PROGRAMACIÓN ORIENTADA A OBJETOS

Polimorfismo y Clases Abstractas

Integrantes:

Jaya Taipe Jefferson Fernando

Ortiz Córdova Nick Mateo

Proaño Panchi Isaac Sebastián

Vilaña Puga Nayeli Estefanía

NRC: 1322

Docente: Ing. Luis Enrique Jaramillo

Fecha: 17/02/2025

Objetivo general:

Entender y utilizar los principios de polimorfismo y clases abstractas en la creación de sistemas orientados a objetos con el fin de optimizar la reutilización, adaptabilidad y preservación del código, fomentando buenas prácticas de programación y favoreciendo la escalabilidad del software.

Objetivos específicos:

- Examinar la utilización de clases abstractas para construir jerarquías de clases reutilizables.
- Describir la noción de polimorfismo y su influencia en la adaptabilidad del diseño de software.
- Poner en práctica ejemplos concretos que demuestren la aplicación de polimorfismo y clases abstractas en un lenguaje de programación enfocado en objetos.

Marco Teórico

El polimorfismo se refiere a la habilidad de un objeto para adoptar múltiples formas, y en el ámbito de la programación, implica que un objeto puede ser considerado como un objeto de diversas clases, en función del contexto.

Existen múltiples métodos para aplicar el polimorfismo en Python, tales como la sobrecarga de operadores, la herencia de clases y el uso de interfaces gráficas. Por ejemplo, en la herencia de clases, se pueden crear subclases que heredan métodos y propiedades de una clase base y también pueden proporcionar su propia implementación de esos métodos. De esta forma, una subclase puede ser tratada como un objeto de la clase base, pero también puede proporcionar su propia lógica. Existen dos tipos principales de polimorfismo:

Polimorfismo en tiempo de compilación:

Se logra a través de la sobrecarga de métodos u operadores. El compilador decide cuál método usar en función de los parámetros o el contexto.

Polimorfismo en tiempo de ejecución:

Se logra a través de la herencia y la sobreescritura de métodos. La decisión de qué método ejecutar se toma en tiempo de ejecución, dependiendo del tipo de objeto que invoque el método.

Ventajas:

Mejora la flexibilidad y la escalabilidad del código.

Permite usar el mismo nombre de función para diferentes tipos de objetos, lo que hace el código más limpio y fácil de mantener.

Ejemplo práctico:

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

// Clase derivada 1
class Perro extends Animal {
    @Override
    void hacerSonido() {
        System.out.println("El perro ladra");
    }
}

// Clase derivada 2
class Gato extends Animal {
    @Override
    void hacerSonido() {
        System.out.println("El gato maulla");
    }
}
```

```java
public class PolimorfismoEjemplo {
    public static void main(String[] args) {
        // Polimorfismo en tiempo de ejecución
        Animal miAnimal;

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

        miAnimal = new Gato();
        miAnimal.hacerSonido(); // Salida: El gato maulla

        // Polimorfismo en tiempo de compilación
        System.out.println(sumar(5, 10)); // Salida: 15
        System.out.println(sumar(5.5, 10.5)); // Salida: 16.0
    }

    // Sobrecarga de métodos (polimorfismo en tiempo de compilación)
    static int sumar(int a, int b) {
        return a + b;
    }

    static double sumar(double a, double b) {
        return a + b;
    }
}
```

Una clase abstracta es una clase que no puede asignarse de manera directa, sino que únicamente puede emplearse como una clase progenitor para generar clases descendientes. El concepto es que las clases descendientes deben ofrecer aplicaciones específicas para los métodos abstractos establecidos en la clase abstracta.

Ejemplo:

```python
 import abc

class ClaseAbstracta(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def metodo_abstracto(self):
        pass

class ClaseConcreta(ClaseAbstracta):
    def metodo_abstracto(self):
        return "Implementación concreta del método abstracto"
```

En este caso, Clase Abstracta representa una clase abstracta mientras que Clase Concreta es una clase hija que ofrece una aplicación específica del método abstracto establecido en Clase Abstracta.

Características principales de las clases abstractas:

No se pueden instanciar: No es posible crear un objeto directamente a partir de una clase abstracta.

Métodos abstractos: Una clase abstracta puede incluir métodos abstractos, que son aquellos sin implementación. Las clases derivadas deben ofrecer su propia implementación de estos métodos.

Métodos concretos: También puede tener métodos con implementación completa, que pueden ser heredados tal como están por las clases derivadas.

Herencia: Las clases abstractas suelen ser heredadas por otras clases que deben implementar los métodos abstractos que se han definido.

Ventajas de las clases abstractas:

Estructura común: Ofrecen una interfaz estándar para todas las clases que heredan de ellas.

Evitan errores: Obligan a las clases derivadas a implementar ciertos métodos, asegurando que no se omitan implementaciones esenciales.

Reutilización de código: Permiten establecer métodos compartidos que las clases derivadas pueden utilizar, disminuyendo la duplicación.

Ejemplo práctico:

```java
// Clase abstracta
abstract class Animal {
    // Método abstracto (sin implementación)
    abstract void hacerSonido();

    // Método concreto (con implementación)
    void dormir() {
        System.out.println("El animal está durmiendo");
    }
}

// Clase derivada 1
class Perro extends Animal {
    @Override
    void hacerSonido() {
        System.out.println("El perro ladra");
    }
}

// Clase derivada 2
class Gato extends Animal {
    @Override
    void hacerSonido() {
        System.out.println("El gato maulla");
    }
}
```

```java
public class ClaseAbstractaEjemplo {
    public static void main(String[] args) {
        // No se puede instanciar la clase abstracta
        // Animal miAnimal = new Animal(); // Esto causaría un error c

        // Crear instancias de las clases derivadas
        Animal miPerro = new Perro();
        Animal miGato = new Gato();

        // Llamar a los métodos
        miPerro.hacerSonido(); // Salida: El perro ladra
        miPerro.dormir();      // Salida: El animal está durmiendo

        miGato.hacerSonido();  // Salida: El gato maulla
        miGato.dormir();       // Salida: El animal está durmiendo
    }
}
```

Recomendaciones:

- Aplicar polimorfismo en sus proyectos mediante sobrecarga de operadores, herencia de clases e interfaces gráficas, para hacer el código más flexible y escalable.
- Implementar clases abstractas al código para edificar una estructura común, obligando a las clases derivadas a implementar métodos esenciales, permitiendo reutilizar métodos en concreto y evitar errores
- Comprender y aplicar los tipos de polimorfismo, dependiendo del objetivo implementar polimorfismo en tiempo de compilación (uso de sobrecarga de métodos) o en tiempo de ejecución (uso de herencia y sobreescritura de métodos).

Conclusiones:

El polimorfismo nos permite mejorar la escalabilidad del código, es decir que nos deja usar el nombre de la función para diferentes tipos de objetos.

Las clases abstractas son útiles para evitar errores, ya que, al obligar a implementar métodos esenciales a las clases derivadas reducen la duplicación de código.

Los pilares de la Programación Orientada a Objetos están estrechamente relacionados, como es el caso del polimorfismo y la herencia. Además, no solo es necesario conocer, sino comprender y aplicar.