



Instituto Politécnico Nacional  
Unidad Profesional  
Interdisciplinaria De Ingeniería  
Campus Zacatecas



Ingeniería en Sistemas Computacionales

## Práctica 7 – Herencia Múltiple

Alumna: Vanessa Melenciano Llamas

Boleta: 2020670081

Profesora: Monreal Mendoza Sandra Mireya

Materia: Programación Orientada a Objetos

Grupo: 2CM2

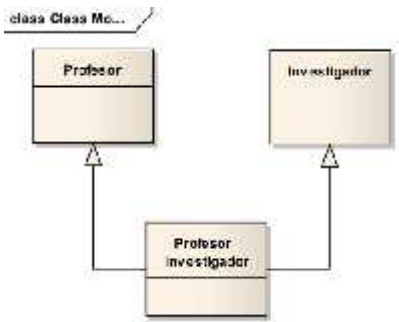
Fecha: 2 de diciembre de 2020

# Índice

<b>Introducción .....</b>	<b>3</b>
<b>Objetivos .....</b>	<b>4</b>
<b>Desarrollo .....</b>	<b>4</b>
<b>Diseño de solución .....</b>	<b>4</b>
<b>Funciones .....</b>	<b>5</b>
<b>Errores detectados .....</b>	<b>11</b>
<b>Posibles mejoras .....</b>	<b>11</b>
<b>Conclusión .....</b>	<b>12</b>
<b>Referencia .....</b>	<b>12</b>

# Introducción

Puede darse a la hora de programar algún caso donde se requiera que una clase tenga características provenientes de dos clases como es el caso de un profesor, el estudiante ante todo es un ciudadano y tendrá ciertas características que afecten su comportamiento con respecto a esta condición, pero también es hijo de alguien y esto puede incurrir en otras características adicionales. A esto se le denomina herencia múltiple.



La herencia múltiple se da cuando una clase derivada hereda de varias clases base (posiblemente no relacionadas). Este tipo de herencia no es soportado por todos los lenguajes de programación orientados a objetos (por ejemplo, Java). (Ottogalli Fernández, Martínez Morales, & León Guzmán, 2011). Para estos casos se usan las interfaces.

Una interface es la definición de un conjunto de métodos que una clase implementa, una funcionalidad abierta a la reutilización y extensibilidad.

En la interfaz se pueden modificar algunas propiedades y elegir sólo lo necesario para una sección de un proyecto si el proyecto ya existiese, de tal forma que si se tiene un proyecto existente se pueden generar los objetos necesarios para cada una de las capas del patrón, acoplándose a lo ya generado con anterioridad, esto sólo ocurrirá si la primera parte se generara con esta misma herramienta (Camarena, Trueba, Martínez, & López, 2012)

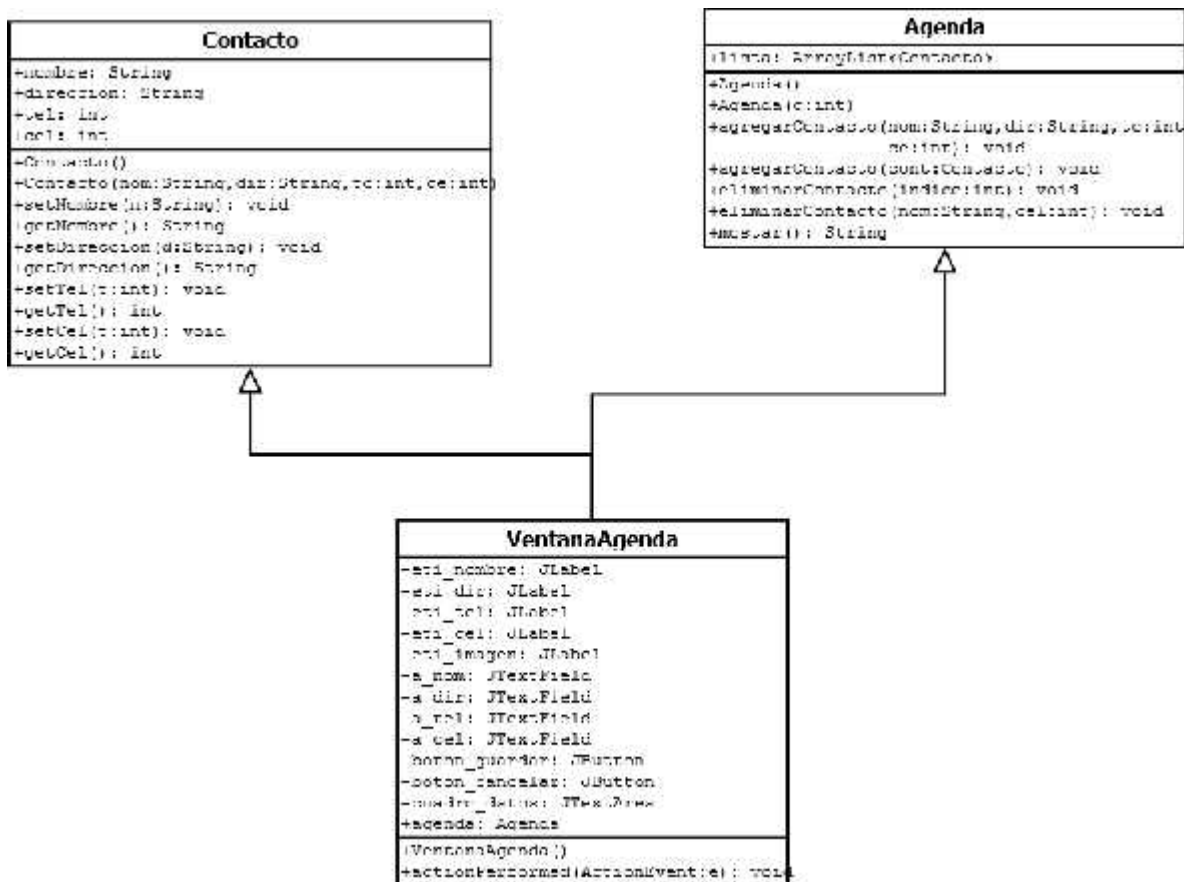
# Objetivos

Desarrollar un programa donde se aplique la herencia múltiple y las interfaces gráficas de usuario (GUI) o ventanas.

## Desarrollo

### Diseño de solución

Para la solución de la práctica, primero se realizó el diagrama UML:



La clase de contacto y agenda, ya se tenía realizada, así que fue solo cuestión de implementarlas en la clase VentanaAgenda, para la herencia múltiple por medio de una interfaz.

Para la clase VentanaAgenda, tiene una interfaz de usuario, por lo tanto en los atributos tiene los correspondientes a JLabel, JTextField, JButton, JTextArea, para armarla, todo esto dentro del método VentaAgenda, es decir, el constructor. Y dentro del método actionPerformed, se llevaba a cabo llamar a las clases de las que se hereda, al aplicar las acciones de cada uno de los botones.

# Funciones

## ➤ Clase Contacto

```
public class Contacto {
    private String nombre;
    private String direccion;
    private int tel;
    private int cel;
    public Contacto() {
        nombre = "";
        direccion = "";
        tel = 0;
        cel = 0;
    }
    public Contacto(String nom, String dir, int tc, int ce) {
        nombre = nom;
        cel = ce;
        tel = tc;
        direccion = dir;
    }
    public void setNombre(String n) {
        nombre = n;
    }
    public String getNombre() {
        return nombre;
    }
    public void setDireccion(String d) {
        direccion = d;
    }
    public String getDireccion() {
        return direccion;
    }
    public void setTel(int t) {
        tel = t;
    }
    public int getTel() {
        return tel;
    }
    public void setCel(int t) {
        cel = t;
    }
    public int getCel() {
        return cel;
    }
}
```

Esta clase contiene los datos que debe tener un contacto, nombre, dirección, teléfono y celular, cada uno con su respectivo getter y setter.

## ➤ Clase Agenda

```
public class Agenda {
    public ArrayList<Contacto> lista;

    public Agenda() {
        lista = new ArrayList();
    }
    public Agenda(int c){
        lista = new ArrayList(i:c);
    }
    public void agregarContacto(String nom, String dir, int tc, int ce){
        Contacto c = new Contacto(nom, dir, tc, ce);
        lista.add(c);
    }
    public void agregarContacto(Contacto cont){
        if(cont.getTel() != 0){
            lista.add(cont);
        }
    }
    public void eliminarContacto(int indice){
        lista.remove(indice);
    }
    public void eliminarContacto(String nom, int cel){
        for(Contacto c:lista){
            if(c.getNombre().equals(nom) && c.getCel()==cel){
                lista.remove(c);
                break;
            }
        }
    }
    public void eliminarContacto(int indice){
        lista.remove(indice);
    }
    public void eliminarContacto(String nom, int cel){
        for(Contacto c:lista){
            if(c.getNombre().equals(nom) && c.getCel()==cel){
                lista.remove(c);
                break;
            }
        }
    }
    public String mostar(){
        String datos = "";
        for(Contacto i: lista){
            if(i !=null){
                datos += "Nombre: " + i.getNombre() +
                    "   Direccion: " + i.getDireccion() +
                    "   Telefono: " + i.getTel() +
                    "   Cel: " + i.getCel() + " \n";
            }
        }
        return datos;
    }
}
```



Esta clase, se encarga de guardar una lista de contacto, así que, dentro de los atributos, se encuentra un *ArrayList* de tipo *Contactos*, que es donde se irán almacenando conforme se agreguen por medio de los métodos llamados *agregarContacto*. La clase también incluye dos constructores, uno con y otro sin parámetros. Después tiene métodos para eliminar contactos, que, aunque no se usan en la siguiente clase, decidí conservarlos. Por último, se tiene la clase mostrar, la cual fue modificada para que pudiera retornar los resultados, y así poderlos mostrar en la interfaz.

#### ➤ VentanaAgenda

```
public class VentanaAgenda extends JFrame implements ActionListener{
    private JLabel eti_nom;
    private JLabel eti_dir;
    private JLabel eti_tel;
    private JLabel eti_cel;
    private JLabel eti_imagen;
    private JTextField a_nom;
    private JTextField a_dir;
    private JTextField a_tel;
    private JTextField a_cel;
    private JButton boton_guardar;
    private JButton boton_cancelar;
    private JTextArea cuadro_datos;
    public Agenda agenda;

    public VentanaAgenda() {
        agenda = new Agenda();
        setTitle( title: "SUPER AGENDA");
        setSize( width: 900, height: 900);
        setDefaultCloseOperation( operation: EXIT_ON_CLOSE);
        setLayout( manager: null);
    }
}
```

Los atributos de la clase, se declara lo necesario para la interfaz, así como la agenda tipo Agenda, para la cual se usará la clase ya creada.

En el constructor, primero se crea la agenda, usando el constructor sin parámetros, después se le da nombre y medidas a la ventana.

```

eti_nom = new JLabel( string: "Nombre");
eti_nom.setBounds( x: 30, y: 80 , width: 70, height: 30);
add( comp: eti_nom);
eti_dir= new JLabel( string: "Dirección");
eti_dir.setBounds( x: 30, y: 120 , width: 70, height: 30);
add( comp: eti_dir);
eti_tel= new JLabel( string: "Teléfono");
eti_tel.setBounds( x: 30, y: 160 , width: 70, height: 30);
add( comp: eti_tel);
eti_cel= new JLabel( string: "Celular");
eti_cel.setBounds( x: 30, y: 200 , width: 70, height: 30);
add( comp: eti_cel);

a_nom = new JTextField();
a_nom.setBounds( x: 100, y: 80, width: 290, height: 30);
add( comp: a_nom);
a_dir = new JTextField();
a_dir.setBounds( x: 100, y: 120, width: 290, height: 30);
add( comp: a_dir);
a_tel = new JTextField();
a_tel.setBounds( x: 100, y: 160, width: 290, height: 30);
add( comp: a_tel);
a_cel = new JTextField();
a_cel.setBounds( x: 100, y: 200, width: 290, height: 30);
add( comp: a_cel);

eti_imagen = new JLabel(new ImageIcon( string: "imagen3.jpg"));
eti_imagen.setBounds( x: 405, y: 72, width: 148, height: 181);
add( comp: eti_imagen);

```

Cada una de las etiquetas de tipo JLabel y JTextField, donde se colocarán los datos, se crean, se les da medida y se agregan a la ventana.

Para la imagen, se usó atributo tipo JLabel, y se agregó como ImageIcon, con sus respectivas medidas.



```

    boton_guardar = new JButton();
    boton_guardar.setText( text: "Guardar");
    boton_guardar.setBounds( x: 90, y: 260, width: 85, height: 20);
    boton_guardar.addActionListener( l: this);
    add( comp: boton_guardar);
    boton_cancelar= new JButton();
    boton_cancelar.setText( text: "Cancelar");
    boton_cancelar.setBounds( x: 260, y: 260, width: 85, height: 20);
    boton_cancelar.addActionListener( l: this);
    add( comp: boton_cancelar);

    cuadro_datos = new JTextArea();
    cuadro_datos.setBounds( x: 40, y: 310, width: 550, height: 150);
    add( comp: cuadro_datos);
    setVisible( b: true);
}

```

Para el caso de los botones, fue el mismo proceso que las etiquetas, solo que en este caso es de tipo JButton y se agregó un método que viene de JButton, para darle acción al apretar el botón.

Y por último, el cuadro donde saldrán los resultados, es de tipo JTextArea.

```

public void actionPerformed(ActionEvent e) {
    if(e.getSource() == boton_guardar) {
        Contacto c = new Contacto();
        int tel, cel;

        try {
            c.setNombre( a: a_nom.getText());
            c.setDireccion( a: a_dir.getText());
            tel = Integer.parseInt( a: a_tel.getText());
            c.setTel( a: tel);
            cel = Integer.parseInt( a: a_cel.getText());
            c.setCel( a: cel);
        } catch (NumberFormatException ex) {
            JOptionPane.showMessageDialog( parentComponent: null, message: "Ingrese datos correctamente");
        }

        agenda.agregarContacto( cont: c);
        cuadro_datos.setText( a: agenda.mostrar());
        a_nom.setText( a: null);
        a_dir.setText( a: null);
        a_tel.setText( a: null);
        a_cel.setText( a: null);
    }
}

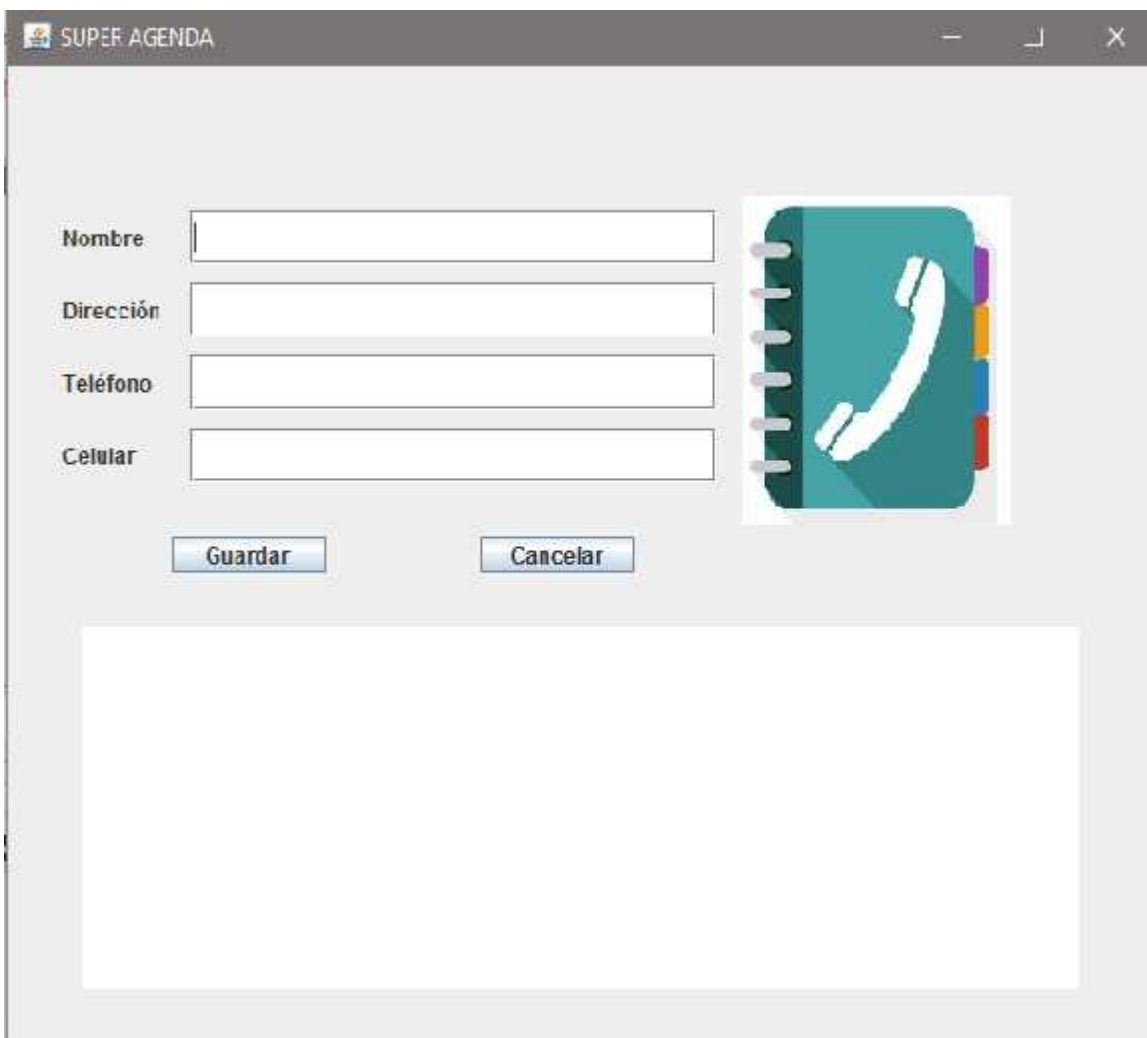
```

Para darle acción a los botones, se usó el método actionPerformed, donde se usa un if y else if para separar los dos botones, el primero, el que se encarga de guardar los datos colocados, se encarga de

mandar esos datos a la clase de Agenda, específicamente al método de agregarContactos. El try y catch se usa en caso de que el usuario no haya colocado correctamente los datos, porque los números no son de tipo numérico.

```
}  
    else if(e.getSource()==boton_cancelar){  
        a_nom.setText("");  
        a_dir.setText("");  
        a_tel.setText("");  
        a_cel.setText("");  
    }  
}  
  
public static void main(String args[]){  
    VentanaAgenda VG = new VentanaAgenda();  
}
```

En el segundo botón, solo se encarga de borrar los datos que están en ese momento en la etiqueta encargada de agregarlos.



Nombre:

Dirección:

Teléfono:

Celular:

Nombre: Maria Mendoza Salazar Dirección: REVOLUCION MEXICANA 60, EJIDAL , GUADALUPE , Z  
Telefono: 9321570 Cel: 1822434

Nombre: Mario Rodarte de la Rosa Dirección: OLIVO 206, CENTRO , FRESNILLO , ZAC , C.P.99000  
Telefono: 9242795 Cel: 7681216

Nombre: Sebastian Trejo Gonzales Dirección: CALLE TRANSITO PESADO 16, CENTRO , ZACATEC  
Telefono: 7681216 Cel: 9277180

## Errores detectados

Lo que más se complicó al realizar la práctica, fue comprender el concepto general, de que, al realizar la interfaz de usuario, se estaba implementando la herencia múltiple. Además de algunos conceptos aún no muy claros que se usan al momento de crear una interfaz.

## Posibles mejoras

En esta práctica, lo más complicado fue hacer una interfaz, ya que las otras clases, ya se tenían realizadas. Así que las posibles mejoras que se pueden dar, es practicar más utilizando las interfaces

## Conclusión

Al desarrollar la práctica, pudo ser más clara la implementación de la herencia múltiple en POO, con ayuda de las interfaces de usuario, esto porque en Java la herencia múltiple no es permitida, ya que puede causar ambigüedad en el código, y se busca mayor simplicidad.

## Referencia

- Barnes, D., & Kolling, M. (2007). *Programación orientada a objetos con Java* (3ª edición ed.). Madrid, España: PEARSON EDUCACIÓN.
- Camarena, J., Trueba, A., Martínez, M., & López, M. (2012). Automatización de la codificación del patrón modelo vista controlador (MVC) en proyectos orientados a la Web. *Ciencia Ergo Sum*, pp. 239-250.
- Ottogalli Fernández, K. A., Martínez Morales, A. A., & León Guzmán, L. (2011). NASPOO: una notación algorítmica estándar para Programación Orientada a Objetos. *Universidad Privada Dr. Rafael Belloso Chacín*, pp. 81-102.