Arquitectura de Software y Diseño

Construyendo sistemas escalables y organizados 🛠️

Probabilidad y Estadística

Aprendiendo con diversión 🎓

25 de noviembre de 2025

¿Qué vamos a aprender?

Nota importante

¡Hoy pensamos como arquitectos de software! 🏛

Objetivo 1: Entender la Programación Orientada a Objetos (POO)

Objetivo 2: Conocer arquitecturas Cliente-Servidor

Objetivo 3: Aprender el patrón MVC

Objetivo 4: Introducción a Bases de Datos

Ejemplo

Como diseñar un edificio: necesitamos planos, estructura y cimientos sólidos

Programación Orientada a Objetos (POO)

Nota importante

¡Modelar el mundo real en código!

POO organiza el código en 'objetos' que representan cosas reales

Ejemplo

En un ERP necesitamos representar:

- Usuarios (con nombre, email, contraseña)
- Billeteras (con saldo, transacciones)
- Facturas (con número, cliente, items, total)

Cada uno es un 'objeto' con sus propios datos y comportamientos

Clases y Objetos

Concepto	Descripción	Analogía
Clase	El molde o plantilla	Plano de una casa
Objeto	Una instancia de la clase	Casa construida
Atributos	Datos del objeto	Color, tamaño, habitaciones
Métodos	Acciones del objeto	Abrir puerta, encender luz

Ejemplo

Clase Usuario es el molde. usuario1 y usuario2 son objetos creados con ese molde

Ejemplo: Clase Usuario

Problema

Crear una clase para representar usuarios del sistema

Ejemplo

```
class Usuario:
    def __init__(self, nombre, email):
        self.nombre = nombre
        self.email = email
        self.activo = True

    def desactivar(self):
        self.activo = False

    def mostrar_info(self):
        return f"{self.nombre} - {self.email}"

# Crear objetos
```

Ejemplo: Clase Billetera

Problema

Modelar una billetera digital con operaciones

Ejemplo

```
class Billetera:
    def __init__(self, propietario):
        self.propietario = propietario
        self.saldo = 0
        self.transacciones = []

    def depositar(self, monto):
        self.saldo += monto
        self.transacciones.append(f"+{monto}")

    def retirar(self, monto):
        if self.saldo >= monto:
            self.saldo -= monto
```

Encapsulamiento: Proteger los Datos

Nota importante

¡No permitir modificaciones no autorizadas! 🔒

El saldo de una billetera no debe modificarse directamente

Ejemplo

Forma incorrecta:

$$billetera.saldo = 1000000 q Modificación directa$$

Ejemplo

Forma correcta:

$$billetera.depositar(1000000) A través de método$$

Ejemplo Completo: Clase Factura

Ejemplo

```
class Factura:
    def __init__(self, numero, cliente):
        self.numero = numero
        self.cliente = cliente
        self.items = []
        self.total = 0

    def agregar_item(self, producto, precio):
        self.items.append({
            "producto": producto,
            "precio": precio
        })
        self.total += precio

    def aplicar_iva(self):
        self.total = self.total * 1.19
```

Arquitectura Cliente-Servidor

Nota importante

¿Quién pide datos y quién los entrega? 🖳↔🖫

Cliente	Servidor
Solicita datos	Entrega datos
Interfaz de usuario	Lógica de negocio
Frontend/App	Backend
HTML, CSS, JS	Python, Java, Node.js

Ejemplo

WhatsApp Web (cliente) pide mensajes al servidor de WhatsApp

Flujo de una Petición

Nota importante

El ciclo de vida de una consulta

1 Cliente: Usuario hace clic en 'Ver saldo'

2 Cliente: Envía petición HTTP al servidor

3 Servidor: Recibe petición, consulta base de datos

4 Servidor: Obtiene saldo de la cuenta

5 Servidor: Envía respuesta al cliente

6 Cliente: Muestra el saldo en pantalla

Ejemplo

Todo esto pasa en milisegundos

Monolito vs. Microservicios

Monolito	Microservicios
Todo en un solo bloque	Servicios separados
Más simple al inicio	Más flexible
Difícil escalar partes	Escala por componente
Un error tumba todo	Fallas aisladas
Despliegue completo	Despliegue independiente

Ejemplo

ERP Monolito: Todo junto. Microservicios: Facturación, Inventario, RRHH separados

Ejemplo: ERP con Microservicios

Nota importante

Cada servicio tiene una responsabilidad específica ◎

Ejemplo

Arquitectura de un ERP escalable:

- Servicio de Autenticación (Login/Logout)
- Servicio de Facturación (Crear/Consultar facturas)
- Servicio de Inventario (Stock de productos)
- Servicio de RRHH (Empleados y nómina)
- Servicio de Reportes (Análisis y estadísticas)

Todos se comunican mediante APIs

Patrón MVC: Modelo-Vista-Controlador

Nota importante

Separar responsabilidades para código más limpio

Capa	Responsabilidad	Ejemplo
Modelo	Datos y lógica de negocio	Clase Usuario, Factura
Vista	Interfaz visual	HTML, CSS, pantallas
Controlador	Conecta Modelo y Vista	Recibe clicks, actualiza datos

Ejemplo

Usuario hace clic → Controlador → Modelo consulta DB → Vista muestra resultado

Ejemplo MVC: Login de Usuario

Problema

Sistema de login usando patrón MVC

Ejemplo

```
# MODELO (datos)
class UsuarioModelo:
    def verificar_credenciales(usuario, password):
        # Consulta base de datos
        return usuario_valido

# VISTA (interfaz)
def mostrar_formulario_login():
    # Muestra campos de usuario y contrasena
    pass

# CONTROLADOR (logica)
def procesar_login(usuario, password):
```

Bases de Datos: Persistencia de Datos

Nota importante

¿Por qué necesitamos guardar la información?

Las variables en memoria se borran al cerrar el programa

Necesitamos almacenamiento permanente

Ejemplo

Si apagas WhatsApp y lo vuelves a abrir, tus mensajes siguen ahí

Esos mensajes están en una base de datos

$$MemoriaRAM(temporal) ß BasedeDatos(permanente)$$

SQL vs. NoSQL

SQL (Relacional)	NoSQL (No Relacional)
Tablas con filas y columnas	Documentos, clave-valor
Relaciones entre tablas	Datos anidados
Estructura rígida	Estructura flexible
MySQL, PostgreSQL	MongoDB, Redis

Ejemplo

SQL: Excel con tablas relacionadas. NoSQL: Colección de documentos JSON

Modelar Relaciones: Clientes y Compras

Problema

Relacionar clientes con sus compras en SQL

Tabla Clientes

id	nombre	email
1	Ana López	ana@mail.com
2	Juan Pérez	juan@mail.com

Tabla Compras

id	cliente$_id$	monto
101	1	50000
102	1	30000
103	2	75000

cliente$_id$*conectalastablas*(*claveforánea*)

Propiedades ACID

Nota importante

Garantías de integridad en transacciones

A Atomicidad: Todo o nada (transferencia completa o se revierte)

C Consistencia: Datos siempre válidos (no saldos negativos)

I Aislamiento: Transacciones no interfieren entre sí

D Durabilidad: Datos guardados no se pierden

Ejemplo

Transferencia bancaria: Si falla a la mitad, se revierte todo (Atomicidad)

Ejemplo ACID: Transferencia Bancaria

Problema

Transferir 50,000 *decuenta A a cuenta B*

```
INICIO TRANSACCION
  1. Verificar saldo cuenta A >= 50000
  2. Restar 50000 de cuenta A
  3. Sumar 50000 a cuenta B
  4. Registrar movimiento

SI todo OK, COMMIT (confirmar)
SI algo falla, ROLLBACK (revertir todo)
FIN TRANSACCION
```

Nota importante

¡El dinero nunca se pierde a la mitad! $

¡Tu Turno de Practicar!

Problema

Diseñar clases POO para un sistema de biblioteca

Clases necesarias:

- Libro (título, autor, ISBN, disponible)
- Usuario (nombre, email, $libros_prestados$)
- Biblioteca (inventario, usuarios)

Métodos sugeridos:

- $prestar_libro(usuario, libro)$
- $devolver_libro(usuario, libro)$

Nota importante

Piensa en encapsulamiento: ¿qué atributos deben ser privados?

¡Resumen de lo Aprendido!

- POO modela el mundo real con clases y objetos
- Encapsulamiento protege datos sensibles
- Cliente-Servidor separa interfaz de lógica de negocio
- Microservicios permiten escalar por componentes
- MVC separa responsabilidades (Modelo-Vista-Controlador)
- Bases de datos persisten información permanentemente
- SQL es relacional, NoSQL es flexible
- ACID garantiza transacciones seguras

Nota importante

¡Ya puedes diseñar arquitecturas de software profesionales!

¡Sigue aprendiendo!