Trabajo Práctico 2

SEDE:	**Leandro N. Alem**
FACULTAD:	**FACULTAD DE INGENIERÍA, TECNOLOGÍA Y ARQUITECTURA**
CARRERA:	**INGENIERÍA EN SISTEMAS DE INFORMACIÓN** **MODALIDAD** ☒Presencial ☐Digital
NOMBRE DE LA ASIGNATURA:	**PARADIGMAS Y LENGUAJES DE PROGRAMACIÓN II** **MODALIDAD** ☒Presencial ☐Presencial Digital ☐Presencial Mixta ☐Digital
NOMBRE Y APELLIDO DEL DOCENTE	**Mgter. Marcelo Arzamendia - Lic. Itatí Malañuk**
DNI, NOMBRE Y APELLIDO DEL/LOS ESTUDIANTE/S	
DNI DEL/LOS ESTUDIANTE/S	Ver sección de “Nombre y Apellidos de los estudiantes
FECHA Y HORARIO DE REALIZACIÓN Y/O DE ENTREGA: Martes 07 de Octubre de 2025 (Hora comienzo 15:00 hs entrega 18:10 hs)	

TIPO DE EVALUACIÓN:	☐Trabajo Práctico I / 1er evaluación (Régimen especial) ☒Trabajo Práctico II / 2da evaluación (Régimen especial) ☐Evaluación Parcial / 3ra evaluación (Régimen especial) ☐Evaluación Integradora / 4ta evaluación (Régimen especial)	
MODALIDAD DE INSTANCIA EVALUATIVA	☐Individual ☐Parejas ☒Grupal	☒Sincrónica ☐Asincrónica

CONTENIDOS A EVALUAR:
Enunciar los contenidos involucrados en la evaluación.

EJE TEMATICO 1: Paradigma de Programación Orientado a Objetos
1.1. Introducción y características fundamentales del Paradigma Orientada a Objetos.
Conceptos introductorios al paradigma. Clases y objetos. Objetos tangibles e intangibles del dominio del problema y objetos del espacio de soluciones. Modelos. Brecha semántica.
Complejidad en sistemas de software. Propiedades deseables: Reusabilidad y extensibilidad, entre otras.
1.2. Objetos, clases y relaciones.
Clases, objetos y relaciones. Relaciones de generalización/ especialización, todo/ parte, asociación y uso. Cardinalidad. Modelos de especificación.
1.3. Mecanismos de abstracción. Herencia y polimorfismo.
Abstracción. Encapsulamiento. Permisos de acceso. Interfase (protocolo). Clasificación. Mecanismo de herencia. Polimorfismo: estático y dinámico. Mensajes.
1.4. Lenguajes Orientados a Objetos.
Entornos de Programación. Lenguajes. Sintaxis y semántica. Especificación de Clases. Variables y constantes. Métodos. Envío de mensajes. Creación y destrucción de objetos. Especificación de herencia y polimorfismo. Asociación temprana y tardía. Control de acceso y herencia. Interfaces. Estudio de paquetes o librerías. Interfase grafica del usuario.
Otros tópicos. Resumen de semejanzas y diferencias entre lenguajes de Programación OO modernos. Reutilización: clases predefinidas.

CRITERIOS DE EVALUACIÓN

1. Responsabilidad del alumno [Res. 98/17]
   a. Desarrollar la evaluación en formato digital entregando en formato PDF;
   b. Consignar en cada hoja: Nombre, Apellido, DNI, Carrera, tipo de Evaluación (Parcial), fecha
2. Criterios de Evaluación [R.R. Nro. 185/10]
   a. Producción escrita clara y adecuada
   b. Claridad conceptual
   c. Capacidad de análisis de las situaciones planteadas
   d. Selección de la información pertinente según la situación
   e. Aplicación de los conceptos centrales a la situación/caso planteado
   f. Postura personal fundada y fundamentada
3. Aspectos referidos al Lenguaje
   a. Coherencia textual
   b. Ortografía
   c. Legibilidad
   d. Prolijidad de texto

CONSIGNAS DE EVALUACIÓN / ASIGNACIÓN DE PUNTAJE

Trabajo Práctico 2. Sistema extendido: POO, GUI y buenas prácticas

Objetivo: el objetivo de este trabajo es que se consolide y amplíe los conocimientos de programación orientada a objetos (POO), aplicando conceptos más avanzados de diseño. Asimismo, se busca fomentar la integración de prácticas de modelado en UML, implementación en Java o Python y el desarrollo de una interfaz gráfica simple, con el propósito de simular un sistema de gestión empresarial cercano a un caso real.

Pautas

El presente trabajo práctico integrador se debe realizar con las siguientes pautas:

1. El siguiente trabajo práctico tiene nota grupal y personal (el código es grupal y el coloquio personal)
2. Los grupos deben ser de 2 personas.
3. En el presente documento se presentan los requerimientos funcionales.
4. La rúbrica tiene una suma 100 puntos (nota 10 diez)
6. Se deberá presentar el desarrollo del informe y el sistema generado.
7. Para aprobar se deberá presentar en coloquio del grupo explicando toda la estructura y códigos implementados.

Introducción: En este trabajo se plantea el desarrollo de un sistema para la gestión de una empresa, ampliando las funcionalidades del Trabajo Práctico 1. El sistema debe permitir administrar personas (clientes, empleados, proveedores), productos y servicios, facturas, pagos y notificaciones. La solución deberá aplicar conceptos de POO (herencia, abstracción, encapsulamiento, polimorfismo, interfaces, clases abstractas), así como buenas prácticas de diseño (principios SOLID), incorporando también una interfaz gráfica que facilite la interacción con el usuario.

Contexto del problema: Una empresa desea desarrollar un sistema para administrar la información de las personas relacionadas con ella, los productos y servicios que ofrece, las facturas emitidas y la gestión interna de pagos y departamentos.

De todas las personas vinculadas a la empresa interesa almacenar datos básicos como nombre, domicilio, DNI, teléfono, entre otros. Estas personas pueden clasificarse en clientes, empleados y proveedores. De los clientes se necesita administrar su límite de crédito, historial de compras y la categoría a la que pertenecen (regular, premium o corporativo). De los empleados se requiere registrar su salario, puesto (administrativo, técnico o gerente), la fecha de ingreso y el departamento al que pertenecen. En el caso de los proveedores, es importante guardar su razón social, número de identificación fiscal y los productos que suministran a la empresa.

La empresa ofrece tanto productos como servicios. De ellos se debe conocer el código, el nombre, el precio, el tipo (por ejemplo, servicio técnico, consultoría o venta de insumos) y el proveedor que los ofrece. Cada transacción comercial se registra mediante una factura. Una factura incluye su número, la fecha de emisión, el cliente que realiza la compra, el empleado que gestionó la operación y la lista de productos o servicios adquiridos. Además, es necesario reflejar la forma de pago (efectivo, tarjeta o transferencia). Cada factura está vinculada a un pago, que se representa con un recibo donde se consigna el monto abonado, la fecha, el método de pago y el estado (pendiente, cancelado o parcial).

Por último, la empresa organiza a sus empleados en departamentos, que cuentan con un nombre, un presupuesto asignado y un responsable a cargo.

De esta manera, el sistema debe permitir instanciar clientes, empleados, proveedores, productos, servicios, facturas y pagos, para luego poder mostrar en pantalla el detalle completo de cada factura con la información del cliente, el empleado que lo atendió, los productos o servicios adquiridos, la forma de pago y el total a abonar.

La empresa desea contar con un sistema que permita instanciar clientes, empleados, productos/servicios y facturas, de modo que se pueda mostrar en pantalla el detalle completo de cada factura (cliente, empleado que lo atendió, productos/servicios adquiridos y el total).

Ampliación respecto al TP1
En esta segunda parte, la empresa desea mejorar y extender el sistema para incorporar nuevas funcionalidades:
• Facturación avanzada (impuestos, descuentos, múltiples formas de pago).
• Gestión de estados de facturas y recibos (pendiente, parcial, cancelado).
• Generación de documentos de factura (mínimo por pantalla; opcional PDF o txt).
• Notificaciones al cliente mediante un sistema extensible de envío de mensajes (ejemplo: Email, SMS), respetando el principio de inversión de dependencias (DIP).
• Soporte a futuras extensiones sin modificar código base, siguiendo el principio de abierto/cerrado (OCP).
• Interfaz gráfica para ingresar datos, mostrar resultados y mensajes.
• Aplicación explícita de principios SOLID y buenas prácticas de POO.

Requerimientos del Trabajo Práctico 2

El sistema deberá cumplir con los siguientes requerimientos funcionales y de diseño:

Modelado UML:
- Diagrama de clases que muestre las clases principales, relaciones (asociación, composición, agregación, herencia).

Clases principales:
- Persona y sus subclases: Cliente, Empleado, Proveedor.
- Producto / Servicio.
- Factura, ItemFactura, Recibo.
- Interfaces: Notificador/MessageSender, DocumentoImprimible.
- Clases abstractas: por ejemplo, Pago o Persona (donde aplica).

(Las clases listadas son mínimas. Los ejemplos son sugeridos, no limitativos).

Implementación en Java o Python:

CRUD (Crear, Leer, Actualizar y Eliminar) básico para Clientes, Productos/Servicios y Empleados (con GUI).

Crear Factura:
-Asociar cliente, empleado (genera la factura) y lista de ítems de la factura (producto/servicio + cantidad).
-Calcular subtotal, impuestos y total.
-Subtotal (suma de precios × cantidades).
-Impuestos (pueden definir reglas, ej: 21% IVA).
-Total (subtotal + impuestos – descuentos que aplican).

-Registrar forma de pago (EFECTIVO, TARJETA, TRANSFERENCIA) y generar un recibo con estado inicial Pendiente.

Gestión de estados Factura/Recibo.
-Permitir pago parcial (resta al saldo y cambia estado a Parcial).
-Si se paga el total: estado = Cancelado.
-Si no se ha pagado nada: estado = Pendiente.

Documento de Factura: implementar clase DocumentoImprimible con salida por pantalla (mínimo).

Notificaciones: al crear o modificar una factura, enviar una notificación simulada al cliente.

GUI: Debe permitir, al menos:
-Crear un cliente, crear un producto/servicio, crear una factura (seleccionando cliente, empleado, ítems).
-Listar facturas (con totales y estado). Disparar notificación al cliente.

Buenas prácticas de diseño

- ✓ Atributos privados, acceso con getters/setters cuando corresponda.
- ✓ Uso de clases abstractas e interfaces en los casos que lo requieran (ej: Pago, Notificador, DocumentoImprimible). Se espera al menos un ejemplo de cada uno.
- ✓ Incorporar polimorfismo en operaciones que tengan variantes (ej: cálculo de impuestos según producto, envío de notificaciones con distintos medios, generación de documentos con diferentes formatos).
- ✓ Aplicar principios SOLID en el diseño. En el informe se debe indicar explícitamente cómo se aplicaron (ejemplos sugeridos: SRP separando cálculo de totales; DIP en NotificationService; OCP en permitir nuevos notifiers sin modificar el servicio). Los ejemplos son sugeridos, no limitativos.
- ✓ Se aceptan otras decisiones de diseño siempre que estén bien fundamentadas.
- ✓ Manejo de errores y validaciones: precios no negativos, cantidades > 0, cliente con datos mínimos, entre otros.

Entrega:

-Código completo en Java o Python.
-Informe:
- Introducción.
- Desarrollo:
  - UML del sistema.
  - Decisiones de diseño (por qué se usaron clases abstractas, interfaces, polimorfismo).
  - Explicación de cómo se aplicaron principios SOLID.
  - Limitaciones y posibles mejoras futuras.
  - Link al repositorio en GitHub.
- Conclusión (reflexión sobre trabajo, dificultades y aprendizajes).

Rubrica de Puntos

	[Grupal] Introducción y Marco teórico	[Grupal] Correcta implementación.	[Grupal] Organización del código y salida del programa	[Personal] Coloquio/Presentación Técnica. Vocabulario técnico
Puntos	**20**	**30**	**20**	**30**

La asignación de puntaje se basa en el puntaje máximo que se encuentra definido en la rúbrica, y la asignación de dicho puntaje en base a la siguiente tabla para evaluar la entrega.

Entrega Excelente (100%)	Se aborda el tema. Se presenta la idea y se profundiza la misma o agregando valor. La entrega se realiza en tiempo y forma. El trabajo está estructurado y completado al 10. El trabajo se presentó con todos los lineamientos propuestos en tiempo y forma.
Terminado Satisfactorio (80%)	Se aborda el tema, pero se encuentra en un 75% el punto abordado. La entrega se realiza, pero existen puntos faltantes para completar la idea de la funcionalidad requerida. Se entrega en tiempo y forma. No se extendió en lo que se propuso como idea. El trabajo o ítem falto algunos puntos a tener en cuenta para completarlo.
Básico (60%)	Se aborda el tema, pero con un nivel escaso de comprensión y de realización. Se encuentra realizado al 50% del ítem solicitado. No se extendió en mejorar o perfeccionar. Se encuentra deficiente la organización del trabajo. No se detalla profundidad en parte del ítem o se argumenta. No se presentan todos los lineamientos propuestos para el ítem.
No realizado/Escaso (0%)	Solo se menciona el tema o no se aborda. No presenta información relacionada al ítem solicitado. O se realizó, pero con error en el abordaje para su funcionamiento o publicación. No se estructuró el trabajo.

INTRODUCCIÓN

En el presente Trabajo Práctico se desarrolla el análisis, diseño e implementación de un sistema de gestión empresarial como parte del Trabajo Práctico N°2. El objetivo principal fue aplicar de manera integral los conceptos de Programación Orientada a Objetos (POO), diagramación UML y principios SOLID.

El sistema desarrollado permite modelar un sistema empresarial donde se gestionan clientes, empleados, proveedores, productos y servicios, facturación, pagos y notificaciones. A diferencia del Trabajo Práctico N°1, en esta instancia se profundiza en la estructura del software, aplicando polimorfismo, herencia, encapsulamiento, abstracción, interfaces y clases abstractas. Además, se incorpora una interfaz gráfica de usuario (GUI) para facilitar la interacción. Por último, también se anexa un método para persistir los datos ingresados.

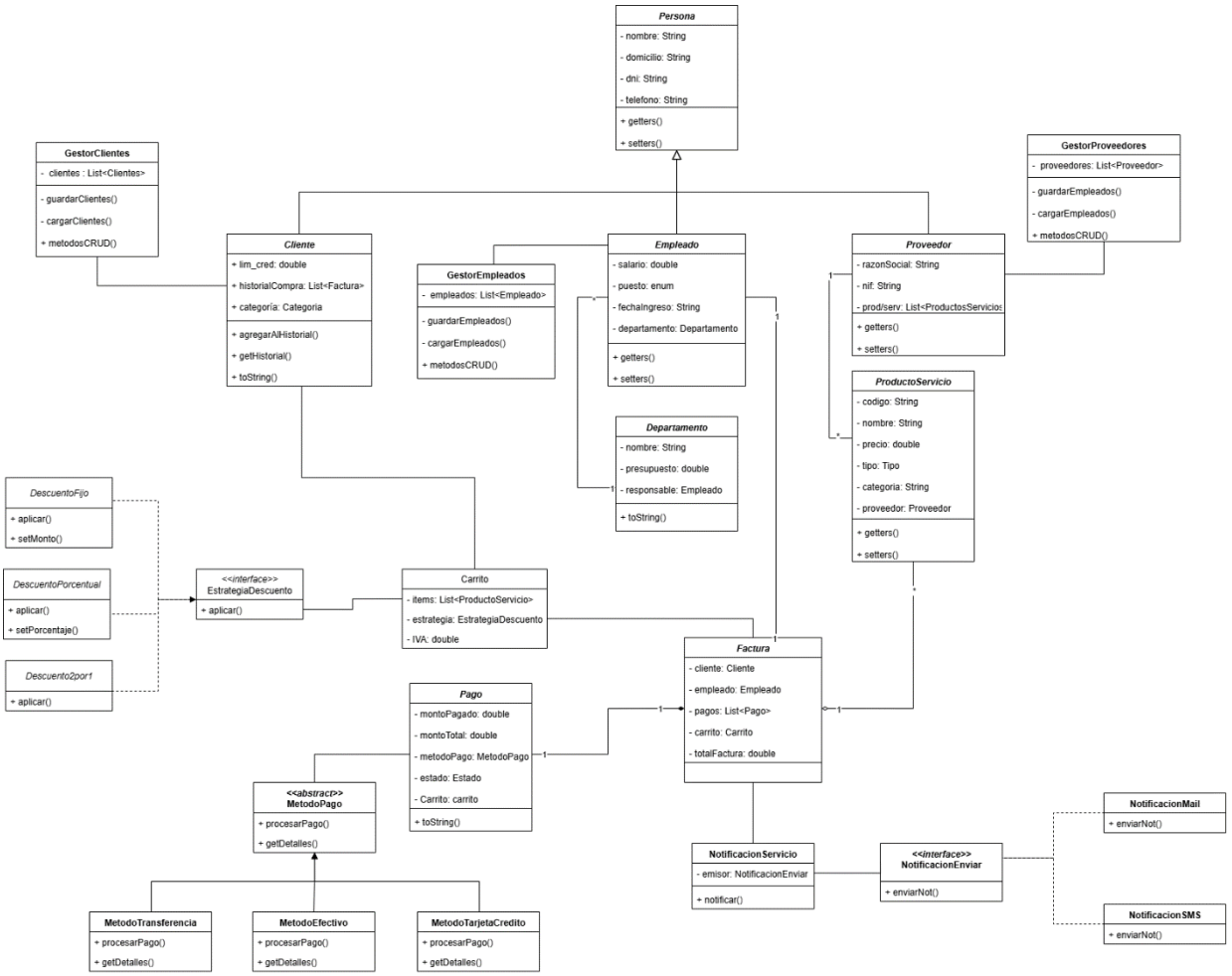
DESARROLLO

UML

Para representar la arquitectura del sistema se utiliza un diagrama UML de clases que permite visualizar la estructura interna de las clases principales y sus relaciones.
En el diagrama se puede observar:

- La clase abstracta Persona, de la cual heredan Cliente, Empleado y Proveedor.
- La asociación entre Proveedor y ProductoServicio, que representa la relación de suministro.
- La composición entre Factura y sus ítems (ItemFactura), ya que la vida útil de los ítems depende de la factura.
- Las interfaces NotificacionEnviar y DocumentoImprimible, que definen contratos para extensiones futuras.
- Las relaciones entre Factura, Cliente, Empleado y Recibo, que permiten modelar el ciclo completo de una transacción.

A continuación, se adjunta el Diagrama de Clases UML terminado, especificando las clases con sus respectivos atributos y métodos, sus relaciones y la cardinalidad de las mismas



DISEÑO

Durante el desarrollo del sistema se ha tomado diversas decisiones de diseño orientadas a aplicar la modularidad, extensibilidad y mantenibilidad del código.

- Se utilizó clases abstractas para representar conceptos generales ***MetodoPago***. Esto permite definir comportamientos comunes y dejar que las subclases implementen detalles específicos (MetodoEfectivo, MetodoTransferencia, MetodoTarjetaDeCredito).

Clase Abstracta "***MetodoPago***":

```
public abstract class MetodoPago {

    public abstract double procesarPago(double monto, Carrito carrito);

    public abstract String getDetalles();
}
```

Clases que Heredan a "***MetodoPago***":

```
public class MetodoEfectivo extends MetodoPago{

    @Override
    public double procesarPago(double monto, Carrito carrito) {
        if(monto < 0|| monto > carrito.calcularTotal()){
            System.out.println("\nMonto inválido");
        }
        System.out.println("\nProcesando $" + monto + " con Efectivo...");
        return monto;
    }

    @Override
    public String getDetalles() {
        return "Efectivo.";
    }

}
```

```
public class MetodoTransferencia extends MetodoPago{
    private final String numeroCuenta;
    private final String clave;

    public MetodoTransferencia(String numeroCuenta, String clave){
        if (numeroCuenta == null || numeroCuenta.isEmpty()) {
            throw new IllegalArgumentException("\nNumero de cuenta bancaria invalido");
        }
        if (clave == null || clave.isEmpty()) {
            throw new IllegalArgumentException("\nClave bancaria invalida");
        }

        this.numeroCuenta = numeroCuenta;
        this.clave = clave;
    }


    @Override
    public double procesarPago(double monto, Carrito carrito) {
        if (monto <= 0 || monto > carrito.calcularTotal()) {
            System.out.println("\nMonto inválido");
        }

        System.out.println("\nProcesando $" + monto + " con Transferencia Bancaria");
        return monto;
    }

    @Override
    public String getDetalles() {
        return "Transferencia bancaria a la cuenta: " + numeroCuenta;
    }
```

```java
public class MetodoTarjetaCredito extends MetodoPago{
    private final String numeroTarjeta;
    private final String fechaVencimiento;
    private final String cvv;

    public MetodoTarjetaCredito(String numeroTarjeta, String fechaVencimiento, String cvv){
        if (numeroTarjeta == null || numeroTarjeta.length() != 16) {
            throw new IllegalArgumentException("\nNumero de tarjeta invalido");
        }
        if (fechaVencimiento == null || !fechaVencimiento.matches("\\d{2}/\\d{2}")) {
            throw new IllegalArgumentException("\nFecha de expiración invalida");
        }
        if (cvv == null || cvv.length() != 3) {
            throw new IllegalArgumentException("\nCVV Invalido");
        }

        this.numeroTarjeta = numeroTarjeta;
        this.fechaVencimiento = fechaVencimiento;
        this.cvv = cvv;
    }

    @Override
    public double procesarPago(double monto, Carrito carrito) {

        if (monto <= 0 || monto > carrito.calcularTotal()) {
            System.out.println("\nMonto inválido");
        }

        System.out.println("\nProcesando $" + monto + " con Tarjeta de Credito");
        return monto;
    }

    @Override
    public String getDetalles() {
        return "Tarjeta de Crédito terminada en: " + numeroTarjeta.substring(numeroTarjeta.length() - 4);
    }
}
```

- Se optó por el uso de interfaces: NotificacionEnviar para definir contratos que permiten desacoplar el código y habilitar futuras extensiones sin modificar clases existentes (NotificaciónEmail, NotificaciónSMS).

Interfaz "**NotificaciónEnviar**":

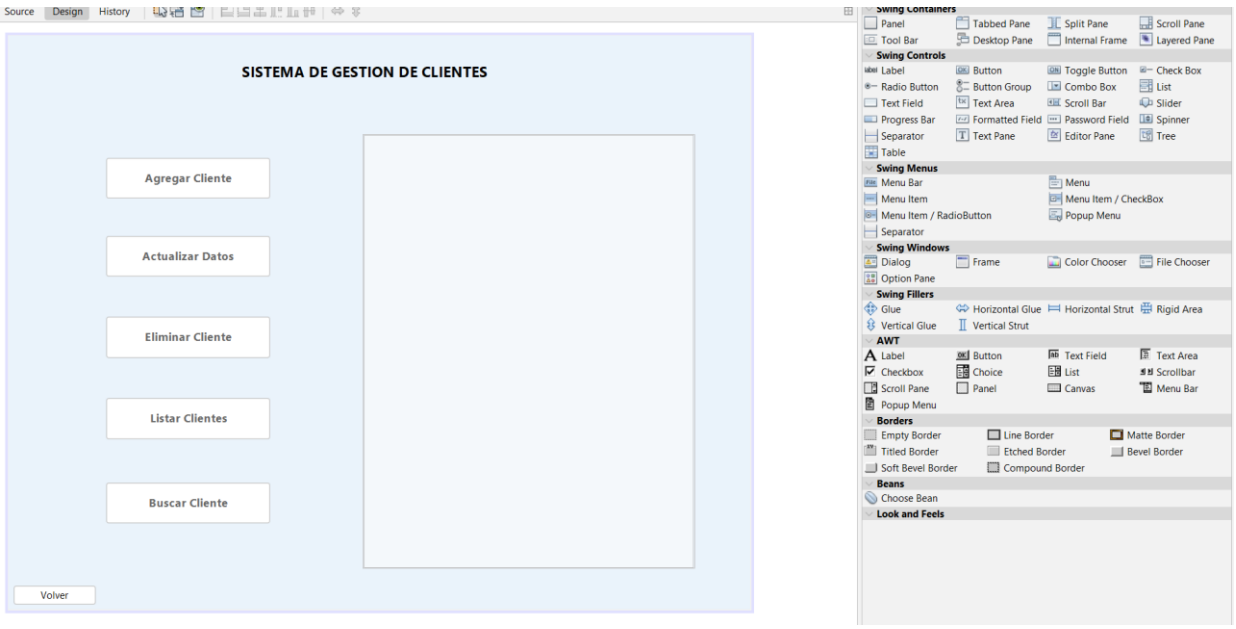
```java
public interface NotificacionEnviar {
    // Se implementa una interfaz para enviar notificaciones por distintos medios.
    void enviarNot(String destinatario, String mensaje);
}
```

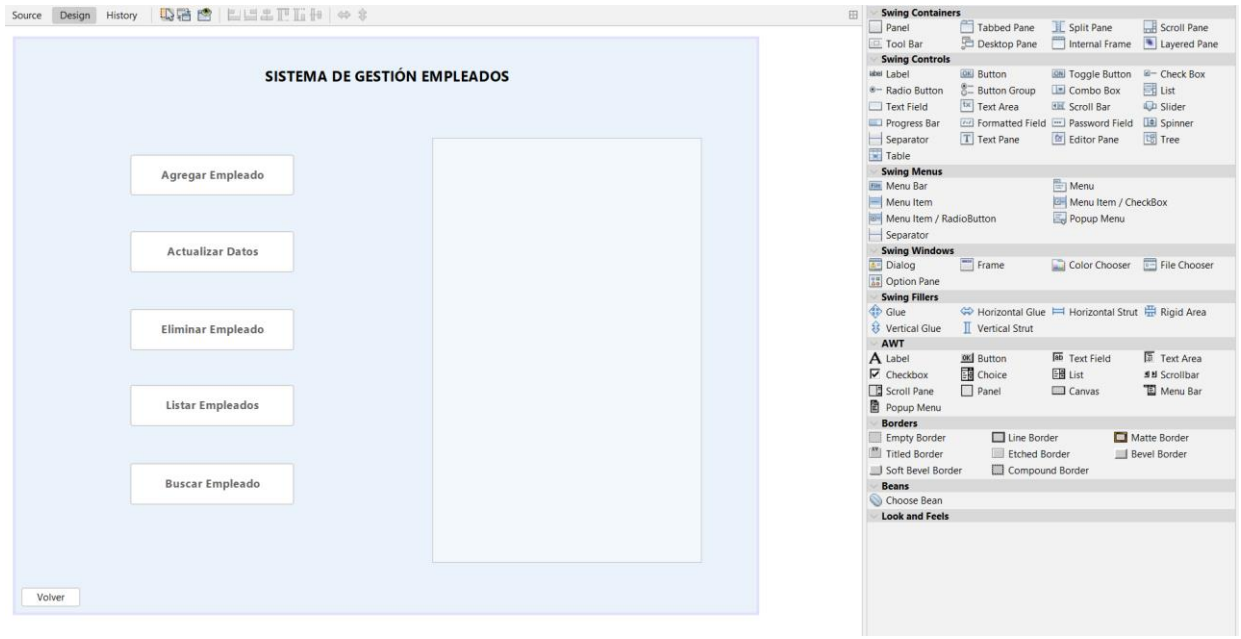
Clases que implementan la Interfaz:

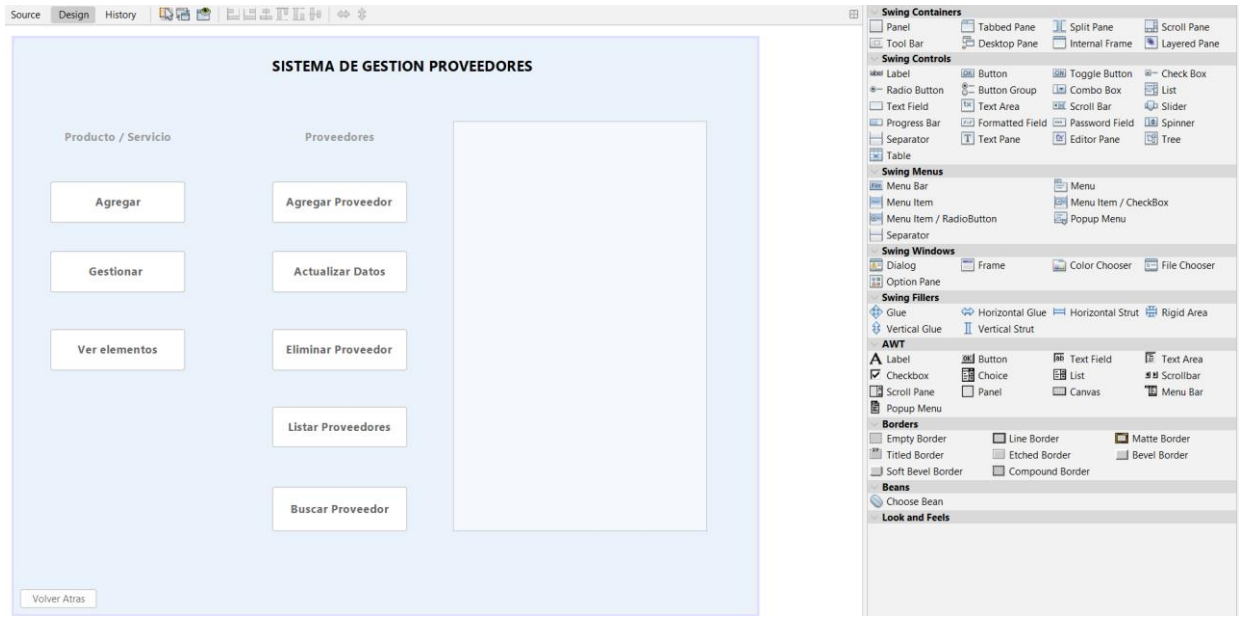
```java
public class NotificacionMail implements NotificacionEnviar{

    @Override
    public void enviarNot(String destinatario, String mensaje) {
        System.out.println("=== ENVIANDO EMAIL ===");
        System.out.println("Para: " + destinatario);
        System.out.println("Mensaje: " + mensaje);
        System.out.println("Email enviado exitosamente!");
        System.out.println("======================");

    }
}
```

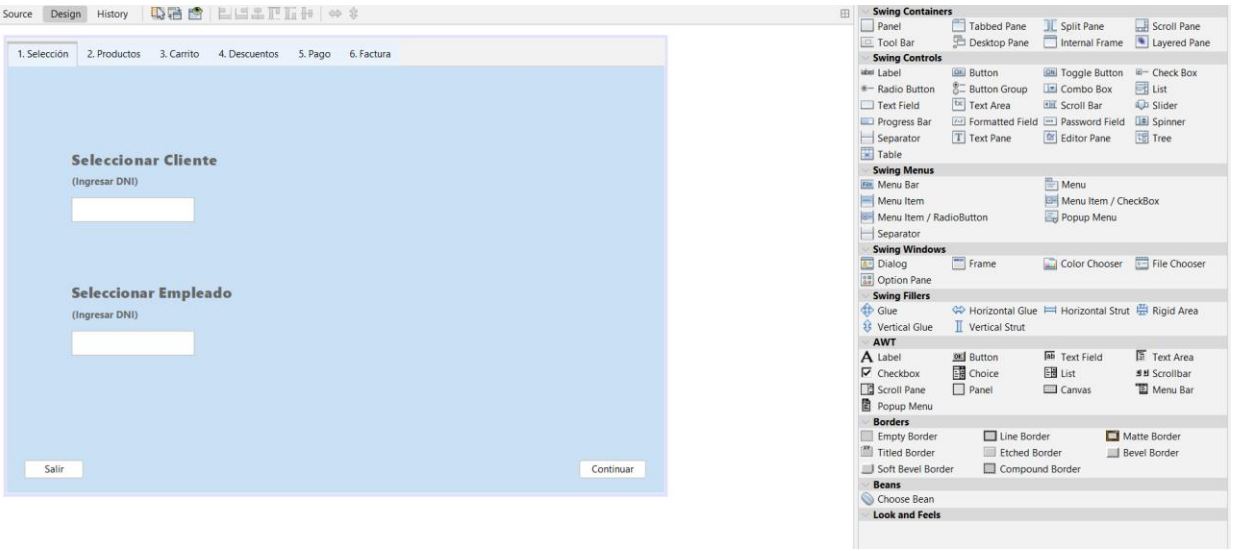
```java
public class NotificacionSMS implements NotificacionEnviar{

    @Override
    public void enviarNot(String destinatario, String mensaje) {
        // Simulación de envío de SMS
        System.out.println("=== ENVIANDO SMS ===");
        System.out.println("Número: " + destinatario);
        System.out.println("Mensaje: " + mensaje);
        System.out.println("SMS enviado exitosamente!");
        System.out.println("====================");

    }
```

- Se aplicó polimorfismo en los métodos de notificación, en las estrategias de descuentos y en los diferentes métodos de pago, logrando un sistema flexible.
- Se organizó el código en componentes separados para cada responsabilidad (gestión de clientes, productos, facturación y notificaciones), lo que mejora la legibilidad y entendimiento del código.
- Se incorporó una interfaz gráfica sencilla para realizar operaciones CRUD y emitir facturas, mejorando la experiencia del usuario final.




Source
Design
History
SISTEMA DE GESTIÓN EMPLEADOS
Agregar Empleado
Actualizar Datos
Eliminar Empleado
Listar Empleados
Buscar Empleado
Volver
Swing Containers
Panel
Tabbed Pane
Split Pane
Scroll Pane
Tool Bar
Desktop Pane
Internal Frame
Layered Pane
Swing Controls
Label
Button
Toggle Button
Check Box
Radio Button
Button Group
Combo Box
List
Text Field
Text Area
Scroll Bar
Slider
Progress Bar
Formatted Field
Password Field
Spinner
Separator
Text Pane
Editor Pane
Tree
Table
Swing Menus
Menu Bar
Menu
Menu Item
Menu Item / CheckBox
Menu Item / RadioButton
Popup Menu
Separator
Swing Windows
Dialog
Frame
Color Chooser
File Chooser
Option Pane
Swing Fillers
Glue
Horizontal Glue
Horizontal Strut
Rigid Area
Vertical Glue
Vertical Strut
AWT
Label
Button
Text Field
Text Area
Checkbox
Choice
List
Scrollbar
Scroll Pane
Panel
Canvas
Menu Bar
Popup Menu
Borders
Empty Border
Line Border
Matte Border
Titled Border
Etched Border
Bevel Border
Soft Bevel Border
Compound Border
Beans
Choose Bean
Look and Feels


Source
Design
History
SISTEMA DE GESTION PROVEEDORES
Producto / Servicio
Proveedores
Agregar
Gestionar
Ver elementos
Agregar Proveedor
Actualizar Datos
Eliminar Proveedor
Listar Proveedores
Buscar Proveedor
Volver Atras
Swing Containers
Panel
Tabbed Pane
Split Pane
Scroll Pane
Tool Bar
Desktop Pane
Internal Frame
Layered Pane
Swing Controls
Label
Button
Toggle Button
Check Box
Radio Button
Button Group
Combo Box
List
Text Field
Text Area
Scroll Bar
Slider
Progress Bar
Formatted Field
Password Field
Spinner
Separator
Text Pane
Editor Pane
Tree
Table
Swing Menus
Menu Bar
Menu
Menu Item
Menu Item / CheckBox
Menu Item / RadioButton
Popup Menu
Separator
Swing Windows
Dialog
Frame
Color Chooser
File Chooser
Option Pane
Swing Fillers
Glue
Horizontal Glue
Horizontal Strut
Rigid Area
Vertical Glue
Vertical Strut
AWT
Label
Button
Text Field
Text Area
Checkbox
Choice
List
Scrollbar
Scroll Pane
Panel
Canvas
Menu Bar
Popup Menu
Borders
Empty Border
Line Border
Matte Border
Titled Border
Etched Border
Bevel Border
Soft Bevel Border
Compound Border
Beans
Choose Bean
Look and Feels



- Se utilizó la Interfaz "***Serializable***" que se implementa en las clases que requieran, para indicar que sus objetos (Personas, Departamentos, Productos y Servicios) pueden ser convertidos (serializados) en un flujo de bytes y luego recreados (deserializados) en otro momento (Gestores de Clientes, Empleados y Proveedores). De esta forma los datos se guardan en un archivo .dat admitiendo la persistencia de datos.

```
import java.util.ArrayList;
import java.util.List;

public class GestorClientes {
    private List<Cliente> clientes;
    private final String ARCHIVO_CLIENTES = "clientes.dat";

    public GestorClientes() {
        this.clientes = new ArrayList<>();
        cargarClientes(); // metodo para cargar clientes al iniciar
    }

    // Metodos para guardar los clientes en un archivo
    private void guardarClientes() {
        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(ARCHIVO_CLIENTES))) {
            oos.writeObject(clientes);
            System.out.println("Clientes guardados en: " + ARCHIVO_CLIENTES);
        } catch (IOException e) {
            System.err.println("Error al guardar clientes: " + e.getMessage());
        }
    }

    @SuppressWarnings("unchecked")
    private void cargarClientes() {
        File archivo = new File(ARCHIVO_CLIENTES);
        if (!archivo.exists()) {
            System.out.println("ℹ No existe archivo de clientes, se creará uno nuevo");
            return;
        }

        try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(ARCHIVO_CLIENTES))) {
            clientes = (List<Cliente>) ois.readObject();
            System.out.println("Clientes cargados: " + clientes.size() + " registros");
        } catch (IOException | ClassNotFoundException e) {
            System.err.println("Error al cargar clientes: " + e.getMessage());
            // Si hay error, empezamos con lista vacía
            clientes = new ArrayList<>();
        }
    }
```

```java
public class GestorEmpleados {
    private List<Empleado> empleados;
    private final String ARCHIVO_EMPLEADOS = "empleados.dat";

    public GestorEmpleados() {
        this.empleados = new ArrayList<>();
        cargarEmpleados();
    }


    // Metodos para guardar empleados en un archivo
    private void guardarEmpleados() {
        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(ARCHIVO_EMPLEADOS))) {
            oos.writeObject(empleados);
            System.out.println("Empleados guardados en: " + ARCHIVO_EMPLEADOS);
        } catch (IOException e) {
            System.err.println("Error al guardar empleados: " + e.getMessage());
        }
    }

    @SuppressWarnings("unchecked")
    private void cargarEmpleados() {
        File archivo = new File(ARCHIVO_EMPLEADOS);
        if (!archivo.exists()) {
            System.out.println("Creando Archivo Empleados");
            return;
        }

        try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(ARCHIVO_EMPLEADOS))) {
            empleados = (List<Empleado>) ois.readObject();
            System.out.println("Empleados cargados: " + empleados.size() + " registros");
        } catch (IOException | ClassNotFoundException e) {
            System.err.println("Error al cargar empleado: " + e.getMessage());
            empleados = new ArrayList<>();
        }
    }
```

```java
public class GestorProveedores {
    private List<Proveedor> proveedores;
    private final String ARCHIVO_PROVEEDORES = "proveedores.dat";

    public GestorProveedores() {
        this.proveedores = new ArrayList<>();
        cargarProveedores();
    }


    // Metodos para guardar proveedor en un archivo
    private void guardarProveedores() {
        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(ARCHIVO_PROVEEDORES)))
            oos.writeObject(proveedores);
            System.out.println("Proveedores guardados en: " + ARCHIVO_PROVEEDORES);
        } catch (IOException e) {
            System.err.println("Error al guardar proveedores: " + e.getMessage());
        }
    }

    @SuppressWarnings("unchecked")
    private void cargarProveedores() {
        File archivo = new File(ARCHIVO_PROVEEDORES);
        if (!archivo.exists()) {
            System.out.println("No existe archivo de proveedores, se creará uno nuevo");
            return;
        }
        try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(ARCHIVO_PROVEEDORES))) {
            proveedores = (List<Proveedor>) ois.readObject();
            System.out.println("Proveedores cargados: " + proveedores.size() + " registros");
        } catch (IOException | ClassNotFoundException e) {
            System.err.println("Error al cargar proveedores: " + e.getMessage());
            proveedores = new ArrayList<>();
        }
    }
```

POO

En la implementación se aplicó los principales conceptos de la Programación Orientada a Objetos:

- **Clases y objetos**: se representan las entidades del dominio (clientes, empleados, productos, facturas) como clases con atributos y métodos que reflejan su comportamiento.
- **Abstracción**: se encapsulan comportamientos comúnes en clases abstractas para optimizar el código y tener una jerarquía clara
- **Encapsulamiento**: se utilizan atributos privados y métodos getters/setters para controlar el acceso a la información.
- **Herencia**: se reutilizan comportamientos compartidos en la jerarquía de Persona y en clases relacionadas con pagos
- **Polimorfismo**: se implementan comportamientos variables a través de subclases e interfaces, como en el envío de notificaciones (Mail, SMS), métodos de pago y estrategias de descuento.

Principios SOLID aplicados

Uno de los objetivos principales del trabajo fue aplicar los principios SOLID en el diseño:

SRP (Single Responsibility Principle):
Cada clase tiene una única responsabilidad. Por ejemplo, la clase GestorClientes se encarga únicamente de las operaciones sobre clientes, mientras que la clase Factura gestiona el cálculo de totales y estados.

OCP (Open/Closed Principle):
Se diseñó el sistema para ser abierto a la extensión y cerrado a la modificación. Por ejemplo, si deseamos agregar un nuevo canal de notificación, simplemente creamos una nueva clase que implemente la interfaz NotificacionEnviar, sin modificar el código existente.

LSP (Liskov Substitution Principle):
Las subclases pueden reemplazar a sus superclases sin alterar el comportamiento del sistema. Las distintas subclases de MetodoPago funcionan indistintamente dentro de la lógica de Pago.

ISP (Interface Segregation Principle):
En el sistema se definen interfaces específicas y reducidas. En lugar de tener una gran interfaz con muchos métodos, se utilizan intefaces pequeñas enfocadas en tareas concretas.

DIP (Dependency Inversion Principle):
La clase NotificacionServicio depende de la abstracción (la interfaz NotificacionEnviar) y no de implementaciones concretas, lo que permite sustituir mecanismos de notificación sin afectar el resto del sistema.

CONCLUSIÓN

El desarrollo del presente Trabajo Práctico me permitió integrar conocimientos teóricos y prácticos sobre Programación Orientada a Objetos, modelado UML y principios SOLID. A través del diseño e implementación del sistema, puede entender la importancia de estructurar correctamente las clases, definir responsabilidades claras y separar componentes para lograr un software mantenible y extensible.
La incorporación de una interfaz gráfica básica también acerca a la experiencia de desarrollar aplicaciones más completas y usables.