Curso: 2023 – 2024

IES Melchor Gaspar de Jovellanos

Desarrollo de Aplicaciones Multiplataforma

Coordinador - Sergio De Mingo

Realizado por

Alberto Colmenar Casas

Santiago Estevez Corvillo

Índice

1. Resumen

Nuestro proyecto de fin de curso se titula “UniversalBank”. Donde desarrollamos y programamos una aplicación de finanzas personales, con el objetivo de facilitar a los usuarios un manejo y seguimiento de sus finanzas de una forma simple y cómoda, pero sobre todo segura.

Nuestra aplicación permite la gestión de distintas cuentas bancarias, cada una con sus propias tarjetas y movimientos propios. No solo proporciona información detallada sobre los movimientos bancarios del usuario, sino que también ofrece la capacidad de generar informes precisos. La aplicación es totalmente independiente de cualquier tipo de servicio en línea o algún banco, gracias a esta independencia, los usuarios no necesitarán una conexión a internet para gestionar sus finanzas, lo que garantiza una mayor flexibilidad y control sobre sus datos bancarios , permitiendo manejar sus datos sensibles con mucha más seguridad y confianza.

Con todo esto, creemos que esta aplicación va a permitir a los usuarios tomar decisiones más precisas y acertadas sobre su dinero.

2. Abstract

Our end-of-course project is titled "UniversalBank." We developed and programmed a private banking application for personal use, aimed at facilitating users with a simple, comfortable, and above all, secure way to manage and track their finances.

Our application allows the management of different bank accounts, each with its own cards and transactions. It not only provides detailed information about the user's banking transactions but also offers the ability to generate accurate reports. The application is completely independent of any online service or bank. Thanks to this independence, users will not need an internet connection to manage their finances, ensuring greater flexibility and control over their banking data, allowing them to handle their sensitive information with much more security and confidence.

With all of this, we believe that this application will enable users to make more

precise and informed decisions about their money.

3. Planteamiento del problema y justificación

En la actualidad, nos hemos dado cuenta de que hay muy pocas aplicaciones que te ayuden a realizar un seguimiento de tu dinero de forma eficiente sin la necesidad de estar vinculadas a un banco en concreto. Y las pocas que existen no cumplen con las exigencias de los clientes. Es por eso, que desde UniversalBank hemos decidido crear una aplicación que escuche las necesidades de los usuarios y se adapte a ellos, dándoles la oportunidad que se merecen.

En UniversalBank creemos que todo lo relacionado con el dinero personal de nuestros clientes debe tratarse con la máxima seguridad posible. Estamos seguros de que esto no es incompatible con una experiencia amigable y sencilla para el usuario. Con una interfaz simple e intuitiva queremos evitar que los usuarios se sientan perdidos a la hora de buscar lo que necesitan, permitiéndoles realizar sus gestiones de forma rápida y eficaz, reduciendo enormemente el tiempo utilizado.

Estas mejoras hacen que nuestra aplicación facilite la gestión segura del dinero personal de nuestros clientes. Queremos que, cuando uses tu dinero, seas consciente de toda la información necesaria para que puedas tomar la mejor decisión en cada momento.

4. Desarrollo

4.1 Organización/Planificación

Para planificar nuestro proyecto, decidimos utilizar “**GitHub**” y “**Trello**”.

GitHub es una plataforma de desarrollo colaborativo basada en la web que utiliza Git para el control de versiones. Proporciona un conjunto de herramientas y servicios para facilitar la colaboración en proyectos de software. Entre sus principales características se incluyen:

- Permitir a los desarrolladores almacenar y gestionar su código.
- Realizar un seguimiento de los cambios en el código.

- Facilitar la revisión de código, gestión de problemas y discusión entre equipos.
- Automatizar procesos de pruebas y despliegues.
- Ofrecer foros de discusión y sistemas de comentarios.

GitHub ha sido un elemento fundamental a la hora de realizar nuestro proyecto, ofreciéndonos una plataforma centralizada para almacenar y gestionar nuestro código de una forma clara y sencilla, facilitandonos una colaboración fluida entre el equipo.

Trello es una herramienta de gestión de proyectos basada en tableros visuales que permite a los equipos organizar y priorizar tareas de manera colaborativa. Cuenta con una interfaz intuitiva y flexible, proporcionando un espacio virtual donde el equipo puede crear listas de tareas, asignarlas distintas etapas y más.

En nuestro caso, Trello ha ayudado en gran medida en nuestro proyecto, al proporcionarnos una plataforma visual hemos podido planificar nuestro proyecto con gran eficacia y flexibilidad. Además, después de planificarlo, hemos seguido utilizando para ir compartiendo nuestro progreso con las distintas tareas en el grupo, permitiéndonos con un simple vistazo rápido ver que tareas faltan por realizar y quienes las están haciendo.

Además de hacer uso de estas herramientas, hemos estado llevando a cabo reuniones regulares cada dos semanas, tras las sesiones de tutoría, donde los miembros del equipo nos hemos reunido para discutir y definir nuestra estrategia de desarrollo para las próximas semanas. Durante estos encuentros, hemos trabajado en conjunto para planificar nuestras acciones e identificar y solucionar posibles obstáculos que hayan podido ir sucediendo durante el desarrollo.

4.2 Datos técnicos

Nombre de la aplicación: “UniversalBank”.
Plataforma: Ordenador.
Lenguaje de programación:
JavaFX: Para la programación y la parte lógica del programa.
FXML y CSS: Para la parte visual del programa.
Requisitos del sistema:
Requisitos de Hardware
CPU: Procesador de 500 MHz o superior.
RAM: 1GB de memoria RAM.

Espacio en disco: 10 MB.

Resolución de pantalla: 1300x750 píxeles.

Requisitos de Software

Sistema Operativo: Windows XP , MacOS 10.7 o Linux.

Java Development Kit (JDK) 8 o superior.

MongoDB Server 4.0 o superior.

4.3 Funcionalidad

4.3.1 Características

UniversalBank permite al usuario crear una cuenta nueva e iniciar sesión en nuestra aplicación. Esta cuenta estará vinculada con distintas subcuentas que serán interpretadas como cuentas bancarias de distintos bancos.

Cada una de estas subcuentas permiten la creación de tarjetas bancarias, las cuales serán representaciones virtuales de tus tarjetas reales. Se podrán bloquear/cancelar y rehabilitar en cualquier momento, dependiendo de las necesidades del usuario. Además, el usuario podrá gestionar sus movimientos bancarios e incluso crear y visualizar sus hipotecas/préstamos.

El usuario también podrá visualizar sus historial de movimientos e incluso generar un informe con los movimientos bancarios realizados en los últimos años.

4.3.2 Modelo Vista Controlador

Como hemos utilizado JavaFX en nuestro proyecto, decidimos usar el modelo vista controlador para formar y representar nuestra estructura de datos.

- **modelo:** Está formado por clases de objetos que definen los datos y la lógica de nuestro proyecto.
- **Vista:** Lo forman archivos *".fxml"* que dan forma a la interfaz del usuario. Usando la herramienta SceneBuilder.
- **Controlador:** Son clases que están formadas por métodos que ayudan con la interacción entre las clases del modelo y la vista.

Diagramas:

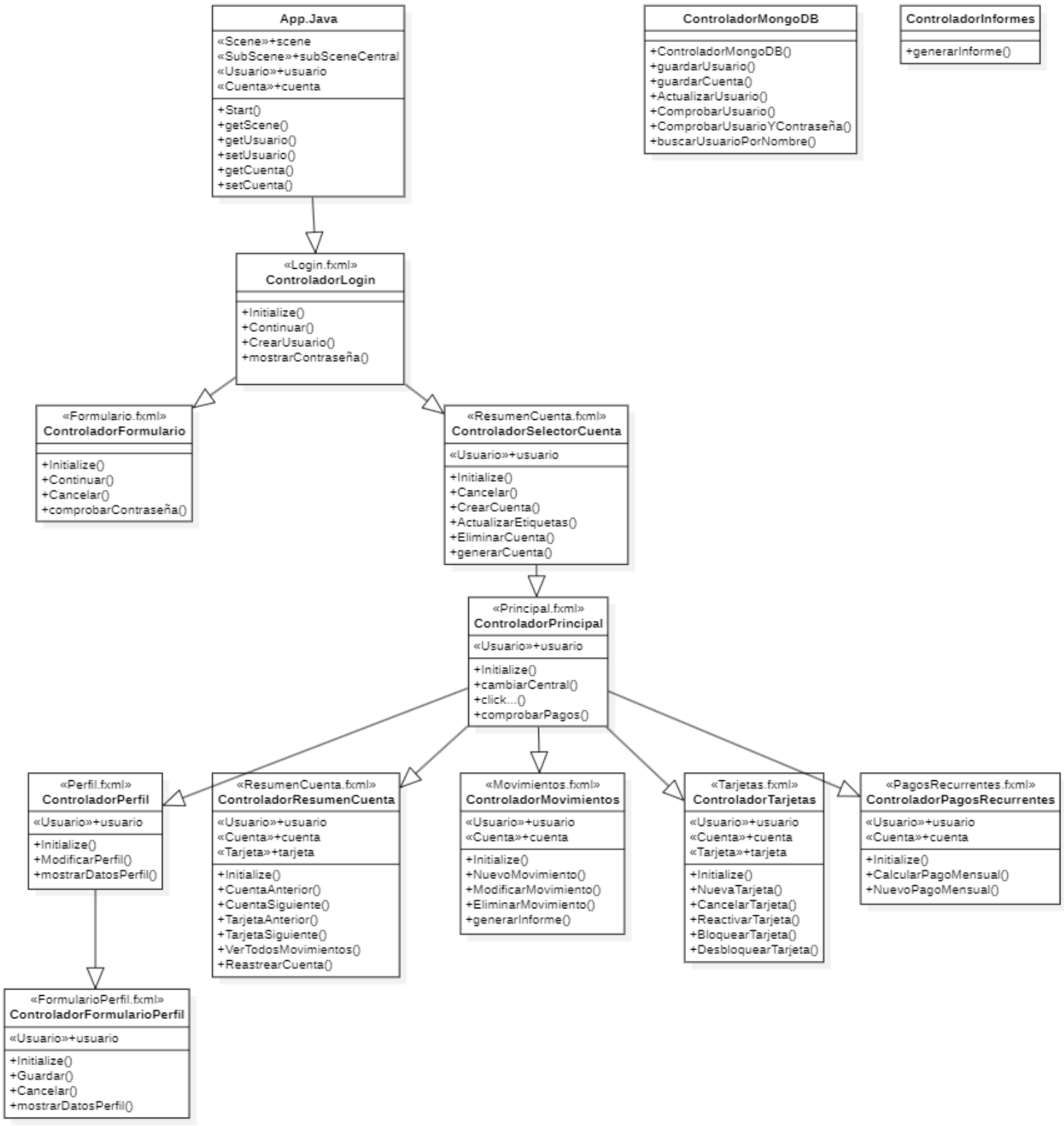

Imagen 1: Diagrama de clases entre controladores.

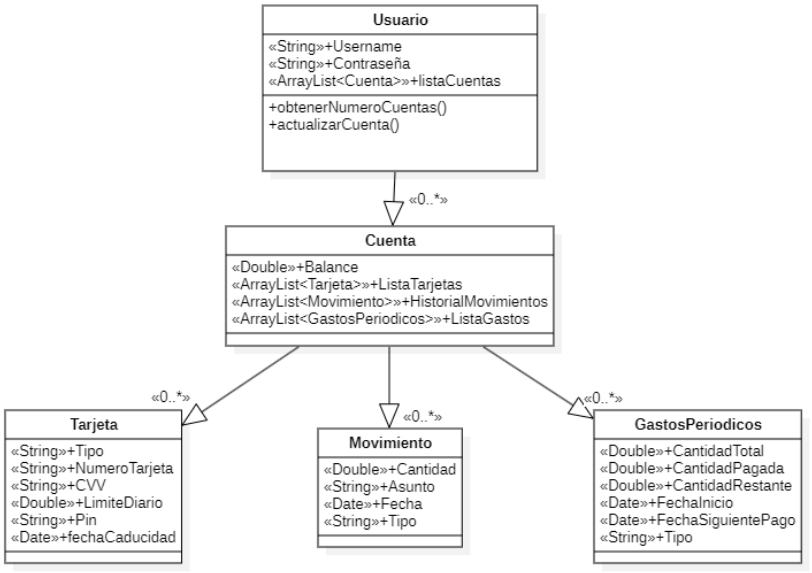

Imagen 2: Diagrama de las clases del proyecto.

4.3.3 Métodos más importantes

Durante el desarrollo de nuestra aplicación, hemos identificado ciertos puntos en los que ha sido necesario crear métodos fundamentales para asegurar su correcto funcionamiento. Algunos de estos métodos son los siguientes:

actualizarEtiquetas(): Este método se encuentra en *"ControladorSelectorCuenta"* y se encarga de comprobar el número de cuentas asociadas al usuario conectado. Por cada cuenta encontrada, la enmarca en una Label y la coloca en la posición correcta en la pantalla. Además, agrega varios eventos a la Label para poder hacer click sobre ella o borrarla.

```
ArrayList<Cuenta> listaCuentas = usuario.getListaCuentas();
int cont = 0;

int numColumns = 2; // Número deseado de columnas

double columnSpacing = 50;
double anchorPaneWidth = 1200;

double totalColumnsWidth = numColumns * 300 + (numColumns - 1) * columnSpacing;
double emptySpace = anchorPaneWidth - totalColumnsWidth;
double leftMargin = emptySpace / 2;

GridPane gridPane = new GridPane();
gridPane.setHgap(columnSpacing);
gridPane.setVgap(20); // Ajustar el espaciado vertical entre las filas

for (Cuenta cuenta : listaCuentas) {
    int id = cuenta.getId();
    Double balance = cuenta.getBalance();

    Label label = new Label("ID:  " + id + "\nBalance: " + balance);
    label.setMinWidth(300);
    label.setMaxWidth(300);
    label.getStyleClass().addAll("card", "cuenta");

    Button btnEliminar = new Button("X");
    btnEliminar.setOnAction(event -> eliminarCuenta(cuenta));

    HBox hbox = new HBox(label, btnEliminar);
    hbox.setSpacing(10);
    hbox.setAlignment(Pos.CENTER); // Alineación centrada horizontalmente

    gridPane.add(hbox, cont % numColumns, cont / numColumns);

    label.setOnMouseClicked(new EventHandler<MouseEvent>() {
        @Override
```

Imagen 3: Método actualizarEtiquetas() de “ControladorSelectorCuenta”.

comprobarPagos(): Este método se utiliza al entrar en *“ControladorPrincipal”*. Y se encarga de comprobar y actualizar los pagos mensuales pendientes en el usuario que se ha conectado.

```
private void comprobarPagos() {
    ArrayList<Cuenta> listaCuentas = usuario.getListaCuentas();

    if (listaCuentas != null && !listaCuentas.isEmpty()) {
        for (Cuenta c : listaCuentas) { // Itineramos sobre las cuentas del usuario
            ArrayList<GastosPeriodicos> listaGastos = c.getListaGastos();

            if (listaGastos != null && !listaGastos.isEmpty()) {
                for (GastosPeriodicos gasto : listaGastos) { //Itineramos sobre los gastos de cada cuenta
                    Date fechaActual = new Date();

                    // Comprobamos que hay pagos pendientes y no salimos hasta solucionarlos:
                    while (!fechaActual.before(gasto.getFechaSiguientePago()) && gasto.getFechaSiguientePago() != null && gasto.getCantidadRestante() > 0) {
                        Double DineroPorPlazo = gasto.getPlazosDinero();            // Dinero que hay que pagar en cada plazo
                        Double DineroRestante = gasto.getCantidadRestante();        // Cuando falta por saldar la deuda
                        Double YaPagado = gasto.getCantidadPagada();                // Cuanto a pagado ya
                        Double DineroARestar = 0.0;

                        if ((DineroRestante - DineroPorPlazo) < 0) { // Si da negativo es que este es el ultimo pago
                            DineroARestar = DineroRestante;
                            gasto.setCantidadRestante(cantidadRestante:0.0);
                        } else {
                            DineroARestar = DineroPorPlazo;
                            double resto = DineroRestante - DineroPorPlazo;

                            System.out.println(resto);
                            gasto.setCantidadRestante(resto);

                            Calendar calendar = Calendar.getInstance();
                            calendar.setTime(gasto.getFechaSiguientePago()); // Se suma un mes a la fecha del siguiente pago
                            calendar.add(Calendar.MONTH, amount:1);
                            Date nuevaFechaSiguientePago = calendar.getTime();

                            gasto.setFechaSiguientePago(nuevaFechaSiguientePago);
                        }
                        System.out.println(DineroARestar + " " + YaPagado);
                        gasto.setCantidadPagada(YaPagado + DineroARestar);
```

Imagen 4: Método comprobarPagos() de “ControladorPrincipal”.

RastrearCuenta(): Un método que en el resumen de la cuenta actual, se encarga de iterar sobre todas las cuentas del usuario actual y determinar cuál es la siguiente cuenta a mostrar, y cual la anterior.

```
private Cuenta RastrearCuenta(String buscada) { // Iteramos sobre la lista de cuentas, para determinar cual es la siguiente o anterior cuenta.
    List<Cuenta> listaCuentas = usuario.getListaCuentas();
    if (listaCuentas.size() <= 1) {
        // Si hay una sola cuenta, simplemente devuelve esa cuenta
        return cuenta;
    }

    int idCuentaActual = cuenta.getId();

    int indiceActual = -1;
    for (int i = 0; i < usuario.getListaCuentas().size(); i++) {
        Cuenta c = usuario.getListaCuentas().get(i);
        if (c.getId() == idCuentaActual) {
            indiceActual = i;
            break;
        }
    }

    int indiceAnterior = (indiceActual - 1 + usuario.getListaCuentas().size()) % usuario.getListaCuentas().size();
    int indiceSiguiente = (indiceActual + 1) % usuario.getListaCuentas().size();

    Cuenta cuentaAnterior = usuario.getListaCuentas().get(indiceAnterior);
    Cuenta cuentaSiguiente = usuario.getListaCuentas().get(indiceSiguiente);

    if (buscada.equals(anObject:"Siguiente")) {
        return cuentaSiguiente;
    } else {
        return cuentaAnterior;
    }
}
```

Imagen 5: Método RastrearCuenta() de “ControladorResumenCuenta”.

CalcularPagoMensual(): Método dedicado a calcular automáticamente el importe a pagar en un nuevo gasto mensual. Se basa en varios campos, y solo lo calcula si estos están con valores válidos. De no estarlo, se asigna un valor por defecto “0.0”.

```
@FXML
public void CalcularPagoMensual() {
    if (txtCantidadTotal.getText().isEmpty() || txtInteresAnual.getText().isEmpty() ||
        txtPlazos.getText().isEmpty()) {

            txtPlazosDinero.setText("0.0");

        return;
    }

    Double total = Double.parseDouble(txtCantidadTotal.getText());
    Double interesAnual = Double.parseDouble(txtInteresAnual.getText());
    int plazos = Integer.parseInt(txtPlazos.getText());
    String mesesOAños = cmbPlazos.getValue();

    // Convertir la tasa de interés anual a mensual
    Double interesMensual = interesAnual / 12 / 100;

    if (mesesOAños.equals(anObject:"Años")) {
        plazos *= 12; // Convertir años a meses
    }

    Double pagoMensual = total * (interesMensual * Math.pow(1 + interesMensual, plazos)) / (Math.pow(1 + interesMensual, plazos) - 1);
    String pagoMensualFormateado = String.format(format:"%.2f", pagoMensual);
    pagoMensualFormateado = pagoMensualFormateado.replace(oldChar:',', newChar:'.'); // Reemplazar coma con punto
    txtPlazosDinero.setText(pagoMensualFormateado);
}
```

Imagen 6: Método CalcularPagoMensual() de “ControladorPagosRecurrentes”.

4.3.4 Conexión con la base de datos MongoDB

Para realizar la conexión entre JavaFX y MongoDB, hemos decidido crear un controlador para que se encargue en exclusiva de realizar todas las conexiones con la base de datos. Este controlador es llamado desde otros controladores, siempre que se quiera hacer una conexión para sacar, o guardar algún dato.

Desde este controlador, podemos encontrar distintos metodos, como por ejemplo:

ControladorMongoDB(): Es el constructor de la clase, y es el encargado de realizar la conexión con la base de datos.

```
public ControladorMongoDB() {
    // Configurar conexión a MongoDB
    ConnectionString connectionString = new ConnectionString(connectionString:"mongodb://localhost:27017");
    CodecRegistry pojoCodecRegistry = fromProviders(PojoCodecProvider.builder().automatic(automatic:true).build());
    CodecRegistry codecRegistry = fromRegistries(MongoClientSettings.getDefaultCodecRegistry(), pojoCodecRegistry);
    MongoClientSettings clientSettings = MongoClientSettings.builder()
            .applyConnectionString(connectionString)
            .codecRegistry(codecRegistry)
            .build();
    MongoClient mongoClient = MongoClients.create(clientSettings);
    MongoDatabase db = mongoClient.getDatabase(databaseName:"UniversalDatas");
    collection = db.getCollection(collectionName:"Usuarios", documentClass:Usuario.class);
}
```

Imagen 7: Constructor de la clase “ControladorMongoDB”.

ActualizarUsuario(): Actualiza un usuario dado por parámetro en la base de datos.

```
public void ActualizarUsuario(Usuario usuario) {
    collection.replaceOne(eq(fieldName:"username", usuario.getUsername()), usuario);
}
```

Imagen 8: Método para actualizar un usuario existente en la base de datos.

ComprobarUsuario(): verifica si en la base de datos existe un usuario con el username dado por parámetro.

```
public boolean ComprobarUsuario(String username) {
    Usuario usuario = buscarUsuarioPorNombre(username);
    return usuario != null;
}
```

Imagen 9: Método que verifica la existencia de un usuario concreto en la base de datos.

4.4 Usabilidad

En las primeras reuniones en las que decidimos las partes más de funcionalidad y datos también planteamos distintas interfaces. Queríamos que fueran claras y donde los datos tuvieran el mayor peso para que los usuarios estuvieran contentos. Estos fueron los primeros bocetos donde se ve cómo queríamos ordenar los espacios de la aplicación.

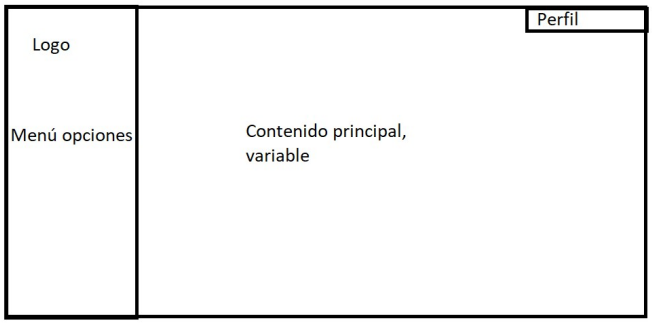

Imagen 10: Boceto de la interfaz principal.

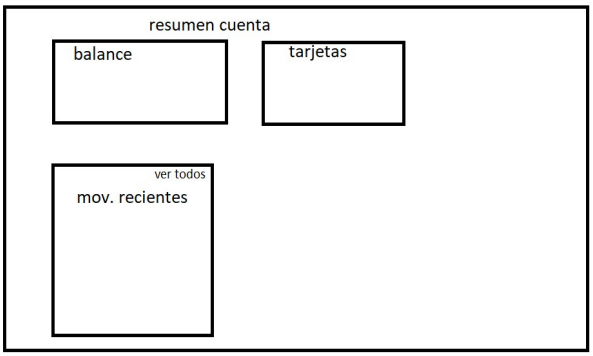

Imagen 11: Boceto de la interfaz de resumen de cuenta.

Con todo ello y después de ver distintos ejemplos en Pinterest decidimos el estilo y colores de la aplicación.

Imagen 12: Ventana de crear usuario.

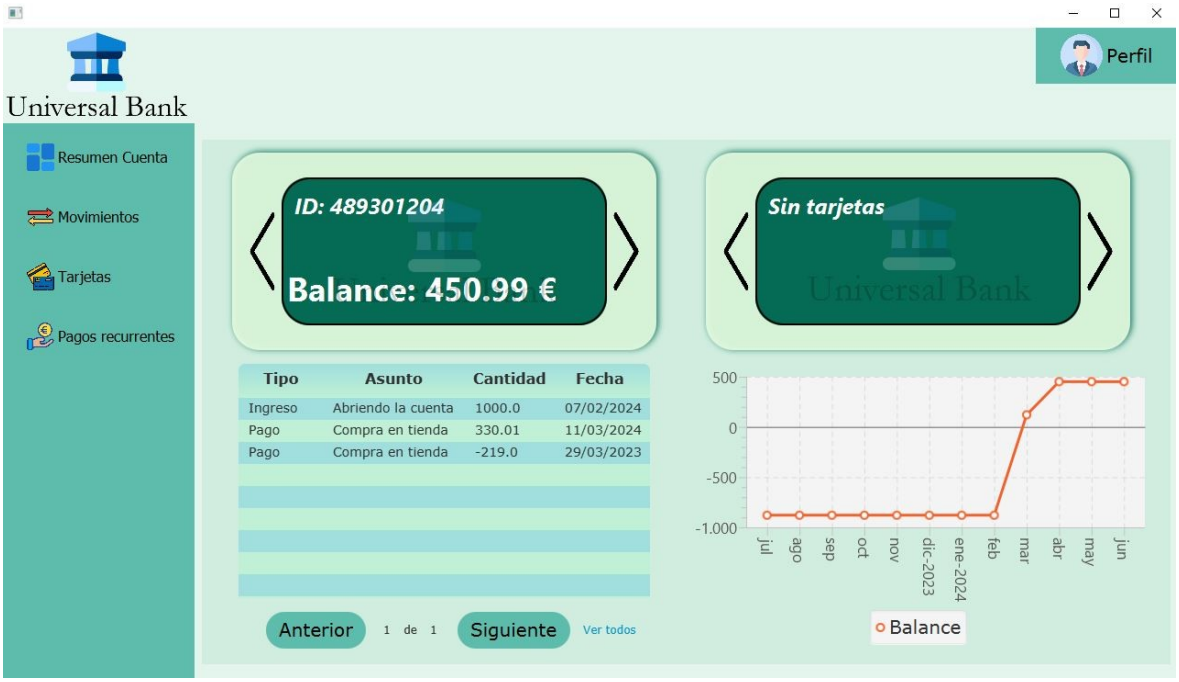

Imagen 13: Ventana de resumen de cuenta.

Nos quisimos desmarcar de los colores típicos de los bancos convencionales para que el usuario nos pueda diferenciar de ellos (como rojo, azul, naranja). Escogimos igualmente una combinación del azul y verde porque transmite tranquilidad.

Hay una tendencia que vimos en los otros proyectos y es el de empaquetar distintos items o información en cajas y darles profundidad. Permite crear esa ilusión de profundidad gracias a los distintos tonos de azul y verde escogidos.

Escogimos una iconografía que tuviera color para que diera una sensación más viva y menos seria.

4.4.1 Navegación

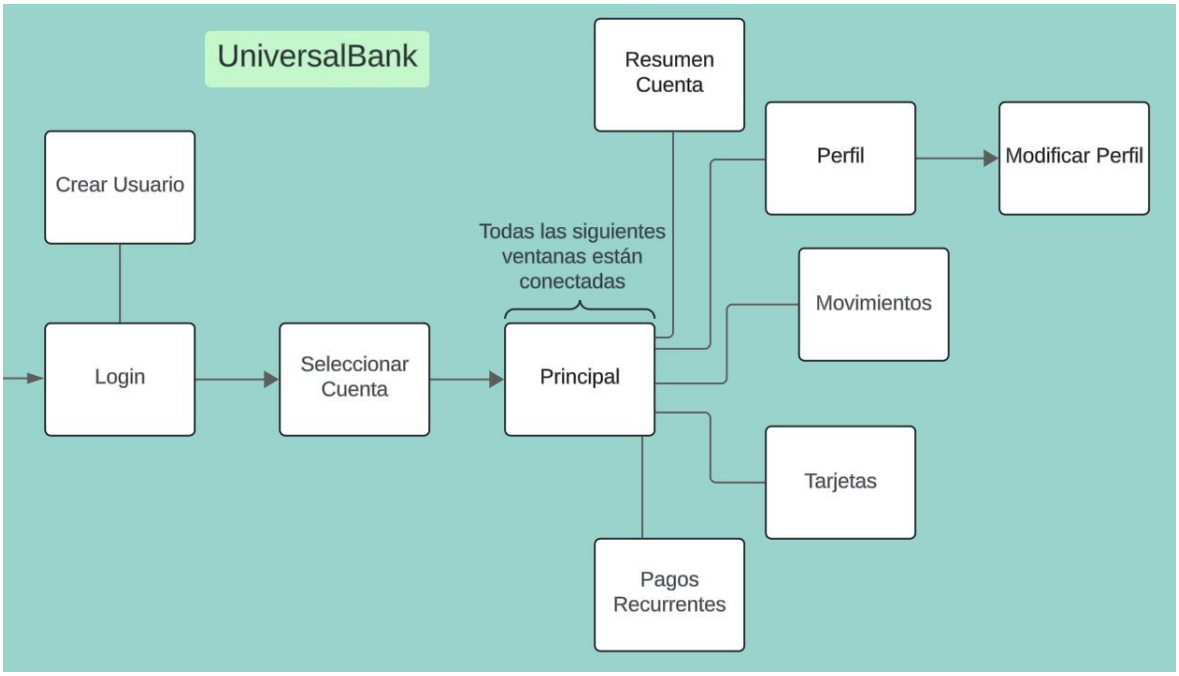

Imagen 13: Mapa de navegación de UniversalBank.

El mapa de navegación de UniversalBank es principalmente lineal hasta llegar a la ventana ‘Principal’ donde luego tiene accesible todas esas opciones ya que lo que cambia es el contenido del centro.

En ‘Principal’ hay una serie de botones en forma de menú que permiten ir a las demás ventanas. Una vez pasado ‘Seleccionar Cuenta’ no puede volver para atrás ya que la aplicación es de uso personal.

4.4.2 Casos de uso

Hemos agrupado funcionalidad que se hace en una misma ventana para simplificar un poco los casos de uso. Por ejemplo: en movimientos el usuario puede crear un nuevo movimiento o borrar o modificar uno existente.

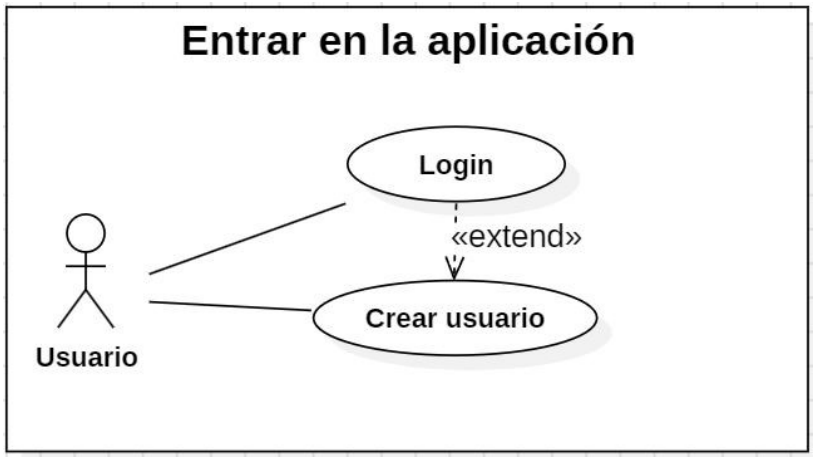

Imagen 14: Caso de uso de entrar a la aplicación.

ID:	1
Nombre:	Entrar en la aplicación

Actores:	Usuario
Disparador:	Iniciar la aplicación
Descripción:	El usuario introduce sus credenciales y entra en la aplicación
Flujo normal:	El usuario tiene que introducir sus credenciales para poder entrar en la aplicación
Flujos alternativos:	Si no tiene usuario todavía tiene que ‘Crear usuario’, rellenar el formulario y después puede entrar en la aplicación

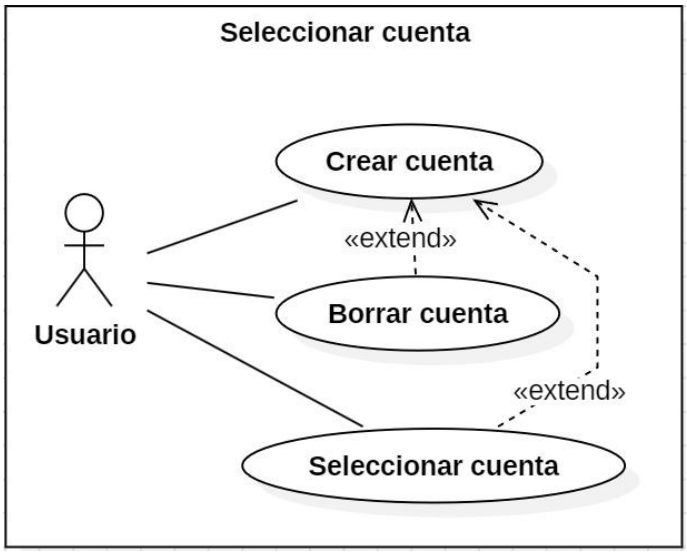

Imagen 15: Caso de uso de seleccionar cuenta.

ID:	2
Nombre:	Seleccionar cuenta

Actores:	Usuario
Disparador:	Haber entrado desde el login
Descripción:	El usuario ve sus cuentas y puede borrar, crear o seleccionar una cuenta
Flujo normal:	El usuario selecciona la cuenta con la que quiere entrar a la aplicación
Flujos alternativos:	Si no tiene cuenta tiene que 'Crear cuenta' y luego seleccionarla

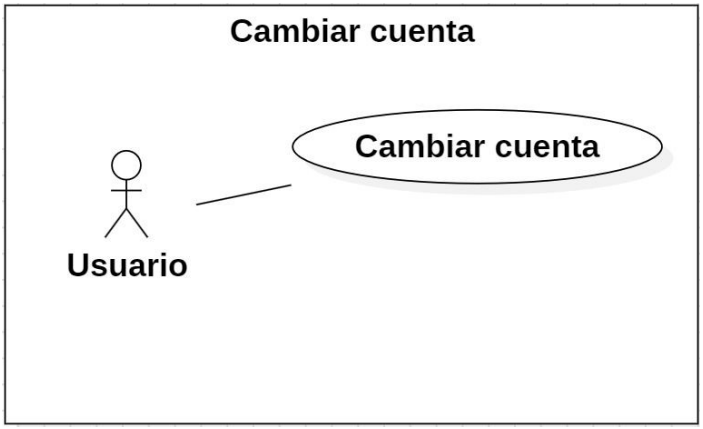

Imagen 16: Caso de uso de cambiar cuenta.

ID:	3
Nombre:	Cambiar cuenta

Actores:	Usuario
Disparador:	Clicar a las flechas del apartado de cuentas

Descripción:	El usuario clica en las flechas para cambiar de cuenta
Precondiciones:	Estar en la ventana ‘Resumen Cuenta’
Flujo normal:	El usuario da a una flecha de cuenta y si tiene otra cuenta cambia toda la información de la ventana ‘Resumen Cuenta’

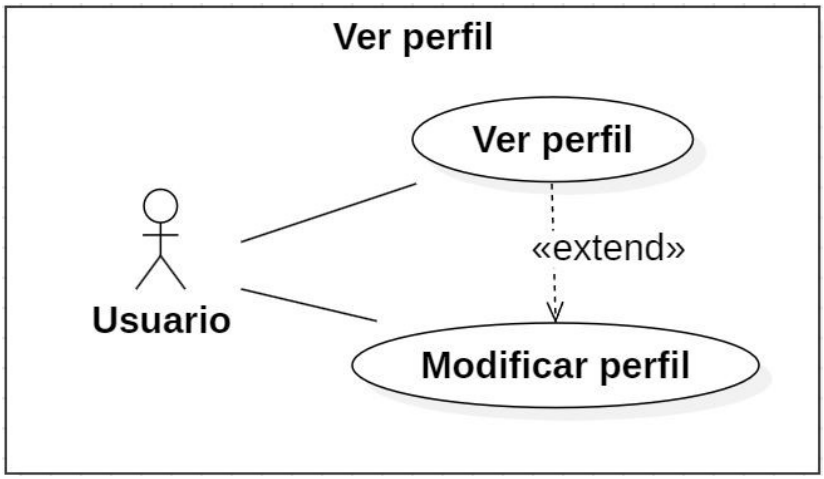

Imagen 17: Caso de uso de ver perfil.

ID:	4
Nombre:	Ver perfil

Actores:	Usuario
Disparador:	Clica en el perfil
Descripción:	El usuario puede ver su información personal y modificala
Flujo normal:	El usuario ve su información y puede modificarla rellenando un formulario

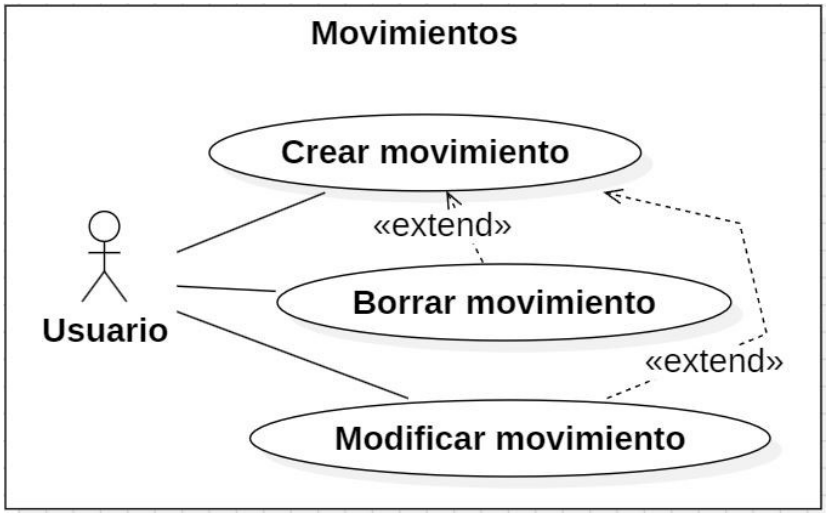

Imagen 18: Caso de uso de movimientos.

ID:	5

Nombre:	Movimiento

Actores:	Usuario
Disparador:	Clica en crear, borrar o modificar movimiento
Descripción:	El usuario puede crear un nuevo movimiento o borrar o modificar uno existe de la tabla
Precondiciones:	Estar en la ventana 'Movimientos'
Flujo normal:	El usuario rellena el formulario para crear un nuevo movimiento y clica el botón 'Nuevo movimiento'
Flujos alternativos:	Selecciona primero un movimiento de la tabla y luego da a 'Modificar' o 'Borrar' movimiento

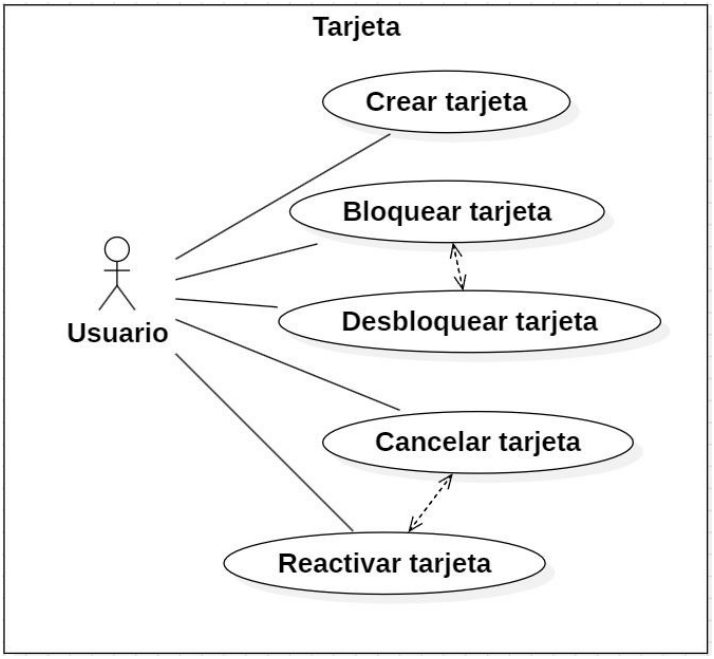

Imagen 19: Caso de uso de tarjeta.

ID:	6
Nombre:	Tarjeta

Actores:	Usuario
Disparador:	Clica en nueva tarjeta o cancelar o bloquear tarjeta
Descripción:	El usuario puede crear una nueva tarjeta para su cuenta o cancelar y reactivar o bloquear y desbloquear su tarjeta
Precondiciones:	Estar en la ventana 'Tarjetas'
Flujo normal:	El usuario rellena el formulario y da al botón 'Nueva Tarjeta'
Flujos alternativos:	El usuario cancela o bloquea una tarjeta y después puede reactivar o desbloquear dicha tarjeta

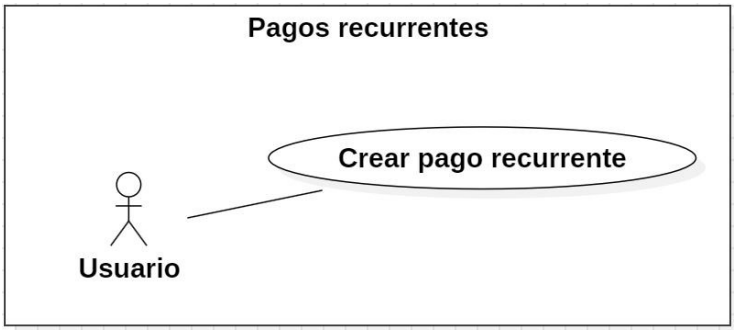

Imagen 20: Caso de uso de pagos recurrentes.

ID:	7
Nombre:	Pagos recurrentes

Actores:	Usuario
Disparador:	Clica en nuevo pago
Descripción:	El usuario rellena el formulario y clica en el botón 'Nuevo pago'
Precondiciones:	Estar en la ventana 'Pagos Recurrentes'
Flujo normal:	El usuario completa el formulario y da a 'Nuevo pago'
Flujos alternativos:	Si el usuario no sabe el dinero por plazo puede dar a 'Calcular' y la aplicación rellena ese campo por él

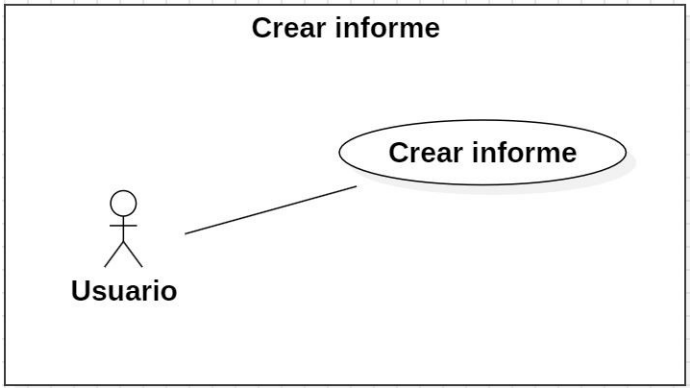

Imagen 21: Caso de uso de crear informe.

ID:	8
Nombre:	Crear informe

Actores:	Usuario
Disparador:	Clica en crear informe
Descripción:	El usuario puede pedir un informe con un gráfico de los últimos 12 meses de movimientos
Precondiciones:	Estar en la ventana 'Movimientos' La plantilla de JasperReport tiene que estar en el mismo directorio que el JAR
Postcondiciones:	Se genera un archivo PDF con el informe

Flujo normal:	El usuario clica en el botón ‘Crear Informe’ y se crea en el mismo directorio un PDF con una gráfica de los últimos 12 meses de movimientos de esa cuenta

4.5 Rendimiento

Durante el desarrollo, hemos estado realizando diversas pruebas para comprobar cómo reaccionaba la aplicación y para ayudarnos a detectar posibles errores.

Por ejemplo, una de las pruebas que llevamos a cabo fue la implementación de métodos en el código que generan datos aleatorios en la base de datos. Esto nos permitió realizar pruebas con datos reales ya añadidos, facilitando la identificación de errores y el análisis del comportamiento de la aplicación en situaciones más cercanas a un entorno de producción.

```java
private static ArrayList<Tarjeta> crearTarjetas() { // Codigo temporal para generar Tarjetas ----------------------------------------------
    ArrayList<Tarjeta> listaTarjetas = new ArrayList<>();

    Random random = new Random();
    int num = random.nextInt(bound:5);

    // Array de tipos de tarjetas disponibles
    String[] tiposTarjeta = {"Crédito", "Débito", "Prepago", "Viaje", "Recompensas", "Corporativa"};

    for (int i = 0; i < num; i++) {
        int id = i + random.nextInt(bound:1000000000);

        String tipo = tiposTarjeta[random.nextInt(tiposTarjeta.length)];

        String NumeroTarjeta = generateCreditCardNumber();

        String CVV = String.format(format:"%03d", random.nextInt(bound:1000));

        double limiteDiario = random.nextDouble() * 2000;
        limiteDiario = Math.round(limiteDiario * 100.0) / 100.0;

        String pin = generateRandomPin();

        Calendar cal = Calendar.getInstance();
        cal.add(Calendar.YEAR, random.nextInt(bound:5) + 1); // Fecha de caducidad aleatoria, entre 1 y 5 años desde la fecha actual
        Date fechaCaducidad = cal.getTime();

        Tarjeta tarjeta = new Tarjeta(id, tipo, NumeroTarjeta, CVV, limiteDiario, pin, fechaCaducidad);
        listaTarjetas.add(tarjeta);
    }

    return listaTarjetas;
}
```

Imagen 22: Método que genera tarjetas con todos sus datos de forma aleatoria.

En este método, estamos generando un listado de Tarjetas de crédito para agregarlo a todas las cuentas del usuario que acaba de iniciar sesión. El método utiliza la clase "Random" de Java para generar números aleatorios que seleccionan distintos valores y se asignan a los datos de la lista.

De manera similar a este método, hemos estado utilizando otros métodos para que, cada vez que se inicia la aplicación con un usuario nuevo, se creen datos ficticios. Esto nos permite realizar pruebas de visualización, modificación, y eliminación de datos de manera efectiva.

5. Conclusiones

5.1 Problemas ocurridos

Durante el desarrollo del proyecto, nos estuvimos enfrentando a diversos problemas técnicos que nos estuvieron ralentizando el progreso.

Uno de los principales problemas que nos ocurrió fue el poder integrar la base de datos en MongoDB con JavaFX y Visual Studio. A pesar de intentarlo durante semanas, no conseguíamos que JavaFX se conectará con MongoDB correctamente. Esto resultó bastante frustrante, pero después de documentarnos varias veces, logramos realizar dicha conexión de forma exitosa.

Otro problema que surgió casi al mismo tiempo que la conexión con MongoDB, fue que Visual Studio no estaba funcionando como cabría esperar, lo que nos impedía ejecutar nuestra aplicación de forma correcta. Estuvimos pensando en cambiarnos de plataforma a NetBeans, pero luego de un tiempo, pudimos solucionar el problema y poder empezar a trabajar en nuestro proyecto con más tranquilidad.

También hemos tenido otros problemas menores, como que por ejemplo una de las ramas secundarias de GitHub dejó de funcionar y nos obligó a crear otra de cero. Pero en general, los mayores problemas vinieron las primeras semanas del proyecto.

5.2 Futuras actualizaciones

5.2.1 Programación Multihilo

Una de las mejoras significativas que implementaremos en nuestro proyecto es la programación multihilo para realizar comprobaciones iniciales de actualización de datos de cada cuenta.

Actualmente, el sistema está utilizando un enfoque secuencial para verificar y actualizar los datos, lo cual puede ser ineficiente y lento, especialmente cuando se maneja una gran cantidad de datos. Al utilizar multihilo, las tareas de verificación y

actualización pueden ejecutarse de manera concurrente, aprovechando al máximo los recursos del procesador y reduciendo significativamente el tiempo de espera para el usuario. Esto no solo mejora la eficiencia del sistema, sino que también proporciona una experiencia de usuario más fluida y receptiva.

5.2.2 Compatibilidad con Android y Diseño Responsive

Otra área clave de mejora es hacer que la aplicación sea compatible con dispositivos Android y que tenga un diseño responsive. En la actualidad, el uso de dispositivos móviles es predominante, y tener una aplicación que funcione bien en teléfonos y tabletas es esencial para alcanzar a una audiencia más amplia.

El diseño responsive garantiza que la aplicación se vea y funcione correctamente en una variedad de dispositivos y tamaños de pantalla, proporcionando una experiencia consistente y agradable a todos los usuarios, independientemente del dispositivo que utilicen.

5.2.3 Mejora de la Seguridad de la Aplicación

La seguridad es un aspecto crítico, especialmente cuando se trata de una aplicación como la nuestra, en la que se están tratando datos sensibles constantemente.

Mejorar la seguridad de nuestra aplicación implica implementar prácticas y tecnologías que protejan los datos del usuario contra accesos no autorizados y amenazas externas. Para nosotros, la seguridad de nuestros usuarios es nuestra máxima prioridad, por eso tenemos planeado seguir mejorando continuamente la seguridad en nuestra aplicación.

5.2.4 Centralización de la Base de Datos en un Servidor Externo

Centralizar la base de datos en un servidor externo puede ofrecer varios beneficios importantes. Actualmente, como la base de datos está alojada localmente, puede haber ciertas limitaciones.

Al mover la base de datos a un servidor externo, podríamos ofrecer a nuestros clientes medidas mucho más seguras y flexibles. Esto no solo mejorará la fiabilidad del sistema, sino que también facilitará el mantenimiento de la base de datos, permitiendo que el equipo de desarrollo se enfoque más en mejorar otras funcionalidades de la aplicación.

6. Bibliografía

MongoDB

https://www.mongodb.com/docs/

Git

https://www.youtube.com/watch?v=i_23KUAEtUM

Visual

https://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html

https://edencoding.com/style-tableview-javafx/

https://www.tutorialspoint.com/javafx/line_chart.htm

Documentación

https://creately.com/blog/es/diagramas/relaciones-de-diagrama-de-clases-uml-explicadas-con-ejemplos/#Directed

Inspiración

https://dribbble.com/shots/19067141-Form-registration-for-a-banking-investment-application

https://dribbble.com/shots/15022670/attachments/6746101?mode=media

https://dribbble.com/shots/21043784-Bank-Dashboard-Digital-Banking

Dudas puntuales

https://stackoverflow.com

https://chatgpt.com