



Universidad De Talca

Facultad De Ingeniería

Departamento De Ciencias De La Computación

Informe Tarea 3 Programación Avanzada

Tarea Simulación Bancaria con GUI

Fecha: 14/06/2021

Autores: Valentina Yáñez

Antonia Fuenzalida

Diego Fernández

e-mail: vyañez20@alumnos.utalca.cl

afuenzalida20@alumnos.utalca.cl

[dfernandez19@alumnos.utalca.c](mailto:dfernandez19@alumnos.utalca.cl)

Introducción

En este informe damos a conocer nuestra interfaz gráfica de usuario basada en una simulación bancaria, esta simulación permite a los clientes saber el rendimiento de sus productos, además permite contemplar diferentes escenarios y situaciones que el cliente podría presentar, todo mediante una GUI. Primero abordamos el programa en un UML, en este solo complementamos con lo nuevo que sería la interfaz, luego utilizamos nuestro código anterior y se llevó a cabo la interfaz la cual fue creada en javaFXML, donde aplicamos lo aprendido en la unidad tres de Programación Avanzada.

Análisis del problema

Este programa es una simulación bancaria que permite a los clientes saber los movimientos de sus cuentas, su rendimiento, además contempla diferentes situaciones a las que el cliente podría presentarse. Esta simulación guarda el nombre y la cédula del cliente con la cual se puede identificar la cuenta de él. Consta de tres productos financieros. Su cuenta de ahorro en donde el cliente recibe un interés mensual, una cuenta de ahorro que no recibe interés y un certificado de depósito a término, en esta cuenta el cliente no puede consignar ni retirar dinero.

Este problema es la continuación de la tarea dos en donde aplicamos nuevas funcionalidades a la simulación bancaria y ahora debíamos implementar las funcionalidades en una interfaz gráfica de usuario, en el cual se debía visualizar y simular un banco. Todo esto aplicado en un UML para luego poder pasarlo a código java.

Solución del problema

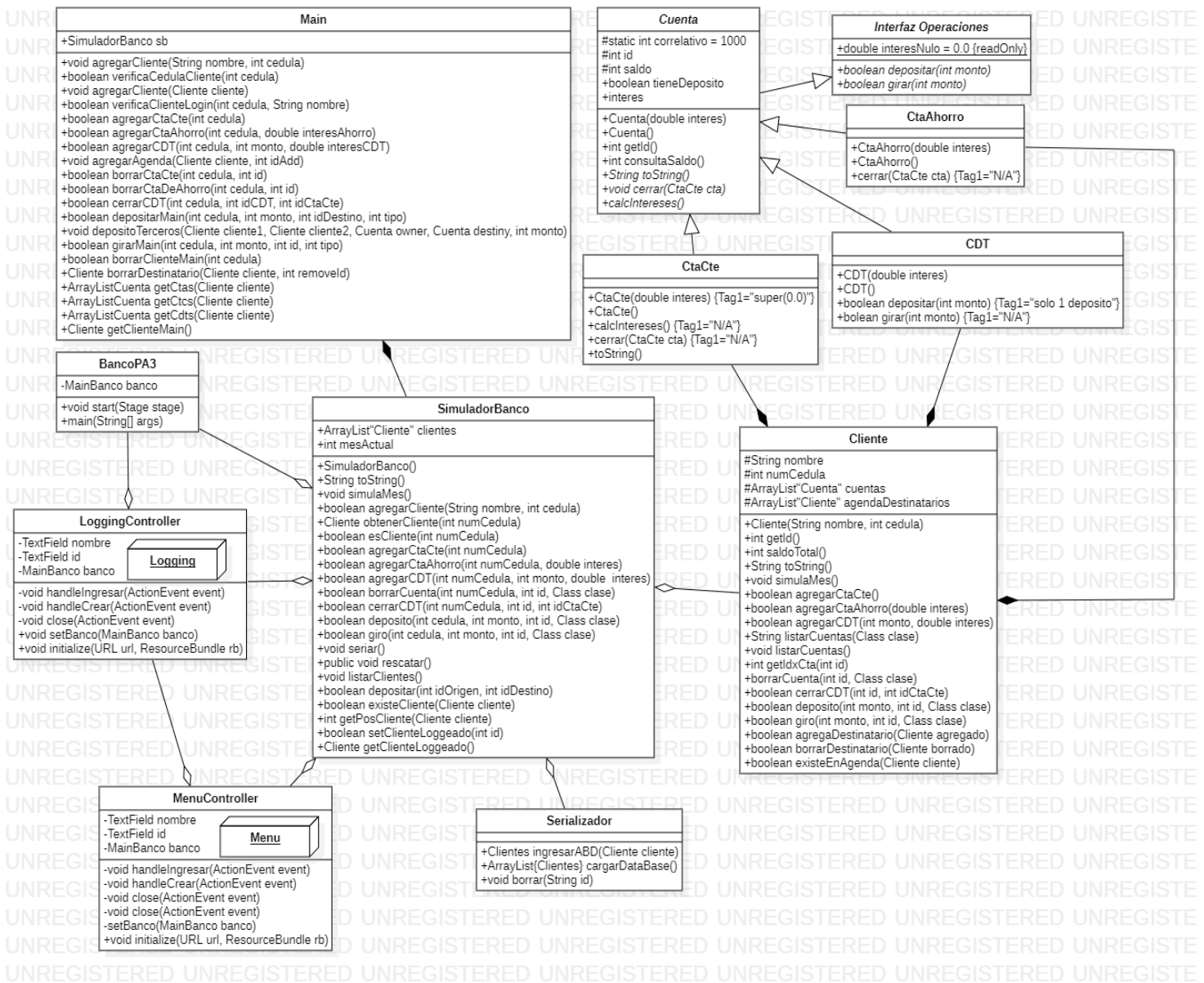
- **Descripción de la solución**

En esta sección se da a conocer nuestra solución aplicada en UML.

Este UML fue reutilizado de nuestro trabajo anterior y ahora solo lo complementamos con lo nuevo que aplicamos para crear la interfaz.

El profesor nos proporcionó una base del UML y nosotros lo terminamos.

Para que el UML no fuera tan largo el profesor nos dejó describir solo los controladores y las vistas principales.



- **Implementación:**

Este simulador cuenta con funcionalidades las cuales fueron aplicadas en una GUI en JavaFXML. En esta parte del informe explicaremos lo nuevo que fue implementado, Todas las nuevas funcionalidades que implementamos dentro del modelo de nuestro programa son idénticas a la versión previa del programa orientado a consola, solamente cambia un poco el cuerpo ya que las salidas ahora ya no se ven por consola, sino que se ven por interfaz de usuario.

```
public void agregarCliente(String nombre, int cedula){
    sb.agregarCliente(nombre, cedula);
    sb.seriar();
}
```

Se implementó un sistema de logeo por el cual se verifica en el modelo si el cliente que se está intentando logear realmente existe.

```
public boolean verificaClienteLogin(int cedula, String nombre){
    if (sb.esCliente(cedula)) {
        if (sb.verificaNombre(cedula, nombre)){
            sb.setClienteLoggeado(cedula);
            return true;
        }
    }
    return false;
}
```

Funciones de manipulación de cuentas para agregar una nueva cuenta a un cliente del banco.

```
public boolean agregarCtaCte(int cedula){
    if (sb.esCliente(cedula)) {
        if (sb.agregarCtaCte(cedula)) {
            sb.seriar();
            return true;
        }
    }
    return false;
}

public boolean agregarCtaAhorro(int cedula, double interesAhorro){
    if (sb.esCliente(cedula)) {
        if (sb.agregarCtaAhorro(cedula, interesAhorro)) {
            sb.seriar();
            return true;
        }
    }
    return false;
}

public boolean agregarCDT(int cedula, int monto, double interesCDT){
    if (sb.esCliente(cedula)) {
        if (sb.agregarCDT(cedula, monto, interesCDT)) {
            sb.seriar();
            return true;
        }
    }
    return false;
}
```

Gestión de cuentas para borrar. Borra una cuenta dentro de las cuentas de un cliente, dependiendo del tipo de cuenta que se llama, se borra una cuenta.

```
public boolean borrarCtaCte(int cedula, int id){
    if (sb.esCliente(cedula) && sb.borrarCuenta(cedula, id, CtaCte.class)) {
        sb.seriar();
        return true;
    }
    return false;
}
public boolean borrarCtaDeAhorro(int cedula, int id){
    if (sb.esCliente(cedula) && sb.borrarCuenta(cedula, id, CtaAhorro.class)) {
        sb.seriar();
        return true;
    }
    return false;
}
}
public boolean cerrarCDT(int cedula, int idCDT, int idCtaCte){
    if (sb.esCliente(cedula) && sb.cerrarCDT(cedula, idCDT, idCtaCte)) {
        sb.seriar();
        return true;
    }
    return false;
}
}
```

Agregar agenda. Agrega un destinatario a la agenda de un cliente.

```
public void agregarAgenda(Cliente cliente, int idAdd){
    for (Cliente aux : sb.clientes) {
        if(aux.getId() == idAdd){
            cliente.agregaDestinatario(aux);
        }
    }
    sb.seriar();
}
```

Deposito entre cuentas. Depositar entre las cuentas internas del cliente.

```
public boolean depositarMain(int cedula, int monto, int idDestino, int tipo){
    Class<? extends Cuenta> clase;
    switch(tipo) {
        case 1: clase = CtaCte.class; break;
        case 2: clase = CtaAhorro.class; break;
        default: clase = null; break;
    }
    if (sb.esCliente(cedula) && sb.deposito(cedula, monto, idDestino, clase)) {
        sb.seriar();
        return true;
    }
    return false;
}
}
```

Deposito terceros. Para depositar a cuentas terceras.

```
public void depositoTerceros(Cliente cliente1, Cliente cliente2, Cuenta owner, Cuenta destiny, int monto) {
    sb.depositar(cliente1.getId(), cliente2.getId(), owner.getId(), destiny.getId(), monto);
    sb.seriar();
}
public boolean girarMain(int cedula, int monto, int id, int tipo){
    Class<? extends Cuenta> clase = (tipo==1)? CtaCte.class : ((tipo==2)? CtaAhorro.class:null);
    if (sb.esCliente(cedula) && sb.giro(cedula, monto, id, clase)) {
        sb.seriar();
        return true;
    }
    return false;
}
```

Borra un cliente según su cédula, esto se realiza solo si el saldo de sus cuentas es 0, al momento de borrar al cliente se elimina de la base de datos.

```
public boolean borrarClienteMain(int cedula){
    Cliente clieAux = sb.obtenerCliente(cedula);
    if(sb.existeCliente(clieAux) && clieAux.saldoTotal()==0){
        Serializador s = new Serializador();
        sb.clientes.remove(clieAux);
        s.borrar(Integer.toString(clieAux.getId()));
        sb.seriar();
        return true;
    }
    return false;
}
```

Borra un destinatario de la Agenda por cédula. Si los clientes existen se borra al cliente de la agenda, en caso de no existir no se borra nada.

```
public Cliente borrarDestinatario(Cliente cliente, int removeId) {
    for (Cliente aux : sb.clientes) {
        if(aux.getId() == removeId){
            System.out.println("Voy a borrar a: "+aux);
            cliente.borrarDestinatario(aux);
        }
    }
    System.out.println(cliente.agendaDestinatarios);
    sb.seriar();
    return cliente;
}
```

Obtención de cuentas específicas de un cliente por clase. Recorren el ArrayList de cuentas del cliente y genera un ArrayList con las cuentas filtradas a un tipo de cuenta específica.

```
public ArrayList<Cuenta> getCtas(Cliente cliente) throws IOException{
    sb.rescatar();
    ArrayList<Cuenta> cuentas = new ArrayList<Cuenta>();
    for (Cuenta cuenta : cliente.cuentas){
        if(cuenta.getClass().equals(CtaAhorro.class)){
            cuentas.add(cuenta);
        }
    }
    return cuentas;
}

public ArrayList<Cuenta> getCtes(Cliente cliente) throws IOException {
    sb.rescatar();
    ArrayList<Cuenta> cuentas = new ArrayList<Cuenta>();
    for (Cuenta cuenta : cliente.cuentas){
        if(cuenta.getClass().equals(CtaCte.class)){
            cuentas.add(cuenta);
        }
    }
    return cuentas;
}

public ArrayList<Cuenta> getCdts(Cliente cliente) throws IOException {
    sb.rescatar();
    ArrayList<Cuenta> cuentas = new ArrayList<Cuenta>();
    for (Cuenta cuenta : cliente.cuentas){
        if(cuenta.getClass().equals(CDT.class)){
            cuentas.add(cuenta);
        }
    }
    return cuentas;
}
```

Obtiene el cliente logeado. Al momento de logearse un usuario, se guarda el cliente que se logeo en el simulador bancario.

```
public Cliente getClienteMain() throws IOException {
    sb.rescatar();
    return sb.getClienteLoggeado();
}
```

Las funciones anteriormente explicadas son las que se encuentra en nuestro modelo de programa, ya que nuestro programa utiliza una arquitectura diseño Modelo Vista Programador (MVC).

A continuación, se explicará las implementaciones del programa, omitiendo los detalles irrelevantes de este.

Dado que tenemos que implementar un nuevo tipo de arquitectura de diseño en nuestro código, lo que hicimos fue crear una nueva clase main, llamándose esta bancoPA3, lo que hace es instanciar la primera ventana de vista, la cual sería el login.

Controladores y vistas del programa

Todas las vistas están enlazadas con su propio controlador, cada controlador va a ir compartiendo una instancia de cliente y banco.

- En el controlador de login, lo que se hace es esperar a un evento. En nuestro controlador de login existen dos eventos, el evento ingresar y el evento crear, en el evento crear se instancia una nueva ventana con una vista que se llama agregar cliente, que es para crear un nuevo cliente. En evento ingresar lo que se hace es verificar la existencia del cliente en el banco y de existir se instancia una nueva ventana llamada menú y se cierra la ventana anterior.
- El MenuController espera y responde a los siguientes eventos:

```
@FXML protected void handleCerrar(ActionEvent event) { ...
@FXML private void handleCtaView(ActionEvent event) throws IOException { ...
@FXML private void handleCtcView(ActionEvent event) throws IOException { ...
@FXML private void handleCdtView(ActionEvent event) throws IOException { ...
@FXML private void handleDestinatario(ActionEvent event){ ...
@FXML private void handleCtaT(ActionEvent event){ ...
@FXML private void handleCtcT(ActionEvent event){ ...
@FXML private void handleVerDestinatario(ActionEvent event) { ...
```

- HandleCerrar, al escuchar este evento nuestro controlador lo que va a hacer es crear una instancia de un nuevo login y cerrar lo que estaba abierto con anterioridad.
- HandleCtaView, este evento dependiendo de si tiene una cuenta o no, va a crear una instancia dentro de la misma ventana de una vista, esta vista va a ser personalizada dependiendo de la existencia de una cuenta de ahorro en el cliente.
- HandleCtaView, este evento dependiendo de si tiene una cuenta o no, va a crear una instancia dentro de la misma ventana de una vista, esta vista va a ser personalizada dependiendo de la existencia de una cuenta corriente en el cliente.
- HandleCtdView, este evento dependiendo de si tiene una cuenta o no, va a crear una instancia dentro de la misma ventana de una vista, esta vista va a ser personalizada dependiendo de la existencia de una cuenta Ctd en el cliente.
- HandleDestinatario, aquí se crea una instancia dentro de la misma ventana de la vista TransferirDestinatario, para poder realizar una transferencia a terceros.
- HandleCtaT, acá se crea una instancia dentro de la misma ventana de la vista TransferirCta, esta es para manejar el dinero de las mismas cuentas de ahorro del cliente.
- HandleCtcT, acá se crea una instancia dentro de la misma ventana de la vista TransferirCtc, esta es para manejar el dinero de las mismas cuentas corrientes del cliente.
- HandleVerDestinatario, esta instancia una nueva vista VerDestinatario dentro de la misma ventana y esta vista nos permite manipular y ver los destinatarios de la agenda de los clientes.

- **Modo de uso:**

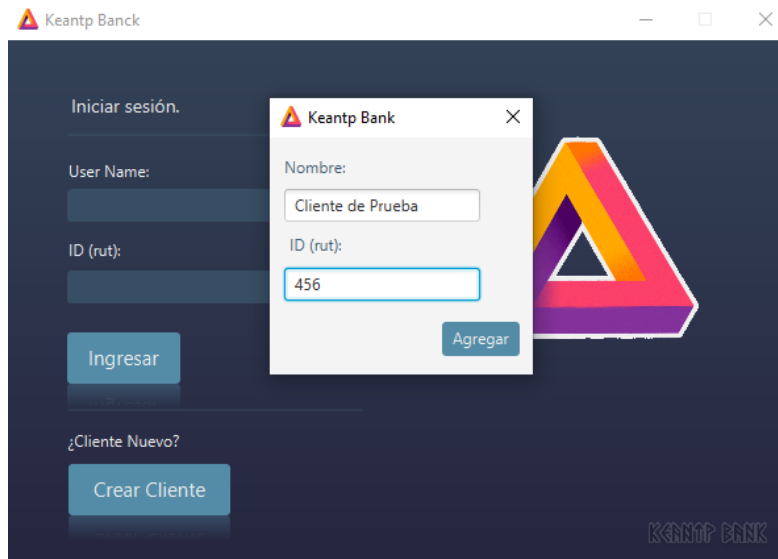
Nuestro programa fue escrito en el lenguaje Java, su JDK es javaSe-16 y su IDE de desarrollo fue NetBeans 12, el editor de texto fue Visual studio code, para implementar la GUI se ocupó java 11, en su versión de JavaFXML.

Para poder utilizarlo se ejecuta el código con vscode en la carpeta .vscode hay perfiles de ejecución que se pueden utilizar, el de launch external terminal, lo ejecuta en una terminal a parte y launch app lo ejecuta en la terminal de visual, existe una opción alternativa a visual estudio se llama “vscodium”, de todas maneras debe ser posible ejecutarlo por símbolos del sistema mediante las líneas de comandos del lenguaje, en la carpeta “src” se contiene el path de estructuras y la carpeta “bin” contiene los archivos compilados. Las entradas del programa son por teclado, al momento de ejecutar el programa se cargará la “base de datos” de la carpeta “DataBase”, este programa utiliza serialización a los clientes del arraylist de clientes de simuladorbanco, en la carpeta “DataBase” se encuentran los clientes ordenados por id, cada vez que se realice una operación se cargará el cambio en la base de datos para así evitar perdida de información.

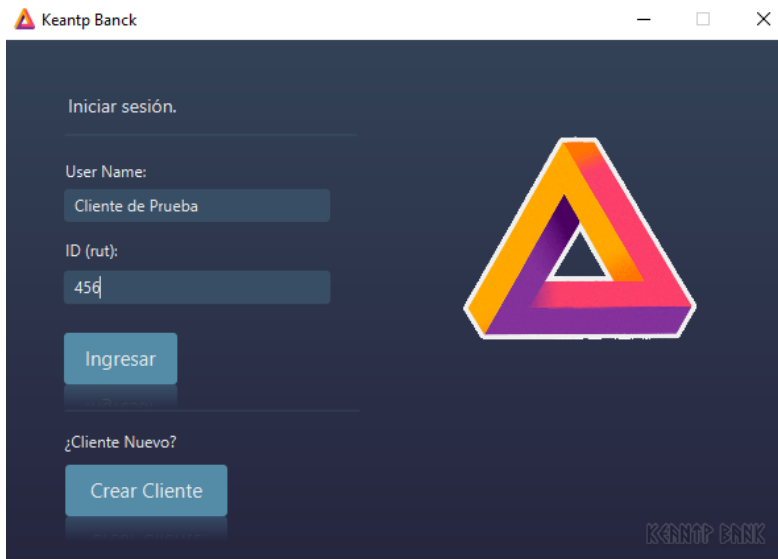
Pruebas

A continuación, se darán a conocer nuestras pruebas realizadas al momento de compilar el código. Los casos de prueba van progresivos, ya que estamos mostrando todas las pruebas del programa.

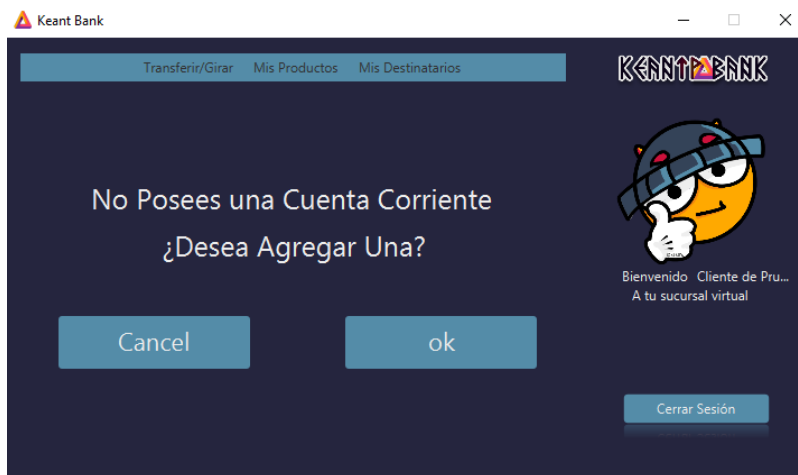
Agrega un cliente.



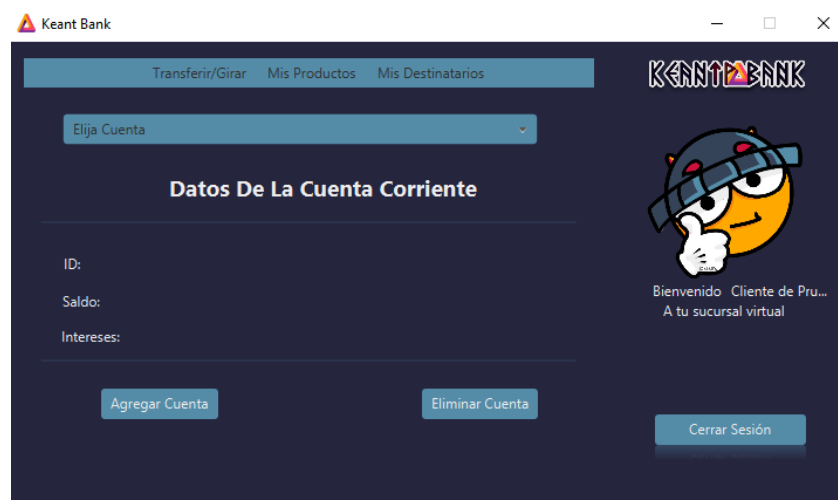
Logear con el cliente creado.




Vamos si tiene cuenta corriente y como no tiene muestra un mensaje para que pueda agregar una cuenta.



Se crea la cuenta corriente.



Se intenta transferir dinero.



Keant Bank

Transferir/Girar Mis Productos Mis Destinatarios

Trasferir/Girar entre Cuenta Corriente


Seleccione su cuenta corriente:

>>id: 1001, saldo: 0<<

Monto:

100000

Girar Transferir Cerrar Sesión



KENTBANK

Bienvenido Cliente de Pru...
A tu sucursal virtual

Verifica si el dinero que se transfirió está.



Keant Bank

Transferir/Girar Mis Productos Mis Destinatarios

>>id: 1001, saldo: 100000<<

Datos De La Cuenta Corriente

ID:	1001
Saldo:	\$100000
Intereses:	0.0%

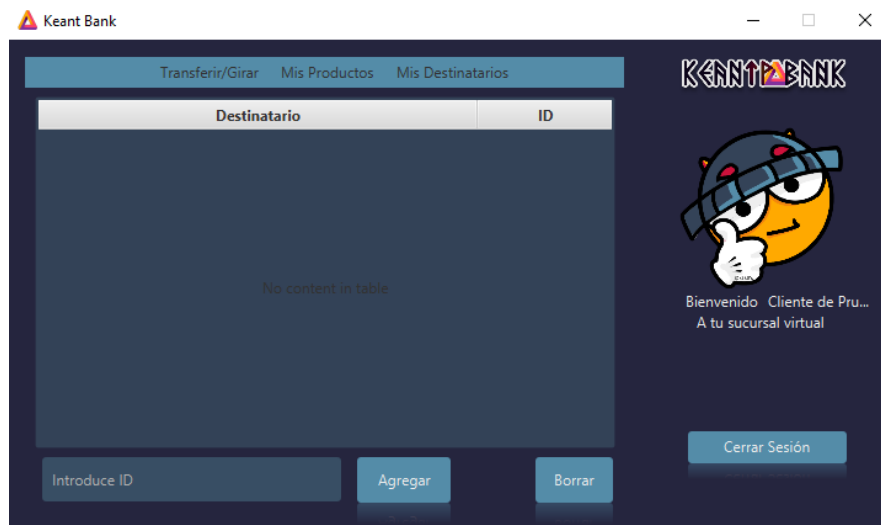
Agregar Cuenta Eliminar Cuenta Cerrar Sesión



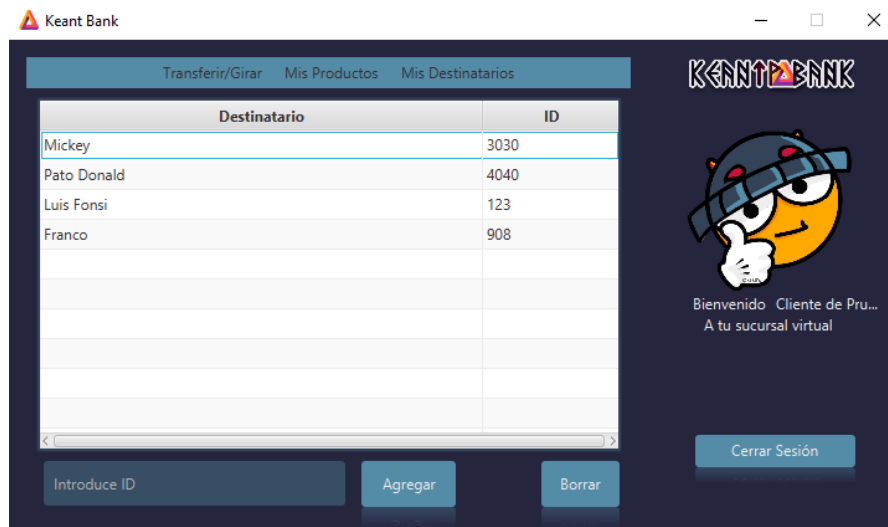
KENTBANK

Bienvenido Cliente de Pru...
A tu sucursal virtual

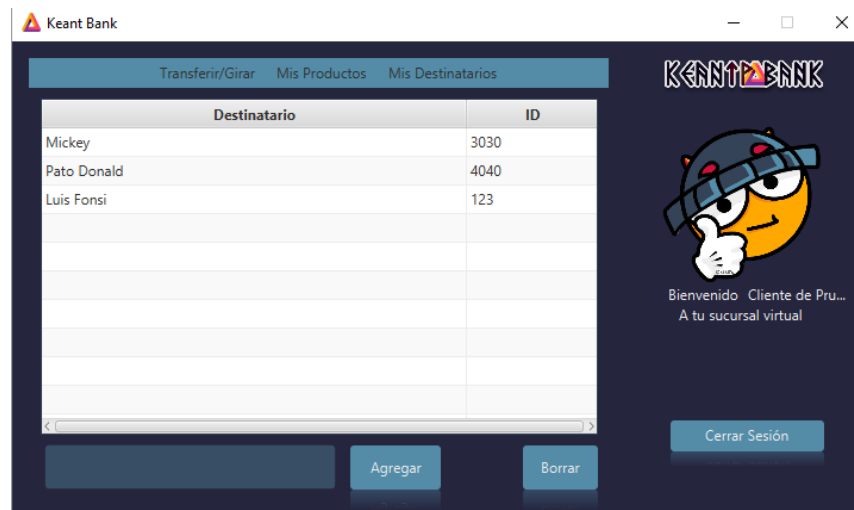
Testeo de la agenda de contactos, como no tiene contactos no hay nada.



Se agregan cuatro clientes.



Se borra el último contacto.



Se intenta mandar dinero a uno de los contactos.

