



PROGRAMACIÓN ORIENTADA A OBJETOS

EJEMPLOS PRÁCTICOS

COLECCIONES POLIMÓRFICAS, CONTROLADORES Y SÍNTESIS DE LA UNIDAD

Las colecciones polimórficas y los controladores son dos ejemplos prácticos de cómo el polimorfismo puede ser utilizado en el diseño de software.

Colecciones polimórficas: En muchos lenguajes de programación, las colecciones (como listas, conjuntos y mapas) se implementan utilizando interfaces genéricas que permiten almacenar y manipular objetos de diferentes tipos de manera polimórfica (Oviedo Regino, 2015).

Por ejemplo, en Java, la interfaz "List<E>" define un contrato para listas genéricas que pueden contener elementos de cualquier tipo:

Figura 1. Interfaz List.

```
java
List<Figura> figuras = new ArrayList<>();
figuras.add(new Circulo(5));
figuras.add(new Rectangulo(4, 6));
figuras.add(new Triangulo(3, 8));
```

En este caso, la lista "figuras" puede contener instancias de "Circulo", "Rectangulo", "Triangulo" o cualquier otra clase que implemente la interfaz "Figura". Esto permite escribir código genérico que opere sobre la lista de figuras sin preocuparse por los tipos específicos de cada elemento.

Controladores: En el patrón de diseño Modelo-Vista-Controlador (MVC), los controladores son responsables de procesar las solicitudes del usuario y de orquestar la interacción entre el modelo y la vista. El polimorfismo se puede utilizar para diseñar controladores flexibles y extensibles (Ruiz Rodríguez, 2009).

Por ejemplo, se puede definir una interfaz "Controlador" que declare los métodos comunes a todos los controladores:

Figura 2. Interfaz Controlador.

```
java
public interface Controlador {
    void manejarSolicitud(HttpServletRequest request, HttpServletResponse response);
}

public class ControladorUsuario implements Controlador {
    @Override
    public void manejarSolicitud(HttpServletRequest request, HttpServletResponse response) {
        // lógica para manejar solicitudes relacionadas con usuarios
    }
}

public class ControladorProducto implements Controlador {
    @Override
    public void manejarSolicitud(HttpServletRequest request, HttpServletResponse response) {
        // lógica para manejar solicitudes relacionadas con productos
    }
}
```

Al utilizar la interfaz "Controlador", se puede escribir código genérico que trabaje con diferentes tipos de controladores de manera polimórfica. Por ejemplo, se podría tener un "DispatcherServlet" que reciba todas las solicitudes y las delegue al controlador apropiado basándose en la URL:

Figura 3. DispatcherServlet

```
java
public class DispatcherServlet extends HttpServlet {
    private Map<String, Controlador> controladores;

    @Override
    protected void service(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        String url = request.getRequestURI();
        Controlador controlador = controladores.get(url);
        controlador.manejaSolicitud(request, response);
    }
}
```

En este ejemplo, el "DispatcherServlet" utiliza un mapa para almacenar las instancias de los controladores y delega las solicitudes al controlador apropiado según la URL. Gracias al polimorfismo, el "DispatcherServlet" puede trabajar con cualquier tipo de controlador que implemente la interfaz "Controlador", lo que facilita la extensibilidad del sistema. Si se necesita añadir un nuevo tipo de controlador, solo se requiere crear una nueva clase que implemente la interfaz y registrarla en el mapa de controladores.

En esta unidad, se ha explorado dos conceptos fundamentales de la programación orientada a objetos: la herencia y el polimorfismo. Se ha visto cómo la herencia permite crear jerarquías de clases y reutilizar código, mientras que el polimorfismo permite diseñar sistemas flexibles y extensibles que pueden trabajar con objetos de diferentes tipos de manera abstracta.

Se inició examinando las jerarquías de herencia, incluyendo conceptos como la herencia simple y múltiple, el uso de la palabra clave "extends" (o equivalente), la herencia de atributos y métodos, y los constructores en la herencia. También se discutieron algunos problemas comunes que pueden surgir al utilizar la herencia de manera incorrecta, como la herencia excesiva o mal aplicada.

Luego, se adentró en la sobrecarga y sobrescritura de métodos, que son dos mecanismos clave para implementar el polimorfismo. Se aprendió sobre las diferencias entre la sobrecarga y la sobrescritura, el uso del modificador "override", los conceptos de polimorfismo estático y dinámico, las reglas de visibilidad y tipo que deben seguirse al sobrescribir métodos.

En la última parte de la unidad, se exploró el diseño basado en polimorfismo y cómo puede ser utilizado para crear sistemas más mantenibles y extensibles. Se discutió el uso de interfaces y clases abstractas como contratos, la sustitución de objetos en tiempo de ejecución, la aplicación del principio de sustitución de Liskov y las ventajas del polimorfismo en el mantenimiento del software. También se presentaron ejemplos prácticos, como las colecciones polimórficas y los controladores en el patrón MVC.

En resumen, la herencia y el polimorfismo son herramientas poderosas que, cuando se utilizan adecuadamente, pueden mejorar significativamente la calidad y mantenibilidad del software. Al comprender y aplicar estos conceptos, los desarrolladores pueden crear sistemas más flexibles, extensibles y fáciles de mantener a largo plazo.

La importancia de estos conceptos se extiende más allá del ámbito académico y tiene una gran relevancia en el desarrollo de software en el mundo real. En el contexto laboral, los desarrolladores a menudo se enfrentan a sistemas complejos y en constante evolución, donde la capacidad de escribir código mantenible y extensible es crucial.



Al aplicar los principios de la herencia y el polimorfismo, los desarrolladores pueden crear abstracciones de alto nivel que permitan a los sistemas adaptarse a los cambios de requisitos y que faciliten la incorporación de nuevas funcionalidades sin afectar al código existente. Esto se traduce en un menor coste de mantenimiento y evolución del software, así como en una mayor agilidad para responder a las necesidades cambiantes del negocio.

Además, el uso adecuado de la herencia y el polimorfismo fomenta la reutilización de código, lo que reduce la duplicación y mejora la consistencia en todo el sistema. Esto no solo ahorra tiempo y esfuerzo a los desarrolladores, sino que también contribuye a la creación de un software más robusto y menos propenso a errores.

En un mercado laboral cada vez más competitivo, los desarrolladores que dominen estos conceptos y sean capaces de aplicarlos de manera efectiva tendrán una ventaja significativa. Las empresas buscan profesionales que puedan escribir código de alta calidad, mantenible y escalable, y el dominio de la herencia y el polimorfismo es un indicador clave de esa capacidad.

En conclusión, la herencia y el polimorfismo son pilares fundamentales de la programación orientada a objetos y su dominio es esencial para cualquier desarrollador que aspire a crear software de calidad en el ámbito profesional. Al comprender y aplicar estos conceptos, los desarrolladores pueden crear sistemas más flexibles, mantenibles y adaptables, lo que les permitirá enfrentar con éxito los desafíos del desarrollo de software en el mundo laboral.

Bibliografía

-  Oviedo Regino, E. M. (2015). Lógica de programación orientada a objetos. Ecoe Ediciones. <https://elibro.net/es/ereader/tecnologicadeloriente/70431?page=1>
-  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?page=9>