DIAGRAMAS DE CLASES EN PROGRAMACION ORIENTADA A OBJETOS

Creador de oportunidades

María Paula Ramírez Gómez

Gestor del Conocimiento

William Alexander Matallana

Universidad de Cundinamarca, Extensión Chía

Ingeniería de Sistemas y Computación

Ingeniería de Software I

2026

Contenido

Introducción

La Programación Orientada a Objetos (POO) constituye uno de los paradigmas más utilizados en el desarrollo de software moderno, debido a su capacidad para modelar sistemas complejos a partir de objetos que representan entidades del mundo real. Para facilitar el diseño, comprensión y documentación de estos sistemas, se emplea el Lenguaje Unificado de Modelado (UML), el cual proporciona diferentes tipos de diagramas, siendo uno de los más importantes el diagrama de clases.

El diagrama de clases permite representar gráficamente la estructura estática de un sistema, mostrando las clases que lo conforman, sus atributos, métodos y las relaciones existentes entre ellas. Esto facilita la comunicación entre desarrolladores, analistas y demás involucrados en el proyecto, además de reducir errores antes de iniciar la implementación del software.

En el presente documento se abordará la teoría básica de los diagramas de clases y los principales tipos de relaciones entre clases, destacando su importancia dentro de la ingeniería de software y la programación orientada a objetos.

Desarrollo

Teoria del diagrama de clases

El diagrama de clases es uno de los diagramas estructurales más importantes del Lenguaje Unificado de Modelado (UML) y se utiliza para representar la estructura estática de un sistema orientado a objetos. Su principal función es mostrar las clases que conforman un sistema, junto con sus atributos, métodos y las relaciones que existen entre ellas, permitiendo comprender cómo está organizado el software antes de su implementación.

Cada clase en UML se representa mediante un rectángulo dividido generalmente en tres secciones. En la primera se ubica el nombre de la clase, que normalmente se escribe con la primera letra en mayúscula. En la segunda sección se incluyen los atributos, los cuales representan las características o datos que posee la clase. Finalmente, en la tercera sección se describen los métodos u operaciones, que corresponden a las acciones o comportamientos que puede ejecutar dicha clase.

Además, los diagramas de clases permiten especificar la visibilidad de los atributos y métodos mediante símbolos que indican el nivel de acceso. El símbolo (+) representa elementos públicos accesibles desde cualquier clase; el símbolo (-) indica elementos privados accesibles solo dentro de la misma clase; y el símbolo (#) representa elementos protegidos, visibles únicamente para la clase y sus subclases. Este manejo de visibilidad está directamente relacionado con el principio de encapsulamiento en la programación orientada a objetos, el cual busca proteger la integridad de los datos y controlar su acceso.

El uso de diagramas de clases facilita la planificación del software, permite identificar responsabilidades de cada clase, reduce la probabilidad de errores de diseño y mejora la comunicación entre los miembros del equipo de desarrollo. Asimismo, sirve como documentación técnica del sistema, lo que resulta especialmente útil en etapas posteriores como mantenimiento, escalabilidad o integración de nuevas funcionalidades.

Tipos de Relaciones en Diagramas de Clases

Las relaciones entre clases constituyen un elemento fundamental en los diagramas UML, ya que permiten representar cómo interactúan las distintas entidades del sistema. Estas relaciones ayudan a comprender la dependencia, asociación o jerarquía entre clases, lo cual es clave para diseñar software modular, reutilizable y fácil de mantener.

Asociación:

Es una relación general entre dos o más clases en la que una clase utiliza o se comunica con otra. Por ejemplo, un cliente que realiza un pedido en un sistema de ventas.

Ejemplo:

Un **Cliente** realiza un **Pedido**. El cliente puede existir sin pedidos y el pedido sin ese cliente específico (conceptualmente).

```
class Cliente {
    String nombre;
}

class Pedido {
    Cliente cliente;
}
```

Agregación:

Representa una relación de tipo “todo-parte” donde una clase contiene a otra, pero ambas pueden existir de manera independiente. Un ejemplo es un equipo compuesto por jugadores, donde los jugadores pueden existir sin el equipo.

Ejemplo:

Un **Equipo** tiene **Jugadores**, pero los jugadores pueden existir sin el equipo.

```java
class Jugador {
    String nombre;
}

class Equipo {
    List<Jugador> jugadores;
}
```

Composición:

Es una forma más fuerte de agregación. En este caso, si el objeto principal desaparece, las partes también lo hacen. Por ejemplo, una casa y sus habitaciones; si la casa deja de existir, las habitaciones también.

Ejemplo:

Una **Casa** tiene **Habitaciones**.

Si la casa desaparece, las habitaciones también.

```java
class Persona {
    String nombre;
}

class Empleado extends Persona {
    double salario;
}
```

Herencia o generalización:

Permite que una clase hija herede atributos y métodos de una clase padre. Esta relación favorece la reutilización del código y la organización jerárquica de las clases.

Ejemplo:

Un **Empleado** hereda de la clase **Persona**.

```java
class Persona {
    String nombre;
}

class Empleado extends Persona {
    double salario;
}
```

Dependencia:

Se presenta cuando una clase utiliza temporalmente a otra para realizar alguna operación, sin establecer una relación permanente. Generalmente ocurre cuando una clase recibe otra como parámetro o la usa dentro de un método.

Ejemplo:

Una **Factura** usa una **Calculadora** para obtener el total.

```java
class Calculadora {
    double calcularTotal(double valor) {
        return valor * 1.19;
    }
}

class Factura {
    void generar() {
        Calculadora calc = new Calculadora();
        calc.calcularTotal(100);
    }
}
```

Conclusiones

Los diagramas de clases son una herramienta esencial en la ingeniería de software, ya que permiten visualizar de forma clara la estructura de un sistema antes de su implementación. Esto facilita la planificación, mejora la comunicación entre los miembros del equipo y contribuye a reducir errores durante el desarrollo.

Asimismo, comprender los diferentes tipos de relaciones entre clases permite diseñar sistemas orientados a objetos más organizados, escalables y mantenibles, lo cual impacta directamente en la calidad y eficiencia del software desarrollado.

Referencias

Sommerville, I. (2011). *Ingeniería del software* (9.ª ed.). Pearson Educación.

Larman, C. (2003). *UML y patrones: Introducción al análisis y diseño orientado a objetos y al desarrollo iterativo* (3.ª ed.). Prentice Hall.

Fowler, M. (2004). *UML esencial: Un breve manual del lenguaje estándar de modelado de objetos* (3.ª ed.). Addison-Wesley.

Pressman, R. S. (2010). *Ingeniería del software: Un enfoque práctico* (7.ª ed.). McGraw-Hill.

Object Management Group. (2017). *Especificación del Lenguaje Unificado de Modelado (UML) versión 2.5.1*. https://www.omg.org/spec/UML/

Ejercicio de práctica

Planteamiento del ejercicio

Se desea modelar un sistema sencillo de gestión de pedidos para una tienda en línea. En este sistema, un pedido está compuesto por varios productos, y cada producto tiene información básica como nombre y precio. Se busca representar esta estructura mediante un diagrama de clases y posteriormente implementarlo en Java aplicando conceptos de programación orientada a objetos.

En este caso se utilizará una relación de **composición**, ya que un pedido está formado por productos y estos dependen directamente del pedido dentro del contexto del sistema.

Diagrama de clases: