Manual Técnico

CARLOS DÍAZ

Introducción

El presente Técnico tiene como objetivo proporcionar una guía detallada sobre el uso del sistema USAC International Bank, una aplicación de escritorio que simula el funcionamiento de una entidad bancaria.

Este sistema ha sido desarrollado con Java y una interfaz gráfica creada con Swing, siguiendo el patrón Modelo-Vista-Controlador (MVC). Está diseñado para permitir a los usuarios realizar operaciones bancarias básicas, como la gestión de cuentas, depósitos, retiros, historial de transacciones y generación de reportes.

El sistema se ejecuta de manera local y no almacena datos de manera persistente; es decir, toda la información ingresada se perderá al cerrar la aplicación.

Los usuarios podrán realizar las siguientes acciones dentro del sistema:

- Inicio de sesión: Acceso con credenciales predefinidas para el administrador.
- Registro de usuarios: Creación de clientes con su información básica (CUI, nombre y apellido).
- Gestión de cuentas: Creación de cuentas bancarias asociadas a clientes registrados.
- Depósitos y retiros: Movimientos de dinero en las cuentas existentes, con validaciones de montos.
- Consulta de cuentas: Visualización de cuentas asociadas a un cliente mediante su CUI.
- Historial de transacciones: Registro de operaciones realizadas en cada cuenta.
- Generación de reportes: Exportación de transacciones en archivos PDF.
- Bitácora del sistema: Registro detallado de las acciones realizadas.

Este manual explica cómo utilizar cada una de estas funciones de manera sencilla y clara, garantizando una experiencia de usuario óptima.

Requerimientos De Hardware

Componente	Mínimo	Recomendado
Procesador	Intel Core i3 o equivalente	Intel Core i5 o superior
Memoria RAM	4 GB	8 GB o más
Espacio en Disco	500 MB disponibles	1 GB disponibles
Resolución de Pantalla	1024x768 píxeles	1366x768 píxeles o más
Periféricos	Teclado y mouse	Teclado, mouse y monitor Full HD

Requerimientos de SoftWare

Componente	Mínimo	Recomendado
Sistema Operativo	Windows 7 / macOS 10.12 / Ubuntu 18.04	Windows 10+ / macOS 11+ / Ubuntu 20.04+
Java JDK	Java SE 8	Java SE 17 o superior
Entorno de Ejecución	JRE 8	JRE 17
Entorno de Desarrollo (opcional)	NetBeans 12 / IntelliJ IDEA Community / Eclipse	IntelliJ IDEA Community o NetBeans 15
Bibliotecas Adicionales	iTextPDF (para generación de reportes)	iTextPDF 7.2.2 o superior

Login

Este método valida las credenciales de inicio de sesión comparando el usuario y la contraseña ingresados con valores predefinidos.

Parámetros:

- usuario: Nombre de usuario ingresado.
- contraseña: Contraseña ingresada.
- loginFrame: Ventana de inicio de sesión.
- Funcionamiento:
- Si el usuario y la contraseña son correctos, muestra un mensaje de bienvenida, cierra la ventana de inicio de sesión y abre el menú principal.
- Si son incorrectos, muestra un mensaje de error indicando que los datos son inválidos.

```
public class loginController {
    private static final String USUARIO_CORRECTO= "ipcD1s2025";
    private static final String CONTRASEÑA_CORRECTA = "ipcLogin";

    public void validarCredenciales (String usuario, String contraseña, JFrame loginFrame){
        if (usuario.equals(USUARIO_CORRECTO) && contraseña.equals(CONTRASEÑA_CORRECTA)){
            alertas.mostrarOk("Bienvenido");
            loginFrame.dispose();
            menuPrincipal menu = new menuPrincipal();
            menu.setVisible(true);
        }else{
            alertas.mostrarError("Datos de ingreso inválidos, revise sus credenciales");
        }
    }
}
```

Menú Principal

- Atributos:
- view: Es la vista del menú principal (tipo menuPrincipal).
- banco: Es un objeto de la clase Banco que representa la lógica de negocio de la aplicación.
- Constructor:
- Se inicializa la vista (view) y se llaman los métodos de configuración de eventos (listeners).
- initListeners():
- Se asignan acciones a los botones del menú principal
- Métodos:
- Cada uno de los métodos como mostrarDatosEstudiante(), crearCuenta(), etc., abre una nueva ventana o vista específica para cada funcionalidad (por ejemplo, creación de cuenta, retiro, etc.) y les asigna su correspondiente controlador.

```
public class menuPrincipalController {
    private menuPrincipal view;
    Banco banco = new Banco();
    public menuPrincipalController (menuPrincipal view){
        this.view = view;
        initListeners();
    }

    private void initListeners(){
        view.getBtnDatosEstudiante().addActionListener(e -> mostrarDatosEstudiante());

        // Botón "Crear Cuenta"
        view.getBtnCrearCuenta().addActionListener(e -> crearCuenta());

        // Botón "Retiros"
        view.getBtnRetiros().addActionListener(e -> realizarRetiro());
```

```
    private void mostrarDatosEstudiante() {
        System.out.println("Mostrando datos del estudiante");
        datosEstudiante datos = new datosEstudiante();
        datos.setVisible(true);
        datos.setLocationRelativeTo(null);
    }

    private void crearCuenta() {
        System.out.println("Creando cuenta");
        crearCuenta vistaCrearCuenta = new crearCuenta();
        new crearCuentaController(vistaCrearCuenta,banco);
        vistaCrearCuenta.setVisible(true);
        vistaCrearCuenta.setLocationRelativeTo(null);
    }
```

Crear Cuentas

- Atributos:
- vista: Representa la vista de creación de cuentas (CrearCuentaView).
- banco: Es el modelo que maneja las operaciones bancarias (BancoModel).
- Constructor:
- Al inicializarse, carga los clientes disponibles del banco y establece los "listeners" (acciones de los botones).
- Método cargarClientes():
- Obtiene la lista de clientes del banco y los muestra en el combo box de la vista, en formato "CUI - nombre completo".
- Método crearCuenta():
- Toma el cliente seleccionado del combo box.
- Si no se selecciona ningún cliente, muestra una advertencia.
- Si se selecciona un cliente, intenta crear una cuenta para ese cliente usando el CUI. Si la creación es exitosa, muestra un mensaje de éxito; si no, muestra un error (si el cliente ha excedido el límite de cuentas).
- Método initListeners():
- Asocia el botón de "Crear Cuenta" con el método crearCuenta(), para que se ejecute cuando el usuario haga clic en dicho botón.

```java
import Models.BancoModel;
import View.*;
import View.AlertasViews;

/**
 * @author charl
 */
public class crearCuentaController {
    private CrearCuentaView vista;
    private BancoModel banco;

    public crearCuentaController(CrearCuentaView vista, BancoModel banco) {
        this.vista = vista;
        this.banco = banco;
        cargarClientes();
        initListeners();
    }

    private void cargarClientes(){
        ClienteModel[] clientes = banco.getClientes();
        vista.getCmbxClientes().removeAllItems();
        for (ClienteModel cliente : clientes){
            if (cliente != null){
                vista.getCmbxClientes().addItem(cliente.getCUI() + " - "+ cliente.getNombreCliente()+ " "+ cliente.getApellidoCliente());
            }
        }
    }
```

Creación de usuarios

- Atributos:
- vistaCrearUsuario: Es la vista donde se ingresa la información del nuevo usuario (CrearUsuarioView).
- banco: Es el modelo que maneja la lógica de negocio para el registro de clientes (BancoModel).
- Constructor:
- Al inicializarse, configura el "listener" para el botón de "Crear Usuario". Esto conecta el botón con el método registrarCliente().
- Método initListeners():
- Asocia el botón "Crear Usuario" con el método registrarCliente(), que se ejecuta cuando el usuario hace clic en el botón.
- Método registrarCliente():
- Obtiene los datos del nuevo cliente (CUI, nombre, apellido) desde los campos de texto en la vista.
- Realiza varias verificaciones:
  - Si algún campo está vacío o incompleto, muestra una advertencia.
  - Si el CUI ya existe en el banco (verificado por el método existeCliente()), muestra una advertencia.
  - Si el cliente no existe, intenta registrarlo usando el método banco.registrarCliente(). Si el registro es exitoso, muestra un mensaje de éxito y limpia los campos. Si el límite de clientes es alcanzado, muestra un mensaje de error.
- Método existeCliente():
- Verifica si el CUI del nuevo cliente ya está registrado en la base de datos del banco. Devuelve true si el cliente existe, o false si no.

```
import Models.BancoModel;
import View.AlertasViews;
import View.CrearUsuarioView;
/**
 *
 * @author charl
 */
public class registroUsuarioController {
    private CrearUsuarioView vistaCrearUsuario;
    private BancoModel banco;

    public registroUsuarioController(CrearUsuarioView vistaCrearUsuario, BancoModel banco) {
        this.vistaCrearUsuario = vistaCrearUsuario;
        this.banco=banco;
        initListeners();
    }
    private void initListeners(){
        vistaCrearUsuario.getBtnCrearUsuario().addActionListener(e-> registrarCliente());
    }

   private void registrarCliente(){
       String CUI = vistaCrearUsuario.getBtnCUI().getText();
       String nombre = vistaCrearUsuario.getBtnNombre().getText();
       String apellido = vistaCrearUsuario.getBtnApellido().getText();

       if (CUI.isBlank() || nombre.isEmpty() || apellido.isBlank()){
            AlertasViews.mostrarAdvertencia("verifique que los campos esten llenos!");
            return;
       }else if(existeCliente(CUI)){
           AlertasViews.mostrarAdvertencia("El CUI: "+CUI+" ya existe!");
       }else if (banco.registrarCliente(CUI,nombre,apellido)){
           AlertasViews.mostrarOk("Cliente registrado exitosamente");
           vistaCrearUsuario.getBtnCUI().setText("");
           vistaCrearUsuario.getBtnNombre().setText("");
           vistaCrearUsuario.getBtnApellido().setText("");
```

mostrar Cuentas por cliente

- Atributos:
- vista: Representa la vista donde el usuario puede buscar cuentas asociadas a un cliente (BuscarCuentasAsociadasView).
- banco: Es el modelo que contiene la lógica de negocio y los datos de los clientes y cuentas (BancoModel).
- Constructor:
- Al inicializarse, carga la lista de clientes en la tabla y configura los "listeners" para los eventos (acciones del usuario).
- Método cargarClientes():
- Carga todos los clientes del banco y los muestra en una tabla dentro de la vista. Cada fila muestra el CUI y el nombre del cliente.
- Método initListeners():
- Asocia el botón "Buscar Cuentas" a la acción buscarCuentasAsociadas(), que se ejecutará cuando el usuario haga clic en dicho botón.
- Método buscarCuentasAsociadas():
- Obtiene el CUI ingresado por el usuario en el campo de texto.
- Si el campo está vacío, muestra una advertencia.
- Si el CUI no corresponde a ningún cliente en el banco, muestra un error.
- Si el cliente existe, obtiene las cuentas asociadas a ese cliente y las muestra en una tabla.

```java
import Models.CuentaModel;
import View.AlertasViews;
import View.BuscarCuentasAsociadasView;
import javax.swing.table.DefaultTableModel;
/**
 *
 * @author charl
 */
public class buscarCuentasAsociadasController {
    private BuscarCuentasAsociadasView vista;
    private BancoModel banco;

    public buscarCuentasAsociadasController(BuscarCuentasAsociadasView vista, BancoModel banco) {
        this.vista = vista;
        this.banco = banco;
        cargarClientes();
        initListeners();

    }

    private void cargarClientes(){
      ClienteModel[] clientes = banco.getClientes();
      DefaultTableModel model = (DefaultTableModel) vista.getTableUsuarios().getModel();
      model.setRowCount(0);

      for(ClienteModel cliente : clientes){
          if(cliente != null){
              Object[] row = {cliente.getCUI(), cliente.getNombreCliente(), cliente.getApellidoCliente()};
              model.addRow(row);
          }
      }
    }
```

Depositos

- Atributos:
- banco: Es el modelo que gestiona los datos y la lógica del banco (BancoModel).
- vista: Es la vista de depósitos donde el usuario puede seleccionar la cuenta y realizar un depósito (DepositosView).
- Constructor:
- Inicializa el controlador con la vista y el banco.
- Configura los "listeners" (acciones del usuario) y carga las cuentas disponibles.
- Método cargarCuentas():
- Carga las cuentas de todos los clientes en el combo box de la vista, mostrando el ID de la cuenta junto con el nombre del cliente.
- Método hacerDeposito():
- Obtiene la cuenta seleccionada y el monto del depósito ingresado por el usuario.
- Verifica si se seleccionó una cuenta y si el monto es mayor que 0. Si no es así, muestra una advertencia.
- Busca la cuenta correspondiente entre todos los clientes y realiza el depósito.
- Si la cuenta no existe, muestra una advertencia. Si el depósito es exitoso, muestra un mensaje de éxito; si se exceden las transacciones permitidas (25), muestra un mensaje de error.
- Método initListener():
- Asocia el botón "Depositar" con el método hacerDeposito(), que se ejecuta cuando el usuario hace clic en el botón.

```
private void hacerDeposito() {
try {
    String objetoSeleccionado = (String) vista.getComboCuentas().getSelectedItem();
    if (objetoSeleccionado == null) {
        AlertasViews.mostrarAdvertencia("Seleccione una cuenta!");
        return;
    }

    String idCuenta = objetoSeleccionado.split(" - ")[0];
    double monto = Double.parseDouble(vista.getTxtMonto().getText().trim());

    if (monto <= 0) {
        AlertasViews.mostrarAdvertencia("El monto debe ser mayor a 0");
        return;
    }

    ClienteModel[] clientes = banco.getClientes();
    boolean cuentaEncontrada = false;

    for (ClienteModel cliente : clientes) {
        if (cliente != null) {
            CuentaModel cuenta = cliente.buscarCuenta(idCuenta);
            if (cuenta != null) {
                cuentaEncontrada = true;
                if (cuenta.realizarDeposito(monto)) {
                    AlertasViews.mostrarOk("Depósito Exitoso, cantidad Depositada: Q" + monto);
                } else {
                    AlertasViews.mostrarError("Se han excedido las transacciones (25)");
                }
                break;
            }
        }
    }
```

Retiros

- Atributos:
- banco: Es el modelo que gestiona los datos y la lógica del banco (BancoModel).
- vista: Es la vista de retiros donde el usuario puede seleccionar la cuenta y realizar un retiro (RetirosView).
- Constructor:
- Inicializa el controlador con la vista y el banco.
- Configura los "listeners" (acciones del usuario) y carga las cuentas disponibles.
- Método cargarCuentas():
- Carga las cuentas de todos los clientes en el combo box de la vista, mostrando el ID de la cuenta junto con el nombre del cliente.
- Método hacerDebito():
- Obtiene la cuenta seleccionada y el monto del retiro ingresado por el usuario.
- Verifica si se seleccionó una cuenta y si el monto es mayor que 0. Si no es así, muestra una advertencia.
- Busca la cuenta correspondiente entre todos los clientes y realiza el retiro.
- Si el saldo de la cuenta es insuficiente, muestra un mensaje de error. Si el retiro excede el límite de transacciones permitidas (25), muestra un error. Si el retiro es exitoso, muestra un mensaje de éxito.
- Método initListener():
- Asocia el botón "Depositar" con el método hacerDebito(), que se ejecuta cuando el usuario hace clic en el botón.

```
private void hacerDebito() {
 try {
    String objetoSeleccionado = (String) vista.getComboCuentas().getSelectedItem();
    if (objetoSeleccionado == null) {
        AlertasViews.mostrarAdvertencia("Seleccione una cuenta!");
        return;
    }

    String idCuenta = objetoSeleccionado.split(" - ")[0];
    double monto = Double.parseDouble(vista.getTxtMonto().getText().trim());

    if (monto <= 0) {
        AlertasViews.mostrarAdvertencia("El monto debe ser mayor a 0");
        return;
    }

    ClienteModel[] clientes = banco.getClientes();
    boolean cuentaEncontrada = false; //

    for (ClienteModel cliente : clientes) {
        if (cliente != null) {
            CuentaModel cuenta = cliente.buscarCuenta(idCuenta);
            if (cuenta != null) {
                cuentaEncontrada = true; //

                if (cuenta.getSaldoCuenta() > 0) {
                    if (cuenta.getSaldoCuenta() >= monto) {
                        if (cuenta.realizarRetiro(monto)) {
                            AlertasViews.mostrarOk("Retiro Exitoso, cantidad Debitada: Q" + monto);
                        } else {
                            AlertasViews.mostrarError("Se han excedido las transacciones (25)");
                        }
                    } else {
                        AlertasViews.mostrarError("Saldo de la cuenta insuficiente");
```

Historial de transacciones

- Atributos:
- banco: Modelo que gestiona los datos y lógica del banco (BancoModel).
- vista: Vista del historial de transacciones (HistorialTransaccionesView), donde se muestra la información al usuario.
- Constructor:
- Inicializa el controlador con el banco y la vista.
- Configura el "listener" para el botón de búsqueda.
- Método buscarCuentaYMostrarHistorial():
- Obtiene el ID de la cuenta desde el campo de texto en la vista.
- Si el ID está vacío, muestra una advertencia.
- Busca la cuenta en los clientes del banco. Si no se encuentra, muestra un error.
- Si la cuenta se encuentra, llena los campos de la vista con el CUI y el nombre del cliente y luego llama al método mostrarHistorialTransacciones para mostrar las transacciones asociadas a la cuenta.
- Método mostrarHistorialTransacciones():
- Muestra las transacciones de la cuenta en una tabla de la vista.
- Para cada transacción, calcula el saldo variable (actualizado después de cada transacción).
- Las transacciones se muestran con los siguientes datos:
  - ID de la transacción
  - Fecha y hora de la transacción
  - Tipo de transacción (depósito o débito)
  - Monto de débito
  - Monto de depósito
  - Saldo actualizado después de cada transacción
- Método initListener():
- Asocia el botón de búsqueda a la acción buscarCuentaYMostrarHistorial() para realizar la búsqueda y mostrar el historial cuando el usuario lo solicite.
  -

.

```java
package Controllers;
import Models.ClienteModel;
import Models.BancoModel;
import Models.CuentaModel;
import Models.TransaccionModel;
import View.HistorialTransaccionesView;
import View.AlertasViews;
import java.text.SimpleDateFormat;
import javax.swing.table.DefaultTableModel;
/**
 *
 * @author charl
 */
public class historialTransaccionesController {
    private BancoModel banco;
    private HistorialTransaccionesView vista;

    public historialTransaccionesController(BancoModel banco, HistorialTransaccionesView vista) {
        this.banco = banco;
        this.vista = vista;
        initListener();
    }
```

```java
private void buscarCuentaYMostrarHistorial(){
    String idCuenta =  vista.getTxtSolicitarIdCuenta().getText().trim();
    if(idCuenta.isEmpty()){
        AlertasViews.mostrarAdvertencia("coloque un Id");
        return;
    }

    ClienteModel clienteEncontrado=null;
    CuentaModel cuentaEncontrada = null;

    ClienteModel[] clientes = banco.getClientes();

    for(ClienteModel cliente : clientes){
        if(cliente != null){
            CuentaModel cuenta = cliente.buscarCuenta(idCuenta);
            if(cuenta !=null ){
            clienteEncontrado = cliente;
            cuentaEncontrada = cuenta;
            break;
            }
        }
    }
    if (clienteEncontrado == null || cuentaEncontrada == null){
        AlertasViews.mostrarError("No se encontro la Cuenta");
        return;
    }
    vista.getTxtMostrarCui().setText(clienteEncontrado.getCUI());
    vista.getTxtMostrarNombre().setText(clienteEncontrado.getNombreCliente());
    vista.getTxtMostrarApellido().setText(clienteEncontrado.getApellidoCliente());

    mostrarHistorialTransacciones(cuentaEncontrada);
}
```

Reportes en PDF de Historial de transacciones

Atributos:

1. banco: Modelo que contiene la lógica del banco (gestiona clientes, cuentas, transacciones).
2. vista: Vista de la interfaz gráfica donde el usuario interactúa con el sistema.

Constructor:

- Inicializa el controlador con el modelo del banco y la vista de generación de reportes.
- Configura los "listeners" para los botones de generación de reportes en la vista.

Métodos:

1. initListeners():
   - Asocia las acciones de los botones en la vista para generar los reportes de transacciones, depósitos y retiros.
2. generarReporteTransacciones(), generarReporteDepositos(), generarReporteRetiros():
   - Estos métodos siguen una estructura similar:
     - Recuperan el ID de la cuenta del campo de texto en la vista.
     - Buscan la cuenta dentro de los clientes del banco.
     - Si la cuenta es encontrada, se genera el reporte correspondiente:
       - Transacciones: Reporte de todas las transacciones de la cuenta.
       - Depósitos: Reporte de solo los depósitos realizados en la cuenta.
       - Retiros: Reporte de solo los retiros realizados de la cuenta.
     - Si no se encuentra la cuenta, se muestra una advertencia.
3. generarReportePDF():
   - Este es el núcleo de la generación del PDF.
   - Crea un archivo PDF utilizando la biblioteca iText y agrega:
     - Información del cliente (CUI, nombre y cuenta).
     - Una tabla con los detalles de las transacciones:
       - ID de transacción, fecha, tipo (depósito o débito), monto de débito y monto de depósito.
     - Si el tipo de transacción es "todos", se incluyen todas las transacciones, de lo contrario, solo se incluyen las del tipo especificado (depósito o débito).
4. obtenerFechaHoraActual():
   - Genera una cadena de texto con la fecha y hora actual en formato ddMMyyyyHHmmss, utilizada para nombrar el archivo PDF de forma única.

Flujo de trabajo:

1. El usuario ingresa un ID de cuenta en la vista y selecciona el tipo de reporte que desea generar (transacciones, depósitos o retiros).
2. Si el ID de cuenta es válido, el controlador genera el reporte correspondiente en formato PDF.
3. El archivo PDF se guarda en una ruta predefinida con un nombre que incluye la fecha y hora actual.

Manejo de Errores:

- Si no se encuentra la cuenta o si el campo de ID de cuenta está vacío, el controlador muestra mensajes de advertencia o error usando la clase AlertasViews.
- Se captura cualquier excepción relacionada con la creación del PDF (DocumentException o IOException) y se imprime el error en consola.

Reportes en PDF de Historial de transacciones

```
private void generarReportePDF(ClienteModel cliente, CuentaModel cuenta, String filePath, String titulo, Str
Document document = new Document();
try {
    PdfWriter.getInstance(document, new FileOutputStream(filePath));
    document.open();

    // Agregar información del cliente
    document.add(new Paragraph(titulo));
    document.add(new Paragraph("CUI: " + cliente.getCUI()));
    document.add(new Paragraph("Nombre: " + cliente.getNombreCliente() + " " + cliente.getApellidoCliente()
    document.add(new Paragraph("Cuenta: " + cuenta.getIdCuenta()));
    document.add(new Paragraph(" "));

    // Crear tabla para las transacciones
    PdfPTable table = new PdfPTable(5);
    table.addCell("ID Transacción");
    table.addCell("Fecha");
    table.addCell("Tipo");
    table.addCell("Monto Débito");
    table.addCell("Monto Depósito");

    TransaccionModel[] transacciones = cuenta.getTransacciones();
    SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy HH:mm");

    for (TransaccionModel transaccion : transacciones) {
        if (transaccion != null) {
            // Mostrar todas las transacciones si el tipo es "todos"
            if (tipoTransaccion.equals("todos") || transaccion.getTipo().equals(tipoTransaccion)) {
                table.addCell(transaccion.getIdTransaccion());
                table.addCell(dateFormat.format(transaccion.getFecha()));
                table.addCell(transaccion.getTipo());
                table.addCell(String.valueOf(transaccion.getMontoDebito()));
                table.addCell(String.valueOf(transaccion.getMontoDeposito()));
```

```
private void generarReporteRetiros() {
    String idCuenta = vista.getTxtCuenta().getText().trim();
    if (idCuenta.isEmpty()) {
        AlertasViews.mostrarAdvertencia("Por favor, ingrese el ID de la cuenta.");
        return;
    }
```

Conclusiones y recomendaciones

- El sistema bancario desarrollado en Java con NetBeans y Swing cumple con los objetivos establecidos al permitir la gestión eficiente de cuentas y transacciones bancarias. La interfaz gráfica de usuario, construida con Swing, facilita la interacción con los administradores, quienes pueden crear usuarios, gestionar cuentas, realizar depósitos y débitos, y generar historiales de transacciones en formato PDF mediante la biblioteca iText. Esta solución.
- Este sistema bancario provee una base sólida para la gestión de cuentas de usuarios dentro de una entidad bancaria, con funcionalidades clave como la búsqueda por CUI y la creación de reportes PDF. Aunque en esta versión no se contempla la interacción entre usuarios, el sistema está diseñado de manera escalable, lo que permitirá la futura implementación de nuevas funcionalidades, como la gestión de transacciones entre clientes, la inclusión de métodos de autenticación más avanzados o la conexión con bases de datos reales. La utilización de tecnologías como iText para la generación de documentos PDF proporciona un valor añadido al sistema, permitiendo la documentación precisa y oficial de las transacciones, lo que facilita la auditoría y el seguimiento.

Recomendaciones

- Asegúrese de ingresar datos correctos:
- Al crear un nuevo usuario o cuenta, verifique que la información ingresada, como el CUI, nombre, y otros datos, sea correcta y esté bien escrita. Esto evitará errores en la búsqueda o en la realización de transacciones.

- Para mantener un control adecuado sobre las cuentas, revise frecuentemente el historial de transacciones de cada cuenta. Esto le permitirá detectar cualquier error o actividad no deseada de manera temprana.

:

- Cuando necesite un respaldo o informe de las transacciones realizadas, utilice la función de generación de PDF. Este documento le permitirá tener un registro oficial que puede ser guardado o impreso para futuras referencias.