Tipo especial de objeto que modifica el estado del jugador (crece, invencible, fuego).
Meta / Bandera	Punto de finalización del nivel. Indica que el jugador completó el nivel.
Física / MotorMovimiento	Gestiona gravedad, colisiones, y velocidad del jugador y los enemigos.

SECCIÓN 2 EJERCICIOS EN UMBRELLO

EJERCICIO #2-A

Enunciado: Estamos desarrollando un sistema para una clínica veterinaria y el análisis ha indicado que se requiere la clase “Medico” (sin tilde porque es nombre de clase) ilustre la misma con al menos cinco atributos y cinco métodos.

Análisis de requerimientos:

El sistema de clínica veterinaria requiere modelar la entidad **Médico Veterinario** como pieza fundamental del negocio.

Requerimientos Identificados:

1. Información Personal

- Identificación única del médico
- Datos de contacto para comunicación

2. Información Profesional

- Credenciales y licencias médicas
- Área de especialización veterinaria
- Estado de disponibilidad para atención

3. Funcionalidades de Atención

- Registro de consultas con pacientes
- Capacidad de diagnóstico veterinario
- Prescripción de tratamientos médicos

4. Gestión Operativa

- Control de disponibilidad del médico
- Consulta de información profesional

ESPECIFICACIONES DEL SISTEMA

Clase Principal: **Medico**

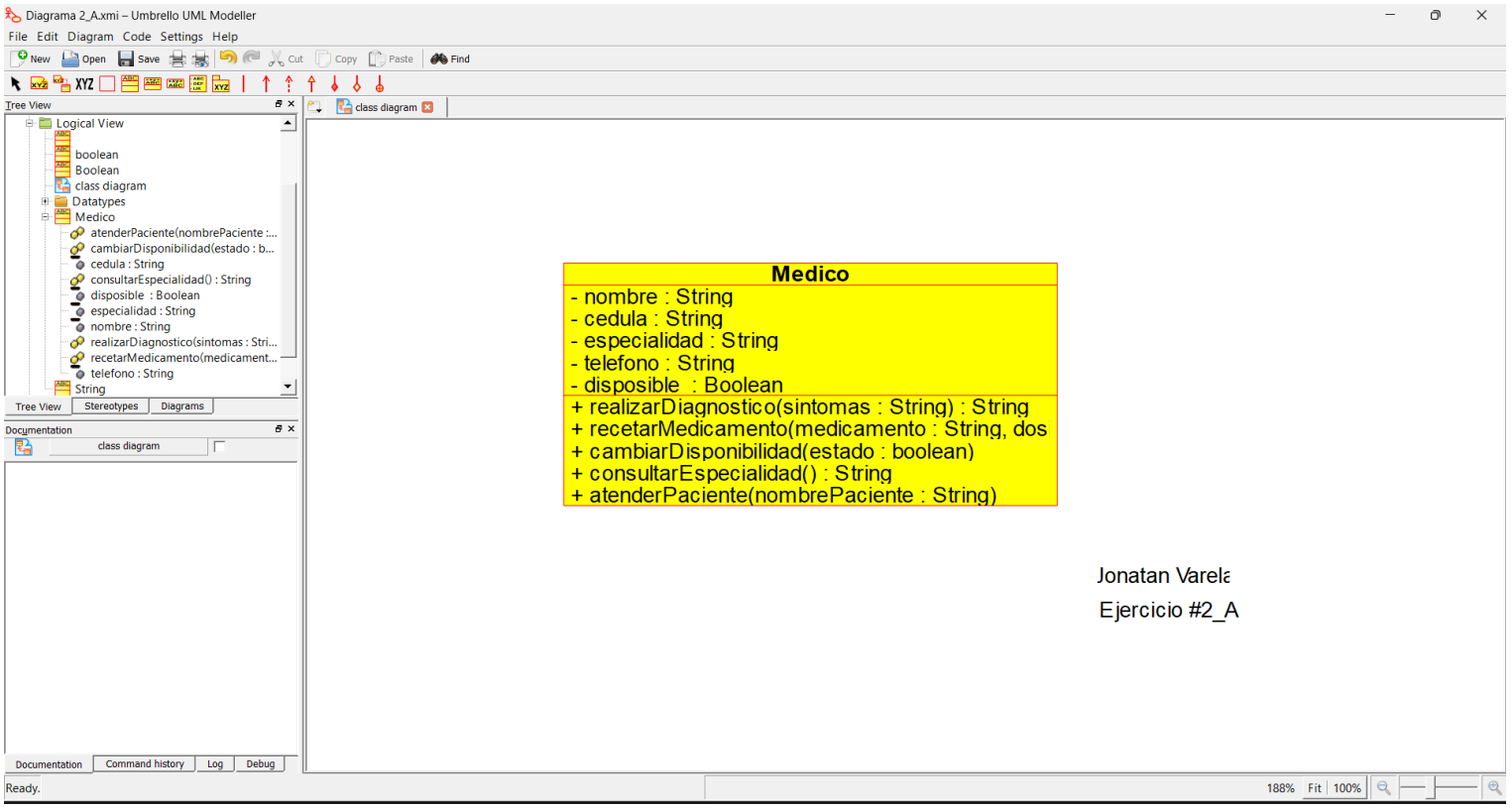
Atributos de la Clase

#	Atributo	Tipo	Visibilidad	Propósito
1	`nombre`	String	Private	Almacena el nombre completo del médico veterinario
2	`cedula`	String	Private	Número de cédula profesional que lo identifica legalmente
3	`especialidad`	String	Private	Área de especialización (cirugía, medicina general, etc.)
4	`telefono`	String	Private	Número de contacto para emergencias y coordinación
5	`disponible`	boolean	Private	Indica si el médico está disponible para atender pacientes

Métodos de la Clase

#	Método	Parámetros	Retorno	Descripción
1	`atenderPaciente()`	`nombrePaciente: String`	void	Registra la atención médica de un paciente animal
2	`realizarDiagnostico()`	`sintomas: String`	String	Analiza síntomas y genera diagnóstico veterinario
3	`recetarMedicamento()`	`medicamento: String dosis: String`	void	Prescribe medicamento con su dosificación específica
4	`cambiarDisponibilidad()`	`estado: boolean`	void	Actualiza el estado de disponibilidad del médico
5	`consultarEspecialidad()`	-	String	Retorna la especialidad del médico veterinario

Diagrama en umbrello



EJERCICIO #2-B

Enunciado:

Considerando como son las relaciones en la UNAH entre una clase y un docente realice la misma en UML denotando todos los atributos de ambas clases. (No se piden métodos)

Análisis de requerimientos:

Problema a resolver:

Se requiere representar, de forma estructurada y formal, la relación entre un docente y las clases que imparte en la UNAH, para comprender cómo se organiza la información y cómo interactúan los elementos dentro del sistema académico.

Necesidades del sistema:

1.

Identificar claramente las entidades principales: Clase y Docente.

2.

Especificar todos los atributos relevantes de cada entidad.

3.

Representar la relación entre la clase y el docente (por ejemplo, un docente puede impartir múltiples clases).

4.

Permitir que el diseño sea compatible con la implementación orientada a objetos en Java.

Actores principales:

• Docente: Persona encargada de impartir clases.

• Clase: Unidad académica que se imparte a estudiantes.

Funcionalidades principales que debe cumplir el modelo UML:

1.

Mostrar la asociación entre docentes y clases (cardinalidad y tipo de relación).

2.

Permitir visualizar los atributos de cada clase de forma clara.

3.

Facilitar la implementación futura en Java mediante clases y atributos.

Requerimientos enumerados:

1.

Definir una clase Docente con sus atributos correspondientes.

2.

Definir una clase Clase con sus atributos correspondientes.

3.

Establecer la relación entre Docente y Clase indicando cardinalidad (por ejemplo, un docente puede impartir varias clases).

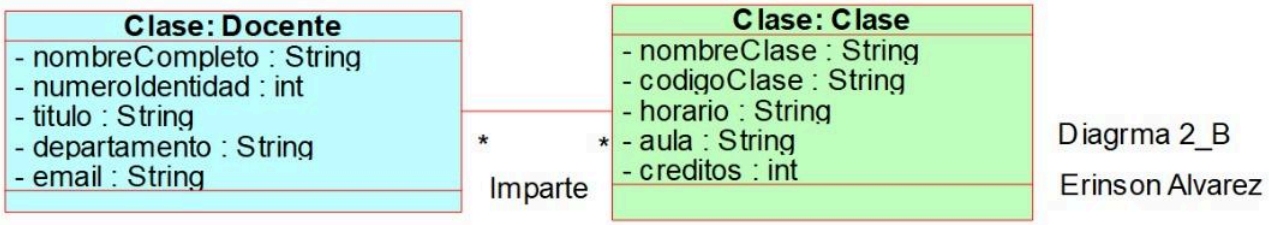
4.

No se requiere definir métodos.

Especificaciones del enunciado:

Clase	Atributos
Docente	- `nombreCompleto` : String- `numeroIdentidad` : String- `titulo` : String- `departamento` : String- `email` : String
Clase	- `nombreClase` : String- `codigoClase` : String- `horario` : String- `aula` : String- `creditos` : int

diagrama en umbrello



EJERCICIO #2-C

Enunciado: Una persona puede ser estudiante, docente, empleado de servicio, empleado administrativo o empleado de seguridad. Represente esto en UML mostrando todos los atributos de cada clase y al menos un método distintivo.

ANÁLISIS DE REQUERIMIENTOS

El sistema requiere modelar diferentes tipos de personas dentro de una institución educativa/empresarial, identificando características comunes y específicas de cada rol.

Requerimientos Identificados:

1. Clasificación de Personas

- Identificación de cinco roles distintos en la institución
- Cada rol tiene características y responsabilidades únicas
- Todos comparten atributos básicos como personas

2. Atributos Comunes

- Información personal básica compartida
- Datos de identificación y contacto
- Información demográfica

3. Especialización por Rol

- Estudiantes: enfocados en actividades académicas
- Docentes: responsables de la enseñanza
- Empleados de Servicio: mantenimiento de instalaciones
- Empleados Administrativos: gestión documental
- Empleados de Seguridad: vigilancia y protección

4. Comportamientos Distintivos

- Cada tipo de persona realiza actividades específicas de su rol
- Métodos que reflejan las responsabilidades principales

ESPECIFICACIONES DEL SISTEMA

Arquitectura: Modelo de Herencia

Patrón de diseño: Jerarquía de clases con herencia simple

Clase Base: Persona

Atributos Generales

#	Atributo	Tipo	Visibilidad	Propósito
1	`nombre`	String	Private	Nombre completo de la persona
2	`identificacion`	String	Private	Documento de identidad único (DNI, pasaporte)
3	`edad`	int	Private	Edad actual en años
4	`direccion`	String	Private	Dirección de residencia completa

Método General

#	Método	Parámetros	Retorno	Descripción
1	`mostrarInformacion()`	-	String	Retorna información básica de la persona

Clase Especializada: Estudiante

Atributos Específicos

#	Atributo	Tipo	Visibilidad	Propósito
1	`carrera`	String	Private	Programa académico que cursa
2	`matricula`	String	Private	Número de matrícula estudiantil
3	`promedio`	double	Private	Promedio académico acumulado

Método Distintivo

#	Método	Parámetros	Retorno	Descripción
1	`estudiar()`	`materiaParameter: String`	void	Registra actividad de estudio de una materia

Clase Especializada: Docente

Atributos Específicos

#	Atributo	Tipo	Visibilidad	Propósito
1	`materia`	String	Private	Materia o asignatura que imparte
2	`titulo`	String	Private	Título académico (Licenciado, Magister, Doctor)
3	`aniosExperiencia`	int	Private	Años de experiencia en docencia

Método Distintivo

#	Método	Parámetros	Retorno	Descripción
1	`impartirClase()`	`tema: String`	void	Imparte clase sobre un tema específico

Clase Especializada: EmpleadoServicio

Atributos Específicos

#	Atributo	Tipo	Visibilidad	Propósito
1	`area`	String	Private	Área o zona de trabajo asignada
2	`turno`	String	Private	Horario de trabajo (mañana, tarde, noche)
3	`equipoAsignado`	String	Private	Herramientas o equipo de trabajo

Método Distintivo

#	Método	Parámetros	Retorno	Descripción
1	`realizarMantenimiento()`	`ubicacion: String`	void	Ejecuta tareas de mantenimiento en ubicación específica

Clase Especializada: EmpleadoAdministrativo

Atributos Específicos

#	Atributo	Tipo	Visibilidad	Propósito
1	`departamento`	String	Private	Departamento administrativo de adscripción
2	`puesto`	String	Private	Cargo o posición laboral
3	`salario`	double	Private	Remuneración mensual

Método Distintivo

#	Método	Parámetros	Retorno	Descripción
1	`gestionarDocumento()`	`tipoDoc: String`	void	Procesa y gestiona documentos administrativos

Clase Especializada: EmpleadoSeguridad

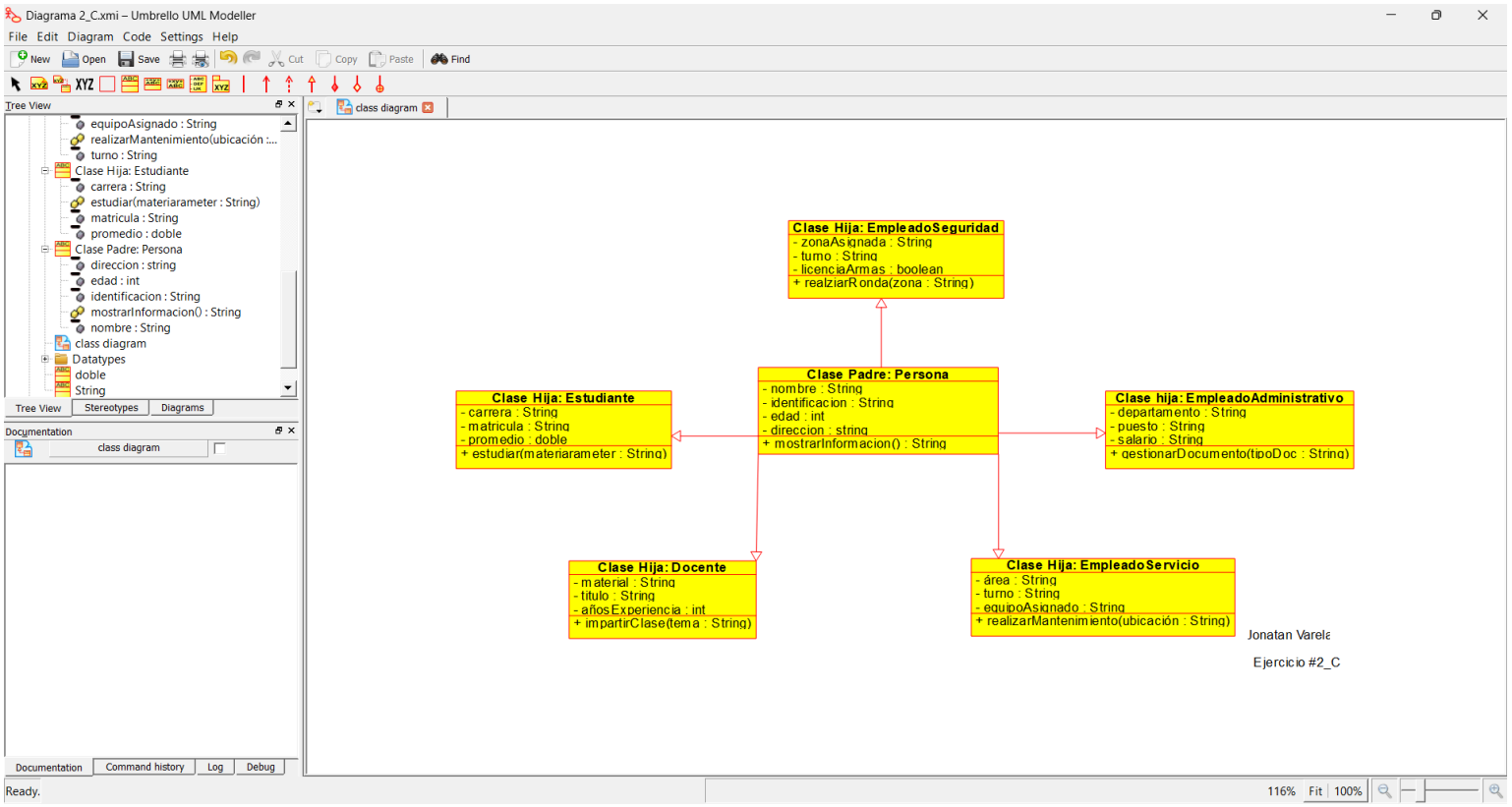
Atributos Específicos

#	Atributo	Tipo	Visibilidad	Propósito
1	`zonaAsignada`	String	Private	Zona de vigilancia asignada
2	`turno`	String	Private	Turno de guardia (diurno, nocturno)
3	`licenciaArmas`	boolean	Private	Indica si posee licencia para portar armas

Método Distintivo

#	Método	Parámetros	Retorno	Descripción
1	`realizarRonda()`	`zona: String`	void	Ejecuta ronda de seguridad en zona específica

Diagrama en Umbrello



EJERCICIO #2-D

Enunciado: Haga el UML para la clase “Cajero” que representa un ATM de cualquier banco del país. Debe incluir la mayor cantidad de métodos y atributos posibles.

Análisis de requerimientos:

Objetivo: Modelar la entidad Cajero como pieza fundamental del sistema bancario automatizado, representando un Cajero Automático (ATM) de cualquier banco del país

Entradas

- Datos de identificación del cajero → ID único, ubicación, banco propietario.
- Estado operativo → Nivel de billetes, horario, bloqueo, saldo disponible.
- Datos de transacción → Monto, tipo de operación, cuenta origen/destino, PIN.
- Datos de mantenimiento → Último servicio, intentos fallidos, capacidad máxima.

Salidas

- Confirmación de operación → Recibo impreso, mensaje en pantalla.
- Billetes dispensados → Combinación óptima según monto solicitado.
- Reporte diario → Total de retiros, depósitos, transacciones, efectivo dispensado.
- Estado del cajero → Disponible, en mantenimiento, bloqueado, bajo nivel de billetes.

Procesos

- Iniciar sesión → Leer tarjeta, validar PIN (máx 3 intentos).
- Consultar saldo → Mostrar saldo de cuenta vinculada.
- Retirar efectivo → Validar saldo, calcular billetes, dispensar, actualizar inventario.
- Depositar efectivo → Recibir billetes, acreditar cuenta, actualizar cajero.
- Transferir fondos → Validar cuentas, debitar origen, acreditar destino.
- Pagar servicios → Validar referencia, procesar pago, imprimir recibo.
- Generar reporte → Consolidar transacciones del día.
- Registrar transacción → Guardar en log interno.
- Notificar bajo nivel → Alerta automática al banco.
- Bloquear/desbloquear → Seguridad ante fallos o mantenimiento.

Restricciones

- Un cajero solo puede procesar una transacción a la vez.
- No se permite el retiro si el monto excede el saldo disponible en el cajero o en la cuenta.
- El PIN debe validarse en máximo 3 intentos; al fallar, se bloquea la tarjeta o el cajero.
- El cajero solo opera dentro del horario establecido (ej. 06:00–22:00).
- No se pueden dispensar billetes si el nivel es insuficiente para el monto solicitado.
- El sistema debe mantener integridad: Cajero → Transacción → Cuenta → Billetes.
- Los reportes diarios no pueden modificarse una vez generados

ESPECIFICACIONES DEL SISTEMA

Clase Principal: **Cajero**

Atributos de la Clase

Atributo	Tipo	Visibilidad	Descripción
`idCajero`	`String`	-	Identificador único del cajero (ej: ATM-001)
`ubicacion`	`String`	-	Dirección física del cajero
`banco`	`String`	-	Nombre del banco propietario
`estado`	`String`	-	Operativo, En Mantenimiento, Fuera de Servicio
`nivelBilletes`	`Map<Denominacion, Integer>`	-	Cantidad actual de billetes por denominación

`capacidadMaxima`	`int`	-	Máximo de billetes por ranura
`ultimoMantenimiento`	`Date`	-	Fecha del último servicio técnico
`logTransacciones`	`List<Transaccion>`	-	Historial completo de operaciones
`pinIntentosFallidos`	`int`	-	Contador de intentos fallidos de PIN
`bloqueado`	`boolean`	-	`true` = cajero bloqueado por seguridad
`horaOperativaInicio`	`Time`	-	Hora de apertura (ej: 06:00)
`horaOperativaFin`	`Time`	-	Hora de cierre (ej: 22:00)
`saldoDisponible`	`double`	-	Total en efectivo dentro del cajero

Métodos de la Clase

Método	Tipo Retorno	Visibilidad	Parámetros	Descripción

iniciarSesion(tarjeta: Tarjeta, pin: String)	boolean	+	tarjeta: Tarjeta, pin: String	Valida tarjeta y PIN
validarPin(pin: String)	boolean	+	pin: String	Verifica PIN (máx 3 intentos)
consultarSaldo(cuenta: Cuenta)	double	+	cuenta: Cuenta	Muestra saldo actual
retirarEfectivo(monto: double, cuenta: Cuenta)	boolean	+	monto: double, cuenta: Cuenta	Procesa retiro
calcularBilletes(monto: double)	Map<Denominacion, Integer>	+	monto: double	Calcula combinación óptima
dispensarBilletes(billetes: Map<Denominacion, Integer>)	boolean	+	billetes	Expulsa billetes físicos
depositarEfectivo(monto: double, cuenta: Cuenta)	boolean	+	monto, cuenta	Recibe efectivo
transferirFondos(monto: double, origen: Cuenta, destino: Cuenta)	boolean	+	monto, origen, destino	Transferencia entre cuentas
pagarServicios(servicio: String, referencia: String, monto: double)	boolean	+	servicio, referencia, monto	Pago de servicios

imprimirRecibo(transaccion: Transaccion)	void	+	transaccion	Imprime comprobante
actualizarNivelBilletes(denominacion: Denominacion, cantidad: int)	void	+	denominacion, cantidad	Actualiza inventario
verificarHorarioOperativo()	boolean	+	-	Verifica horario permitido
bloquearCajero()	void	+	-	Bloquea por seguridad
desbloquearCajero()	void	+	-	Reactiva el cajero
registrarTransaccion(transaccion: Transaccion)	void	+	transaccion	Guarda en log
generarReporteDiario()	Reporte	+	-	Genera reporte del día
notificarBajoNivelBilletes()	void	+	-	Alerta al banco
cerrarSesion()	void	+	-	Finaliza sesión
reiniciarSistema()	void	+	-	Reinicia software del cajero

Diagrama en umbrello

EJERCICIO #2-E

Enunciado

Realice el diseño de una aplicación para la gestión de pedidos. La aplicación debe permitir manejar clientes (nombre, dirección, teléfono y e-mail) que realizan pedidos de productos (se registra la cantidad en stock). Un cliente puede tener una o varias cuentas de pago. Cada cuenta está asociada a una tarjeta de crédito y tiene una cantidad disponible de dinero que el cliente puede aumentar para realizar pedidos.

Un cliente puede iniciar un pedido únicamente si al menos una de sus cuentas tiene dinero disponible. Los pedidos pueden ser simples o compuestos. Un pedido simple está asociado a una sola cuenta de pago y contiene como máximo 20 unidades (sumadas entre todas sus líneas). Un pedido compuesto contiene dos o más pedidos (simples o compuestos). Todos los pedidos simples que componen un pedido compuesto deben pertenecer al mismo cliente y usarse cuentas del mismo cliente. Solo se permiten peticiones de productos que estén en stock.

Existe una clase única (singleton) responsable del cobro, orden de distribución y confirmación de pedidos. El cobro se procesa una vez al día: se revisan todos los pedidos con estado “pendiente de cobro” y se intenta cobrar la cuenta asociada. Si la cuenta no tiene suficiente dinero, el pedido se rechaza; si el pedido rechazado forma parte de un pedido compuesto, se rechaza todo el pedido compuesto. Cuando un pedido está cobrado y listo, se ordena su distribución y, al confirmarse la entrega, el pedido pasa a estado confirmado.

Análisis de requerimientos

Requerimientos funcionales (RF)

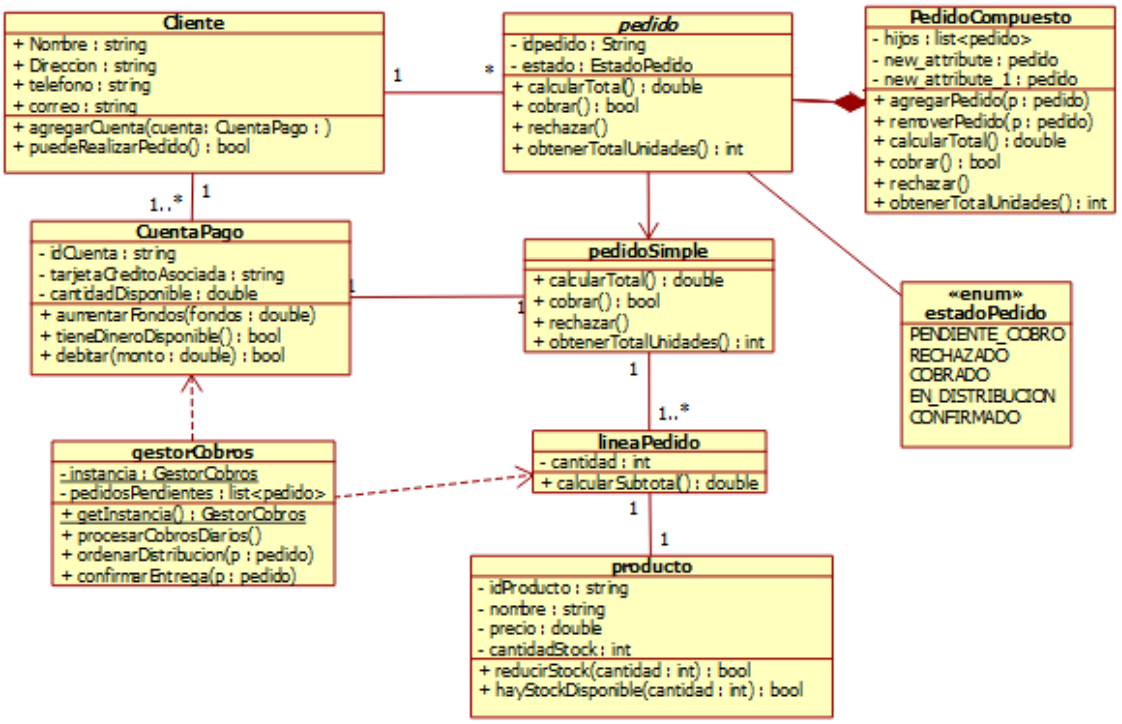
- RF1. Gestionar clientes: crear/editar/borrar clientes con nombre, dirección, teléfono, e-mail.
- RF2. Gestionar cuentas de pago por cliente: crear, asociar tarjeta, aumentar fondos, consultar saldo
- RF3. Gestionar productos: crear/editar/borrar producto con cantidadEnStock.
- RF4 — Crear pedidos: permitir pedidos simples y compuestos.
- RF5. Restricción de inicio de pedido: cliente solo puede iniciar pedido si tiene alguna cuenta con saldo disponible.
- RF6. Pedido simple: asociado a una cuenta, máxima 20 unidades (sumadas de todas las líneas).

- RF7. Pedido compuesto: contiene 2+ pedidos (simples o compuestos). Garantizar que los pedidos simples que lo componen se cobren desde cuentas del mismo cliente.
- RF8.Solo pedir productos con stock suficiente. Al aceptar una línea, reducir stock solo al confirmar cobro/creación según política.
- RF9. Proceso diario de cobros: revisar pedidos pendientes, cobrar cuentas, rechazar si saldo insuficiente (rechazo de pedidos compuestos como un todo).
- RF10. Orden de distribución y confirmación: cuando cobrado, ordenar distribución; al entregar, marcar confirmado.
- RF11. Singleton de gestión de cobros/distribución: única instancia global.

Requerimientos no funcionales (RNF)

- RNF1. Persistencia: datos deben poder exportarse/guardarse (por ejemplo base de datos / ficheros).
- RNF2.Usabilidad: interfaz clara para crear clientes/cuentas/productos/pedidos
- RNF3. Consistencia: garantizar integridad (ej. no permitir pedido por encima de stock, no permitir duplicado de cuentas con mismo id)
- RNF4. Auditabilidad: registrar operaciones de cobro y rechazos (logs).

Diagrama



EJERCICIO #2-F

Enunciado: Especificar un diagrama de clases que describa redes de ordenadores.

I. Los elementos que se pueden incluir en la red son:

1.Servidor, PC, Impresora.

2.Hub, Cable de red.

II. Los PCs pueden conectarse con un único Hub, los servidores con uno o varios.

III. Los Servidores y PCs pueden generar mensajes, con una cierta longitud.

IV. Los Hubs tienen un número de puertos, algunos de los cuales pueden usarse para conectar con otros Hubs.

V. Tienen cierta probabilidad de “perder” mensajes.

VI. Las impresoras pueden averiarse, con cierta probabilidad, durante cierto tiempo.

Análisis de Requerimientos:

El objetivo es modelar una red de ordenadores identificando las clases principales, sus atributos, comportamientos (métodos) y las relaciones entre ellas, basándonos en los requisitos.

1. Identificación de Clases y Abstracción
Para evitar duplicidad y modelar correctamente las características comunes, utilizaremos la herencia.

1.1. Clase Base (Abstracta): `ElementoDeRed`

- Todos los elementos mencionados (PC, Servidor, Impresora, Hub, Cable) son, en esencia, elementos de una red. Esta clase servirá como base abstracta.

1.2. Clases de Dispositivos:

- Identificamos que `PC` y `Servidor` comparten una característica (generar mensajes). Creamos una clase abstracta `Computadora` para agruparlos.
- `PC`: Hereda de `Computadora`.
- `Servidor`: Hereda de `Computadora`.

- `Impresora`: Es un dispositivo, pero no una computadora.
- Crearemos una clase abstracta `Dispositivo` (que hereda de `ElementoDeRed`) de la cual heredarán `Computadora` e `Impresora`.

1.3. Clases de Infraestructura:

- `Hub`: Elemento de conexión.
- `CableDeRed`: Elemento físico.

- Podemos agruparlos bajo una clase abstracta `EquipoConexion` (que hereda de `ElementoDeRed`), aunque `CableDeRed` también podría heredar directamente de `ElementoDeRed`. Para este ejercicio, `Hub` heredará de `EquipoConexion` y `CableDeRed` de `ElementoDeRed`.

1.4. Clases Adicionales:

- `Mensaje`: Se menciona explícitamente que las computadoras generan mensajes con longitud. Esto debe ser una clase propia.

2. Identificación de Atributos y Métodos

2.1. `Computadora` (Abstracta):

- Método: `generarMensaje(longitud: int)`.

2.2. `Impresora`:

- Atributos: `probabilidadAveria: float`, `tiempoAveria: int`.
- Método: `averiarse()`.

2.3. `Hub`:

- Atributos: `numeroPuertos: int`, `probabilidadPerdida: float`.
- Método: `perderMensaje()`.

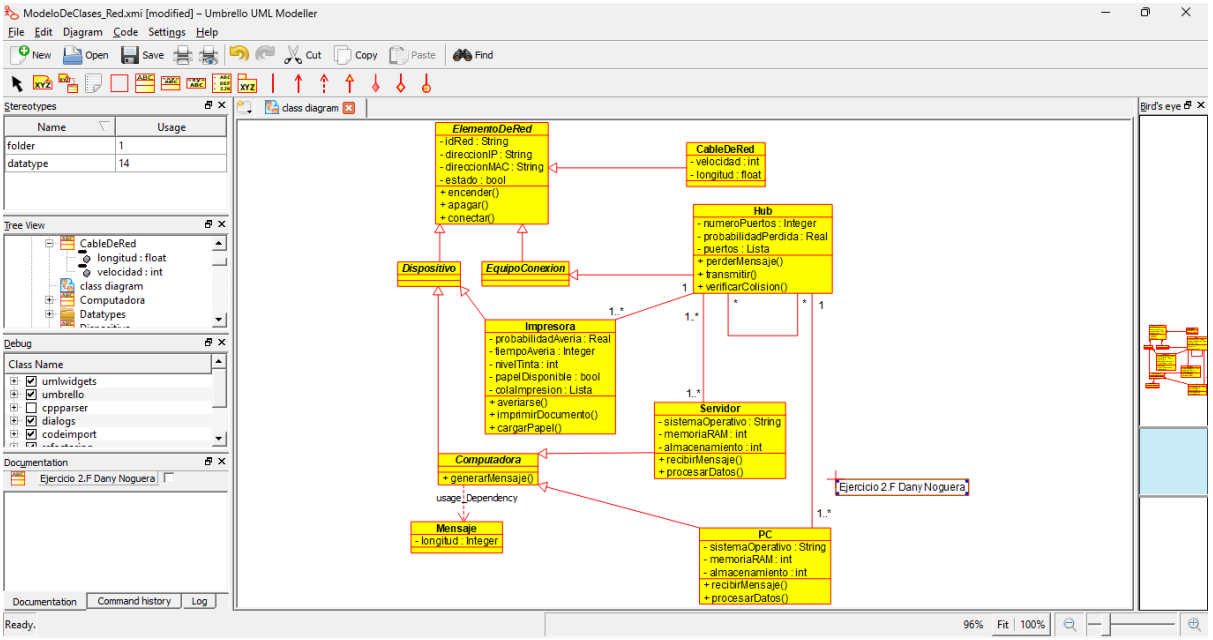
2.4. `Mensaje`:

- Atributo: `longitud: int`.

3. Identificación de Relaciones

- **Herencia.**
- **Asociación (PC <-> Hub):**
    - Un PC se conecta a un único Hub.
    - Un Hub puede tener muchos PC.
    - Multiplicidad: PC [1..*] --- [1] Hub.
- **Asociación (Servidor <-> Hub):**
    - Un `Servidor` se conecta a uno o varios Hubs.
    - Un Hub puede tener muchos `Servidores`.
    - Multiplicidad: `Servidor` [1..*]* --- *[1..*] Hub.
- **Asociación (Hub <-> Hub):**
    - Un Hub puede conectarse con otros Hubs.
    - Esta es una asociación reflexiva (o recursiva).
    - Multiplicidad: Hub [0..*]* --- *[0..*] Hub.
- **Dependencia (Computadora -> Mensaje):**
    - La clase `Computadora` (y por ende PC y `Servidor`) *utiliza* o *crea* la clase `Mensaje` a través de su método `generarMensaje`.
    - Relación: `Computadora` -- (Usage) --> `Mensaje`.
- **Asociación (Impresora <-> Hub) (Implícita):**
    - El enunciado no especifica cómo se conecta la impresora. Asumiremos el escenario más común: se conecta a la red a través de un Hub, de forma similar a un PC.
    - Multiplicidad: `Impresora` [1..*] --- [1] Hub.

Diagrama en Umbrello



SECCIÓN 3 UMBRELLO Y JAVA

EJERCICIO #3-1

Enunciado:

3.1 (copiado de discoduroderoer) Haz una clase llamada Persona que siga las siguientes condiciones:

- Sus atributos son: nombre, edad, DNI, sexo (H hombre, M mujer), peso y altura. No queremos que se accedan directamente a ellos. Piensa que modificador de acceso es el más adecuado, también su tipo. Si quieres añadir algún atributo puedes hacerlo.
- Por defecto, todos los atributos menos el DNI serán valores por defecto según su tipo (0 números, cadena vacía para String, etc.). Sexo sera hombre por defecto, usa una constante para ello.
- Se implantaran varios constructores: ○ Un constructor por defecto. ○ Un constructor con el nombre, edad y sexo, el resto por defecto. ○ Un constructor con todos los atributos como parámetro.
- Los métodos que se implementaran son:
  ○ calcularIMC(): calculara si la persona esta en su peso ideal (peso en kg/(altura^2 en m)), si esta fórmula devuelve un valor menor que 20, la función devuelve un -1, si devuelve un número entre 20 y 25 (incluidos), significa que esta por debajo de su peso ideal la función devuelve un 0 y si devuelve un valor mayor que 25 significa que tiene sobrepeso, la función devuelve un 1. Te recomiendo que uses constantes para devolver estos valores.

■ esMayorDeEdad(): indica si es mayor de edad, devuelve un booleano.
■ comprobarSexo(char sexo): comprueba que el sexo introducido es correcto. Si no es correcto, sera H. No sera visible al exterior.
■ toString(): devuelve toda la información del objeto.
■ generaDNI(): genera un número aleatorio de 8 cifras, genera a partir de este su número su letra correspondiente. Este método sera invocado cuando se construya el objeto. Puedes dividir el método para que te sea más fácil. No será visible al exterior.
■ Métodos set de cada parámetro, excepto de DNI.

Ahora, crea una clase ejecutable que haga lo siguiente:
● Pide por teclado el nombre, la edad, sexo, peso y altura.

● Crea 3 objetos de la clase anterior, el primer objeto obtendrá las anteriores variables pedidas

por teclado, el segundo objeto obtendrá todos los anteriores menos el peso y la altura y el

último por defecto, para este último utiliza los métodos set para darle a los atributos un valor.

● Para cada objeto, deberá comprobar si está en su peso ideal, tiene sobrepeso o por debajo de

su peso ideal con un mensaje.

● Indicar para cada objeto si es mayor de edad.

● Por último, mostrar la información de cada objeto.

Análisis de requerimientos

El sistema requiere modelar personas con información personal y física, implementando cálculos médicos y validaciones automáticas. Se identificaron las siguientes necesidades principales:

1. Gestión de Datos Personales

El sistema debe almacenar información básica de identificación de cada persona, incluyendo nombre completo, edad, sexo y un documento de identidad único. El DNI debe generarse automáticamente al momento de crear cada objeto, garantizando que cada persona tenga un identificador único sin intervención del usuario.

2. Registro de Datos Físicos

Se requiere capturar y almacenar medidas corporales específicas: peso en kilogramos y altura en metros. Estos datos son fundamentales para realizar cálculos médicos posteriores y deben permitir valores decimales para mayor precisión.

3. Cálculo del Índice de Masa Corporal (IMC)

El sistema debe implementar la fórmula médica estándar del IMC para determinar si una persona se encuentra en su peso ideal, por debajo o con sobrepeso. La clasificación se basa en rangos establecidos: valores menores a 20 indican peso bajo, entre 20 y 25 peso ideal, y superiores a 25 sobrepeso.

4. Verificación de Mayoría de Edad

Se necesita una funcionalidad que determine automáticamente si una persona es mayor de edad según el estándar de 18 años, retornando un valor booleano que facilite la toma de decisiones en el sistema.

5. Validación y Normalización de Datos

El sexo ingresado debe validarse automáticamente, aceptando únicamente los valores 'H' para hombre o 'M' para mujer. En caso de recibir un valor inválido, el sistema debe asignar 'H' como valor predeterminado sin generar errores.

6. Encapsulamiento y Seguridad

Todos los atributos deben estar protegidos contra acceso directo externo, permitiendo únicamente su modificación a través de métodos controlados. El DNI, una vez generado, no debe poder ser modificado ni consultado externamente, garantizando la integridad del identificador.

7. Flexibilidad en la Creación de Objetos

El sistema debe ofrecer múltiples formas de crear personas: con valores predeterminados, con información parcial (nombre, edad y sexo), o completamente vacío para llenarse posteriormente mediante métodos set. Esta flexibilidad permite adaptarse a diferentes escenarios de uso.

8. Representación Completa de Información

Se requiere un método que presente toda la información almacenada de forma legible y organizada, permitiendo visualizar el estado completo de cada objeto persona de manera clara y profesional.

Especificaciones de cada ejercicio

Arquitectura General

El sistema utiliza Programación Orientada a Objetos con una única clase Persona que encapsula datos personales y físicos, implementando cálculos médicos y validaciones automáticas.

Clase: Persona

Atributos Privados:

Los atributos de instancia incluyen nombre, edad, dni, sexo, peso y altura. Todos son privados para proteger la información. El DNI se genera automáticamente y no puede modificarse. El sexo solo acepta 'H' o 'M', validándose automáticamente.

El sistema define constantes para manejar valores especiales: SEXO_DEFECTO ('H'), PESO_BAJO (-1), PESO_IDEAL (0) y SOBREPESO (1).

Constructores:

Se implementan tres constructores para flexibilidad: uno por defecto que inicializa valores en cero y cadenas vacías, otro que recibe nombre, edad y sexo, y un tercero completo (que fue eliminado porque el DNI se genera automáticamente). Todos los constructores invocan generaDNI() para asignar un identificador único.

Métodos Principales:

El método calcularIMC() implementa la fórmula peso/altura² y clasifica el resultado en tres categorías según rangos médicos estándar. El método esMayorDeEdad() verifica si la persona tiene 18 años o más. El método toString() genera una representación formateada con toda la información del objeto.

Validación y Seguridad:

El método privado comprobarSexo() normaliza automáticamente valores inválidos a 'H'. Los métodos privados generaDNI() y generaLetrasDNI() crean identificadores únicos usando el algoritmo español oficial. El DNI no tiene getter para proteger su acceso externo.

Métodos de Acceso:

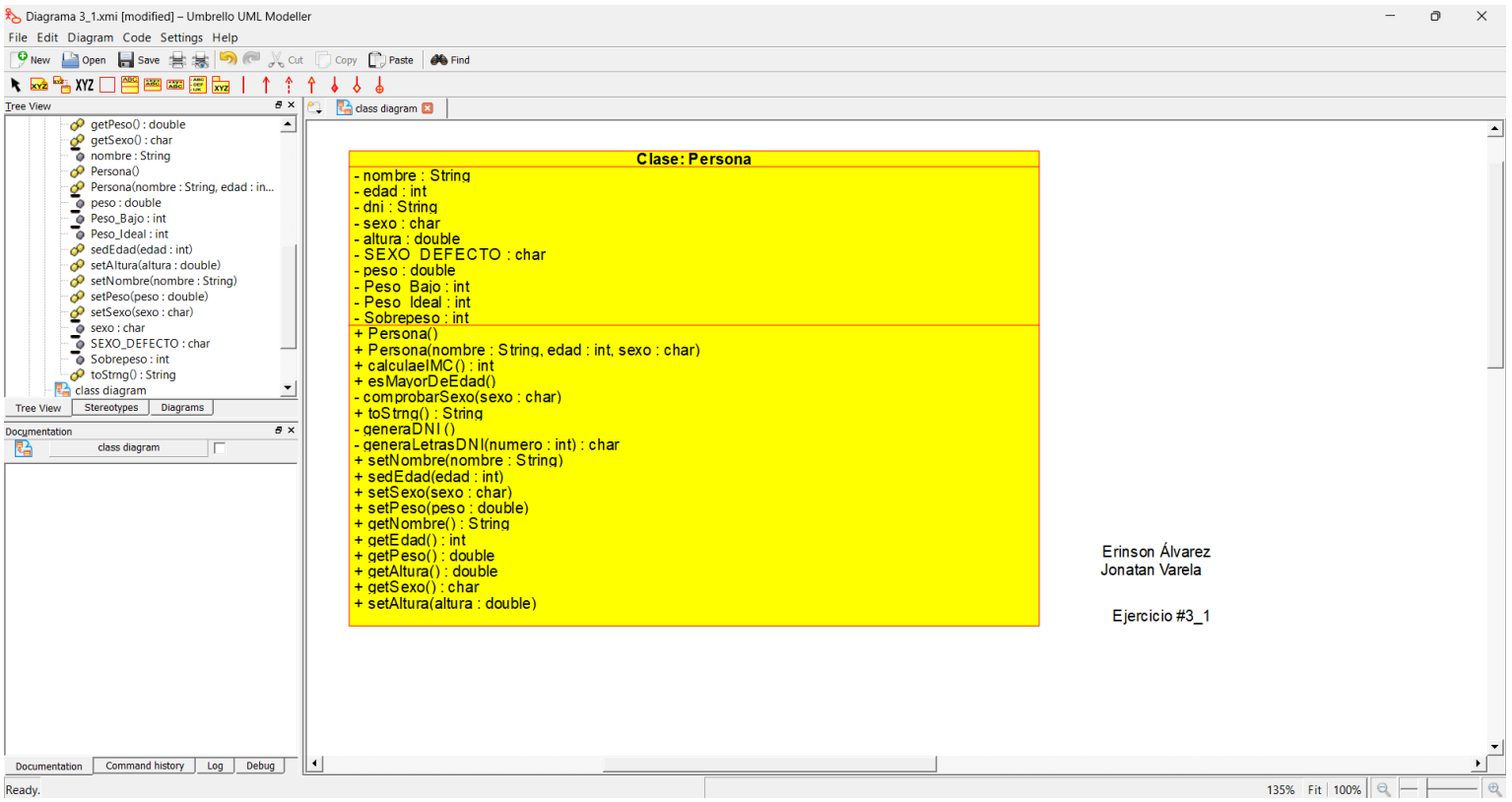
Se proporcionan setters para modificar nombre, edad, sexo, peso y altura de forma controlada. Los getters permiten consultar todos los atributos excepto el DNI, manteniendo su confidencialidad

.

Decisiones Clave de Diseño

El encapsulamiento estricto protege todos los atributos. La generación automática del DNI garantiza identificadores únicos sin intervención del usuario. La validación silenciosa del sexo prioriza la continuidad del sistema. El uso de constantes mejora la legibilidad del código. Los múltiples constructores ofrecen flexibilidad en la creación de objetos.

Diagrama De umbrello



EJERCICIO #3-2

Enunciado

Crea una clase Cuenta con los métodos ingreso, reintegro y transferencia. La clase contendrá un constructor por defecto, un constructor con parámetros, un constructor copia y los métodos getters y setters.

Análisis de Requerimientos

El objetivo es crear una clase Cuenta que represente una cuenta bancaria simple. Debe manejar un saldo y un número de cuenta (o titular) para identificarla.

Requerimientos Funcionales

- Creación de Cuentas: Debe permitir la creación de cuentas de tres maneras:
  - ☐ Constructor por defecto (valores iniciales predeterminados).
  - ☐ Constructor con parámetros (para inicializar el titular y el saldo).
  - ☐ Constructor copia (para crear una nueva cuenta idéntica a otra existente).
- Consulta/Modificación de Atributos: Debe incluir métodos getters y setters para acceder y modificar los atributos de la cuenta.
- Ingreso: Un método para añadir una cantidad al saldo.
- Reintegro (Retiro): Un método para sustraer una cantidad del saldo. Debe manejar la posibilidad de saldo insuficiente.
- Transferencia: Un método para mover una cantidad de dinero de la cuenta actual a otra cuenta.

Especificaciones

Atributo	Tipo de Dato	Descripción
titular	String	Nombre del titular de la cuenta.
saldo	double	Cantidad de dinero en la cuenta.

Métodos

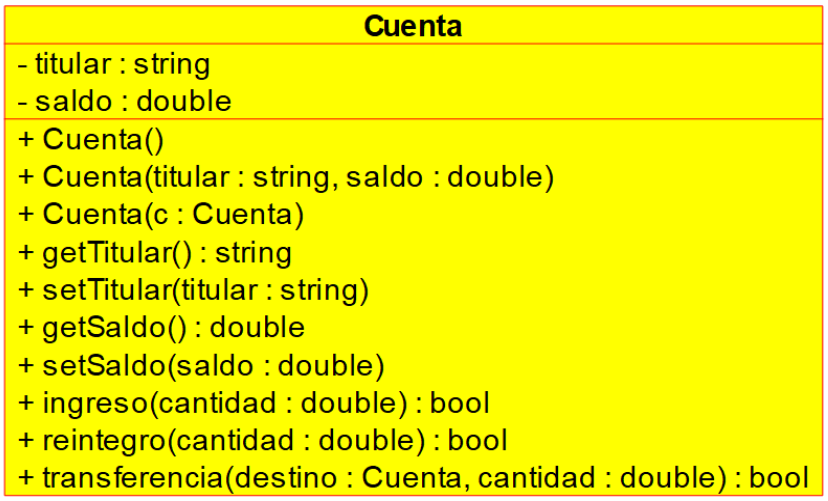
Método	Tipo	Descripción	Parámetros	Valor de Retorno
Cuenta()	Constructor	Por defecto. Inicializa titular a "" y saldo a 0.0.	Ninguno	N/A
Cuenta(String, double)	Constructor	Con parámetros.	String t, double s	N/A
Cuenta(Cuenta)	Constructor	Copia.	Cuenta c	N/A
getTitular()	Getter	Obtiene el titular.	Ninguno	String
setTitular(String)	Setter	Establece el titular.	String titular	void
getSaldo()	Getter	Obtiene el saldo.	Ninguno	double
setSaldo(double)	Setter	Establece el saldo.	double saldo	void
ingreso(double)	Funcional	Añade la cantidad al saldo si es positiva.	double cantidad	boolean

reintegro(double)	Funcional	Retira la cantidad del saldo si es positiva y hay saldo suficiente.	double cantidad	boolean
transferencia(Cuenta, double)	Funcional	Transfiere una cantidad a otra cuenta.	Cuenta destino, double cantidad	boolean

Promt GEMINI (BARD):

Prompt	**Propósito**
Crea una clase Cuenta con los métodos ingreso, reintegro y transferencia. La clase contendrá un constructor por defecto, un constructor con parámetros, un constructor copia y los métodos getters y setters.	*Generación del código base y la estructura del informe.*

Diagrama en Umbrello UML



EJERCICIO #3-3

Enunciado

Crea una clase Contador con los métodos para incrementar y decrementar el contador. La clase contendrá un constructor por defecto, un constructor con parámetros, un constructor copia y los métodos getters y setters.

Análisis de Requerimientos

El sistema requiere gestionar un valor interno (valor de tipo int).

Requerimientos de Datos: La clase necesita un atributo privado (private int valor) para almacenar el conteo.

Requerimientos Estructurales (Constructores):

- Debe existir un constructor por defecto que inicie el contador a cero.
- Debe existir un constructor que acepte un valor inicial para el contador.
- Debe existir un constructor copia que inicialice el nuevo contador con el valor de otro objeto Contador.

Requerimientos Funcionales:

- Un método incrementar() que aumente el valor en uno.
- Un método decrementar() que disminuya el valor en uno, asegurando que el contador nunca sea negativo (debe detenerse en cero).
- Métodos de acceso (getValor() y setValor()) para consultar y modificar el valor, respectivamente. El *setter* debe validar que no se asignen valores negativos.

Especificaciones

La clase principal del diseño es Contador, la cual maneja el atributo privado valor: int. El único atributo es valor, con visibilidad privada (-).

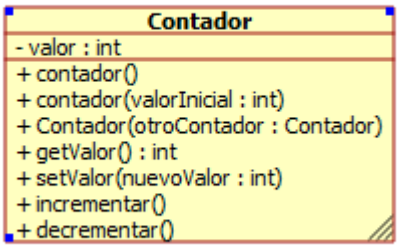
Constructores

1. Contador(): Constructor público por defecto. Inicializa valor = 0.
2. Contador(valorInicial: int): Constructor público con parámetros. Asigna el valor inicial y lo corrige a cero si es negativo.
3. Contador(otroContador: Contador): Constructor público copia.

Métodos u Operaciones

1. getValor(): Público (+), devuelve int.
2. setValor(nuevoValor: int): Público (+), devuelve void.
3. incrementar(): Público (+), devuelve void.
4. decrementar(): Público (+), devuelve void.

Diagrama en Umbrella UML



EJERCICIO #3-4

Enunciado

Crea una clase Fecha. La clase contendrá además de constructores, métodos set y get y el método toString, un método para comprobar si la fecha es correcta y otro para modificar la fecha actual por la del día siguiente.

Análisis de Requerimientos

Objetivo: Desarrollar una clase Fecha que represente una fecha válida en el calendario gregoriano, con capacidad para:

- Validar si una fecha es correcta.
- Avanzar al día siguiente.
- Gestionar acceso y modificación segura de sus componentes (día, mes, año).

Entradas

Tipo	Descripción	Ejemplo
Datos de entrada (constructor)	Día, mes y año	`31, 12, 2025`
Datos de entrada (setters)	Valor individual de día, mes o año	`setDia(15)`
Fecha por defecto	Fecha inicial en caso de error	`1/1/2000`

Salidas

Tipo	Descripción	Formato
Representación textual	Fecha formateada	`dd/mm/aaaa` → `31/12/2025`

Resultado de validación	¿Es válida la fecha?	`true` / `false`
Fecha siguiente	Día siguiente a la actual	`01/01/2026`
Acceso a componentes	Valores individuales	`getDia()` → `31`

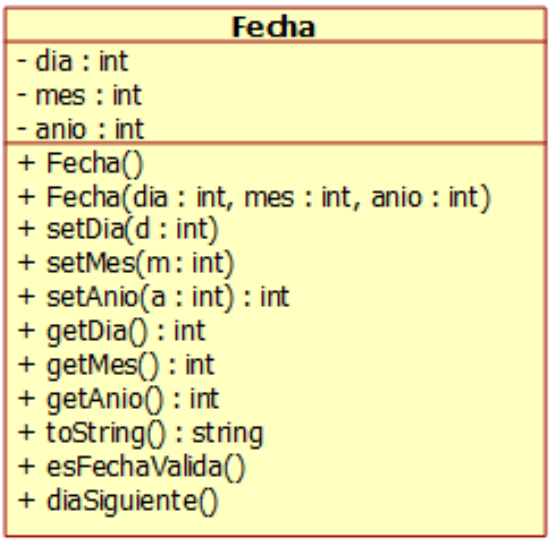
Procesos

Proceso	**Descripción**
Construir fecha	Inicializar con valores por defecto o parámetros.
Validar fecha	Comprobar que día, mes y año forman una fecha válida (incluye años bisiestos).
Avanzar al día siguiente	Incrementar día; si excede, pasar al siguiente mes; si excede, al siguiente año.
Modificar componentes	Asignar nuevo día, mes o año con validación.
Obtener componentes	Devolver día, mes o año actual.
Mostrar fecha	Formatear como cadena legible (`toString`).

Restricciones

Restricción	Descripción
Rango de día	1 ≤ día ≤ 31 (según mes y año).
Rango de mes	1 ≤ mes ≤ 12.
Rango de año	Año ≥ 1 (sin límite superior definido).
Febrero en año no bisiesto	Máximo 28 días.
Febrero en año bisiesto	Máximo 29 días.
Fecha inválida en constructor	Se asigna `1/1/2000` por defecto.
Setters inválidos	No modifican el valor si es incorrecto (con mensaje opcional).
Año bisiesto	Cumple regla: divisible por 4, pero no por 100, salvo por 400.

Diagrama en Umbrello UML


Fecha
- dia : int
- mes : int
- anio : int
+ Fecha()
+ Fecha(dia : int, mes : int, anio : int)
+ setDia(d : int)
+ setMes(m : int)
+ setAnio(a : int) : int
+ getDia() : int
+ getMes() : int
+ getAnio() : int
+ toString() : string
+ esFechaValida()
+ diaSiguiente()