DISEÑO DE SISTEMAS

Juan Carlos Molina Lozano
Docente

CONTENIDO

- Introducción al Diseño de Sistemas
- Un poco de Historia
- Características del Diseño de Sistemas
- Fases del Diseño de Sistemas
- Procesos en e Diseño de Sistemas
- Metodologias de Desarrollo
- Problemas asociados al Desarrollo de Software
- Conceptos: Objetos, clase y abstracción Asociaciones: dependencias, agregación y composición, Encapsulamiento, herencia y Polimorfismo

QUÉ ES EL DISEÑO DE SISTEMAS Y PARA QUÉ SIRVE?

El diseño de sistemas es una fase fundamental en el desarrollo de software que permite transformar requisitos funcionales y no funcionales en una estructura técnica organizada y eficiente. Su objetivo principal es establecer la arquitectura, los componentes y la interacción entre las partes de un sistema de información, garantizando que sea escalable, mantenible y alineado con las necesidades del usuario y la organización.

UN POCO DE HISTORIA ACERCA DEL DISEÑO DE SISTEMAS

El concepto de diseño de sistemas surge con el desarrollo de la informática en la segunda mitad del siglo XX. En los años 60 y 70, con el auge de la programación estructurada, se comenzó a enfatizar la importancia de una planificación detallada antes de la implementación de software. Posteriormente, en los años 80 y 90, con la aparición de la programación orientada a objetos y las metodologías ágiles, el diseño de sistemas evolucionó hacia enfoques más flexibles y adaptativos.

CARACTERÍSTICAS DEL DISEÑO DE SISTEMAS

Debe ser evaluable contra los requisitos del software

Debe ser una guía clara para la implementación del producto

Debe ser expresado en términos no ambiguos

Debe poder justificar cada decisión de diseño

Modularidad: División del sistema en componentes independientes.

Escalabilidad: Capacidad del sistema para adaptarse al crecimiento.

Mantenibilidad: Facilidad de modificar y actualizar el sistema.

Reusabilidad: Uso de componentes ya diseñados para otros proyectos.

Eficiencia: Optimización de recursos y rendimiento.

Seguridad: Protección contra vulnerabilidades y amenazas.

FASES DEL DISEÑO DE UN SISTEMA

El diseño de sistemas puede dividirse en las siguientes fases:

1. **Análisis de requisitos**: Recopilación y validación de necesidades.
2. **Diseño conceptual**: Identificación de entidades clave y relaciones.
3. **Diseño lógico**: Especificación de la arquitectura y estructura de datos.
4. **Diseño físico**: Selección de tecnologías y optimización de rendimiento.
5. **Programación:** Codificación
6. **Pruebas y validación**: Evaluación de la funcionalidad antes de la implementación
7. **Implementación**
8. **Mantenimiento**

PROCESOS EN EL DISEÑO DE SISTEMAS

Los procesos claves incluyen:

- **Modelado de datos**: Representación de la información del sistema.
- **Definición de arquitectura**: Estructura y relaciones entre componentes.
- **Interacción con el usuario**: Diseño de interfaces y experiencia del usuario.
- **Pruebas de diseño**: Evaluación preliminar de la viabilidad del sistema.

METODOLOGÍAS DE DESARROLLO DE SOFTWARE Y SU PROPÓSITO

Su propósito es mejorar la organización del trabajo, reducir errores

Las metodologías de desarrollo de software guían el proceso de construcción de sistemas, estableciendo principios y buenas y garantizar la calidad del producto final

TIPOS DE METODOLOGÍAS DE DESARROLLO

- **Metodologías Clásicas**:
  - Cascada
  - Espiral
  - Prototipado
- **Metodologías Ágiles**:
  - Scrum
  - Kanban
  - XP (Extreme Programming)

ETAPAS DEL CICLO DE VIDA DE LAS METODOLOGÍAS DE DESARROLLO

1. **Planificación**: Definir requisitos y recursos.
2. **Análisis**: Estudio de factibilidad y modelado del sistema.
3. **Diseño**: Definición de la arquitectura y estructura del software.
4. **Implementación**: Desarrollo y codificación.
5. **Pruebas**: Validación del funcionamiento y corrección de errores.
6. **Despliegue**: Instalación y configuración en el entorno de producción.
7. **Mantenimiento**: Actualizaciones y soporte técnico.

PROBLEMAS ASOCIADOS EN EL DESARROLLO DE UN SISTEMA DE INFORMACIÓN

Algunos problemas comunes incluyen:

- Requisitos poco definidos o cambiantes.
- Mala comunicación entre el equipo de desarrollo y el cliente.
- Falta de pruebas adecuadas.
- Problemas de escalabilidad y rendimiento.
- Exceso de costos y tiempos de entrega prolongados.

CAUSAS DE ESTOS PROBLEMAS

- Falta de análisis adecuado antes de comenzar el desarrollo.
- Uso de metodologías inapropiadas para el proyecto.
- Deficiente gestión del equipo y recursos.
- No considerar la usabilidad y experiencia del usuario.

CÓMO EVITAR ESTOS PROBLEMAS A TRAVÉS DEL DISEÑO DE SISTEMAS

Para evitar estos inconvenientes, es esencial:

- Definir claramente los requisitos desde el inicio.
- Utilizar metodologías de desarrollo adecuadas según el proyecto.
- Mantener una comunicación fluida entre todas las partes involucradas.
- Realizar pruebas constantes en cada fase del desarrollo.
- Incorporar principios de diseño modular y escalable.

ORIENTACIÓN A OBJETOS

Es un modelo de programación que organiza el código en torno a objetos, en lugar de funciones y lógica

Características

- Se enfoca en los objetos que los programadores necesitan manipular.
- Agrupa datos y las operaciones relacionadas con ellos en entidades llamadas "clases".
- Las clases pueden heredar atributos de una superclase, lo que permite reutilizar código y ahorrar tiempo.
- Tiene como objetivo reducir los errores y promover la reutilización del código.

Qué es un paradigma en programación?

Los paradigmas de programación son modelos para resolver problemas comunes con nuestro código. Son caminos, guías, reglas, teorías y fundamentos que agilizan nuestro desarrollo y evitan que reinventemos la rueda.

CLASES EN PROGRAMACIÓN ORIENTADA A OBJETOS

Qué es una Clase?

Una **clase** es un molde o plantilla a partir de la cual se crean objetos. Define los atributos y métodos que tendrán los objetos.

📌 **Ejemplo del Mundo Real: Molde para Galletas** 🍪

Imagina que tienes un **molde para galletas** en forma de estrella ⭐. Puedes usarlo para hacer muchas galletas iguales, pero cada una tendrá su propia masa y decoración.

✅ **El molde** = La clase
✅ **Cada galleta horneada** = Un objeto creado a partir de la clase

OBJETOS EN PROGRAMACIÓN ORIENTADA A OBJETOS

Qué es un Objeto en POO?

- Un **objeto** es una unidad dentro de un programa que representa una entidad del mundo real. Cada objeto tiene:
  ✅ **Atributos** (propiedades que describen al objeto).
  ✅ **Métodos** (acciones que el objeto puede realizar).
  ✅ **Identidad** (cada objeto es único en memoria, incluso si tiene los mismos valores que otro).

📌 Ejemplo del Mundo Real: Un Celular 📱

Si quisiéramos representar un celular en POO, podríamos definirlo así:

Celular	Ejemplo
Atributos	Marca, modelo, color, capacidad de almacenamiento
Métodos	Llamar(), enviar_mensaje(), tomar_foto()

ABSTRACCIÓN EN PROGRAMACIÓN ORIENTADA A OBJETOS

Qué es la Abstracción?

La **abstracción** es el proceso de ocultar detalles complejos y mostrar solo lo esencial.

📌 **Ejemplo del Mundo Real: Conducir un Auto** 🚗

- Cuando conduces un auto, solo te preocupas por:
  ☑ **Acelerar**
  ☑ **Frenar**
  ☑ **Girar el volante**

No necesitas saber cómo funciona el motor internamente.

☑ **¿Cómo aplicamos abstracción en POO?**
Definiendo clases con métodos que oculten la complejidad del código.

☑ **¿Por qué usar abstracción?**
Permite simplificar el código y ocultar detalles innecesarios para el usuario.

ASOCIACIONES ENTRE OBJETOS

Los objetos no existen en aislamiento, interactúan entre sí de diferentes formas.

📌 **Tipos de Asociaciones:**

Tipo	Descripción	Ejemplo
Dependencia	Un objeto usa temporalmente otro.	Un profesor usa una computadora para dar clase.
Agregación	Un objeto es parte de otro, pero puede existir por separado.	Un carro tiene ruedas, pero las ruedas pueden usarse en otro carro.
Composición	Un objeto es parte de otro, pero no puede existir sin él.	Un corazón es parte de un ser humano y no puede existir sin él.

Los objetos interactúan entre sí de diferentes formas. Estas interacciones se llaman **asociaciones** y pueden ser:

- 1️⃣ **Dependencia (Uso Temporal de un Objeto)**
- Un objeto usa otro de manera **temporal**, pero no lo almacena.

📌 **Ejemplo del Mundo Real: Un Profesor y un Proyector 🎥**

- El profesor **usa** un proyector para dar clase, pero el proyector no le pertenece.

AGREGACIÓN DE OBJETOS

Ejemplo del Mundo Real: Un Carro y sus Ruedas 🚙

- Un carro **tiene ruedas**, pero las ruedas **pueden existir por separado**.
- Explicación: Carro tiene Ruedas, pero las Ruedas pueden existir por separado y usarse en otro carro.

COMPOSICIÓN DE OBJETOS

Composición (Un Objeto No Puede Existir sin el Otro)

📌 **Ejemplo del Mundo Real: Un Cuerpo Humano y su Corazón** 💙

- Un corazón no puede existir sin un cuerpo humano.

ENCAPSULAMIENTO, HERENCIA Y POLIMORFISMO EN PROGRAMACIÓN ORIENTADA A OBJETOS

La Programación Orientada a Objetos (POO) es un paradigma de programación basado en la creación y manipulación de objetos. Un objeto es una entidad que combina datos (atributos) y comportamientos (métodos).

Los principales pilares de la POO son:

- Encapsulamiento: Protege los datos del acceso no autorizado.
- Herencia: Permite la reutilización de código entre clases.
- Polimorfismo: Permite la flexibilidad en el uso de métodos.

La POO se usa en lenguajes como Python, Java, C++, entre otros, para construir software modular, reutilizable y escalable.

ENCAPSULAMIENTO

El encapsulamiento es el principio que permite ocultar los detalles internos de un objeto y exponer solo las partes necesarias para su uso. Esto se logra restringiendo el acceso directo a los atributos y proporcionando métodos para interactuar con ellos.

Beneficios del Encapsulamiento

- ☑ Protección de datos: Evita modificaciones no autorizadas.
- ☑ Modularidad: Facilita la organización del código.
- ☑ Mantenimiento: Reduce el impacto de cambios en la implementación interna.

Niveles de Acceso en el Encapsulamiento

Los lenguajes de POO permiten diferentes niveles de acceso a los atributos y métodos de una clase:

- Público (public): Accesible desde cualquier parte del programa.
- Protegido (protected): Accesible solo dentro de la clase y sus subclases.
- Privado (private): Accesible solo dentro de la propia clase.

ENCAPSULAMIENTO

Ejemplo en Python y java:

Aquí, el atributo __nombre es privado y solo se puede acceder a él mediante el método obtener_nombre().

```python
class Persona:
    def __init__(self, nombre, edad):
        self.__nombre = nombre  # Atributo privado
        self.__edad = edad  # Atributo privado

    def obtener_nombre(self):
        return self.__nombre  # Método público para acceder

    def cambiar_edad(self, nueva_edad):
        if nueva_edad > 0:
            self.__edad = nueva_edad

persona = Persona("Ana", 25)
print(persona.obtener_nombre())  # Ana
# print(persona.__edad)  # Error: atributo privado
```

```java
class Persona {
    private String nombre; // Atributo privado

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

    public String getNombre() { // Método getter
        return nombre;
    }

    public void setNombre(String nombre) { // Método setter
        this.nombre = nombre;
    }
}

public class Main {
    public static void main(String[] args) {
        Persona persona = new Persona("Carlos");
        System.out.println(persona.getNombre()); // Carlos

        persona.setNombre("Ana");
        System.out.println(persona.getNombre()); // Ana
    }
}
```

HERENCIA

La herencia permite que una clase (subclase o clase derivada) obtenga atributos y métodos de otra clase (superclase o clase base). Esto fomenta la reutilización de código y la jerarquía de clases.

Beneficios de la Herencia

- ☑ Reutilización de código: Reduce la duplicación.
- ☑ Extensibilidad: Permite agregar nuevas funcionalidades sin modificar la clase base.
- ☑ Estructuración jerárquica: Facilita la organización del código.

Tipos de Herencia

- 📌 Herencia simple: Una subclase hereda de una sola superclase.
- 📌 Herencia múltiple: Una subclase hereda de múltiples superclases.
- 📌 Herencia multinivel: Una subclase hereda de otra subclase.

HERENCIA

Ejemplo en Python y java:

En este ejemplo, Perro y Gato heredan de Animal y redefinen el método hacer_sonido().

```python
class Animal:
    def __init__(self, nombre):
        self.nombre = nombre

    def hacer_sonido(self):
        return "Sonido genérico"

class Perro(Animal):
    def hacer_sonido(self):
        return "Guau!"

class Gato(Animal):
    def hacer_sonido(self):
        return "Miau!"

perro = Perro("Firulais")
print(perro.hacer_sonido())   # Guau!

gato = Gato("Misu")
print(gato.hacer_sonido())   # Miau!
```

```java
// Superclase
class Animal {
    void hacerSonido() {
        System.out.println("El animal hace un sonido");
    }
}

// Subclase que hereda de Animal
class Perro extends Animal {
    void hacerSonido() { // Sobreescritura del método
        System.out.println("El perro ladra: Guau!");
    }
}

public class Main {
    public static void main(String[] args) {
        Perro miPerro = new Perro();
        miPerro.hacerSonido(); // Salida: El perro ladra: Guau!
    }
}
```

POLIMORFISMO

El polimorfismo es la capacidad de un objeto de tomar múltiples formas. Permite que un mismo método tenga diferentes comportamientos según la clase que lo implemente

Beneficios del Polimorfismo

☑ Flexibilidad: Permite diseñar código más adaptable.

☑ Extensibilidad: Facilita la incorporación de nuevas funcionalidades sin modificar el código existente.

☑ Reutilización: Permite el uso de un mismo método para diferentes tipos de objetos.

Tipos de Polimorfismo

📌 Polimorfismo de Sobreescritura (Override): Una subclase redefine un método de su superclase.

📌 Polimorfismo de Sobrecarga (Overload): Métodos con el mismo nombre pero diferentes parámetros (no soportado en Python de manera nativa).

POLIMORFISMO

Ejemplo en Python y java:

Aquí, Aguila y Pinguino sobrescriben el método volar(), demostrando polimorfismo.

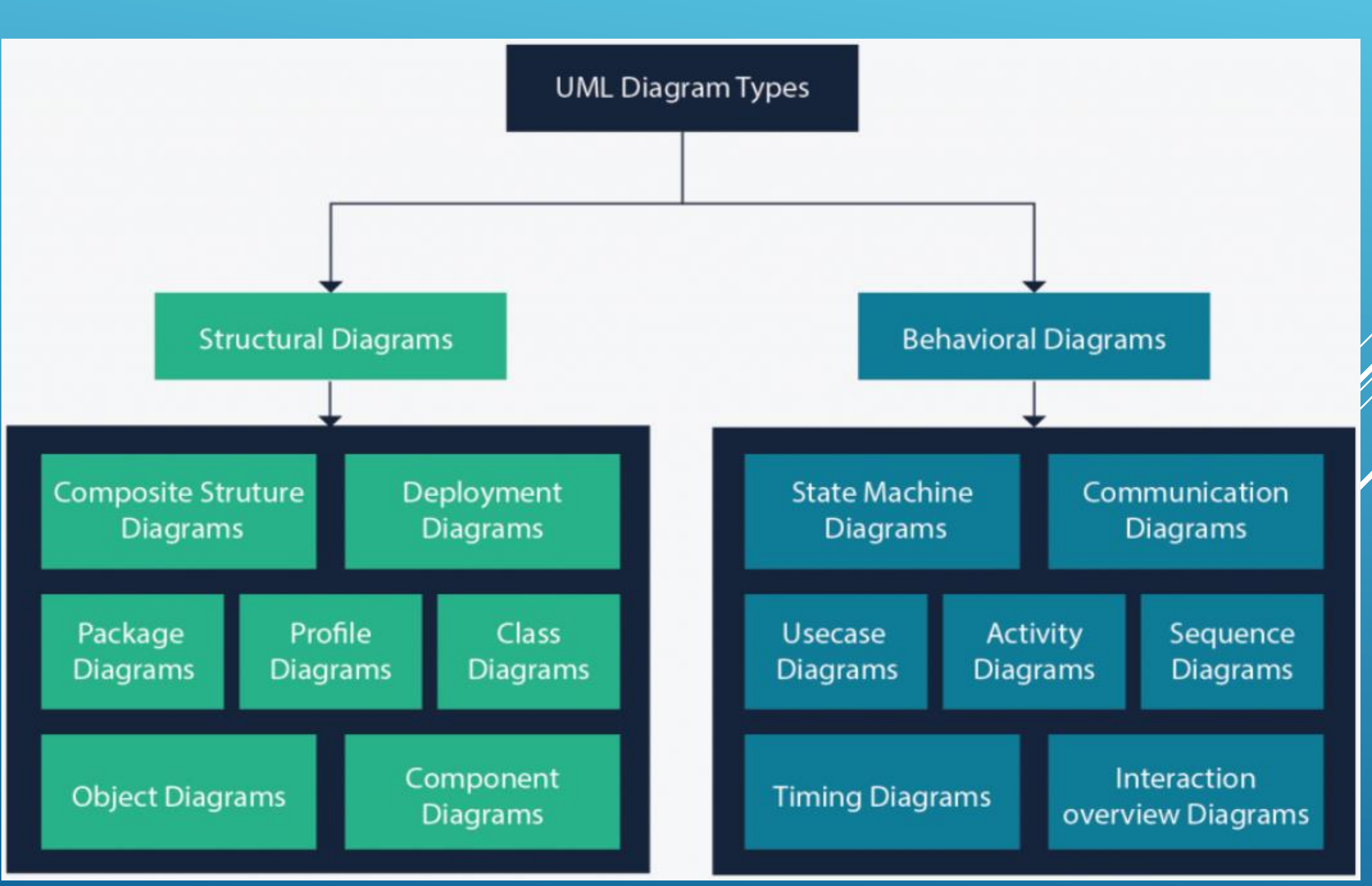
```python
class Ave:
    def volar(self):
        return "Algunas aves vuelan"

class Aguila(Ave):
    def volar(self):
        return "El águila vuela alto"

class Pinguino(Ave):
    def volar(self):
        return "El pingüino no vuela"

aves = [Aguila(), Pinguino()]

for ave in aves:
    print(ave.volar())  # Diferentes comportamientos según la subclase
```

```java
// Superclase
class Animal {
    void hacerSonido() {
        System.out.println("El animal hace un sonido");
    }
}

// Subclases con polimorfismo
class Perro extends Animal {
    void hacerSonido() {
        System.out.println("El perro ladra: Guau!");
    }
}

class Gato extends Animal {
    void hacerSonido() {
        System.out.println("El gato maúlla: Miau!");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal miAnimal; // Referencia de tipo Animal

        miAnimal = new Perro();
        miAnimal.hacerSonido(); // Salida: El perro ladra: Guau!

        miAnimal = new Gato();
        miAnimal.hacerSonido(); // Salida: El gato maúlla: Miau!
    }
}
```

UML (UNIFIED MODELING LANGUAGE)

UML es una notación que se originó como resultado de la unificación de la técnica de modelado de objetos

Tipos de diagramas UML



PREGUNTAS