|  |  |  |
| --- | --- | --- |
|  | Instituto Politécnico Nacional  Unidad Profesional Interdisciplinaria en Ingeniería y Tecnologías Avanzadas |  |

Ingeniería Mecatrónica

Programación Avanzada

**REPORTE DE LA PRÁCTICA No. 3**

**Herencia**

|  |  |  |  |
| --- | --- | --- | --- |
| **Nombre del Alumno:** | Zarazua Aguilar Luis Fernando | | |
| **Grupo:** | 2MV4 | **Fecha:** | 05/Septiembre/2017 |

**Objetivo:** Identificar las características, el uso y las ventajas que posee la programación orientada a objetos al momento de tener Herencia.

**Resumen**

En el presente documento se habla sobre las características, el uso y las ventajas que se tienen al momento de usar la Herencia como una herramienta principalmente de reutilización de código, partiendo sobre de la necesidad de implementar la Herencia cuando se tienen datos en común entre distintas clases con la finalidad de no volver a escribir el mismo código en varias partes del programa en repetidas ocasiones, ejemplificando esa problemática con un ejemplo de una clasificación taxonómica y como se ordena esta por medio del uso de la Herencia.

Posteriormente se habla sobre la ventaja de usar código ya compilado para reducir el tiempo de programación y aumentar la eficacia en su ejecución, para luego introducir los elementos característicos que tiene la Herencia en java como lo son "extends" y "super", además de las características que un objeto con herencia posee como lo es poder ser tratado como el objeto base, pero sin perder la información de todo el objeto con herencia.

**Introducción**

En la programación estructural se acostumbra asignar un proceso para cada evento que ocurre el cual es implementado a través de una llamada a función, sin embargo cuando se programa con objetos se tiene una variedad de procesos distintos para realizar finalmente una misma operación, que se diferencia únicamente por el tipo de objeto a ser evaluado, por ejemplo el cálculo del perímetro de distintas figuras, en el cual para cada objeto dependiendo de su número de lados, de las curvas que posea y de su forma canónica se puede obtener el perímetro correcto, esto se puede llevar a cabo con el uso del polimorfismo que adapta un método para su uso en distintos tipos de objetos.[1]

**Planteamiento del problema.**

En la situación cuando el programador necesita implementar el uso de una clase base, pero para su aplicación es necesario que esta se extienda por medio de la herencia, para poder agregarle atributos que son necesarios para la correcta ejecución del programa, el programador se ve en la necesidad de volver a crear cada uno de los métodos que contiene la clase nuevamente para que estos puedan soportar el ingreso de un nuevo tipo de dato, ya que los métodos originales no fueron pensados para trabajar con objetos más grandes, sin embargo en momento de llevar a cabo la compilación del programa esto ya no resulta un problema gracias al uso del polimorfismo en el cual el lenguaje de POO, puede tratar con objetos más grandes, como si se trataran solamente de un objeto de la clase base. Esto representa una gran ventaja de los objetos en cuestiones de redefinición del programa, ya que si no se tuviera esta posibilidad o se trabajara solamente con estructuras y funciones como en C, el programador forzosamente tendría que redefinir por completo el programa solo por quererle agregar un dato más a su nueva estructura.

**Desarrollo**

Para ejemplificar el uso del polimorfismo se crearon las clases "CarRegistro" y "ParkingAcceso", las cuales no están contenidas en "Parking", pero ejecutan métodos diseñados para esta clase.

Aplicacion01.java

package aplicacion01;

import java.util.Scanner;

public class Aplicacion01 {

public static void main(String[] args) {

Scanner Opcion;

ParkingAcceso UPIITA;

UPIITA=new ParkingAcceso();

int opc;

Opcion=new Scanner(System.in);

do

{

System.out.println("Bienvenido al Estacionamiento UPIITA");

System.out.println("1-Agregar Automovil");

System.out.println("2-Listar Automoviles");

System.out.println("3-Verificar Acceso");

System.out.println("5-Salir");

opc=Opcion.nextInt();

//IngresaCoche(UPIITA);

switch(opc)

{

case 1: IngresaCoche(UPIITA);//Pasa un ParkingAcceso que contiene un Parking.

break;

case 2: ListarCoches(UPIITA);

break;

case 3: VerificarCoche(UPIITA);

break;

}

}while(opc!=5);

}

public static boolean IngresaCoche(Parking P)

{

if (P==null)

{System.out.println("Error al crear objeto");

return false;}

else

{

String Propietario;

String Marca;

int Modelo;

Scanner Texto;

Texto=new Scanner(System.in);

System.out.println("Desea agregar información adicional (S/N)");

if (Texto.nextLine().equals("S")){

System.out.println("Ingrese el nombre del propietario:");

Propietario=Texto.nextLine();

System.out.println("Ingrese la marca del automovil:");

Marca=Texto.nextLine();

System.out.println("Ingrese el modelo");

Modelo=Texto.nextInt();

CarRegistro c;

c=new CarRegistro(Propietario, Marca, Modelo);

P.addCar(c);}

else{

CarRegistro c;

c=new CarRegistro();

P.addCar(c);

}

return true;

}

}

public static void ListarCoches(Parking p)//Acepta un objeto tipo Parking

{

CarRegistro Aux;

int i;

int n=p.getNumCar();

for (i=0;i<n;i++)

{

Aux=(CarRegistro)p.getCar(i);

System.out.println(i+".-"+Aux.getModelo());

System.out.println("Propietario: "+Aux.getPropietario());

System.out.println("Marca: "+Aux.getMarca());

}

}

public static void VerificarCoche(ParkingAcceso P)

{

Scanner Texto;

Texto=new Scanner(System.in);

System.out.println("Ingrese el numero de coches a verificar");

int n=Texto.nextInt();

CarRegistro C=(CarRegistro)P.getCar(n);

if (C!=null){

System.out.println("El acceso es: ");

System.out.println(C.isAcceso());

}

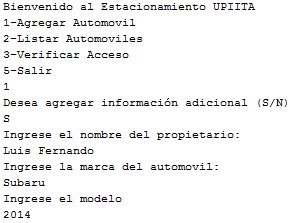
}

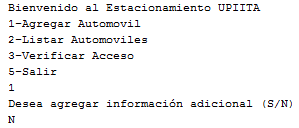
}

Observaciones: El objeto UPIITA originalmente era un objeto tipo "Parking", en lugar de ser un "ParkingAcceso", sin embargo como VerificarCoche necesita un dato más el cual es Acceso se tiene que cambiar por un objeto de tipo ParkingAcceso, el cual contiene un método para obtener el acceso, por lo cual Parking puede guardar cualquier tipo de objetos "Carro" incluyendo a "CarRegistro", pero no posee el método para leer el Acceso. Al agregar todos los objetos como CarRegistro se asegura de tener en "ParkinAcceso" solo "CarRegistro".

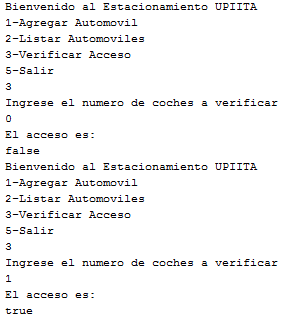
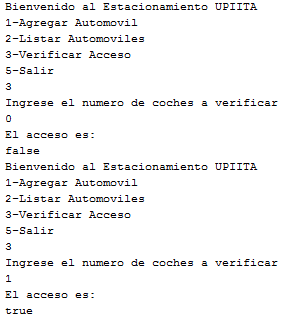
Al usar P.addCar(c), se está usando el polimorfismo ya que addCar solo podría agregar objetos tipo Carro, pero al tener heredado a "Carro" el objeto de tipo "CarRegistro", agrega por polimorfismo al objeto de tipo "CarRegistro" sin perder sus atributos.

Los métodos ListarCoches y VerificarCoche pueden aceptar "ParkinAcceso", ya que también es un parking, pero al momento de regresar el dato como sus métodos solo regresan "Carro" para almacenarlo se necesita un cast como CarRegistro C=(CarRegistro)P.getCar(n), en el cual recupera todos los atributos que contiene un CarRegistro.





Figuras 1 y 2. Ejecución del programa con ingreso de datos.

Figuras 3 y 4. Lectura de los datos y finalización del programa.

**Resultados y conclusiones**

"ParkingAcceso" por tener de Herencia a Parking, puede guardar los objetos de tipo "Carro" que le sean ingresados.

Cuando se crea un objeto de tipo "CarRegistro" y este es ingresado como un elemento de "Parking" por medio del método "addCar" lo ingresa sin problemas, ya que se trata finalmente de un "Carro".

Una vez que se eligió si el automóvil tiene información adicional o no, este es creado por el constructor de "CarRegistro", el cual indica que si no tiene información su acceso es falso y en caso de poseer información su acceso es verdadero, en cada caso con su constructor se ejecuta por medio de "super" equivaliendo a un constructor "Carro".

Al momento de leer el acceso se usa el método "getCar" que regresa un "Carro", por lo tanto como "ParkingAcceso" contiene elementos "CarRegistro", puede devolver el "Carro" sin problemas, pero usando el cast también puede devolver el "CarRegistro" completo.

En conclusión las clases y objetos con herencia permiten reutilizar el código ya escrito anteriormente, brindando la posibilidad de una mejor ejecución del programa, además de poder ampliar la funcionalidad que se quiere dar en una clase, todo esto sin tener que crear los objetos con código repetido de nuevo para cada uno de ellos y en cambio tratarlos como un nuevo objeto más grande que puede ser manipulado con las funciones de la clase base sin perder información.

**Referencias**

|  |  |
| --- | --- |
| [1] | H. M. Deitel y P. J. Deitel, Cómo programar en C/C++ y Java, Pearson Educación, 2004. |
| [2] | E. Peñaloza Romero, Fundamentos de Programación, México: Alfaomega, 2004. |