UML ~ Diagrama de interacción

Definicion: Permiten visualizar y describen aspectos dinámicos del sistema para su modelado y su interacción unos con otros a lo largo del tiempo

¿Quién esté interesado? → Programadores (ob, comp), Arquitectos (comp) y interesados (comp)

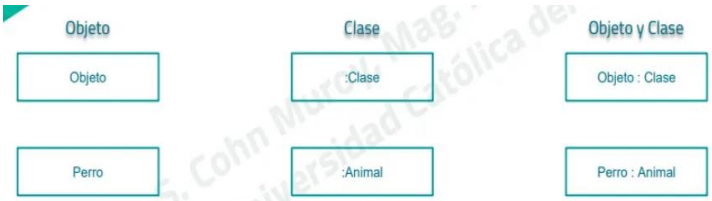
¿Que facilita? → el flujo de mensajes, acciones y eventos durante la ejecucion de un caso de uso o escenario

Diagrama de Secuencia

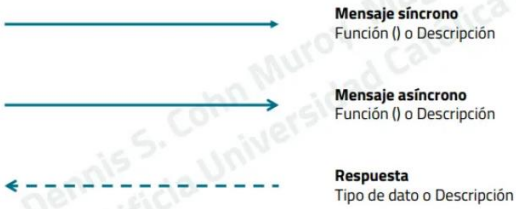
Uso → Presentar interraciones ordenadas , se leen de arriba a abajo, los primeros se ejecutaran primero

Representacion

- Objetos



- Mensajes



- Fragmentos de secuencia o interaccion

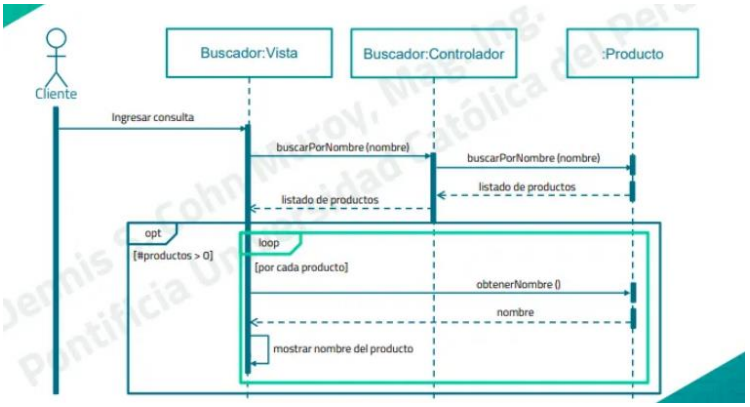
  Permiten incluir estructuras de control y estructuras de iteración sobre un diagrama de secuencia

  - Estructura de control

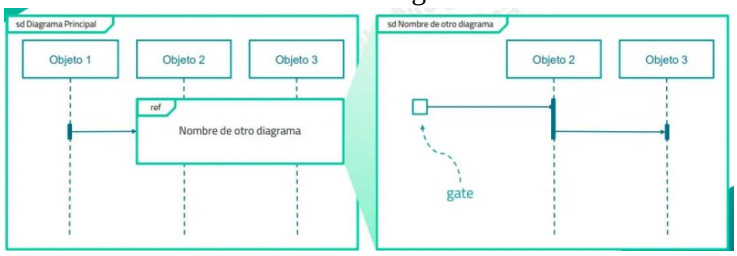
  alt → permite multiples opciones

  opt → permite solo una condicion

  - Estructura iterativas



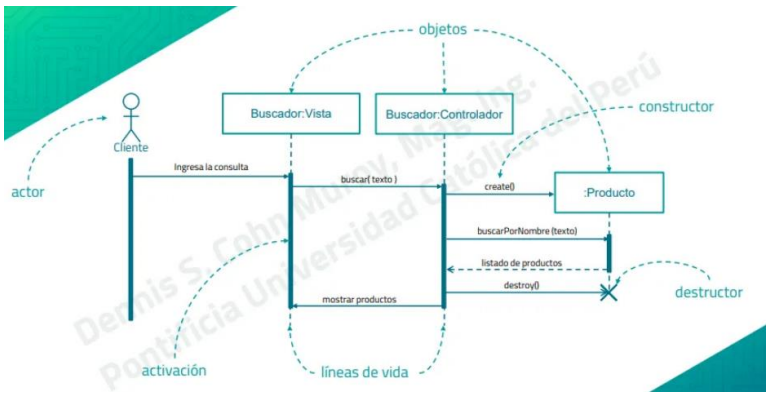
  - Referenciar a otros diagramas

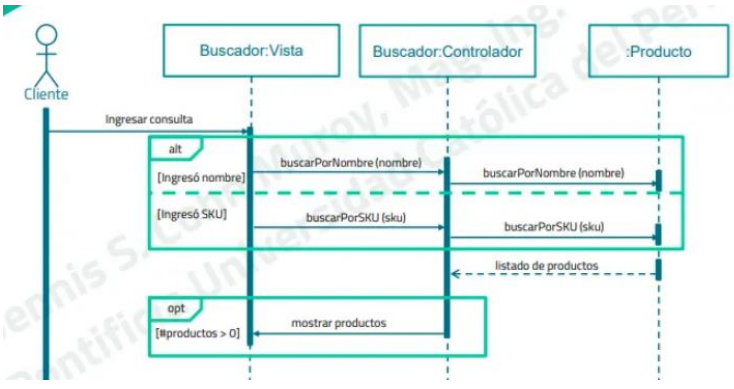


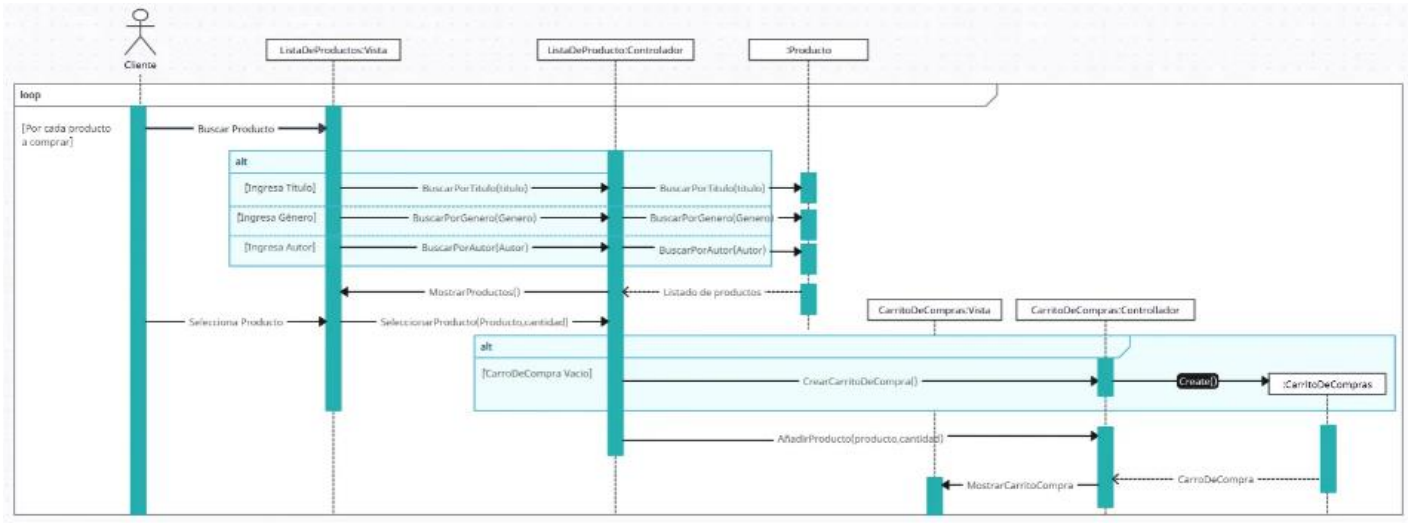
  - Tipos de fragmento de secuencia

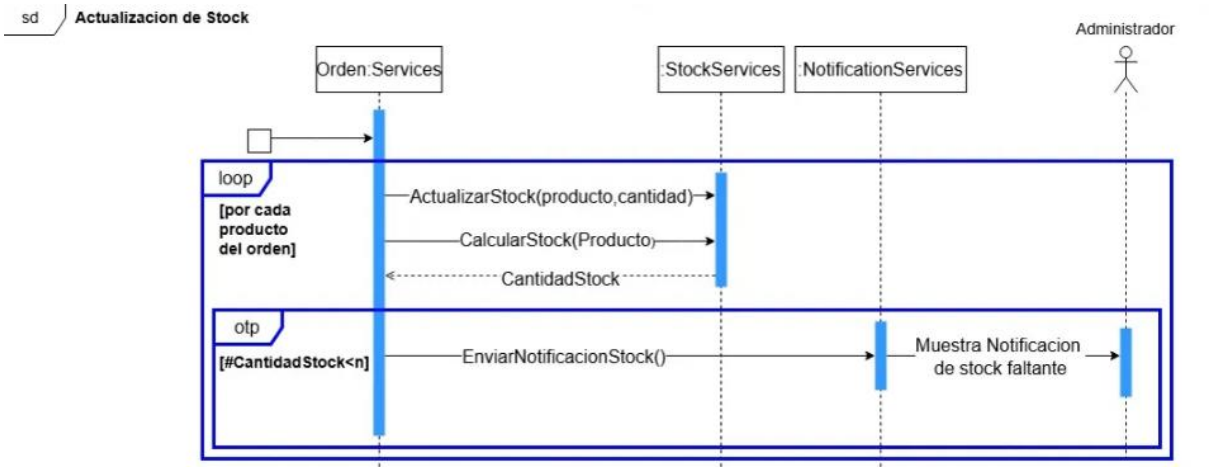
Operador	Acción
alt	Divide un fragmento en base a condiciones booleanas.
opt	Define un fragmento opcional.
par	Indica que un conjunto de mensajes se ejecuta en paralelo.
loop	Indica que un conjunto de mensajes se ejecuta de forma repetitiva.
critical	Indica que un fragmento debe ejecutarse de forma ininterrumpida.
neg	Segmento que se ejecuta cuando se ha producido un error.
ref	Referencia a otro diagrama.

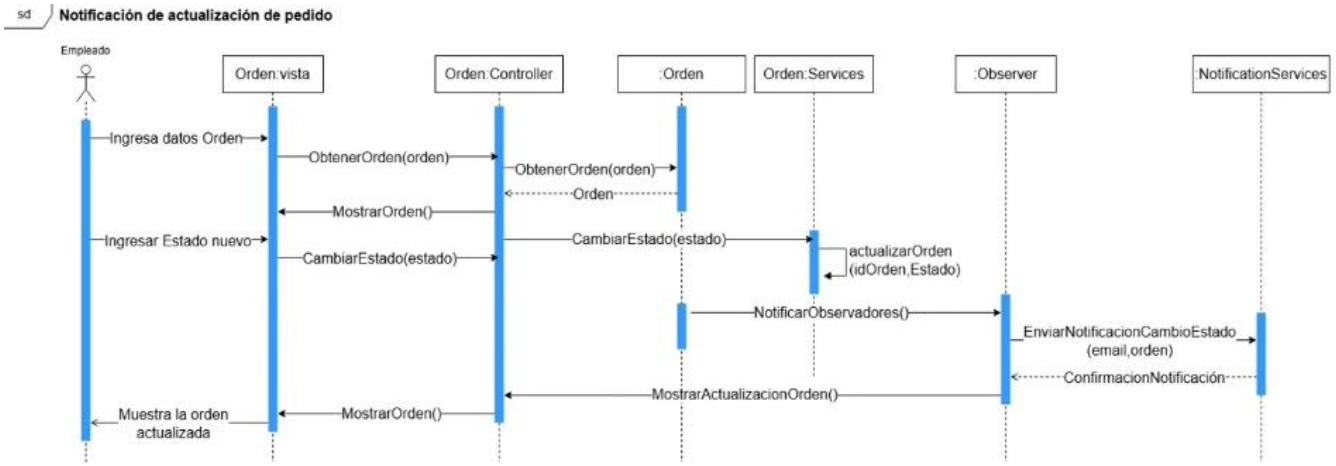
Ejemplos:






Cliente
ListaDeProductos:Vista
ListaDeProducto:Controlador
:Producto
loop
[Por cada producto a comprar]
Buscar Producto
alt
[Ingresa Título]
BuscarPorTitulo(titulo)
[Ingresa Género]
BuscarPorGenero(Genero)
[Ingresa Autor]
BuscarPorAutor(Autor)
MostrarProductos()
Listado de productos
Selecciona Producto
SeleccionarProducto(Producto,cantidad)
CarritoDeCompras:Vista
CarritoDeCompras:Controlador
[CarroDeCompra Vacio]
CrearCarritoDeCompra()
Create()
:CarritoDeCompras
AñadirProducto(producto,cantidad)
MostrarCarritoCompra
CarroDeCompra


sd
Actualizacion de Stock
Orden:Services
:StockServices
:NotificationServices
Administrador
loop
[por cada producto del orden]
ActualizarStock(producto,cantidad)
CalcularStock(Producto)
CantidadStock
otp
[#CantidadStock<n]
EnviarNotificacionStock()
Muestra Notificacion de stock faltante


sd
Notificación de actualización de pedido
Empleado
Orden:vista
Orden:Controller
:Orden
Orden:Services
:Observer
:NotificationServices
Ingresa datos Orden
ObtenerOrden(orden)
ObtenerOrden(orden)
Orden
MostrarOrden()
Ingresar Estado nuevo
CambiarEstado(estado)
CambiarEstado(estado)
actualizarOrden (idOrden,Estado)
NotificarObservadores()
EnviarNotificacionCambioEstado (email,orden)
ConfirmacionNotificación
MostrarActualizacionOrden()
Muestra la orden actualizada
MostrarOrden()

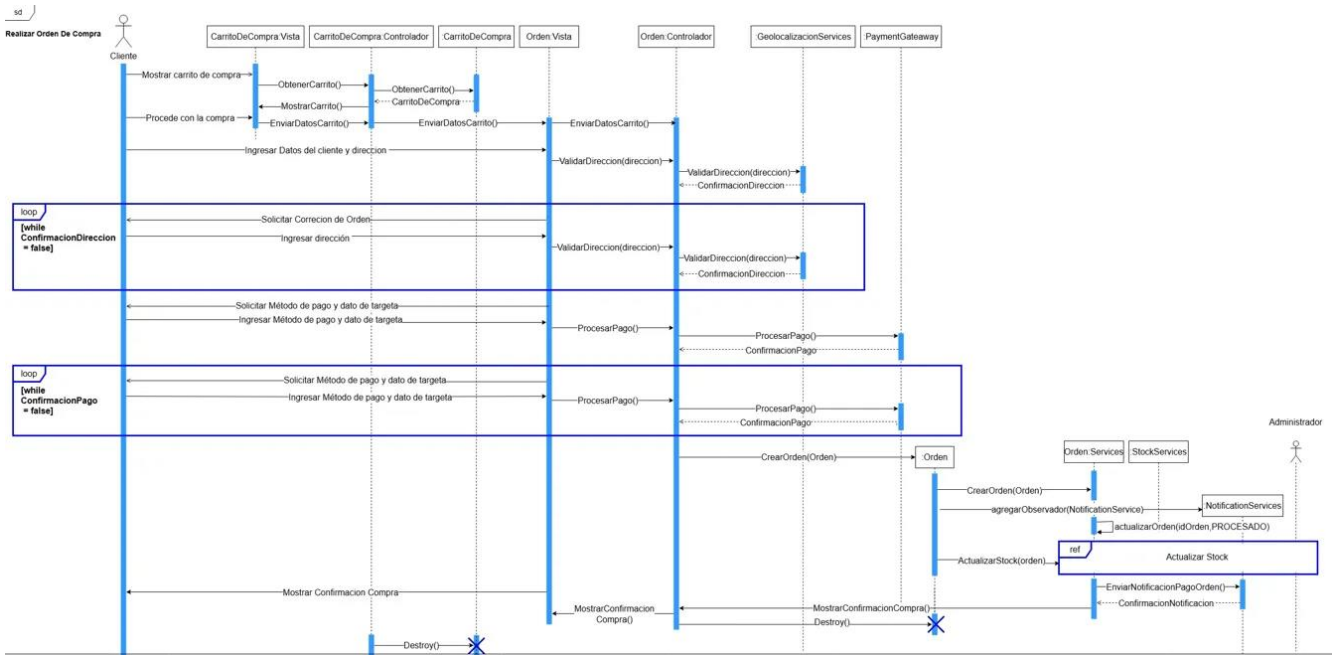
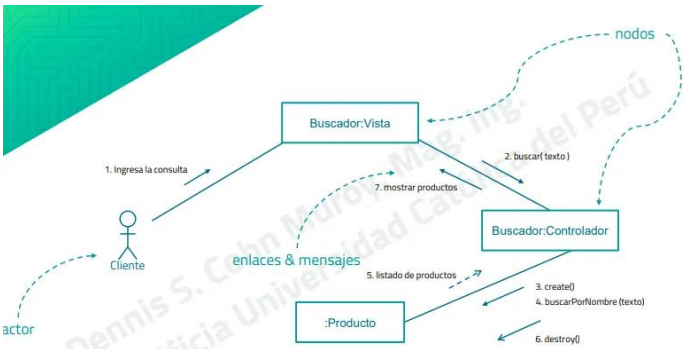

sd
Realizar Orden De Compra
Cliente
CarritoDeCompra:Vista
CarritoDeCompra:Controlador
CarritoDeCompra
Orden:Vista
Orden:Controlador
GeolocalizacionServices
PaymentGateaway
Mostrar carrito de compra
ObtenerCarrito()
ObtenerCarrito()
CarritoDeCompra
MostrarCarrito()
Procede con la compra
EnviarDatosCarrito()
EnviarDatosCarrito()
EnviarDatosCarrito()
Ingresar Datos del cliente y direccion
ValidarDireccion(direccion)
ValidarDireccion(direccion)
ConfirmacionDireccion
loop
[while ConfirmacionDireccion = false]
Solicitar Correcion de Orden
Ingresar dirección
ValidarDireccion(direccion)
ValidarDireccion(direccion)
ConfirmacionDireccion
Solicitar Método de pago y dato de targeta
Ingresar Método de pago y dato de targeta
ProcesarPago()
ProcesarPago()
ConfirmacionPago
loop
[while ConfirmacionPago = false]
Solicitar Método de pago y dato de targeta
Ingresar Método de pago y dato de targeta
ProcesarPago()
ProcesarPago()
ConfirmacionPago
Administrador
CrearOrden(Orden)
Orden
Orden:Services
StockServices
CrearOrden(Orden)
agregarObservador(NotificationService)
NotificationServices
actualizarOrden(idOrden,PROCESADO)
ref
Actualizar Stock
ActualizarStock(orden)
EnviarNotificacionPagoOrden()
ConfirmacionNotificacion
Mostrar Confirmacion Compra
MostrarConfirmacion Compra()
MostrarConfirmacionCompra()
Destroy()
Destroy()

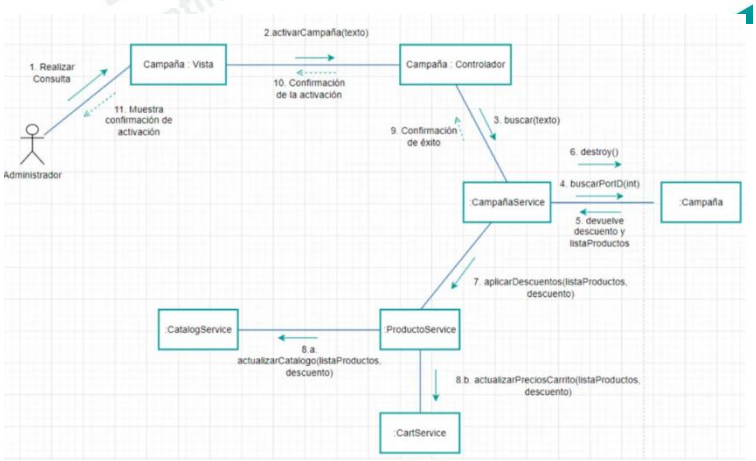
Diagrama de comunicion

Uso → Se centra en las relaciones estructurales entre objetos (ocomponentes) y sus interacciones. Queremos saber quien se comunica con quien

Representacion (misma que la anterior)

Ejemplos

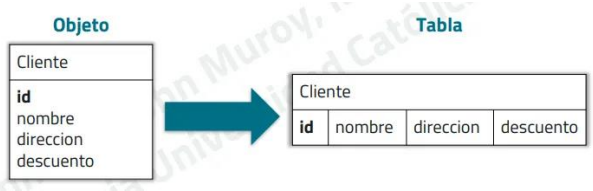




Patrones de Persistencia de objetos

Uso: En la relacion entre objetos y tablas, estos guian el modelamiento de las tablas en base a las clases de diseño

Patron: Identity Field



Objetivo →Mantener un identificador único que permita relacionar un objeto en la memoria con su representación correspondiente en la base de datos

¿Como funciona?

Tienes un objeto (Cliente), este se almacena en la base de datos, y cada registro en la BD tiene ID, lo q hace el patron es garantizar que el ID tmb se almacene en el objeto del Cliente, de manera que la aplicacion sabe que registro en la BD pertenece al objeto.

Uso: Almacena el id del registro de base de datos en el objetos

Caso de uso: Mapear objetos en memoria y registros en base de datos

¿Como seleccionamos el identificador?

La llave debe ser única e inmutable.

- Llave representativa vs. Llave no representativa
  - LLave Representativa ~ (Ejemplo: DNI)
    - Se usa un campo con significado para el usuario
    - La llave representativa podría requerir edición (error humano).
  - LLave no Representativa ~ (Ejemplo: Valor entero aleatorio o incremental)
    - Usa una valor sin significado directo para el usuario
    - Evita problemas como la modificación manual
- Llave simple vs. Llave compuesta
  - Llave simple ~ (Ejemplo: unico campo)
    - Es un único campo que actúa como identificador (por ejemplo, un número ID).
  - LLave compuesta *~ (Ejemplo: Varios Campos)*
    - Combina varios campos para formar una llave unica (nombre+FechaNacimiento)
    - Necesitan utilizar lógica adicional para su creación y asegurar que la combinacion sea unica

¿Como calculamos/generamos el identificador?

- Que la base de datos la autogenere. Esto es comun cuando se utiliza claves primarias autoincrementales
- GUID (Global Unique IDentifier): Valor único entre todas las bases de datos.Util si los datos

deben ser distribuidos entre multiples bases de datos
- Crear tu propio método

Ejemplo Caso: Sistema de Gestión de Clientes

Tienes un Sistema de Gestión de Clientes para una empresa que necesita almacenar información de sus clientes en una base de datos. Cada cliente tiene un nombre, dirección, correo electrónico y número de teléfono. Además, para cada cliente necesitas asegurarte de que puedas identificar de manera única quién es quién, tanto en la base de datos como en tu aplicación.

El ID aquí es el Identity Field que se utiliza para mapear este objeto Clienteen la aplicación con su registro correspondiente en la base de datos. El patrón Identity Field asegura que, al cargar o actualizar clientes, puedas identificar exactamente qué registro en la base de datos corresponde al objeto en la memoria.

Uso del patrón en la persistencia

Cuando guardas un nuevo cliente en la base de datos o recuperas uno existente, el ID se encarga de establecer esta relación.

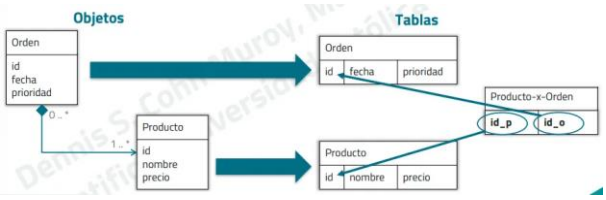
Beneficios del Patrón Identity Field

1. El ID asegura que cada cliente tenga un identificador único, lo que evita duplicaciones o errores al manejar los registros.

2. Al tener el ID almacenado en el objeto, puedes buscar directamente el registro correspondiente en la base de datos sin tener que hacer búsquedas complejas.

3. Este patrón es fácil de implementar en cualquier sistema que maneje persistencia de #

Patron: Association Key Mapping



Objtivo → la relacion entre dos clases se representan a travez de una tabla intermedia

Caso de uso: Cuando la multiplicidad entre clases es de “muchos por muchos”

Ejemplo Caso: Relación entre Estudiantes y Cursos

Un estudiante puede estar inscrito en varios cursos, y un curso puede tener varios estudiantes inscritos. Esta es una relación de muchos a muchos.

Para representar esta relación en la base de datos, necesitamos tres tablas, Estudiantes, Cursos,Estudiantes_Cursos

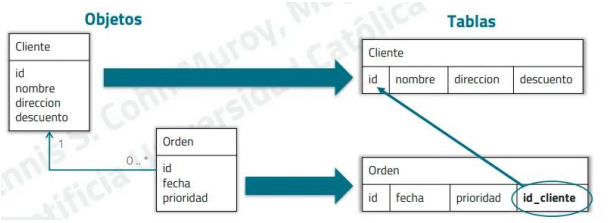
Para agregar un estudiante a un curso o inscribir a un estudiante en varios cursos, primero creas las entradas correspondientes en las tablas Estudiantes y Cursos. Luego, agregas los registros en la tabla intermedia Estudiantes_Cursos para representar la asociación.

Beneficios del Patrón Association Key Mapping

1. Flexibilidad: Al utilizar una tabla intermedia, puedes agregar o eliminar relaciones sin necesidad de modificar las tablas principales.

2. Escalabilidad: Esta estrategia se puede utilizar con grandes cantidades de datos. A medida que crece el número de estudiantes y cursos, la tabla intermedia maneja eficientemente las relaciones entre ellos.objetos, y se puede adaptar a diferentes formas de generar identificadores (autoincrementales, UUIDs, llaves compuestas, etc.).

Patron: Foreigh Key Mapping



Objtivo →Ayuda a representar relaciones entre diferentes objetos o entidades en la base de datos usando llaves foráneas

Caso de uso: Cuando la multiplicidad entre clases es de “1 a 1”, o de “ 1 a muchos”

¿Como funciona?

Cuando tienes una relación entre dos objetos o clases en tu sistema, como entre un **Cliente** y sus **Órdenes** (pedidos que ha realizado), esa relación se representa en la base de datos mediante una llave foránea.

Ejemplo Caso: Relación entre Clientes y Órdenes

Tienesun sistema para una tienda online. Los Clientes pueden hacer Órdenes de productos. Esta es una relación de "uno a muchos" (un cliente puede hacer muchas órdenes, pero una orden solo pertenece a un cliente).

En la clase Orden,se utiliza un objeto de la clase Cliente para representar la relación entre una orden y el cliente que la hizo. El Cliente en la clase Orden es equivalente a la columna Cliente_ID en la tabla Órdenes de la base de datos.

Beneficios del Patrón Foreign Key Mapping

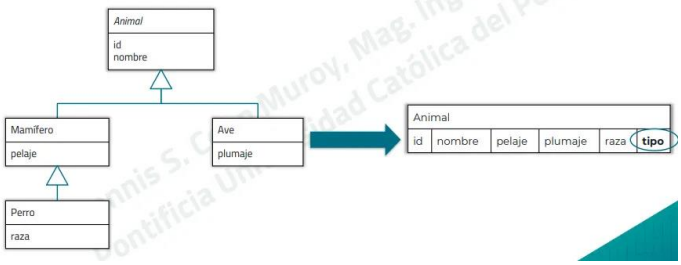
1. El uso de llaves foráneas permite que las relaciones entre objetos de la aplicación estén correctamente mapeadas en la base de datos. Esto hace que sea fácil recuperar o manipular datos relacionados (como encontrar todas las órdenes de un cliente específico)

2. Mediante la llave foránea, puedes realizar consultas complejas en la base de datos, como encontrar todas las órdenes de un cliente, o unir datos de múltiples tablas fácilmente.

Patrones para las Herencias de clases

Se puede combinar los 3 patrones dentro del diseño para un sistema

Patron: Single table Inheritance



→ la herencia se mapea contra una unica tabla que contiene todos los atributos de cada uno de los objetos

Ventajas
- Una unica tabla que mantener
- No se necesita “joins” para obtener datos
- El mover atributos entre clases no implica un cambio en la tabla

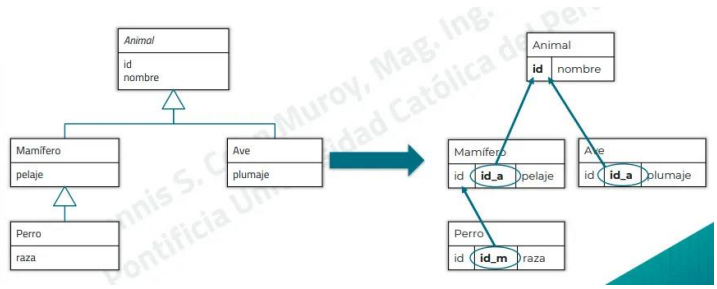
Desventajas
- Complejidad para determinar la importancia de las columnas
- Desperdicio de espacio por valores nulos
- Alto numero de registros, menor desempeño
- Complejidad al agregar nuevas clases hijas

Ejemplo: Relación de Empleados

Supongamos que estamos diseñando un sistema para una empresa. Tenemos una clase base llamada Empleado, y varias clases hijas, como Gerente, Desarrollador y Diseñador. Cada tipo de empleado tiene atributos diferentes.

ID	Nombre	Salario	Bono	LenguajePrograma cion	HerramientaDis eno
1	Ana Pérez	5000	1000	NULL	NULL
2	Juan López	4000	NULL	Java	NULL
3	Pedro Ramírez	3500	NULL	NULL	Photoshop

Patron: Class Table Inheritance



→Cada clase que forma parte de la jerarquía de la herencia es representada por su propia tabla

Ventajas
- No hay desperdicio de espacio.
- Modelo de datos entendible.
- Relación directa entre las clases y las tablas.
- Menor impacto al agregar nuevas clases hijas.

Desventajas
- Consultas complejas conmúltiples “joins”.
- El mover atributos entre clases implica un cambio en la tabla.
- La clase “padre” puede ser un cuello de botella.

Ejemplo: Sistema de Vehículos

Estamos diseñando un sistema para gestionar diferentes tipos de vehículos. Tenemos una clase base llamada Vehículo y varias clases hijas como Coche y Motocicleta, que heredan de Vehículo.

1. **Vehículos:**

ID	Marca	Modelo
1	Toyota	Corolla
2	Honda	Civic
3	Harley	Davidson

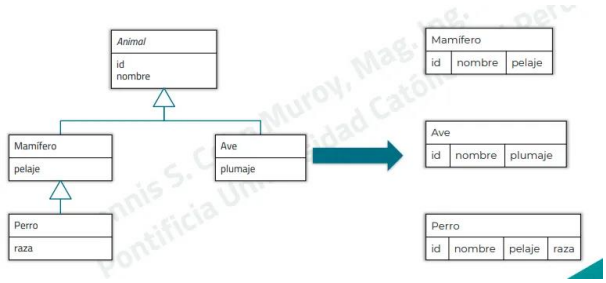
1. **Coches:**

Veh_ID	Puertas
1	4
2	4

1. **Motocicletas:**

Vehiculo_ID	TieneSidecar
3	false

Patron: Concrete Table Inheritance



→Cada clase concreta que forma parte de la jerarquía dela herencia es representada por su propia tabla

Ventajas
- No hay desperdicio de espacio
- Si se requiere leer datos de una clase concreta, no se requiere utilizar “joins”.
- Una tabla es consultada únicamente cuando se utiliza la clase correspondiente.
- Menor impacto al agregar nuevas clases hijas

Desventajas
- El mover atributos entre clases implica un cambio en la tabla.
- El modificar los atributos de la clase “padre” puede impactar en varias tablas.
- Consultas complejas si se requieren los registros sobre la clase “padre”.

Ejemplo: Sistema de Gestión de Empleados

Tenemos un sistema de gestión de empleados en una empresa, donde existen diferentes tipos de empleados con características particulares. Tenemos una clase padre llamada **Empleado**, de la cual heredan clases concretas como Gerente y Desarrollador.

1. **Tabla Gerentes:**

ID	Nombre	Salario	Bono
1	Ana Pérez	5000	1000
2	Juan López	4500	800

1. **Tabla Desarrolladores:**

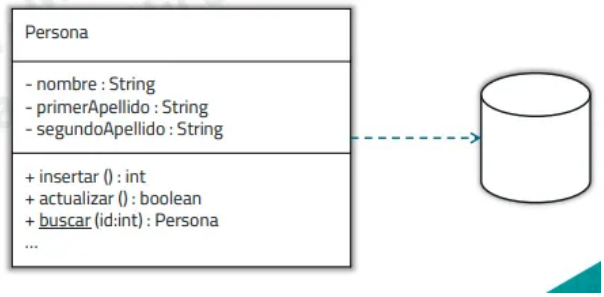
ID	Nombre	Salario	LenguajeProgramacion
3	Pedro Ramírez	4000	Java
4	Laura Martínez	4200	Python

No se necesita una tabla separada para la clase base **Empleado**, ya que la información común (nombre y salario) está repetida en las tablas de **Gerentes** y **Desarrolladores**.

Patrones de Interacción con Base de datos

Uso: En la intereccion entre objetos y tablas, estos definen los objetos responsables para la lectura y escritura de datos en las tablas

Patron: Active Record



→ Un mismo objeto gestiona la lógica de la entidad y la interacción con el repositorio de datos

Consideraciones:
- Correspondencia atributos - columnas.
- Contienen los SQL.
- Métodos estáticos de búsqueda que retornan “Active Records”
- Métodos de actualización e inserción utilizan datos de los atributos.

Ventajas:
- Fácil de implementar.
- Correspondencia objeto-tabla.
- Eficaz cuando el modelo del dominio es simple.

Desventajas:
- Complejidad cuando no hay correspondencia objeto - tabla.
- Dificultad para trabajar con herencias y colecciones.
- Fuerte acoplamiento al diseñar la clase y la tabla.
- El objeto conoce la existencia del repositorio y su estructura.

¿Como funciona?

La idea clave del patrón Active Record es que cada clase en tu aplicación está directamente asociada con una tabla en la base de datos, y cada instancia de esa clase representa una fila (o registro) en la tabla.

- Los atributos del objeto corresponden a las columnas de la tabla en la base de datos.

- El objeto contiene métodos estáticos para buscar registros en la base de datos.

- Los métodos CRUD se realizan directamente en el objeto utilizando los atributos del mismo.

¿Cuando usar Active Record?

- Tienes un modelo de datos simple, con entidades que se corresponden directamente con las tablas de la base de datos.

- Quieres una implementación rápida y sencillasin necesidad de manejar relaciones complejas entre objetos.

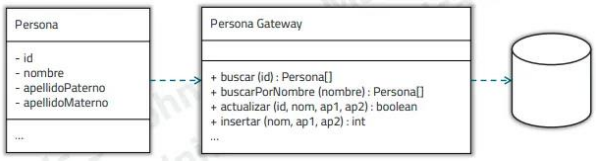
- Tu aplicación no requiere de abstracción o separación de las capas de lógica de negocio y persistencia.

Ejemplo Caso: Sistema de Biblioteca

Tebenis una aplicación sencilla para gestionar un sistema de biblioteca, y queremos manejar la información de los libros en nuestra base de datos.

Si queremos usar esta clase en nuestra aplicación, simplemente interactuamos con los métodos de la clase **Libro** para realizar operaciones en la base de datos

Patron: Table Data Gateway



→El objeto Gateway actua como intermediario entre la BD y la logica de la aplicacion, encargado de gestionar las operaciones CRUD en la BD.

Consideraciones:
- Correspondencia atributos - columnas.
- Gateway contiene los SQL.
- Métodos de búsqueda deben retornar listas.
- Métodos mapean los parámetros a las sentencias SQL.

Ventajas
- Fácil de implementar.
- Encapsula la lógica para interactuar con el repositorio.
- Se puede utilizar de forma combinada con consultas y procedimientos almacenados.

Desventajas
- Dependencia circular entre el Gateway y la Entidad.
- El objeto entidad conoce la existencia un repositorio.

¿Cuando usar Table Data Gateway?

- Aplicaciones sencillas o medianas, donde el modelo de datos no es demasiado complejo.

- Proyectos donde se quiere centralizar el acceso a la base de datos en un solo lugar, y evitar que las entidades gestionen directamente las operaciones CRUD.

- Sistemas donde se desea tener control directo sobre las consultas SQL que se ejecutan.

Ejemplo Caso: Sistema de Gestion de empleados

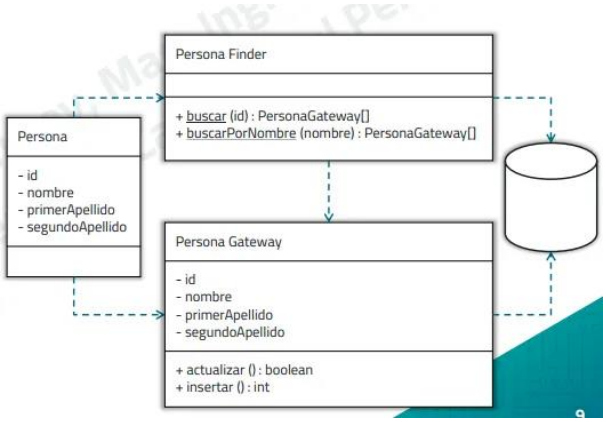
Tienes una aplicación de gestión de empleados en una empresa. Tenemos una tabla de base de datos llamada empleados que almacena los datos de cada empleado.

En nuestra aplicación tenemos una clase que representa la entidad **Empleado**. Esta clase simplemente contiene los atributos del empleado, pero no tiene lógica para interactuar directamente con la base de datos.

Ahora creamos una clase **EmpleadoGateway** que será el intermediario entre la clase **Empleado** y la base de datos. Esta clase contiene toda la lógica necesaria para interactuar con la tabla **empleados** en la base de datos, y permite realizar consultas, inserciones, actualizaciones y eliminaciones.

Ahora podemos usar la clase **EmpleadoGateway** en nuestra aplicación para interactuar con la base de datos, sin tener que mezclar la lógica de la entidad **Empleado** con la lógica de acceso a la base de datos.

Patron: Row Data Gateway



→ El objeto Gateway representa la estructura de la tabla y funge como intermediario con el repositorio dedatos.

Consideraciones:
- Gateway contiene los SQL.
- Métodos de búsqueda deben retornar listas.
- Opera a nivel de una fila especifica
- Cada objeto representa una fila única y gestiona las operaciones relacionadas solo con esa fila

Ventajas
- Encapsula la lógica para interactuar con el repositorio.
- Independiza la estructura de la entidad de la estructura de la tabla.
- Se puede utilizar de forma combinada con consultas y procedimientos almacenados

Desventajas
- El objeto entidad conoce la existencia un repositorio.
- Complejidad de mantenimiento de las estructuras: entidad, gateway, repositorio.

¿Cuando usar Row Data Gateway?

Este patrón es ideal para situaciones donde necesitas trabajar directamente con filas individuales de una tabla y realizar operaciones básicas como leer, modificar o eliminar registros de manera controlada. Es comúnmente utilizado en:

- Aplicaciones simples o medianas donde las operaciones de base de datos son claras y no demasiado complejas.
- Sistemas donde las entidades no deben preocuparse por los detalles de la persistencia y necesitas una capa intermedia que gestione esta interacción.

¿Qué hace el Persona Finder?

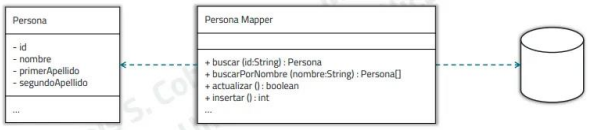
- Persona Finder contiene métodos para buscar personas, como:Estos métodos interactúan con la base de datos para obtener los datos que coincidan con los criterios de búsqueda y retornan una lista de objetos `PersonaGateway`, que representan las filas correspondientes.

  - `buscar(id)`: busca una persona específica por su ID.

  - `buscarPorNombre(nombre)`: busca personas que coincidan con un nombre dado.

- Persona Gateway: Representa una fila individual de la tabla y maneja las operaciones de actualización o inserción para esa fila.

Patron: Data Mapper



→ Transfiere datos entre el objeto entidad y el repositorio de datos

Consideraciones:
- Mapear atributos de la entidad con las columnas del repositorio.
- Mapper contiene los SQL.
- Métodos de búsqueda pueden retornar una entidad o una lista.

Ventajas
- Soporta lógica compleja como colecciones y herencia.
- Independiza la estructura de la entidad de la estructura de la tabla.
- El objeto entidad no conoce la existencia de un repositorio

Desventajas
- Dada la complejidad, se recomienda el uso de librerías y frameworks

¿Cuándo usar Data Mapper?
- Cuando tienes una aplicación con un modelo de dominio complejo, que requiere lógica de negocio avanzada, herencia, colecciones o relaciones complejas entre entidades.

- Cuando necesitas separar claramente la lógica de negocio del acceso a la base de datos para que la aplicación sea más modular y mantenible.
- Cuando quieres que tus objetos de negocio (entidades) estén completamente desacoplados de la forma en que los datos están almacenados en la base de datos.

Conceptos Clave del Data Mapper:

1. Separación de Responsabilidades:
   - Los objetos de tu aplicación, conocidos como entidades, no tienen conocimiento de cómo se almacenan o acceden los datos.

   - El Data Mapper actúa como un intermediario que se encarga de convertir los datos de las tablas en objetos y viceversa, sin que las entidades interactúen directamente con la base de datos.

2. Mapeo de Atributos:
   - El Mapper es el encargado de asociar los atributos de las entidades con las columnas de las tablas en la base de datos.

   - Por ejemplo, si tienes una clase `Persona` con atributos `nombre`, `apellido`, etc., y una tabla en la base de datos con columnas `nombre`, `apellido`, etc., el Mapper será el que realice las conversiones entre ambos.

3. Métodos de Búsqueda y Persistencia:
   - El **Mapper contiene los métodos necesarios para buscar, insertar, actualizar y eliminar entidades en la base de datos. Estos métodos ejecutan las sentencias SQL que realizan estas operaciones.

   - Los métodos de búsqueda pueden retornar una entidad (si solo buscas un objeto) o una lista de entidades (si buscas varios resultados).

Ejemplo sencillo del uso de Data Mapper:

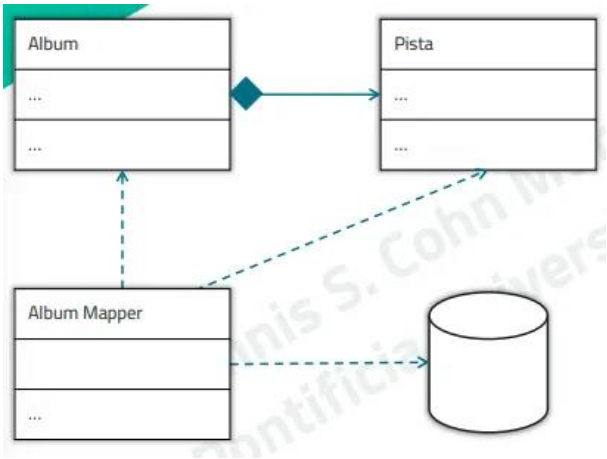
Supongamos que tienes una clase Usuari en tu aplicación y una tabla usuarios en la base de datos.

El Data Mapper sería la clase que se encarga de gestionar la conexión entre esta clase `Usuario` y la tabla `usuarios`.

```
public class UsuarioMapper {

    // Método para buscar un usuario por su ID
    public Usuario buscarPorId(int id) {
        // Aquí iría el código SQL para buscar en la tabla "usuarios"
        String sql = "SELECT * FROM usuarios WHERE id = ?";
        // Ejecutar la consulta y mapear los resultados a un objeto Usuario
        // Simulación: supongamos que obtenemos de la base de datos el siguiente r
esultado:
        return new Usuario(id, "Juan Pérez", "jperez@mail.com");
    }
    // Método para insertar un nuevo usuario en la base de datos
    public void insertar(Usuario usuario) {
        String sql = "INSERT INTO usuarios (nombre, email) VALUES (?, ?)";
        // Ejecutar la consulta SQL usando los atributos de "usuario"
    }
}
```

Patron: Dependet Mapping



→Cuando un objeto aparece en el contexto de otro; éste delega su mapeo al otro objeto.

→En este patrón, la clase dependiente no tiene su propio mapeo a la base de datos de manera independiente; en cambio, delega todo el manejo de su persistencia a la clase "dueña" o principal. Esto significa que la clase dueña controla cómo y cuándo se guardan los datos de la clase dependiente en el repositorio.

Consideraciones:
- La clase “dependiente” depende de la clase “dueña” para gestionar su persistencia.
- La clase “dependiente” debe tener una sola clase “dueña”.
- La clase “dependiente” solo debe ser referenciada desde la clase “dueña”.

Cuándo usar Dependent Mapping?
- Uso ideal cuando tienes objetos que no tienen sentido por sí mismos y que siempre están asociados a un objeto principal.

- En situaciones donde la clase dependiente solo existe en el contexto de la clase dueña y no tiene sentido manejarla de manera separada.

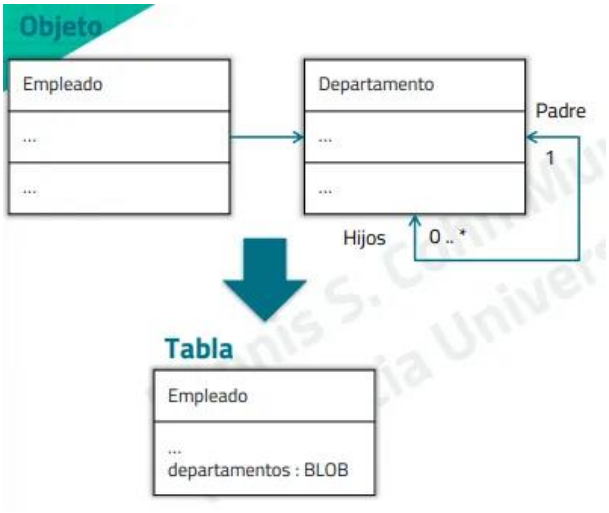
Ejemplo simple del patrón Dependent Mapping:

Imagina que estás desarrollando una aplicación para gestionar pedidos en una tienda en línea. Tienes las siguientes entidades:

- Pedido: representa un pedido que ha realizado un cliente.
- Dirección: representa la dirección de envío asociada al pedido.

En este caso, una Dirección solo tiene sentido en el contexto de un Pedido, ya que no necesitas manejar direcciones de manera independiente. Una dirección siempre está asociada a un pedido, y no puedes guardar una dirección sin guardar primero un pedido.

Patron: Objeto Grande (LOB) Serializado



→Almacena una estructura de objetos serializándolos en un único objeto

Consideraciones:
- La serialización puede ser:
    - ◎ Binaria (BLOB).
    - ◎ Texto (CLOB).
- Cuidado de no duplicar data; salvo que se almacenen “snapshots”.
- No es compatible con sentencias SQL

BLOB
- Fácil de programar.
- Utiliza poco espacio.
- No puede reconstruirse los datos si no se cuenta con el objeto original.
- Un cambio en el objeto puede impedir la lectura de objetos previamente guardados.

CLOB
- Serializa el objeto en una cadena de caracteres (JSON, XML).
- El texto es entendible a simple vista.
- Ocupa más espacio.
- Requiere implementar el serializador/deserializador.
- Más lento que el BLOB.

¿Cómo funciona el patrón de Objeto Grande Serializado?

1. Almacenar el objeto:

- En lugar de mapear cada propiedad o atributo del objeto a una columna en una tabla, tomas el objeto completo, lo serializas (lo conviertes en un formato que puede ser almacenado) y lo guardas en un único campo en la base de datos.

2. Recuperar el objeto:

- Cuando necesitas acceder al objeto, lees el contenido del campo (que contiene el objeto serializado), lo deserializas (es decir, lo conviertes nuevamente a su forma original) y lo utilizas dentro de tu aplicación.

¿Cuándo usar el patrón de LOB Serializado?

- Cuando los objetos son muy complejos: Si tu objeto tiene muchas propiedades o relaciones internas, mapear cada atributo a una columna en una tabla puede ser complicado y poco eficiente.

- Cuando no necesitas hacer consultas SQL sobre el objeto: Si no necesitas buscar o filtrar datos dentro del objeto (por ejemplo, no necesitas buscar contratos basados en la fecha o el cliente), el patrón LOB serializado puede ser una buena opción.

- Cuando el rendimiento y el espacio son una prioridad: Si usas BLOB, puedes almacenar grandes cantidades de datos en un espacio reducido, y la serialización es relativamente rápida.

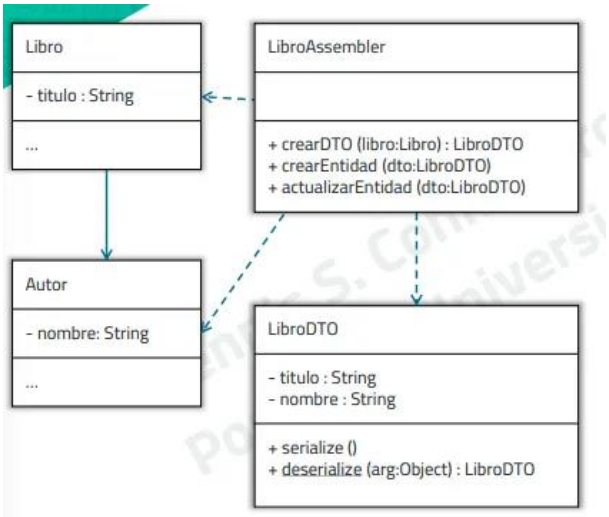
¿Qué es la serialización?

La serialización es el proceso de convertir un objeto (que existe en la memoria de un programa) en un formato que puede ser fácilmente almacenado o transmitido. Este formato puede ser binario (BLOB - Binary Large Object) o de texto (CLOB - Character Large Object).

Ejemplo de uso del patrón LOB Serializado:

Imagina que estás desarrollando una aplicación para gestionar contratos electrónicos en una empresa. Cada contrato contiene múltiples secciones, firmas digitales, fechas, entre otros datos. Estos contratos son complicados, y en lugar de crear una tabla con decenas de columnas para almacenar cada detalle del contrato, decides utilizar el patrón de LOB Serializado.

Patron: Data Transfer Object (DTO)



→ Objeto que traslada información entre sistemas a fin de reducir el número de llamadas a métodos.

Consideraciones:
- Envía y recibir un gran número de registros con una sola llamada.
- El DTO debe ser un objeto serializable.
- Cliente y servidor deben contar con el mismo algoritmo para serializar/deserializar los datos.

¿Qué es un DTO?

Un DTO es un objeto sencillo que tiene atributos pero sin lógica. Es decir, no contiene funciones o métodos complejos, únicamente se utiliza para transportar datos de un lugar a otro, como entre el cliente y el servidor o entre diferentes capas de

una aplicación (por ejemplo, la capa de negocio y la capa de presentación).

Consideraciones importantes del patrón DTO:

- Debe ser serializable:
  - Como los DTO suelen enviarse a través de una red (por ejemplo, desde el servidor al cliente), necesitan ser convertidos a un formato que se pueda transmitir, lo que llamamos serialización.
  - Al ser serializable, el DTO puede transformarse en un formato que pueda viajar por internet o entre diferentes sistemas (como JSON, XML o binario), y luego puede deserializarse para volver a su forma original cuando llega a su destino.
- Reduce la cantidad de llamadas:
  - Sin el uso de un DTO, podrías necesitar hacer muchas llamadas para transferir datos. Por ejemplo, si tienes que enviar 10 atributos diferentes de un objeto, podrías hacer 10 llamadas separadas. Con un DTO, envías o recibes todos los atributos en una sola llamada, lo que mejora la eficiencia.
- Cliente y servidor deben tener el mismo esquema de serialización:
  - El cliente (que puede ser una aplicación web, móvil o un programa) y el servidor (donde se almacena y procesa la información) deben "ponerse de acuerdo" sobre cómo se serializan y deserializan los datos. Ambos sistemas necesitan usar el mismo algoritmo para asegurarse de que los datos se interpreten de la misma forma en ambos lados.

Ventajas del patrón DTO:

- Optimiza el rendimiento:

Al agrupar todos los datos que necesitas enviar o recibir en una sola llamada, reduces el número de conexiones a través de la red, lo que es especialmente útil en sistemas distribuidos o aplicaciones web.

- Facilita la transferencia de grandes volúmenes de datos:

Cuando necesitas transferir muchos registros o atributos, puedes encapsular toda esa información en un DTO y enviarla de una sola vez, lo que es más rápido y eficiente.

- Aislamiento de la lógica del negocio:

El DTO solo transporta datos, no contiene lógica de negocio. Esto significa que se separa la lógica de la aplicación de los datos que viajan entre las capas o sistemas.

Desventajas del patrón DTO:

- Sobrecarga de código:

El uso de DTOs puede requerir más código para definir estos objetos y escribir los métodos necesarios para serializarlos y deserializarlos. Además, debes estar pendiente de mantener sincronizados los DTOs con la estructura de los objetos del lado del servidor.

- Duplicación de estructuras:

A veces, los DTOs pueden terminar siendo una copia de los objetos de negocio o de la base de datos, lo que genera redundancia. Esto puede hacer que se incremente el esfuerzo de mantenimiento, ya que si cambias la estructura del objeto original, también debes cambiar el DTO.

Ejemplo sencillo para entender el uso de un DTO:

Imagina que estás desarrollando una aplicación para gestionar usuarios y necesitas enviar la información de un usuario desde el servidor hasta una aplicación en el cliente (como una app móvil).

Supón que el servidor tiene un objeto de negocio llamado Usuario con los siguientes atributos: id, nombre, apellido, correoElectronico, fechaDeRegistro

Cada vez que el cliente quiere obtener la información de un usuario, en lugar de hacer múltiples llamadas al servidor (una llamada para el nombre, otra para el apellido, etc.), puedes crear un DTO que agrupe toda esa información.

En este caso, el UsuarioDTO es un objeto de transporte que contiene todos los datos que quieres enviar del servidor al cliente. En lugar de hacer múltiples llamadas para cada atributo, puedes hacer una sola llamada que retorne este DTO.

¿Cómo se usaría en la práctica?

1. Servidor:

- En el servidor, cuando quieras enviar la información de un usuario, creas un objeto de tipo `UsuarioDTO` y lo llenas con los datos correspondientes.

2. Cliente:

- En el cliente, recibes el DTO, lo deserializas (por ejemplo, si lo enviaste en formato JSON o XML) y ahora tienes todos los datos del usuario listos para usarse.

Resumen del uso del patrón DTO:

- ¿Cuándo usar un DTO?: Cuando necesitas enviar y recibir muchos datosentre sistemas o entre diferentes capas de tu aplicación. En lugar de hacer muchas llamadas pequeñas, agrupas toda la información en un solo objeto y la transfieres en una sola llamada, mejorando la eficiencia.

- ¿Qué hace un DTO?: Solo transporta datos, no tiene lógica de negocio ni comportamientos complejos. Sirve únicamente como un contenedor de datos.

- ¿Por qué es útil?: Reduce el tráfico de red y simplifica la transferencia de datos entre sistemas, especialmente cuando se trata de grandes cantidades de información o estructuras de datos complejas.