Universidad de las Fuerzas Armadas (ESPE)

Programación Orientada a Objetos

Control de lectura 1

Autores:

Israel Fernando Portilla Santamaría

Eduardo Anibal Mejia Catillo

Bryan Roger Camacho Ramirez

Jenifer Andrea Castro Cuevas

Julio Emerson Coro Pineida

Alexis Sebastian Murminacho Cabascango

Nrc: 1323

lunes, 17 de febrero de 2025

Link del repositorio:

https://github.com/IsraelPortilla/Repositorio_Israel_Portilla/tree/main/Grupo%209/Control%20de%20lectura%201

Link del video:

https://drive.google.com/file/d/14j6MRCBteMzlUBeT7EvZ0FDRnrIXQGih/view?usp=drivesdk

Polimorfismo y Clases Abstractas

1. **Objetivo general**

- Investigar, analizar y aplicar el concepto de polimorfismo y clases abstractas en el paradigma de programación orientada a objetos, mencionando su importancia en el diseño de sistemas prácticos en java.

2. **Objetivos específicos**

- Explicar el concepto de polimorfismo enfocado a POO, diferenciando sus tipos y proporcionando ejemplos de su implementación en Java.
- Explicar el uso de clases abstractas en el desarrollo de sistemas con el paradigma de POO, destacando su papel en la estructuración del código y su relación con el polimorfismo.

Introducción

El paradigma orientado a objetos (POO) es una forma de ver un programa como una serie de instrucciones lineales, lo que permite agrupar datos (objetos) y las operaciones relacionadas con ellos, Este paradigma permite estructurar software de manera modular y reutilizable, tiene como objetivo principal modelar problemas del mundo real a través de clases y objetos, facilitando la organización y mantenimiento del código. Dentro de este enfoque, existen cuatro principios fundamentales: abstracción, encapsulamiento, herencia y polimorfismo. Centralizaremos el informe en el polimorfismo y las clases abstractas.

Es uno de sus pilares fundamentales de POO, ya que permite que una misma interfaz sea utilizada para diferentes tipos de objetos esto significa que un mismo método puede comportarse de

manera diferente según la clase que lo implemente, promoviendo la flexibilidad y escalabilidad en el desarrollo de software. Por otro lado, las clases abstractas son un concepto clave que permite definir estructuras comunes sin necesidad de implementar completamente sus funcionalidades. Estas clases sirven como modelos base para que las subclases hereden y proporcionen implementaciones específicas, fomentando así el diseño estructurado y reutilizable de código.

Existen muchos lenguajes de programación orientados a objetos, aunque no están necesariamente restringidos al paradigma de POO. Por ejemplo, algunos lenguajes, como JavaScript, Python y PHP, permiten estilos de programación tanto procedimentales como orientados a objetos.

Desarrollo

Ampliaremos la información acerca de los temas mencionados polimorfismo y clases abstractas, exploraremos el concepto y su relación entre sí, destacando su relevancia en el desarrollo de sistemas orientados a objetos. Además, se incluyen ejemplos prácticos en Java para ilustrar su aplicación y utilidad.

Polimorfismo

El polimorfismo es uno de los pilares básicos en la programación orientada a objetos, el término polimorfismo tiene origen en las palabras poly-muchos y morfo-formas, y aplicado a la programación hace referencia a que los objetos pueden tomar diferentes formas, en otras palabras, significa que objetos de diferentes clases pueden ser accedidos utilizando el mismo interfaz, mostrando un comportamiento distinto según cómo sean necesitados.

En distintos documentos consultados se da cuenta de diferentes tipos de polimorfismos vamos a mencionarlos brevemente, pero los englobaremos en polimorfismo estáticos y dinámico.

En la investigación hemos visto tres tipos de polimorfismo, a saber, de subtipos, paramétrico y ad hoc, algunos agregan un cuarto el casteo, es decir, la conversión dinámica de un tipo de objeto a otro, este polimorfismo de subtipos tiene lugar en tiempo de ejecución y existen dos vertientes de este, a saber, el single dispatch y el multiple dispatch. El primero es su forma más común, y básicamente, lo que *despacho simple* quiere decir es que, cuando el programa se está ejecutando, se decide qué versión del método se va a invocar basándose en el tipo de la referencia del objeto.

El despacho múltiple, en contraposición, consiste en despachar el método correcto teniendo en cuenta el tipo de parámetros o argumentos.

El polimorfismo paramétrico le confiere al programador, en un lenguaje con tipado estático, la capacidad de escribir código en forma general y abstracta.

El polimorfismo ad hoc consiste en la capacidad que tiene una función o método de arrojar un resultado para diferentes tipos de datos que recibe como argumentos. Dicho de otra forma, una función o método polimórfico es capaz de *despachar* un resultado correcto para cada tipo que reciba.

Con estas explicaciones, lo englobaremos en dinámicos y estáticos:

Polimorfismo estático: El polimorfismo estático, también conocido como sobrecarga de métodos, ocurre cuando una clase tiene múltiples métodos con el mismo nombre pero con diferentes parámetros (diferentes tipos, número de parámetros o ambos). Esto permite que un método realice diferentes funciones basándose en los argumentos que se le pasan. Ejemplos

```
1  public class Calculadora {
2      int suma(int a, int b) {
3          return a + b;
4      }
5
6      double suma(double a, double b) {
7          return a + b;
8      }
9  }
```

Polimorfismo dinámico: En POO es el mecanismo mediante el cual se pueden definir múltiples métodos con el mismo nombre y firma en la superclase y subclase. La llamada a un método anulado se resuelve en tiempo de ejecución, En otras palabras, ocurre cuando la selección del método a ejecutar se determina en tiempo de ejecución. Se logra mediante sobreescritura de métodos, donde una subclase redefine un método de la superclase para proporcionar una implementación específica. Ejemplos:

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

class Gato extends Animal {
    @Override
    void sonido() {
        System.out.println("El gato maúlla");
    }
}
```

Ejemplo general de Polimorfismo:

Se creo un sistema que muestra matricula, marca, modelo, numero de puertas, cilindraje y carga según sea el tipo de carro, se implementa polimorfismo en la herencia de la clase padre "vehículo" a las tres subclases hijas que tienen solo 1 atributo implementado y se las hereda los otros 3, además se puede apreciar el polimorfismo en la instancia de las subclases, dado que solo instanciamos la clase padre, por tener la herencia de ella las subclases se instancian también en ese método se podrá ver en las siguientes imágenes.

Clase padre "Vehiculo"

vehiculo.java | vehiculoturismo.java | vehiculodeportivo.j... | vehiculofurgoneta.j... | main.java

```java
public class vehiculo{
    public String matricula;
    public String marca;
    public String modelo;

    public vehiculo(String matricula,String marca, String modelo){
        this.matricula = matricula;
        this.marca = marca;
        this.modelo = modelo;
    }
    public String getmatricula(){
        return matricula;
    }
     public String getmarca(){
        return marca;
     }
     public String getmodelo(){
        return modelo;
     }
     public String mostrardatos(){
         return ("Matricula: "+matricula+"\nMarca: "+marca+"\nModelo: "+modelo);
     }
```

Subclase hija “vehiculoturismo”

vehiculo.java | vehiculoturismo.java | vehiculodeportivo.j... | vehiculofurgoneta.j... | main.java

```java
public class vehiculoturismo extends vehiculo{
  public int puertas;
    public vehiculoturismo(int puertas, String matricula, String marca, String modelo){
        super(matricula, marca, modelo);
        this.puertas = puertas;
    }
    public int getpuertas(){
        return puertas;
    }
    @Override
    public String mostrardatos(){
        return "Matricula: "+matricula+"\nMarca: "+marca+"\nModelo: "+modelo+"\nNumero de puertas: "+puertas;
    }

}
```

Subclase hija “vehiculodeportivo”

vehiculo.java | vehiculoturismo.java | vehiculodeportivo.j... | vehiculofurgoneta.j... | main.java

```java
public class vehiculodeportivo extends vehiculo{
  public int cilindrada ;
    public vehiculodeportivo(int cilindrada, String matricula, String marca, String modelo){
        super(matricula, marca, modelo);
        this.cilindrada = cilindrada;
    }
    public int getcilindrada(){
        return cilindrada;
    }
    @Override
    public String mostrardatos(){
        return "Matricula: "+matricula+"\nMarca: "+marca+"\nModelo: "+modelo+"\nCilindrada: "+cilindrada;
    }

}
```

Subclase hija “vehiculofurgoneta”

vehiculo.java | vehiculoturismo.java | vehiculodeportivo.j... | vehiculofurgoneta.j... | main.java

```java
public class vehiculofurgoneta extends vehiculo{
  public int carga ;
    public vehiculofurgoneta(int carga, String matricula, String marca, String modelo){
        super(matricula, marca, modelo);
        this.carga = carga;
    }
    public int getcarga(){
        return carga;
    }
    @Override
    public String mostrardatos(){
        return "Matricula: "+matricula+"\nMarca: "+marca+"\nModelo: "+modelo+"\nCarga en Kg: "+carga;
    }

}
```

Programa Main: mostrar vehículos con herencia

```
public class main {
    public static void main(String[]args){
    vehiculo misvehiculos[] = new vehiculo [4];
    misvehiculos[0] = new vehiculo("RCH","BMW","X-Trail");
    misvehiculos[1] = new vehiculoturismo(6,"DHG","Ferrari","Avalon");
    misvehiculos[2] = new vehiculodeportivo(500,"JHF","Mercedes Benz","Yaris");
    misvehiculos[3] = new vehiculofurgoneta(200,"CDA","Chevrolet","Jolion");

    for(vehiculo vehiculos: misvehiculos){
        System.out.println(vehiculos.mostrardatos());
        System.out.println("");
    }
}
}
```

Salida del programa

```
Matricula: RCH
Marca: BMW
Modelo: X-Trail

Matricula: DHG
Marca: Ferrari
Modelo: Avalon
Numero de puertas: 6

Matricula: JHF
Marca: Mercedes Benz
Modelo: Yaris
Cilindrada: 500

Matricula: CDA
Marca: Chevrolet
Modelo: Jolion
Carga en Kg: 200
```

Clases abstractas Así en programación las clases abstractas funcionan como una clase que declara la existencia de métodos, pero no su implementación. Eso es algo que hace después en las diferentes subclases derivadas de la clase abstracta. Una clase abstracta puede contener métodos no abstractos, pero al menos uno de los métodos sí debe serlo. Es decir, Las clases abstractas, no representan algo específico y las podemos usar para crear otras clases. No pueden ser instanciadas, por lo que no podemos crear nuevos objetos con ellas. Ejemplo general:

Clase abstracta “Vehiculo”, define el método abstracto y un método implementable

main.java | vehiculo.java | vehiculoturismo.java | vehiculodeportivo.j...

```java
public abstract class vehiculo {
    public String marca;
    public String modelo;

    public vehiculo(String marca, String modelo) {
        this.marca = marca;
        this.modelo = modelo;
    }

    public abstract void mostrarDetalles();
    public void arrancar() {
        System.out.println("El vehículo está arrancando...");
    }
}
```

Subclase heredara del abstract “vehiculoturismo”

main.java | vehiculo.java | vehiculoturismo.java | vehiculodeportivo.j...

```java
class vehiculoturismo extends vehiculo {
    public int puertas;

    public vehiculoturismo(String marca, String modelo, int puertas) {
        super(marca, modelo);
        this.puertas = puertas;
    }

    @Override
    public void mostrarDetalles() {
        System.out.println("Vehículo Turismo - Marca: " + marca + ", Modelo: " + modelo + ", Puertas: " + puertas);
    }
}
```

Subclase heredara del abstract “vehiculodeportivo”

Run | Debug | Stop | Share | Save | { } Beautify | Language Java

main.java | vehiculo.java | vehiculoturismo.java | vehiculodeportivo.j...

```java
class vehiculodeportivo extends vehiculo {
    public int cilindrada;

    public vehiculodeportivo(String marca, String modelo, int cilindrada) {
        super(marca, modelo);
        this.cilindrada = cilindrada;
    }

    @Override
    public void mostrarDetalles() {
        System.out.println("Vehículo Deportivo - Marca: " + marca + ", Modelo: " + modelo + ", Cilindrada: " + cilindrada + "cc");
    }
}
```

Programa Main: mostrar vehículos con métodos del abstract

main.java | vehiculo.java | vehiculoturismo.java | vehiculodeportivo.j...

```java

public class main {
    public static void main(String[] args) {
        vehiculo miTurismo = new vehiculoturismo("Honda", "Civic", 4);
        vehiculo miDeportivo = new vehiculodeportivo("Ferrari", "F8", 600);

        miTurismo.arrancar();
        miTurismo.mostrarDetalles();

        miDeportivo.arrancar();
        miDeportivo.mostrarDetalles();
    }
}

```

Salida del programa

input

```
El vehículo está arrancando...
Vehículo Turismo - Marca: Honda, Modelo: Civic, Puertas: 4
El vehículo está arrancando...
Vehículo Deportivo - Marca: Ferrari, Modelo: F8, Cilindrada: 600cc


...Program finished with exit code 0
Press ENTER to exit console.
```

Conclusiones

El uso de polimorfismo y clases abstractas en la Programación Orientada a Objetos es fundamental para mejorar la flexibilidad, escalabilidad y reutilización del código, permite que un mismo método se comporte de manera distinta según la clase que lo implemente, facilitando la extensibilidad del sistema. Se presenta en dos formas principales; Polimorfismo estático, que permite definir múltiples métodos con el mismo nombre pero distintos parámetros y polimorfismo dinámico, que permite redefinir el comportamiento de un método heredado en una subclase. Las clases abstractas, por otro lado, proporcionan una estructura base para las clases derivadas, asegurando que todas compartan ciertas características esenciales sin permitir su instanciación directa. Esto promueve un diseño más limpio y modular, donde las subclases implementan métodos específicos mientras reutilizan funcionalidades comunes.

En el desarrollo de software, aplicar correctamente estos conceptos permite crear sistemas más organizados, mantenibles y extensibles, ya que el código puede evolucionar sin necesidad de grandes modificaciones. Estos principios son esenciales en la construcción de arquitecturas robustas, especialmente en proyectos grandes y escalables.

Nota: todos los programas mostrados se encuentran en el repositorio indicado al inicio del documento.

Bibliografía

Bernal Gomez, M.(2014). Polimorfismo.
URL.https://repositorio.konradlorenz.edu.co/handle/001/3970

Hartman, J. (2024, noviembre 25). *Polimorfismo en Java (Estático y dinámico)*. Guru99. https://www.guru99.com/es/polymorphism-in-java.html

Blog IfGeekThen. (s/f). Nttdata.com. Recuperado el 17 de febrero de 2025, de https://ifgeekthen.nttdata.com/s/post/clases-abstractas-e-interfaces-MCRLZG2CKOIFF2JKPMO7XLKRWJAI?language=es

https://www.unirioja.es/cu/jearansa/0910/archivos/EIPR_Tema04.pdf