TECNOLÓGICA DEL ORIENTE
INSTITUCIÓN DE EDUCACIÓN SUPERIOR

PROGRAMACIÓN ORIENTADA A OBJETOS

INTERACCIÓN ENTRE OBJETOS

INTERACCIÓN ENTRE OBJETOS

En un sistema orientado a objetos, los objetos no existen de forma aislada, sino que interactúan y se comunican entre sí para realizar tareas y cumplir con los requisitos del sistema. La interacción entre objetos se produce a través de las asociaciones, que son relaciones entre clases que describen cómo los objetos de una clase se conectan o se relacionan con los objetos de otra clase. Estas asociaciones pueden ser de diferentes tipos, como asociación simple, agregación o composición, dependiendo de la naturaleza de la relación y la dependencia entre los objetos involucrados.

La comunicación entre objetos se realiza mediante el envío de mensajes, que son solicitudes enviadas a un objeto para que ejecute uno de sus métodos. Cuando un objeto recibe un mensaje, invoca el método correspondiente y realiza las acciones solicitadas. Este proceso de envío de mensajes y ejecución de métodos permite que los objetos colaboren y trabajen juntos para lograr los objetivos del sistema. Además, los objetos pueden pasarse como parámetros en los métodos y también pueden ser devueltos como resultados, lo que permite una mayor flexibilidad y reutilización en la interacción entre objetos.

Para reforzar los conceptos generales sobre interacción entre objetos, le invitamos a ver el siguiente vídeo en YouTube.

Inostroza, P. (2020, 26 de diciembre) Clase 22: Clases y Objetos: Interacción entre objetos. [Vídeo] YouTube. https://youtu.be/FLjwjxYZ9Ro

Asociaciones entre clases

En la programación orientada a objetos, las asociaciones representan las relaciones entre las clases. Describen cómo los objetos de una clase están relacionados con los objetos de otra clase. Las asociaciones pueden ser unidireccionales o bidireccionales y pueden tener diferentes multiplicidades, como uno a uno, uno a muchos, muchos a uno y muchos a muchos.

Según Moreno Pérez (2015), las asociaciones son uno de los principales mecanismos para lograr la funcionalidad requerida en un sistema orientado a objetos. Las asociaciones permiten que los objetos colaboren y dependan unos de otros para realizar tareas complejas. Permiten el modelado de relaciones del mundo real y promueven la reutilización del código.

En el desarrollo de **software**, las asociaciones se utilizan ampliamente para modelar las relaciones entre diferentes entidades del sistema. Por ejemplo, en un sistema de gestión universitaria, podría haber una asociación "muchos a muchos" entre las clases "Estudiante" y "Curso", indicando que un estudiante puede inscribirse en muchos cursos y un curso puede tener muchos estudiantes.

Figura 1. Ejemplo en Java

```java
class Estudiante {
    private String nombre;
    private List<Curso> cursos;

    public Estudiante(String nombre) {
        this.nombre = nombre;
        this.cursos = new ArrayList<>();
    }

    public void inscribirEnCurso(Curso curso) {
        cursos.add(curso);
        curso.agregarEstudiante(this);
    }
}
```

```java
class Curso {
    private String nombre;
    private List<Estudiante> estudiantes;

    public Curso(String nombre) {
        this.nombre = nombre;
        this.estudiantes = new ArrayList<>();
    }

    public void agregarEstudiante(Estudiante estudiante) {
        estudiantes.add(estudiante);
    }
}

// Uso
Estudiante estudiante1 = new Estudiante("John");
Estudiante estudiante2 = new Estudiante("Jane");

Curso curso1 = new Curso("Matemáticas");
Curso curso2 = new Curso("Historia");

estudiante1.inscribirEnCurso(curso1);
estudiante1.inscribirEnCurso(curso2);
estudiante2.inscribirEnCurso(curso1);
```

En este ejemplo, las clases “Estudiante” y “Curso” tienen una asociación bidireccional "**muchos a muchos**". Cada “Estudiante” mantiene una lista de “Curso” en los que está inscrito, y cada “Curso” mantiene una lista de “Estudiante” inscritos. Los métodos “inscribirEnCurso” y “agregarEstudiante” se utilizan para establecer la asociación de ambos lados cuando un estudiante se inscribe en un curso.

Mensajes entre objetos (invocación de métodos)

En la programación orientada a objetos, los mensajes son la forma en que los objetos se comunican e interactúan entre sí. Un mensaje es una solicitud de un objeto a otro objeto para realizar una acción o devolver información. En la mayoría de los lenguajes de programación orientados a objetos, los mensajes se envían invocando los métodos de un objeto.

De acuerdo con Oviedo Regino (2015), el envío de mensajes es el proceso mediante el cual un objeto solicita a otro que ejecute uno de sus métodos. Cuando un objeto recibe un mensaje, busca el correspondiente método en su clase (o en sus superclases) y lo ejecuta. Este mecanismo permite que los objetos colaboren para realizar tareas complejas.

En el desarrollo de **software**, el envío de mensajes es fundamental para implementar la funcionalidad del sistema. Los objetos se comunican enviando mensajes entre sí, solicitando servicios y coordinando sus actividades. Por ejemplo, en un sistema de procesamiento de pedidos, un objeto "Pedido" podría enviar un mensaje a un objeto "Inventario" para comprobar la disponibilidad de un producto, y luego enviar un mensaje a un objeto "Envío" para organizar el envío del producto.

Figura 2. Ejemplo en Java

```java
class Inventario {
    private Map<String, Integer> stock;

    public Inventario() {
        stock = new HashMap<>();
        stock.put("Producto1", 10);
        stock.put("Producto2", 5);
    }

    public boolean comprobarDisponibilidad(String producto, int cantidad) {
        if (stock.containsKey(producto) && stock.get(producto) >= cantidad) {
            return true;
        }
        return false;
    }

    public void reducirStock(String producto, int cantidad) {
        if (comprobarDisponibilidad(producto, cantidad)) {
            stock.put(producto, stock.get(producto) - cantidad);
        }
    }
}

class Envio {
    public void organizarEnvio(String producto, String direccion) {
        System.out.println("Organizando envío de " + producto + " a " + direccion);
    }
}
class Pedido {
    private Inventario inventario;
    private Envio envio;

    public Pedido(Inventario inventario, Envio envio) {
        this.inventario = inventario;
        this.envio = envio;
    }

    public void procesarPedido(String producto, int cantidad, String direccion) {
        if (inventario.comprobarDisponibilidad(producto, cantidad)) {
            inventario.reducirStock(producto, cantidad);
            envio.organizarEnvio(producto, direccion);
        } else {
            System.out.println("Producto no disponible: " + producto);
        }
    }
}

// Uso
Inventario inventario = new Inventario();
Envio envio = new Envio();
Pedido pedido = new Pedido(inventario, envio);

pedido.procesarPedido("Producto1", 2, "123 Calle Principal");
```

En este ejemplo, el objeto “Pedido” envía mensajes a los objetos “Inventario” y “Envio” para procesar un pedido. Primero comprueba la disponibilidad del producto en el inventario, luego reduce el stock si el producto está disponible, y finalmente organiza el envío. Esto demuestra cómo los objetos colaboran a través del envío de mensajes para realizar una tarea compleja.

Acoplamiento y cohesión

El acoplamiento y la cohesión son dos principios importantes en el diseño orientado a objetos. El acoplamiento se refiere al grado de interdependencia entre los módulos o clases, mientras que la cohesión se refiere al grado en que los elementos de un módulo o clase están relacionados entre sí. Un buen diseño orientado a objetos tiene como objetivo un bajo acoplamiento y una alta cohesión.

Según López Goytia (2015), el acoplamiento es la medida de la fuerza de la asociación establecida por una conexión de un módulo a otro. Un acoplamiento alto indica una fuerte dependencia entre las clases, lo que hace que el sistema sea más difícil de entender, mantener y modificar. Por otro lado, la cohesión es una medida de cuán relacionadas y enfocadas están las responsabilidades de un elemento (López Goytia, 2015). Una alta cohesión significa que los elementos de una clase están estrechamente relacionados y trabajan juntos para lograr un único propósito.

En la ingeniería de **software**, el acoplamiento y la cohesión son principios rectores para el diseño de sistemas. Un bajo acoplamiento se logra minimizando las dependencias entre las clases, usando abstracciones, y aplicando el principio de inversión de dependencias. Una alta cohesión se logra asegurando que cada clase tiene una única responsabilidad bien definida y que todos sus métodos y propiedades están estrechamente alineados con esa responsabilidad. Esto resulta en un sistema que es más fácil de entender, mantener y modificar.

Figura 3. Ejemplo en Java

```java
// Acoplamiento alto
class Motor {
    public void encender() {
        // lógica para encender el motor
    }
}

class Coche {
    private Motor motor;

    public Coche() {
        motor = new Motor();
    }

    public void arrancar() {
        motor.encender();
    }
}

// Acoplamiento bajo
interface Arrancable {
    void encender();
}

class MotorElectrico implements Arrancable {
    public void encender() {
        // lógica para encender el motor eléctrico
    }
}

class Coche2 {
    private Arrancable motor;

    public Coche2(Arrancable motor) {
        this.motor = motor;
    }

    public void arrancar() {
        motor.encender();
    }
}

// Uso
MotorElectrico motorElectrico = new MotorElectrico();
Coche2 coche = new Coche2(motorElectrico);
coche.arrancar();
```

En este ejemplo, la primera implementación “Coche” tiene un alto acoplamiento con la clase “Motor”. Esto hace que sea difícil cambiar el tipo de motor sin modificar la clase

"Coche". La segunda implementación "Coche2" tiene un bajo acoplamiento gracias al uso de la interfaz "Arrancable". Esto permite que el tipo de motor se cambie fácilmente sin afectar a la clase "Coche2", lo que resulta en un diseño más flexible y mantenible.

Composición y agregación

La composición y la agregación son dos tipos de asociaciones que representan relaciones "todo-parte" entre clases. Ambos implican que una clase (el "todo") está compuesta por una o más instancias de otra clase (las "partes"), pero difieren en la naturaleza de esta relación.

De acuerdo con Ruiz Rodríguez (2009), la composición es una relación todo-parte en la que las partes no pueden existir independientemente del todo. En otras palabras, el ciclo de vida de la parte está completamente gestionado por el todo. Cuando el todo se destruye, todas sus partes también se destruyen. Por otro lado, la agregación es una relación todo-parte en la que las partes pueden existir independientemente del todo (Ruiz Rodríguez, 2009). En una agregación, el todo no gestiona el ciclo de vida de las partes, y si el todo se destruye, las partes pueden seguir existiendo por sí mismas.

En la ingeniería de **software**, la composición y la agregación son patrones importantes para modelar relaciones complejas entre objetos. La composición se utiliza cuando existe una fuerte relación de pertenencia y el todo es responsable de la creación y destrucción de las partes. La agregación se utiliza cuando la relación es más débil y las partes pueden existir independientemente del todo. Por ejemplo, una clase "Universidad" podría tener una composición de "Departamentos" (ya que un departamento no puede existir sin una universidad), pero una agregación de "Estudiantes" (ya que un estudiante puede existir independientemente de una universidad específica).

Figura 4. Ejemplo en Java

```java
// Composición
class Universidad {
    private List<Departamento> departamentos;

    public Universidad() {
        departamentos = new ArrayList<>();
    }

    public void agregarDepartamento(Departamento departamento) {
        departamentos.add(departamento);
    }
}

class Departamento {
    private String nombre;

    public Departamento(String nombre) {
        this.nombre = nombre;
    }
}
```

```java
// Agregación
class Estudiante {
    private String nombre;

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

class Universidad2 {
    private List<Estudiante> estudiantes;

    public Universidad2() {
        estudiantes = new ArrayList<>();
    }

    public void matricularEstudiante(Estudiante estudiante) {
        estudiantes.add(estudiante);
    }
}
```

```
// Uso
Universidad universidad = new Universidad();
Departamento departamento1 = new Departamento("Ciencia de la Computación");
Departamento departamento2 = new Departamento("Matemáticas");
universidad.agregarDepartamento(departamento1);
universidad.agregarDepartamento(departamento2);

Estudiante estudiante1 = new Estudiante("John");
Estudiante estudiante2 = new Estudiante("Jane");
Universidad2 universidad2 = new Universidad2();
universidad2.matricularEstudiante(estudiante1);
universidad2.matricularEstudiante(estudiante2);
```

En este ejemplo, la clase “Universidad” tiene una composición de “Departamento”. Cuando se crea una “Universidad”, se crea una lista vacía de departamentos, y los departamentos se añaden a la universidad. Por otro lado, la clase “Universidad2” tiene una agregación de “Estudiante”. Los estudiantes se crean independientemente y luego se matriculan en la universidad. Esto refleja la naturaleza más débil de la relación en una agregación.

Uso de objetos como parámetros y retorno de métodos

En la programación orientada a objetos, los objetos no sólo representan datos, sino que también encapsulan comportamiento en forma de métodos. Estos métodos a menudo necesitan comunicarse y colaborar con otros objetos para realizar sus tareas. Esto se logra pasando objetos como parámetros a los métodos y devolviendo objetos como valores de retorno de los métodos.

De acuerdo con Vélez Serrano (2011), los objetos se pueden pasar como parámetros a los métodos de la misma manera que los tipos de datos primitivos. Cuando se pasa un objeto como parámetro, en realidad se pasa una referencia a ese objeto. Esto permite que el método acceda y manipule el estado del objeto. De manera similar, los métodos pueden devolver objetos como valores de retorno (Vélez Serrano, 2011), lo que permite una mayor flexibilidad y expresividad en el diseño de las interacciones entre objetos.

En la ingeniería de **software**, pasar objetos como parámetros y devolverlos como valores de retorno es una técnica fundamental para diseñar sistemas flexibles y modulares. Permite que los objetos colaboren de formas complejas sin exponerlos a los detalles internos de los demás. Por ejemplo, un método "procesarPago" en un objeto "Cesta" podría tomar un objeto "Pago" como parámetro y devolver un objeto "Recibo" como resultado.

Figura 5. Ejemplo en Java

```
class Producto {
    private String nombre;
    private double precio;

    public Producto(String nombre, double precio) {
        this.nombre = nombre;
        this.precio = precio;
    }

    public String getNombre() {
        return nombre;
    }

    public double getPrecio() {
        return precio;
    }
}
```

```
class LineaItem {
    private Producto producto;
    private int cantidad;

    public LineaItem(Producto producto, int cantidad) {
        this.producto = producto;
        this.cantidad = cantidad;
    }

    public Producto getProducto() {
        return producto;
    }

    public int getCantidad() {
        return cantidad;
    }

    public double getTotal() {
        return producto.getPrecio() * cantidad;
    }
}

class Recibo {
    private List<LineaItem> items;
    private double total;

    public Recibo(List<LineaItem> items) {
        this.items = items;
        this.total = items.stream().mapToDouble(LineaItem::getTotal).sum();
    }

    public double getTotal() {
        return total;
    }
}

class Cesta {
    private List<LineaItem> items;

    public Cesta() {
        items = new ArrayList<>();
    }

    public void addItem(Producto producto, int cantidad) {
        LineaItem item = new LineaItem(producto, cantidad);
        items.add(item);
    }

    public Recibo checkout() {
        return new Recibo(items);
    }
}
// Uso
Producto producto1 = new Producto("Leche", 2.50);
Producto producto2 = new Producto("Pan", 1.75);

Cesta cesta = new Cesta();
cesta.addItem(producto1, 2);
cesta.addItem(producto2, 1);

Recibo recibo = cesta.checkout();
System.out.println("Total: " + recibo.getTotal()); // Total: 6.75
```

En este ejemplo, la clase “Cesta” tiene un método “addItem” que toma objetos “Producto” e “int” como parámetros, y un método “checkout” que devuelve un objeto “Recibo”. El objeto “Recibo” a su vez se construye a partir de una lista de objetos “LineaItem”, cada uno de los cuales encapsula un objeto “Producto” y una cantidad. Este diseño permite una expresiva modelización del proceso de compra sin exponer los detalles internos de cada clase.

Los fundamentos de la programación orientada a objetos, como los conceptos de clases y objetos, la encapsulación, la abstracción y la interacción entre objetos, son pilares esenciales para el desarrollo de **software** moderno y eficiente. Comprender y aplicar estos principios es crucial para los profesionales de la ingeniería de **software**, dado que les permite diseñar y construir sistemas robustos, modulares y fácilmente mantenibles.

En el contexto laboral, la POO se utiliza ampliamente en el desarrollo de aplicaciones de **software** en diversos dominios, como la banca, la salud, el comercio electrónico, la industria manufacturera y muchos otros. Los ingenieros de **software** emplean los conceptos de clases y objetos para modelar y representar entidades del mundo real, como usuarios, productos, pedidos o transacciones. La encapsulación les permite ocultar los detalles internos de las clases y exponer solo las interfaces necesarias, lo que facilita la gestión de la complejidad y mejora la seguridad del sistema.

La abstracción, a través del uso de clases abstractas e interfaces, permite a los desarrolladores crear diseños flexibles y extensibles. Pueden definir contratos y especificaciones generales que luego pueden ser implementados por diferentes clases concretas, lo que promueve la reutilización de código y la modularidad. Esto es especialmente beneficioso en proyectos a gran escala, donde múltiples equipos trabajan en diferentes componentes del sistema.

Además, la interacción entre objetos es fundamental para construir sistemas funcionales y cohesivos. Los desarrolladores utilizan asociaciones, como la agregación y la composición, para establecer relaciones entre clases y modelar la colaboración entre objetos. Esto les permite diseñar sistemas que reflejen de manera precisa las interacciones y dependencias del mundo real, lo que conduce a un **software** más intuitivo y fácil de entender.

En resumen, los fundamentos de la POO son herramientas esenciales en las competencias de cualquier ingeniero de **software**. Su dominio y aplicación adecuada conducen a un desarrollo de **software** más eficiente, modular y mantenible. En el entorno laboral, estos principios se utilizan constantemente para abordar desafíos complejos y construir soluciones de **software** robustas y escalables. Por lo tanto, invertir tiempo y esfuerzo en comprender y dominar estos conceptos es una inversión valiosa para cualquier profesional de la ingeniería de **software** que busque fortalecer su formación y contribuir al éxito de los proyectos en los que participa.

Bibliografía

- Inostroza, P. (2020, 26 de diciembre). Clase 22: Clases y Objetos: Interacción entre objetos [Vídeo]. YouTube. https://youtu.be/FLjwjxYZ9Ro
- López Goytia, J. L. (2015). Programación orientada a objetos C++ y Java: un acercamiento interdisciplinario. Grupo Editorial Patria. https://elibro.net/es/ereader/tecnologicadeloriente/39461
- Moreno Pérez, J. C. (2015). Programación orientada a objetos. RA-MA Editorial. https://elibro.net/es/ereader/tecnologicadeloriente/106461
- Oviedo Regino, E. M. (2015). Lógica de programación orientada a objetos. Ecoe Ediciones. https://elibro.net/es/ereader/tecnologicadeloriente/70431
- Ruiz Rodríguez, R. (2009). Fundamentos de la programación orientada a objetos: una aplicación a las estructuras de datos en Java. El Cid Editor. https://elibro.net/es/ereader/tecnologicadeloriente/34869
- Vélez Serrano, J. (2011). Diseñar y programar, todo es empezar: una introducción a la Programación Orientada a Objetos usando UML y Java. Dykinson. https://elibro.net/es/ereader/tecnologicadeloriente/63076