Memoría Escrita Del Proyecto Mis Finanzas

Programación Orientada A Objetos

Juan Pablo Mejia Gomez , Jorge Humberto Gaviria Botero ,
Leonard David Vivas Dallos , Jose Daniel Moreno Ceballos y
Tomas Escobar Rivera .

Docente: Ph.D. Jaime Alberto Guzman Luna.

Universidad Nacional De Colombia

Sede Medellín

Facultad De Minas

Medellín, 2023

Tabla de Contenido

1. Descripción general de la solución (análisis, diseño, e implementación):

Mis Finanzas es una plataforma de gestión financiera digital que brinda a los usuarios la capacidad de administrar y controlar sus recursos monetarios de manera eficiente. El propósito fundamental de Mis Finanzas es mejorar la relación que las personas tienen con su dinero, proporcionando diversas funcionalidades diseñadas para ofrecer a los usuarios una amplia gama de opciones sobre cómo utilizar sus fondos y obtener el máximo beneficio de ellos.

Esta plataforma permite a los usuarios realizar un seguimiento detallado de sus ingresos, gastos y ahorros, brindando una visión integral de su situación financiera. Además, ofrece herramientas para establecer y monitorear metas financieras, como ahorros para un objetivo específico o la realización de préstamos.

1.1. Definición de requerimientos:

El programa “Mis Finanzas” define los siguientes requerimientos:

1.1.1. Creación de Usuarios, Cuentas, Bancos, Estados, Movimientos y Metas.
1.1.2. Ofrecer una tipo de acceso administrativo con privilegios para modificar sin restricciones parámetros dentro del programa.
1.1.3. Implementar una capa de persistencia.
1.1.4. Manipulación del saldo de las Cuentas por parte de un Usuario, esto es, invertir, transferir y/o consignar el saldo.
1.1.5. Utilizar un sistema de clasificación de Usuarios llamado “Suscripciones”. La finalidad de este sistema de clasificación debe ser establecer la cantidad de comisión que un Banco le cobra a un Usuario, la probabilidad de que una inversión de saldo sea exitosa, la cantidad de saldo ganado en la inversión y la cantidad de Cuentas diferentes que se pueden asociar a un Usuario.
1.1.6. Asesorar al Usuario en sus inversiones. Es decir, recomendar al Usuario en qué sectores de la economía es conveniente poner su dinero.
1.1.7. Ofrece una amplia variedad de opciones para realizar préstamos incluso con entidades no bancarias.
1.1.8. Realizar el proceso de Compra de Cartera del usuario. Esto es, trasladar una deuda de una cuenta Corriente a otra que pueda ofrecer mejor interés y cambie el plazo en la que debe ser pagada.
1.1.9. Efectuar préstamos en las diferentes Cuentas de Ahorros permite a los individuos y empresas alcanzar sus metas y cubrir gastos de manera más rápida y efectiva. A través de esta modalidad de préstamos, las instituciones financieras ofrecen a los titulares de Cuentas de Ahorro la posibilidad de obtener financiamiento sin cuotas de intereses ni límite de pago.
1.1.10. Brindar la posibilidad al Usuario de poder hacer una previsualización completa de una Deuda a recibir, de manera que el Usuario pueda tener una visión completa de la información de una Deuda antes de recibirla.
1.1.11. Facilitar al Usuario intercambios monetarios entre Cuentas de distinta Divisa por medio de transferencias que utilizan tasas de cambio específicas.

1.2. Diseño de la solución:

1.2.1. Paquetes:
1.2.1.1. baseDatos.
1.2.1.1.1. Clases públicas:
1.2.1.1.1.1. Serializador.
1.2.1.1.1.2. Deserializador.
1.2.1.2. gestorAplicación.
1.2.1.2.1. Interno.
1.2.1.2.1.1. Clases públicas:
1.2.1.2.1.1.1. Ahorros.
1.2.1.2.1.1.2. Corriente.

1.2.1.2.1.1.3. Cuenta.
1.2.1.2.1.1.4. Deuda.
1.2.1.2.1.1.5. Metas.
1.2.1.2.1.1.6. Movimientos.
1.2.1.2.1.1.7. Usuario.
1.2.1.2.1.2. Enums:
1.2.1.2.1.2.1. Suscripcion.
1.2.1.2.1.2.2. Categoria.
1.2.1.2.2. Externo.
1.2.1.2.2.1. Clases públicas:
1.2.1.2.2.1.1. Banco.
1.2.1.2.2.1.2. Estado.
1.2.1.2.2.2. Interfaces:
1.2.1.2.2.2.1. Tablas.
1.2.1.2.2.3. Enums:
1.2.1.2.2.3.1. Cuotas.
1.2.1.2.2.3.2. Divisas.
1.2.1.3. uiMain.
1.2.1.3.1. Clases públicas:
1.2.1.3.1.1. Main.

1.3. Implementación de la solución:

1.3.1. Serialización:
1.3.1.1. Serializar listas con objetos.
1.3.2. Deserializador:
1.3.2.1. Deserializar listas con objetos.
1.3.3. Ahorros:
1.3.3.1. Cuentas ahorros totales.
1.3.3.2. Saldo.
1.3.3.3. Vaciar cuenta.
1.3.4. Corriente:
1.3.4.1. Cupo.
1.3.4.2. Disponible.
1.3.4.3. Plazo de pago.
1.3.4.4. Intereses.
1.3.4.5. Primer Mensualidad.
1.3.4.6. Cuentas corriente totales.
1.3.4.7. Vaciar cuenta.
1.3.5. Cuenta:
1.3.5.1. Titular.
1.3.5.2. Clave.
1.3.5.3. Divisa.
1.3.5.4. Nombre.
1.3.5.5. Id.
1.3.5.6. Banco.
1.3.5.7. Cuentas totales.
1.3.5.8. Gota gota.
1.3.5.9. Vaciar cuenta.
1.3.6. Deuda:
1.3.6.1. Id.
1.3.6.2. Cuenta.
1.3.6.3. Banco.
1.3.6.4. Deudas totales.
1.3.6.5. Conseguir deudas.
1.3.6.6. Conseguir propiedades cuenta
1.3.7. Metas:
1.3.7.1. Nombre.
1.3.7.2. Cantidad.
1.3.7.3. Fecha.
1.3.7.4. Id.
1.3.7.5. DATE_FORMAT.

1.3.7.6. Metas Totales.
1.3.7.7. Dueño.
1.3.7.8. Revisión Metas.
1.3.7.9. Cambio de fecha.
1.3.7.10. Determinar Plazo.
1.3.7.11. Prioridad Metas.

1.3.8. Movimientos:

1.3.8.1. Id.
1.3.8.2. Cantidad.
1.3.8.3. Categoría.
1.3.8.4. Fecha.
1.3.8.5. Destino.
1.3.8.6. Origen.
1.3.8.7. Owner.
1.3.8.8. Nombre categoría.
1.3.8.9. Cantidad categoría.
1.3.8.10. Recomendar fecha.
1.3.8.11. Analizar categoría.
1.3.8.12. Impuestos movimiento.

1.3.9. Usuario:

1.3.9.1. Lista de cuentas asociadas.
1.3.9.2. Lista de movimientos asociados.
1.3.9.3. Lista de metas asociadas.
1.3.9.4. Lista de cuentas corrientes asociadas.
1.3.9.5. Lista de cuentas de ahorro asociadas.
1.3.9.6. Lista de bancos asociados.
1.3.9.7. Lista de usuarios totales.
1.3.9.8. Nombre.
1.3.9.9. Correo.
1.3.9.10. Contraseña.
1.3.9.11. Id.
1.3.9.12. Suscripción.
1.3.9.13. Contador de movimientos.
1.3.9.14. Limite de cuentas que se pueden asociar
1.3.9.15. Asociar bancos.
1.3.9.16. Asociar cuentas.
1.3.9.17. Asociar metas.
1.3.9.18. Asociar movimientos.
1.3.9.19. Verificar credenciales.
1.3.9.20. Verificar el contador de movimientos.
1.3.9.21. Hallar usuario gota gota.
1.3.9.22. Hallar usuario impuestos portafolio.

Ver figura 1 de la tabla

2. *Descripción del diseño estático del sistema en la especificación UML (Diagrama de clases y objetos del sistema):*

Diagrama UML (Debe crear una cuenta en LucidChart para revisar el diagrama desde ésta página) ó ***Ver figura 2 de la tabla***

3. *Descripción de la implementación de características de programación orientada a objetos en el proyecto (indicando los lugares y el modo en que se implementaron):*

3.1. Clases Abstracta (1) y Métodos Abstractos (1):

El programa maneja una clase abstracta, esta clase se llama *Cuenta*. Esta misma hace las veces de Padre de dos subclases, llamadas *Ahorros* y *Corriente*. El sentido del uso de esta clase como abstracta se debe a la intención de simulación de la tipología de cuentas en la vida real. Normalmente se poseen dos tipos de cuenta en cualquier entidad bancaria, cuentas de *Ahorros* y cuentas *Corriente*. Estos dos tipos

de *Cuenta* poseen atributos comunes, como los son *titular*, *clave*, *divisa*, *nombre*, *id* y *banco*, empero, cuando de hablar de dinero se trata, estos dos trabajan de manera muy distinta. Mientras que la cuenta de *Ahorros* posee un atributo de *saldo*, en el cual se almacena la cantidad de dinero propio y a disposición del *usuario*, la cuenta *Corriente* posee dos atributos, *cupo* y *disponible*. El atributo *cupo* hace referencia al tope de dinero que se puede gastar con este tipo de *Cuenta*, en el programa hará referencia a la máxima cantidad de deuda a usar y será un valor de referencia, ahora bien, el atributo *disponible* almacenará la cantidad de dinero que le resta a cada *Usuario* por usar, haciendo las veces de lo que llamamos cupo disponible en una tarjeta de crédito por ejemplo, es decir, la cantidad de deuda que puede adquirir un *Usuario* con una compra.
Por otra parte, el método abstracto (línea 152 clase Cuenta) *vaciarCuenta* que se encuentra en la clase *Cuenta* y en sus subclases *Ahorros* y *Corriente* no retorna ningún valor y recibe como parámetro una cuenta de tipo *Ahorros*. Este es abstracto debido a que hay que hacer la diferencia entre las cuentas de tipo *Ahorros* y las de tipo *Corriente*, ya que se tiene un *saldo* y un *cupo* para cada clase respectivamente. Se crea un movimiento entre la cuenta que llama al método (origen) y la cuenta que recibe como parámetro (destino) por un valor que depende directamente del *saldo*, o bien, disponibilidad del *cupo* (dependiendo del tipo de cuenta), para finalmente asociar este movimiento al usuario.

Ver figuras 3 y 4 de la tabla

3.2. Interfaces (1) diferentes a los utilizados para serializar los objetos en el punto de la persistencia:

El programa hace uso de una interface llamada *Tablas*. El objetivo fundamental de la interface es emular la impresión de los objetos en un formato de tablas y es creada debido a la necesidad de mostrar listas de objetos con características apropiadas y de la misma manera brindar la posibilidad de una elección rápida y agradable a ojos del usuario. *Tablas* se encuentra dentro del paquete *externo*, que a su vez se encuentra dentro del paquete *gestorAplicacion*.

La interface desarrolla un método que devuelve una lista con los nombres de los atributos de la clase que la implementa, para ser usados como encabezados en las tablas. (Previo a ser usadas como encabezados se realiza una limpieza con un método estático de los atributos innecesarios de mostrar en la tabla). Además de esto, la interface posee varios métodos estáticos que reciben la listas de objetos a imprimir y que son los encargados de realizar la impresión de las tablas con una construcción manual, en la que se hace uso del *printf* que nos permite emular la creación del espacio de celdas necesario.

Ver figura 5 de la tabla

3.3. Herencia (1):

Se utiliza herencia en el programa para emular la especialización de un objeto. Tal es el caso de la clase *Metas* y la clase *Deuda*; *Metas* es padre de *Deuda*, ambas clases se encuentran dentro del paquete *interno*. La lógica detrás de esto recae en el hecho de que en el mundo real una deuda es un compromiso de pago, es decir, un contrato; asimismo, una meta también es un contrato para quien se lo propone. *Metas* tiene atributos, *nombre, id, cantidad* y *dueño*, que son básicos para una meta monetaria; *Deuda* los hereda y a su vez define una instancia de *Cuenta* y una instancia de *Banco* que van a estar asociados, porque se propone realizar pagos de deudas al *banco* asociado a través de la *cuenta* asociada. Además, las clases *Ahorros* y *Corriente* heredan de la clase *Cuenta*, estás clases están en el paquete *interno*, su explicación se encuentra en el apartado de *Clases Abstracta (1) y Métodos Abstractos (1)*.

Ver figuras 6 y 7 de la tabla

3.4. Ligadura dinámica (2) asociadas al modelo lógico de la aplicación:

3.4.1. La clase *Cuenta* es una clase abstracta que contiene un método llamado *invertirSaldo*. Este método permite obtener la probabilidad de inversión del *Usuario*, basado en su nivel de *suscripción*, y mediante un valor aleatorio, determina si el usuario puede realizar la inversión. Si el resultado es verdadero, se le muestra al usuario un mensaje indicando que ha sido estafado.
Por otro lado, la clase Ahorros hereda de la clase Cuenta y sobreescribe el mismo método brindando un funcionamiento diferente: en este caso, el *Usuario* no resulta estafado al aplicar el método. Cuando se utiliza la ligadura dinámica, el método que se ejecuta es el definido en la clase Ahorros debido a que para utilizar este método la instancia de *Cuenta* empleada debe pertenecer a la clase *Ahorros*.

En la línea 1725 de la clase *Main*, se aplica la ligadura dinámica. En este punto, se tiene una instancia llamada *c* declarada como tipo *Ahorros*. Gracias a la ligadura dinámica, al llamar al método *invertirSaldo* usando la instancia *c*, se ejecutará el método definido en la clase *Ahorros*, ya que se considera el tipo real del objeto al que hace referencia, que es una instancia de la clase *Ahorros*.

3.4.2. En el código, se encuentra una clase llamada *Corriente* que contiene el método *retornoMensual*. Este método se encarga de calcular los intereses mensuales a aplicar a una instancia de *Corriente*. Para realizar este cálculo, utiliza atributos específicos de la clase *Corriente*, como *interes*, *disponibilidad*, *plazopago* y algunos otros métodos propios de la clase.

Por otro lado, la clase *Cuenta* es la clase padre de *Corriente* y también tiene una definición del método *retornoMensual*. Sin embargo, dado que la clase *Cuenta* no tiene los atributos específicos de la cuenta *Corriente*, realiza el cálculo utilizando la comisión del *Banco* y los impuestos del *Estado* asociados a la *Cuenta*, es decir, el método realiza una operación diferente en *Cuenta* y en *Corriente*.

En la línea 1104 de la clase *Main*, se aplica la ligadura dinámica. En este punto, se tiene una instancia llamada *vistaPrevia* declarada como tipo *Cuenta*. Sin embargo, en tiempo de ejecución, se le asigna un objeto de tipo *Corriente*. Gracias a la ligadura dinámica, al llamar al método *retornoMensual* en la variable *vistaPrevia*, se ejecutará el método definido en la clase *Corriente*, ya que se considera el tipo real del objeto al que hace referencia, que es una instancia de la clase *Corriente*.

Ver figura 8.1 y figura 8.2 de la tabla

3.5. Atributos de clase (1) y métodos de clase (1):

En la clase *Usuario* se encuentra el método estático *verificarCredenciales* y el atributo de clase *usuariosTotales*. El método estático *verificarCredenciales* recibe como parámetros dos atributos de tipo *String*, *nombre* y *contraseña* respectivamente. Posteriormente, se recorre el atributo de clase *usuariosTotales* comprobando que exista una instancia de *Usuario* cuyos atributos *nombre* y *contraseña* sean iguales a los valores pasados por parámetro, en el caso afirmativo se retorna esta misma instancia, de lo contrario, se retorna null. Asimismo, todas las clases del programa contienen atributos de clase cuya función es la de almacenar todas las instancias que se creen de las mismas.

Ver figura 9 y 10 de la tabla

3.6. Uso de constante (1 caso):

Las constantes se encuentran presenten en todas las clases de nuestro programa, específicamente en los atributos *static final* llamados *"nombreD"* que son diferentes para cada clase y que son utilizados posteriormente en el proceso de serialización y deserialización. El atributo *static final DATE_FORMAT*, que se encuentra dentro de la clase *Metas* en la línea 20, se utiliza para evitar crear varios objetos *"SimpleDateFormat"* en diferentes partes del código. La creación de múltiples objetos "*SimpleDateFormat*" puede ser costosa en términos de memoria y rendimiento, por lo que es considerado como buena práctica definir una constante y reutilizarla en el código.

Ver figura 11 y 12 de la tabla

3.7. Encapsulamiento (private, protected y public):

En la clase Metas podemos evidenciar los diferentes tipos de encapsulamiento: Private para el atributo *nombre* que se encuentra en la línea 16, debido a que este se debe tener lo más encapsulado posible por motivos de seguridad. Protected para *id* que se encuentra en la línea 19, ya que se usa en su clase hija *Deuda*, por lo que él protected permite que sea accesada mientras se encapsula lo máximo posible. Finalmente, para Public tenemos varios ejemplos, como lo son métodos y constructores, pero también cabe recalcar el atributo *nombreD*, que se encuentra en la línea 15, que es fundamental para el funcionamiento del serializador, y precisamente es público ya que se debe acceder desde una clase de otro paquete. Asimismo, en la clase *Cuenta* podemos ver una variedad de tipos de encapsulamiento: el atributo *titular*, que se encuentra en la línea 18, es privado debido a que es una información vulnerable y por ende debe de estar bien encapsulada. En adición, hay atributos como *divisa, nombre* e *id*, que se encuentran en la líneas 20, 21 y 22 respectivamente, que son protegidos debido a que se usan en sus clases hijas *Ahorros* y *Corriente*.

Ver figuras 13 y 14 de la tabla

3.8. Sobrecarga de métodos (1 caso mínimo) y constructores (2 casos mínimo):

En la clase *Cuenta* hay tres sobrecargas de constructores, en la clase *Usuario* hay tres sobrecargas de constructores, en la clase *Movimientos* hay diez sobrecargas de constructores y en la clase Metas hay cinco sobrecargas de constructores. El método estático *crearCuenta*, que se encuentra en la línea 53, es sobrecargado una vez en la clase *Cuenta* en la línea 55. Además, el método estático *compraCartera*, que se encuentra en la línea 1216, es sobrecargado una vez en la clase *Main* en la línea 992.

Ver figura 15, 16 y 17 de la tabla

3.9. Manejo de referencias this para desambiguar y this() entre otras. 2 casos mínimo para cada caso:

En los métodos *asociarMeta*, *asociarCuentaAhorros*, *asociarCuentaCorriente*. *asociarBanco*, *asociarMovimiento* se utiliza el operador *this* para referirse, en el paso de argumentos, a la instancia de la clase *Usuario* que llama a ejecución a estos mismos. De igual manera, el método de instancia *invertirSaldo* de la clase *Ahorros,* que se encuentra en la línea 61, utiliza el operador *this* para acceder a diferentes atributos de instancia de la instancia que lo ejecuta. Por otra parte, el método *this()* se utiliza en las clases *Estado*, *Banco* y *Usuario* cuando se ejecutan los constructores vacíos, el *this()* llama a los constructores con parámetros y les pasa valores estándar

con el fin de crear las “instancias por defecto” respectiva a cada clase, esto toma lugar en el método estático *accesoAdministrativo* de la clase *Main*, que se encuentra en la línea 2105.

Ver figura 18, 19 y 20 de la tabla

3.10. Implementación de un caso de enumeración:

La implementación de la enumeración se encuentra con la utilización del enum *Suscripcion*, que se encuentra en el paquete *interno*, para llevar a cabo la funcionalidad de Suscripciones de Usuarios, algunos atributos de instancia para la clase *Usuario*, como *límiteCuentas* que se encuentra en la línea 26, dependen de la instancia enum seleccionada. Además de esta enumeración, se hace uso de otras tres enumeraciones, llamadas *Cuotas, Divisas* y *Categoría*. Estas son usadas con el fin de realizar la caracterización de distintas *cuentas* y/o *metas* en el programa.
Por ejemplo, la enumeración llamada *Cuotas* nos permite almacenar los tipos de cuotas que serán ofrecidos por un banco, con el fin de que las cuentas corriente posean una de ellas y de esta manera se establezca el plazo esperado y determinado por el usuario para saldar sus compras en una cuenta corriente. Además de esto, nos permite determinar la cantidad de cálculos e impresiones a generar en el apartado de *Calculadora Financiera*. Un ejemplo de aplicación del enumerado se encuentra en la línea 17 de la clase *Corriente*, donde es usado como atributo que almacena la cantidad de cuotas que restan para saldar la deuda de la cuenta.
Ahora bien, la enumeración llamada *Divisas*, como bien su nombre lo indica, será la encargada de almacenar las abreviaciones de las divisas con las que trabajará el programa, estas mismas harán parte de los atributos de instancias de Bancos, Cuentas y Estados, que tendrán una divisa asignada en la cual se almacenará el dinero de sus respectivas cuentas. Además de esto, hará parte fundamental de la funcionalidad Cambio de Divisa, pues a partir de estas mismas se realizan las cotizaciones necesarias y se cambian los estados en los que se guardará la cuenta. Un ejemplo de esta se encuentra en la clase *Cuenta*, en la línea 20 como un atributo protected que almacena la divisa en la que se encontrará la cuenta.
Finalmente, hacemos uso del enumerado llamado *Categoría*, el cual contiene la tipología en la que el usuario podrá clasificar sus movimientos, con el fin de que este mismo tenga un control sobre sus movimientos financieros. Además de esto, se hará uso de este mismo en la funcionalidad de *Asesoramiento de Inversión*, con el fin de validar la cantidad de movimientos que han sido clasificados en una *Categoría* específica y dar una opción específica con respecto a estos. Un ejemplo de la implementación de esta se realiza como atributo protegido de la clase *Movimientos*, en la línea 19 y que servirá para clasificar los movimientos según la petición del Usuario.

Ver figura 22 de la tabla

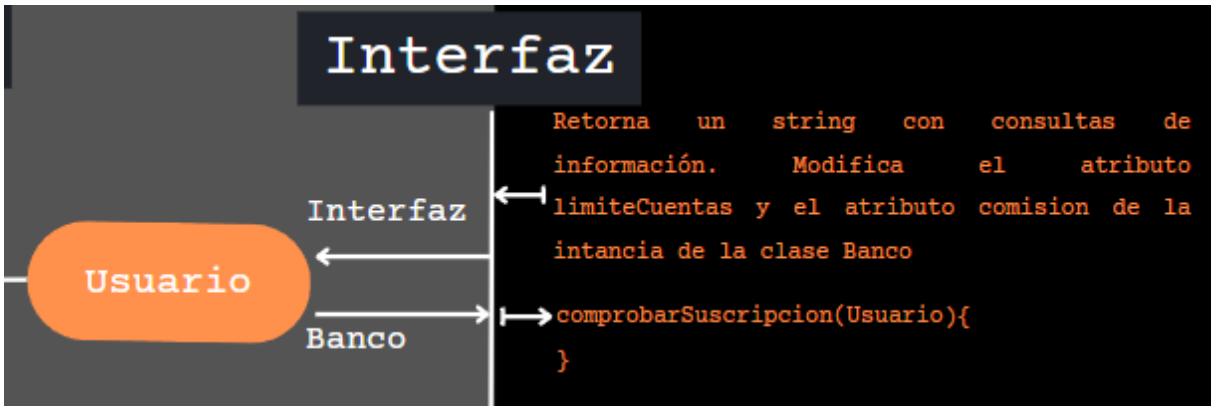
4. *Descripción de cada una de las 5 funcionalidades implementadas (incluye la descripción de la funcionalidad, que objetos intervienen en su implementación con un breve modelo de la secuencia del proceso, y por último, Incorporar una captura de pantalla con los resultados que presenta al usuario):*

4.1. Funcionalidad de Suscripciones de Usuarios:

```
Bienvenido a Gestión Económica, ¿en que te podemos ayudar?
1. Modificar mi suscripción
2. Invertir saldo de mi cuenta
3. Consignar saldo a mi cuenta
4. Transferir saldo entre cuentas
```

4.1.1. **Método comprobarSuscripcion**: El método de instancia *comprobarSuscripcion* que se encuentra en la clase *Banco* tiene como

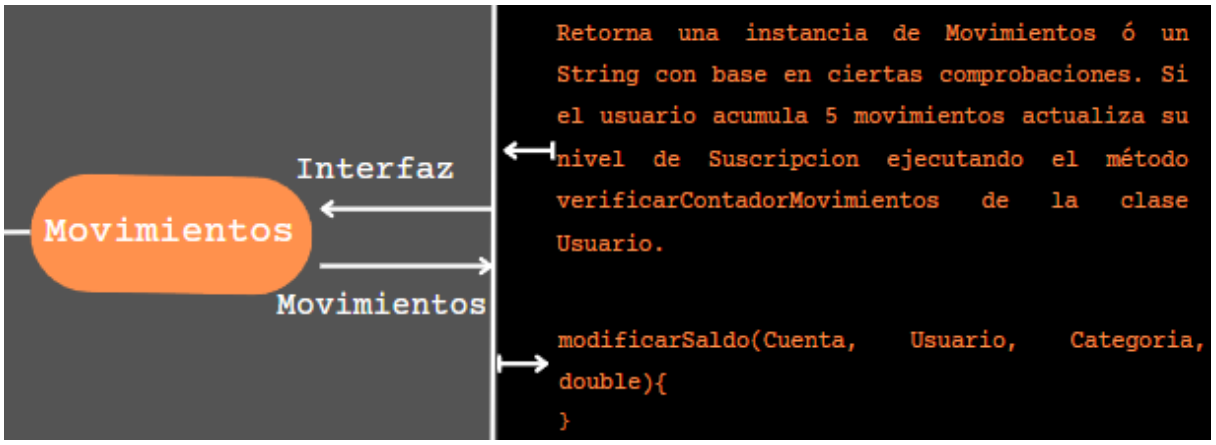
parámetro una instancia de la clase *Usuario*. En este método se consulta el atributo *Suscripcion* de la instancia de *Usuario* dada por parámetro y, con base en este, se modifica el atributo de instancia *limiteCuentas* de tipo *int* de la misma instancia de *Usuario*. Este atributo *limiteCuentas* se utiliza para establecer la cantidad de instancias diferentes de la clase *Cuenta* que se le pueden asociar a través del método de instancia *asociarCuentas,* que se encuentra dentro de la clase *Usuario*, a la misma instancia de *Usuario* pasada por parámetro. Estas cuentas son añadidas al atributo de instancia *cuentasAsociadas* de tipo *ArrayList<Cuenta>*, que se encuentra dentro de la clase *Usuario*. El atributo *comision* se invoca haciendo uso del *this*, luego, este valor se multiplica por K, donde K es un factor que varía con base en el atributo *suscripcion* del *Usuario* pasado por parámetro en el método. El método de instancia *comprobarSuscripcion* es ejecutado en la clase *Main* al momento de llamar al método estático *transferirSaldoCuentasUsuario*.




```
Bienvenido juan, eres un cliente BRONCE de nuestro banco, por eso te cobramos 0.3 de comision
¿Desea cambiar su nivel de suscripción? (Y/N): y
1. DIAMANTE
2. ORO
3. PLATA
4. BRONCE
Seleccione el número de suscripción: 2

El nivel de suscripción del usuario juan se ha actualizado a ORO
```

4.1.2. **Método modificarSaldo**: El método estático *modificarSaldo* que se encuentra en la clase *Movimientos* recibe como parámetros una instancia de *Usuario*, dos instancias de *Ahorros*, un enum de *Categoria* y un dato de tipo *double* llamado *cantidad*. Este método consulta el atributo de instancia *cuentasAsociadas* de tipo *ArrayList<Cuenta>* del usuario pasado por parámetro, posteriormente comprueba que el atributo de instancia llamado *origen* de tipo *Ahorros* pasado por parámetro se encuentre dentro de la lista *cuentasAsociadas* . Si se cumple la condición anterior entonces llama al método *crearMovimiento* de la clase *Movimientos*, posteriormente, éste último es asociado a la instancia de *Usuario* pasada por parámetro usando el método de instancia *asociarMovimiento* de la clase *Usuario*, finalmente, se retorna la instancia de *Movimientos*. El método *crearMovimiento* realiza una consulta al atributo de instancia *saldo* de tipo *double* de la cuenta *origen* pasada por parámetro en el método *modificarSaldo*, consulta el atributo de instancia *comision* de tipo *double* de la instancia de *Banco*, usando el método de instancia *getBanco()* de la clase *Usuario*, asociada a la instancia de *Ahorros destino* pasada por parámetro en el método *modificarSaldo* y, luego, consulta el atributo de instancia *tasa_impuestos* de tipo *double* que se obtiene luego de consultar el atributo de instancia *estadoAsociado* de la instancia de *B*anco asociada al *destino* de tipo *Ahorros* a través del método de instancia *getEstadoAsociado* de la clase *Cuenta* y del método de instancia *getTasa_impuestos* de la clase *Estado*. Posteriormente, dentro del método

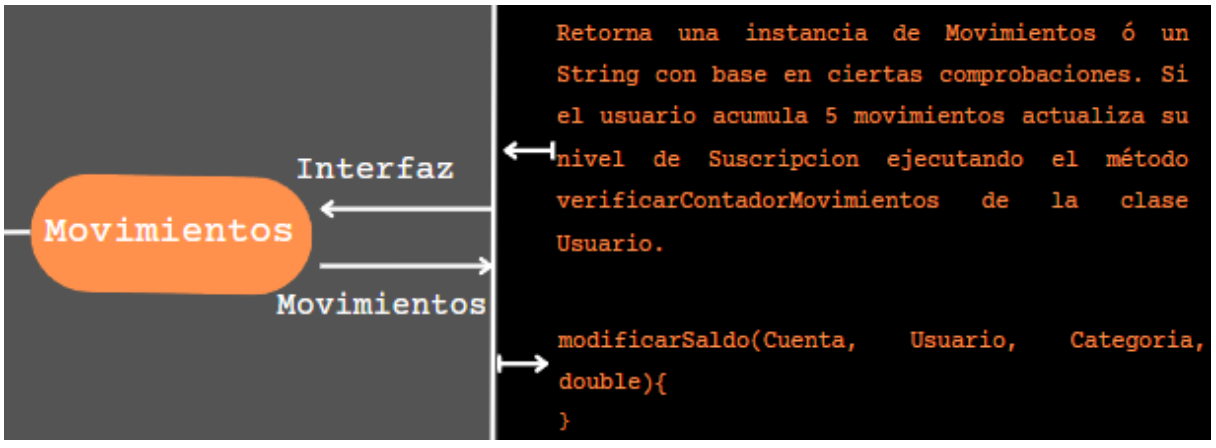
crearMovimiento se evalúa si el atributo *saldo* es menor que el atributo *cantidad*: si la condición es *true* entonces se retorna un *String* y se termina allí la ejecución del método *crearMovimiento* y *modificarSaldo*, este *String* se imprime en el método estático *transferirSaldoCuentasUsuario* que se encuentra en la clase *Main*; por el contrario, si la condición es *false*, entonces se retorna una nueva instancia de la clase *Movimientos*, esta misma instancia se retorna en el método *modificarSaldo* y, por último, se imprime en el método estático *transferirSaldoCuentasUsuario* que se encuentra en la clase *Main*. Cabe aclarar que el formato con el que se imprime una instancia de la clase *Movimientos* fue modificado en la sobreescritura del método *toString* que se encuentra dentro de la clase *Movimientos*. En el método *transferirSaldoCuentasUsuario* que se encuentra en la clase *Main* se ejecuta el método de instancia *verificarContadorMovimientos* que se encuentra dentro de la clase *Usuario*, en este último se comprueba el atributo de instancia *suscripcion* de tipo *Suscripcion*, siendo este un *enum*, y el atributo de instancia *contadorMovimientos* de tipo *int* de la instancia de *Usuario* que ejecuta el método. En este último método se evalúa que el atributo *contadorMovimientos* sea igual a cinco, si la condición es *true*, entonces se modifica el atributo de instancia *contadorMovimientos*, de la instancia de *Usuario* que se utiliza para llamar al método, usando el método *setContadorMovimientos* pasándole cero, se actualiza el atributo de instancia *suscripcion* de la instancia de *Usuario* que se utiliza para llamar al método, con base en la consulta del atributo *suscripcion* del mismo usuario, usando el método setSuscripcion, de lo contrario, si la condición es *false*, entonces se retorna un *String* con consultas de información. El método de instancia *crearMovimiento* de la clase *Movimientos* se encuentra sobrecargado, en este caso se ejecuta el método *crearMovimiento* que tiene como parámetros dos instancia de la clase *Cuenta*, un tipo de dato *double*, una instancia del *enum Categoria* y una instancia de la clase *Date*.




```
Movimiento creado
Fecha: Sun Apr 30 15:57:19 COT 2023
ID: 3
Origen: 0
Destino: 0
Cantidad: 25.0
Categoria: FINANZAS
Debes completar 5 movimientos para ser promovido de nivel, llevas 1 movimiento(s)
```

4.1.3. **Método invertirSaldo**: El método de instancia *invertirSaldo* que se encuentra en la clase *Ahorros* consulta el atributo de instancia *titular* de tipo *Usuario*, de la instancia de *Ahorros* utilizada para ejecutar el método, usando el operador *this* y el método de instancia *getTitular*, posteriormente, verifica el atributo de instancia *suscripcion* de la instancia *titular* y obtiene la constante *probabilidad_Inversion* de tipo *float* asociada a este. Esta última constante se

utiliza para realizar un cálculo aritmético que se almacena dentro de una variable de tipo *double* llamada *rand* y se evalúa que *rand* sea mayor ó igual a uno: si la condición es *true*, entonces se ejecuta el método estático *crearMovimientos* de la clase *Movimientos* y se asocia ésta instancia de *Movimientos* al *titular* de la instancia de *Ahorros* empleada para ejecutar el método *invertirSaldo*, usando el método *asociarMovimiento* de clase *Usuario*. De lo contrario, si la condición es *false*, entonces se retorna un *String*. En el método estático *crearMovimientos* de la clase *Movimientos* se comprueba que la instancia de *Ahorros* pasada por parámetro usando el *this* en el método *invertirSaldo* si se encuentre dentro del atributo de clase *cuentasTotales* de tipo *ArrayList<Cuenta>* de la clase *Cuenta*: si la condición es *true* entonces se verifica que la *categoría* pasada por parámetro sea de tipo *Categoria.PRESTAMO*, si la condición es *true* ó *false*, entonces se retorna una instancia nueva de la clase *Movimientos*, que posteriormente es impresa en el método estático *InvertirSaldoUsuario* de la clase *Main*, por el contrario, si la condición es *false*, entonces se retorna un *String*. Cabe aclarar que el formato con el que se imprime una instancia de la clase *Movimientos* fue modificado en la sobreescritura del método *toString* que se encuentra dentro de la clase *Movimientos*. En el método *InvertirSaldoUsuario* que se encuentra en la clase *Main* se ejecuta el método de instancia *verificarContadorMovimientos* que se encuentra dentro de la clase *Usuario*, en este último se comprueba el atributo de instancia *suscripcion* de tipo *Suscripcion*, siendo este un *enum*, y el atributo de instancia *contadorMovimientos* de tipo *int* de la instancia de *Usuario* que ejecuta el método. En este último método se evalúa que el atributo *contadorMovimientos* sea igual a cinco, si la condición es *true*, entonces se modifica el atributo de instancia *contadorMovimientos*, de la instancia de *Usuario* que se utiliza para llamar al método, usando el método *setContadorMovimientos* pasándole cero, se actualiza el atributo de instancia *suscripcion* de la instancia de *Usuario* que se utiliza para llamar al método, con base en la consulta del atributo *suscripcion* del mismo usuario, usando el método *setSuscripcion*, de lo contrario, si la condición es *false*, entonces se retorna un *String* con consultas de información. El método de instancia *crearMovimiento* de la clase *Movimientos* se encuentra sobrecargado, en este caso se ejecuta el método *crearMovimiento* que tiene como parámetros una instancia de la clase *Ahorros*, un tipo de dato *double*, una instancia del *enum Categoria* y una instancia de la clase *Date*.



```
La inversión de saldo ha sido exitosa:
Movimiento creado
Fecha: Sun Apr 30 15:59:44 COT 2023
ID: 7
Destino: 0
Cantidad: 194.40000547170644
Categoria: FINANZAS

Felicidades, has sido promovido al nivel de DIAMANTE, estos son tus beneficios:

puedes asociar un máximo de 5 cuentas, la probabilidad de ganar en tu inversión es de 0.8
```

4.2. Funcionalidad Asesoramiento de inversiones:

Diagrama de la funcionalidad:

[Diagrama de la funcionalidad](#) (Debe crear una cuenta en LucidChart para revisar el diagrama desde ésta página) **ó** ***Ver figura 23 de la tabla***

Funcionamiento:
La funcionalidad da una recomendación de un portafolio de inversiones en base a las preferencias y características del usuario, como las fechas de sus metas y sus movimientos o el dinero que hay en sus cuentas. Además, provee herramientas que pretenden mejorar aún más la inversión para la satisfacción del usuario.

La funcionalidad comienza haciendo unas comprobaciones, con el fin de que la funcionalidad opere como es esperado. Estas comprobaciones se basan en buscar las listas asociadas al usuario para verificar que no estén vacías. Una vez que se confirma que las listas no están vacías, se le pide al usuario que ingrese su factor de riesgo del 1 al 3, que se guarda como un *int* en la variable *riesgo*, después se le pide al usuario que ingrese la cantidad que desea invertir, que se guarda como un *double* en la variable *invertir*. Luego se ejecutan los siguientes métodos:

revisionMetas(Usuario): Este método de la clase Metas recibe como parámetro el usuario que está ejecutando la aplicación. Primero hace una comprobación, si el usuario tiene solo una meta, se imprime la fecha de esa meta (esto por temas de recursividad). Si no, recorre la lista de metas asociada al usuario y compara cada una de las fechas de las metas, almacenando la fecha más próxima. Luego devuelve la meta que sea más próxima, es decir, la meta que tenga la fecha menor dentro de la lista de metas asociadas al usuario.

*Una vez devuelva la meta se le pregunta al usuario si desea cambiar la fecha. En caso de que la respuesta sea afirmativa, se pasa al siguiente método (*cambioFecha)* y le preguntará a qué fecha desea modificar la meta. Si no, pasará directamente a *determinarPlazo*.

cambioFecha(Metas, String): Este método de la clase Metas toma la meta que devuelve *revisionMetas* junto con una fecha que ingresa el usuario en formato *String* y se cambia la fecha de esa meta a la fecha ingresada por el usuario. En este caso no se hace con *setFecha* ya que hay que hacer un cast porque las fechas se almacenan en formato *Date* y el método está recibiendo la fecha en formato *String*. Finalmente se retorna la meta con la nueva fecha.

determinarPlazo(Metas): Este método de la clase Metas toma la meta que devuelve *cambioFecha* (o en su defecto, si no se ejecuta este método, toma la de revisionMetas) y dependiendo de la fecha de esta meta se modifica una variable de instancia llamada *plazo* de tipo *String* que se encuentra en la clase Metas. Específicamente, la modificación se hace comparando la fecha de la meta que recibió como parámetro con otras dos fechas arbitrarias, y dependiendo del resultado se determina si el plazo de inversión es Corto,

Mediano o Largo, para modificar la variable estática *plazo* de la clase Metas y así asignarle uno de estos 3 valores.

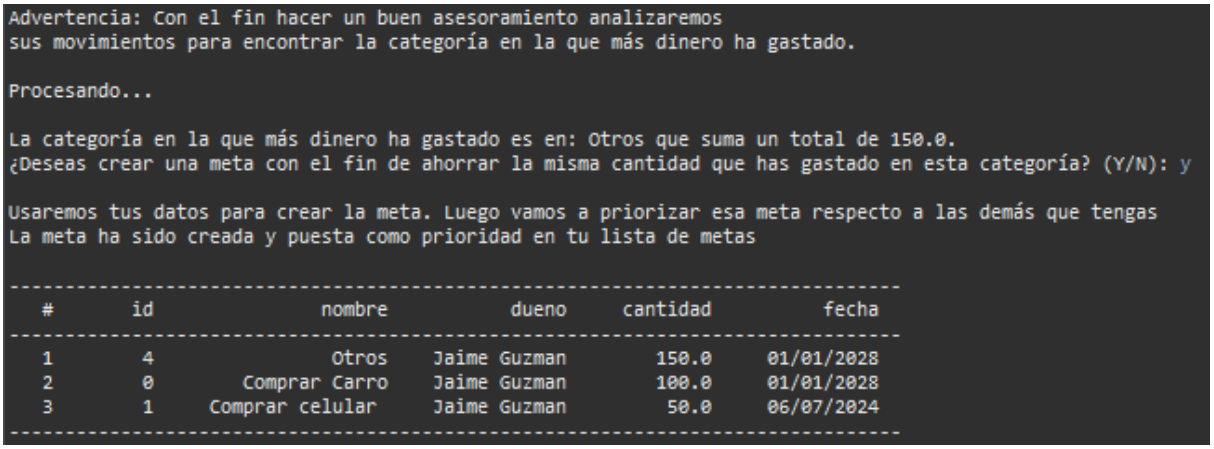
```
¿Cuál es su tolerancia de riesgos?:
1. Baja
2. Media
3. Alta
1

¿Qué cantidad piensa invertir?: 100

Tienes una meta para una fecha muy próxima: Comprar Carro, 100.0, 05/05/2023
¿Desearías cambiar la fecha de esta meta para invertir ese dinero en tu portafolio? (Y/N): y

¿Para qué fecha desearías cambiar la meta? (formato dd/MM/yyyy): 01/01/2028

La fecha ha sido modificada satisfactoriamente y su plazo de inversión es: Plazo Largo
```

Resultado por consola funcionalidad asesor inversiones

*Luego de mostrarle al usuario su plazo de inversión, se le va a advertir que se van a analizar sus movimientos para encontrar la categoría en la que más movimientos ha realizado y de esta obtener el total de dinero gastado.

analizarCategoria(Usuario, String): Este método está ubicado en la clase Movimientos recibe como parámetros el usuario que está ejecutando la aplicación y *plazo* (el *String* que se acaba de modificar en el método *determinarPlazo*) recorre los movimientos del usuario en busca de la categoría que más se repita (del enum Categoría) y suma la cantidad de dinero que ha gastado en cada uno de esos movimientos para obtener el total. Luego, devuelve la información recopilada al main mediante dos variables de instancia llamadas *nombreCategoria* y *cantidadCategoria* que están en la clase Movimientos y son un *String* y un *double* respectivamente. Además, este método toma el plazo y lo usa para modificar una tercera variable de instancia llamada *recomendarFecha* de tipo *String*. Para esto, el método primero aplica un filtro dependiendo del valor de la variable *plazo* (si es Corto, Mediano o Largo), y luego compara la fecha del último movimiento realizado por el usuario con la fecha de la meta que devuelve el método *revisionMetas,* para así determinar el valor de *recomendarFecha.* Con las variables *nombreCategoria, cantidadCategoria* y *recomendarFecha* se le va a preguntar al usuario si desea crear una meta para ahorrar la misma cantidad que ha gastado en esa categoría. En caso afirmativo se crea una nueva meta (con *nombreCategoria, cantidadCategoria* y *recomendarFecha* pasadas como parámetros) en el main. En caso contrario, se saltará el siguiente método (*prioridadMetas*) y se ejecutará directamente *retornoPortafolio*. En adición, se usa el método *asociarMeta* que se encuentran en la clase Usuario y se encargan de guardar la meta recién creada en la lista de metas asociada al usuario.

prioridadMetas(Usuario, Metas): Este método de la clase Metas toma la meta que se crea luego de ejecutar el método *analizarCategoria* y la va a priorizar en la lista de metas asociadas al usuario. Esto debido a que cuando una meta se crea va a estar por defecto de última en la lista de metas asociadas al usuario, pero con este método que recibe como parámetro el usuario que ejecuta la aplicación y la meta que se acaba de crear, se va a colocar en la primera posición. Para esto el método la inserta de primera en la lista y remueve la última posición. Posteriormente se llamará a *verMetas* en el main, que a su vez implementa la interface del programa*,* que muestra la lista de metas asociadas al usuario, donde se puede apreciar como la meta recién creada se encuentra en la primera posición. A su vez, se puede confirmar que la fecha modificada en el método *cambioFecha* efectivamente se cambió.



Resultado por consola funcionalidad asesor inversiones

retornoPortafolio(int, double, String, Usuario): Este método ubicado en la clase Banco recibe como parámetro el factor de riesgo del usuario (*riesgo*), la cantidad de dinero que el usuario desea invertir (*invertir*), el plazo de inversión del usuario (*plazo*), y el usuario que ejecuta la aplicación. El método inicia creando una variable de tipo *double* que se llama *interes* que depende del riesgo. Luego, el programa revisa 2 condiciones: Si el usuario tiene al menos una cuenta de Ahorros o si bien tiene alguna cuenta Corriente y revisa si el saldo o el disponible (dependiendo de la condición anterior va a revisar si tiene saldo, osea cuenta de ahorros, o disponible, osea cuenta corriente) de un de las cuenta asociada al usuario sea mayor o menor que el dinero que desea invertir. Entonces dependiendo del caso se crea una variable llamada *cobro* de tipo *double* que se obtiene luego de una operación aritmética con el saldo o bien el disponible. Esta variable se usa como parámetro para hacer un movimiento entre el usuario y una cuenta de tipo Ahorros que pertenece al usuario *ImpuestosPortafolio* (este usuario se llama mediante un método de la clase Usuario llamado *hallarUsuarioImpuestosPortafolio),* que hace un pequeño cobro, con el fin de simular una comisión o impuesto por mostrarle una recomendación de portafolio al usuario.

Posteriormente se va a evaluar otra condición que depende del método *impuestosMovimiento(double)*, ubicado en la clase movimientos y que recibe como parámetro la variable *interés* y se llama mediante el movimiento que se acaba de crear. Este revisa si el movimiento está dado entre cuentas que tengan el mismo banco o diferentes bancos. En cualquiera de los dos casos se le va a hacer un pequeño cobro al usuario, por un valor que depende de la variable *interes*. En adición, dependiendo del caso (si son bancos iguales o no) el método devuelve true o false, que en *retornoPortafolio* se usa como condición junto con el saldo o disponible del usuario y la variable *invertir* para devolver un número *int* que va desde el 0 hasta el 8, y que es el encargado de determinar el portafolio que se le va a recomendar al usuario.

bancoPortafolio(User) e interesesPortafolio(Banco, user): Estos dos métodos van de la mano. En primer lugar *bancoPortafolio* devuelve uno de los bancos asociados al usuario. Este banco se determina mediante un loop que toma el último banco que se le asoció al usuario. Luego, este banco que devolvió *bancoPortafolio* es pasado como parámetro a *interesPortafolio,* que define una variable de tipo *double* llamada *interes* y posteriormente dentro de un loop se hacen unas operaciones aritméticas que dependen del azar (uso de la librería random) y del banco que se recibe como parámetro para devolver un número de tipo *double* que es menor que 1 y simula un interés que cobra un banco asociado al portafolio. Estos métodos se usan para decirle al usuario que banco está asociado al portafolio (banco que devuelve *bancoPortafolio)* y cuál es la tasa de interes del banco respecto a ese portafolio (número que devuelve *interesesPortafolio*).

```
En base a los datos recolectados, deberías invertir tu dinero en estos sectores:
-Finanzas
-Cuidado de la salud
-Servicios de comunicación

Nota: Hay un banco asociado al portafolio: Banco de Colombia, con una tasa de interes del: 0.91%
```

Resultado por consola funcionalidad asesor inversiones

*Posteriormente se le preguntará al usuario si desea hacer un préstamo, si su respuesta es sí, se ejecutarán los siguientes dos métodos. En caso contrario la funcionalidad acabará en este punto.

gotaGota(double, Usuario, Ahorros): Este método se encuentra en la clase *Cuenta.* El parámetro Ahorros que recibe este método es la cuenta de Ahorros del usuario gotaGota (para hallar al usuario gotaGota y su cuenta se usa el método *hallarUsuariogotaGota* de la clase Usuario). El método *gotaGota* busca la cuenta asociada al usuario que tenga más saldo, o bien, más disponible mediante un loop que compara el saldo (o disponible) de las cuentas de tipo Ahorros del usuario (o cuentas de Corriente). Luego se hace un movimiento entre esta cuenta y la cuenta de ahorros del gotaGota por una cantidad especificada por una variable llamada *cantidadPrestamo* que es el *double* que recibe como parámetro este método y es ingresada por el usuario durante la ejecución del programa. Además devuelve la cuenta de tipo ahorros que determina el loop (la cuenta con más saldo o disponible).

vaciarCuenta(Ahorros): Recibe como parámetro la cuenta de ahorros del gotaGota y se llama mediante la cuenta que devuelve el método *gotaGota.* Se encarga de hacer un movimiento entre estas dos cuentas, donde se hace una transferencia de todo el saldo (o disponible) de la cuenta del usuario a la del gotaGota. Posteriormente esto se le notifica al usuario.

```
Finalmente, para mejorar aún más tu inversión te recomendamos hacer un préstamo. ¿Deseas hacer el préstamo? (Y/N): y

Las tasas de interés de los prestamos estan muy altas pero tenemos la solución perfecta para ti,
aunque no sea la más correcta... Vas a hacer un prestamo con el usuario gota a gota.
Ingrese el monto que desea solicitar prestado: 10000

Era una trampa, ahora el usuario gota a gota vació tu cuenta

Ha sido un placer asesorarte en este proceso, espero que nuestra recomendación haya sido de ayuda.
```

Resultado por consola funcionalidad asesor inversiones

*Luego el usuario se puede dirigir al menú Mis Productos para ver sus cuentas y verificar que efectivamente ya no tiene dinero en la cuenta donde más saldo o disponible tenía.

```
La lista de Cuentas creadas por el Usuario Jaime Guzman son:
CUENTAS DE AHORROS
-------------------------------------------------------------------------------------------------------
  #       id          nombre                titular      clave          saldo               banco
-------------------------------------------------------------------------------------------------------
  1       1         Ahorros             Jaime Guzman       1234        0.0 EUR    Banco de Colombia
-------------------------------------------------------------------------------------------------------
```

*Notar además que los resultados de la funcionalidad están sujetos a los deseos del usuario, ya que este puede responder diferente a las preguntas que se le hacen y por ende no se ejecutarán todos los métodos o bien las comprobaciones no serán las mismas que las mostradas en el ejemplo de las imágenes.

4.3. Funcionalidad Préstamos y Deudas:

Esta Funcionalidad tiene la intención de crear un sistema de préstamos que el usuario pueda realizar en caso de que lo necesite.En esta funcionalidad intervienen 5 clases diferentes, la clase Usuario, la clase Ahorros, la clase Deuda, la clase Movimiento,la clase Banco; estas clases interactúan entre sí de diferentes maneras

- **Usuario:** es el que interactúa con la interfaz y es al que se le asigna la deuda y es el dueño de las cuentas, además nos dice el tipo de suscripción que nos da las condiciones para realizar préstamos. Esta clase contiene el método ComprobarConfiabiliad() en la línea 220 el cual comprueba si el usuario puede realizar préstamos. Para hacerlo es necesario que tenga más de una cuenta de ahorros asociadas, que los bancos de estas cuentas presten dinero y su suscripción le permita realizar una suscripción más.
- **Banco:** Con su atributo préstamo nos dice la cantidad máxima que se puede prestar en cada una de las cuentas dependiendo del banco al que está asociado la cuenta.
- **Ahorros:** Son las cuentas del usuario con las que puede hacer préstamos, estas deben ser de esta clase ahorros. En esta clase se encuentra el método comprobarPrestamo() el cual se encuentra en la línea 125. Este método consigue las cuentas del usuario y consigue el banco asociado a cada cuenta, luego comprueba que el atributo préstamo del banco sea diferente de 0, y retorna una lista con las cuentas las cuales su préstamo es diferente de 0.
- **Movimientos:** En esta clase se encuentran dos metodos RealizarPrestamo() en la línea 281 y PagarDeuda() en la línea 296. RealizarPrestamo() se encarga de hacer las últimas comprobaciones antes de efectuar el préstamo y crea un movimiento ingresando el dinero a la cuenta; el método PagarDeuda() realiza un movimiento restando el dinero de la cuenta y disminuyendo la deuda, en caso de que se pague el total de la cuenta se elimina esta.
- **Deuda:**Las deudas son instancias de la clase deuda,al crear un prestamo se crea una instancia de esta clase, además de contar con el metodo conseguirDeuda() de la línea 93 que busca las deudas correspondientes al usuario debido a que es necesario en varias ocasiones para el correcto funcionamiento de la funcionalidad.

Ver diagrama de la funcionalidad Figura 24

El usuario ingresa al menú de realizar préstamo(opción 6-Gestionar Préstamos),tiene dos opciones,1-Pedir Préstamo 2-Pagar Préstamo.
Pedir Préstamo: En caso de elegir la primera se ejecuta el método comprobarConfiabilidad() comprueba que el usuario tenga cuentas de ahorros creadas, en caso de que no, se le muestra el error al usuario y se termina la funcionalidad, consigue las deudas del usuario con conseguirDeudas() y se comprueba que su suscripción le permite realizar como mínimo un préstamo, en caso de que no pueda realizar un préstamo se le muestra el error.

En caso de que las dos anteriores comprobaciones sean correctas se retorna una lista con las cuentas y se ejecuta el método (2) ComprobarPrestamo() este método recibe las cuentas y como se puede dar que algún banco no realice préstamos es decir que su atributo préstamo sea 0, se consiguen las cuentas que realizan préstamos y se agregan a un arreglo, en caso de que este arreglo sea de tamaño 0, se le muestra al usuario el error de que ninguno de sus bancos realiza préstamos, en el caso contrario se le muestra al usuario las cuentas con las que puede realizar préstamos y se le pide que seleccione una.

(3) Al momento de seleccionar una cuenta se le pide que ingrese la cantidad del préstamo que desea realizar y se ejecuta el metodo realizarPréstamo() el cual recibe la cuenta y la cantidad, comprueba que la cantidad ingresada por el usuario no exceda la cantidad permitida por el banco de la cuenta y el multiplicador de su suscripción, en caso que de no exceda este valor se crea la instancia de la deuda y se realiza el movimiento, en caso contrario se le muestra el error al usuario.

Pagar Préstamo: (4)Con el método de la clase deuda ConseguirDeudas() se buscan las deudas que corresponden al usuario, en caso de que no tenga deudas asociadas se le imprime al usuario el mensaje “Usted no tiene deudas por pagar”, en caso de que si tenga deudas, se le muestran al usuario estas deudas y se le pide al usuario que seleccione la que desee pagar y la cantidad a pagar,(5) se le pasa la cantidad con la deuda y el usuario al método pagarDeuda(), se comprueba la cantidad

Resultados del usuario

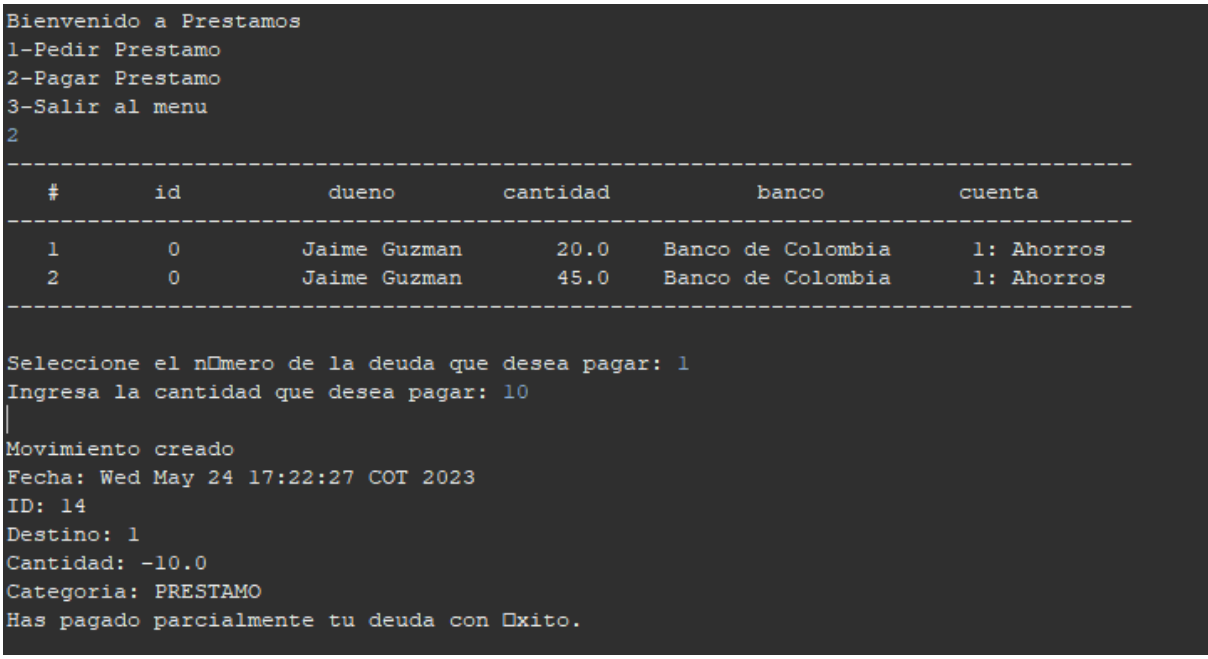
- **Caso 1: El usuario no puede pedir más préstamos**

```
Bienvenido a Prestamos
1-Pedir Prestamo
2-Pagar Prestamo
3-Salir al menu
1
|-------------------------------------------------------------------------------
   #       id          dueno        cantidad          banco          cuenta
---------------------------------------------------------------------------------
   1        0        Jaime Guzman       20.0    Banco de Colombia      1: Ahorros
   2        0        Jaime Guzman       45.0    Banco de Colombia      1: Ahorros
---------------------------------------------------------------------------------
♦Error! La suscripci♦n BRONCE solo permite realizar un total de 2. Usted tiene2/2 Deudas
```

- **Caso 2: El préstamo se realiza de manera exitosa**

```
Bienvenido a Prestamos
1-Pedir Prestamo
2-Pagar Prestamo
3-Salir al menu
1
Estas son las cuentas valida para hacer un prestamo y el valor maximo del prestamo
1-Cuenta: Ahorros Maximo a prestar:100.0
2-Salir al Men□
Seleccione una:
1

Ingrese el valor del prestamo, el valor de este debe ser menor de $100.0
20

Movimiento creado
Fecha: Wed May 24 17:17:21 COT 2023
ID: 11
Destino: 1
Cantidad: 20.0
Categoria: PRESTAMO

|-----------------------------------|

    Prestamo Realizado con Exito

|-----------------------------------|
```

- **Caso 3:Pagar Deuda**



4.4. Funcionalidad Compra de Cartera:

Diagrama de la funcionalidad:

> **Diagrama de la funcionalidad (Debe crear una cuenta en LucidChart para revisar el diagrama desde ésta página) ó ver figura 25 de la Tabla**

Planteamiento:
Lo que se plantea con esta funcionalidad es emular el mecanismo financiero de la Compra de Cartera usado para aliviar momentáneamente el bolsillo de los usuarios que hacen uso de esta, mecanismo que lo que hace es transferir una deuda de una cuenta a otra con la posibilidad de recibir mejores intereses y/o con la posibilidad de cambiar el plazo de pago de dicha deuda. De esta manera, lo que se busca es que el usuario escoja una de las cuentas Corriente de las cuales tiene una deuda, luego realice el mismo proceso con una capaz de recibir dicha deuda verificando los intereses bancarios con los cuales quedaría para posteriormente realizar las comprobaciones correspondientes por parte del usuario y ejecutar los cambios necesarios en las cuentas correspondientes.

Funcionamiento:
La funcionalidad realiza los cambios necesarios a las cuentas escogidas por parte del usuario con base a sus preferencias en términos de cuotas e intereses. Además de esto, se brinda al usuario la posibilidad de una revisión completa de la información necesaria de la cuenta destino, además de una revisión del cómo sería el proceso mes a mes para finalizar con el pago de la deuda con la que pretende finalizar dicha cuenta.

La funcionalidad comienza haciendo unas comprobaciones, con el fin de que la funcionalidad se pueda ejecutar. Estas primeras comprobaciones se basan en la verificación de la posesión por parte del usuario de mínimo dos cuentas corrientes y que al menos una de estas posea deuda alguna. Para la primera verificación se hace uso del getter del atributo CuentasCorrienteAsociadas y se evalúa que hayan por lo menos dos cuentas en este arreglo. Para la segunda comprobación se hace uso del siguiente método:

retornarDeudas(): Al ser un método de instancia de la clase usuario, hace uso del atributo cuentasAsociadas (almacena las cuentas asociadas a un usuario) del usuario para filtrar las cuentas Corriente que un usuario tiene asociadas y verificar que se tenga deuda alguna en esta cuenta (esto se realiza comparando los atributos cupo y disponible de la cuenta (el que sean diferentes nos indica que en algún momento adquirimos una deuda en esta cuenta)).

Aquellas cuentas que pasen dicha verificación serán agregadas a un arreglo llamado *cuentasConDeuda*, atributo que es retornado por el método.

Una vez obtenidas las cuentas con Deuda (almacenadas en el arreglo *cuentasEnDeuda* obtenido con el método anterior), se realiza la verificación de la existencia de al menos una deuda (con el tamaño del arreglo). Si no se pasa alguna de estas dos comprobaciones se otorga el mensaje correspondiente al caso y se finaliza la funcionalidad.

Se realiza la impresión de las cuentas almacenadas en el arreglo *cuentasEnDeuda* para que el usuario escoja a cual de estas cuentas con deuda desea aplicar el mecanismo financiero, elección realizada mediante la elección de un numeral asociado y almacenado como el entero *Cuenta_Compra*. Se realiza la impresión de la información de la cuenta escogida y se pide la confirmación de que dicha cuenta es justamente a la que se desea aplicar la funcionalidad. En caso de que se exprese que no es la cuenta escogida por el usuario, se le da la opción de reelegir dicha cuenta o de salir de la funcionalidad.

Se envía un arreglo con las cuentas asociadas al usuario (menos la cuenta escogida) y la cuenta escogida al método *Capacidad_Endeudamiento*, que validará las cuentas propias capaces de recibir la deuda de la cuenta escogida, luego de esto se enviará el arreglo recibido por este método a uno llamado *verificarTasasdeInteres* que devolverá las tasas de interes que cobrará cada uno de los bancos a las cuentas por recibir esta nueva deuda. A continuación, una explicación más detallada de los dos métodos usados:

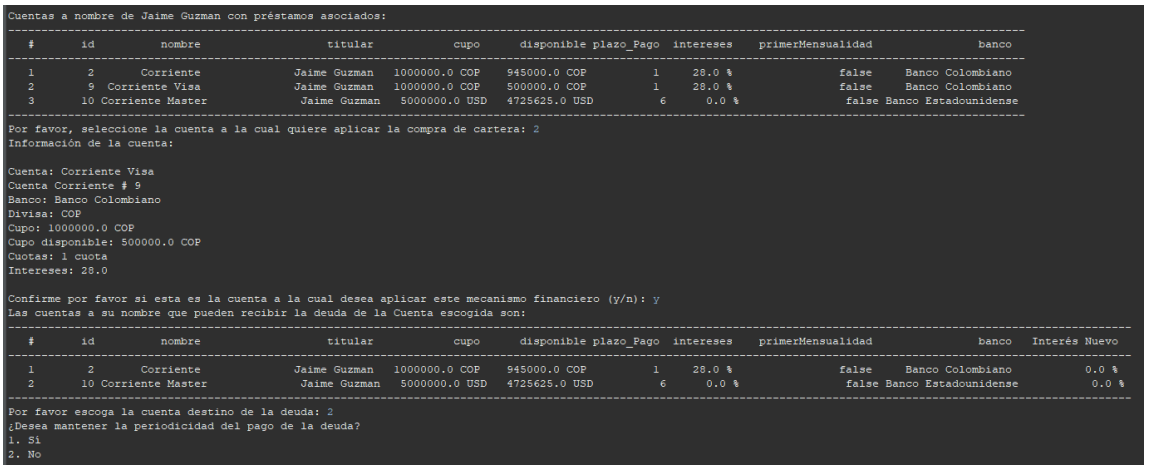
Capacidad_Endeudamiento(ArrayList<Cuenta>, Corriente): Este método de instancia de la clase Usuario tiene como objetivo principal realizar las validaciones necesarias de las cuentas corriente asociadas a un usuario con el fin de devolver un arreglo con las cuentas que sean capaces de recibir la deuda de una cuenta ingresada. Para esto, el método recibe dos parámetros, uno con el arreglo de las cuentas a evaluar y otro con la cuenta de la cual se quiere saldar la deuda. Dentro del arreglo se toma la referencia de la deuda (restando el disponible del cupo) y se almacena como atributo auxiliar. Luego de esto se itera el arreglo de cuentas y se realiza la verificación de que sea una cuenta corriente y así mismo se valida que el disponible de la cuenta en mención sea mayor que la deuda ya definida (esto pues el disponible me indica el dinero del que puedo hacer uso aún en una cuenta). De ser así, la cuenta que se está validando se almacena en el arreglo *cuentasCapacesDeuda*, arreglo que retorna el método. Tener en cuenta que detrás de la verificación de que sea capaz de realizar la deuda, se valida la realización de un cambio de divisa con el fin de tener en consideración que entre las cuentas se tengan divisas distintas, esto se realiza en el método *DineroaTenerDisponible* que se describe a continuación.

DineroaTenerDisponible(Cuenta, Divisas): Este método estático de la clase Cuenta tiene como objetivo principal realizar el cambio de divisa con el fin de que el valor que sea validado sea el correspondiente en cada cuenta según su divisa. Recibe una cuenta Corriente de la cual será extraída la Deuda y la Divisa de esta. El otro parámetro recibido será la Divisa a la cual quiero cambiar la deuda que recibo en el parámetro anterior. Para este proceso se emplea la funcionalidad de Cambio de Divisa, descrita en el documento. Se devuelve la deuda en la otra divisa para que sea verificado por el método anterior.

verificarTasasdeInteres(Usuario, ArrayList<Corriente>): Este método estático de la clase Banco tiene como objetivo principal realizar la verificación de las tasas de interés de un conjunto de cuentas y retornarlas de manera ordenada. La idea de la validación se basa en que cada uno de los bancos brindará determinados descuentos a cada cuenta y a cada usuario, según su suscripción y la cantidad de movimientos que tenga asociados a un banco. Así pues, el método recibe dos parámetros, el primero de ellos un Usuario del cual se tomará el valor de Suscripción, el segundo un arreglo de cuentas Corriente sobre las cuales se determinará la tasa de interés. En este método se hace una iteración de las cuentas Corriente que vienen en el arreglo y las envía a la sobrecarga de este método que se describe más adelante, pero que devuelve la tasa de intereses a cobrar a cada cuenta, para luego añadirlo al *ArrayList<Double> tasasdeInteres* que devuelve el método.

verificarTasasdeInteres(Suscripción, Corriente): Este método estático de la clase Banco tiene como objetivo principal realizar la validación de la tasa de interés a aplicar de una cuenta corriente con base en su Suscripción y en su descuento por parte del Banco. Para realizarlo se reciben dos parámetros, el primero de ellos con la Suscripción del Usuario que será usada como casuística y el segundo de ellos una cuenta Corriente de la cual se extraerá el banco asociado con el fin de validar los descuentos necesarios. El método comienza inicializando el atributo a devolver, llamado interés, con el fin de evitar ambigüedades. Luego de esto se manejan tres atributos de tipo double, uno de ellos (*descuento_movimientos*) que recibirá mediante el método *retornarDescuentosMovimientos* (método desarrollado más adelante) el descuento generado por parte de los movimientos. El segundo atributo de tipo double (*descuento_suscripcion*) es un array con elementos de este tipo y es generado con el método *retornarDescuentosSuscripcion* (método desarrollado más adelante) y almacena los descuento que brinda el banco según la suscripción que un usuario pueda tener. Finalmente se hace uso del tercer atributo (*descuento_total*), que almacenará los atributos finales que un banco brindará a una cuenta según su suscripción y son determinados mediante el método *descuentoTotal* (desarrollado más adelante). Notar que los tres atributos anteriores son atributos auxiliares concatenados uno con otro y el tercero de ellos es resultado de la suma respectiva de los primeros dos. Una vez que se tiene el atributo *descuento_total* procede realizar la casuística de la suscripción del Usuario recibida por parámetro, esto se realiza con un switch, que según la suscripción que se posea, evalúa si no es mayor al interes bancario corriente del Estado (con el fin de verificar que no se retorna un interes negativo pues no tendría sentido) y si no es mayor a este, se realiza la resta del interes del Estado menos el descuento para asignarlo a *interes* y devolver este valor. Notar que el interés bancario corriente del Estado se recibe desde la cuenta, accediendo al banco, luego al estado y posteriormente al atributo de interés (Ruta: *cuenta.getBanco().getEstadoAsociado().getInteres_bancario_corriente()*).

Luego de esta serie de métodos, se verifica que el arreglo devuelto no esté vacío (esto nos indicará de que hay por lo menos una cuenta capaz de recibir la deuda), de no ser pasada la verificación se imprime el mensaje correspondiente y se finaliza la funcionalidad. De ser pasada la verificación, se continúa el programa con la impresión de las cuentas del arreglo en mención de manera que el usuario escoja la cuenta a la cual desea enviar la deuda escogida anteriormente. Dicha elección se almacena en el entero *Cuenta_Destino*.



Luego de escoger la cuenta destino de la deuda, comienza el proceso en el cual el usuario escoge las preferencias con las cuales va a realizar la compra de cartera. Primero, se le pide al Usuario que escoja si desea mantener la periodicidad del pago de la cuenta que tiene (cantidad de cuotas en la que se salda la deuda de la cuenta destino). De ser afirmativa la respuesta se asigna al atributo *eleccion_periodicidad* de tipo Cuotas el atributo *plazo_Pago* de la cuenta destino (la razón por la cuál para la declaración del atributo se asigna por defecto *Cuotas.C1* (1 cuota) se debe a la intención de evitar ambigüedades y la advertencia de que el atributo podría no estar inicializado, sin embargo, por como están las condiciones, se asegura de que se asigne al valor correspondiente). Empero, de ser negativa esta respuesta se da la opción al usuario de que escoja la nueva periodicidad de la

cuenta, esto se realiza mediante la impresión de las posibles cuotas y la elección del usuario. Posteriormente, según esta selección se asigna al atributo *eleccion_periodicidad* el valor correspondiente a la elección.

Una vez tenemos la preferencia del usuario frente a la periodicidad de su pago, se procede a realizar la vista previa de cómo quedaría la cuenta una vez se hayan realizado los cambios según las elecciones hechas por el usuario, para este fin, se hará uso del método *vistaPreviaMovimiento* que nos devolverá una cuenta Corriente nueva con los datos cambiados. El método se describe mejor a continuación.

vistaPreviaMovimiento(Corriente, Cuotas, double, double): Este método estático de la clase Corriente tiene como objetivo principal generar una cuenta corriente semejante a la ingresada por parámetro salvo los parámetros justamente ingresados al método, esto con el fin de generar la vistaPrevia de una cuenta en una instancia independiente a otra. El método recibe 4 parámetros, el primero de ellos la cuenta que tendremos como base para nuestros cambios, el segundo un plazo de tipo Cuotas, el tercero de tipo double, que corresponde a una deuda previa a generar en nuestra nueva cuenta y el cuarto también de tipo double, que corresponde al interés con el cuál quedará la cuenta. Para cumplir con el objetivo del método, se hace uso de la interface Clonable, interface de la cual se sobrescribe e implementa el método *clone()* (desarrollado más adelante) con el fin de recibir un nuevo objeto totalmente independiente del ingresado pero con los mismos atributos de este. Este nuevo objeto se almacena en el atributo de tipo cuenta *Corriente* llamado *cuenta_aux*. Con los setters de la clase corriente (*setDisponible(), setIntereses(), setPlazo_Pago()*) se modifican los atributos de la cuenta *cuenta_aux* para luego devolver esta cuenta corriente.

clone(): Es un método de instancia implementado en la clase Corriente, que se encarga de crear un nuevo objeto de tipo Corriente usando el constructor predeterminado en principio. Posterior a la creación de estos, hace uso de los getters y setters para asignar a este nuevo objeto cada uno de los atributos que tenía el objeto (llamado con *this* al ser de instancia). Luego de hacer los cambios necesarios, devuelve el nuevo objeto.

A continuación, se realiza la validación de la cantidad de cuotas que fueron escogidas por el usuario, almacenadas en el atributo *elección_periodicidad* y si la cantidad de cuotas de este atributo (tomadas con *getCantidad_Cuotas()*) es mayor que 1, se brinda la posibilidad al usuario de que pueda realizar el pago de intereses en el primer mes del pago de la deuda, o si no desea realizarlo pero teniendo en cuenta de que tendrá que pagar dicho valor correspondiente en el segundo mes de cuota. Se realizan dichas validaciones y por tanto se tendrán dos casos, según la elección ya descrita.
En el primero de ellos haremos uso del método *retornoCuotaMensual*, al que enviaremos la deuda que recibe el mecanismo financiero con el fin de retornar la primer cuota a pagar tomando en cuenta que se realizará pago de intereses en dicho mes. En el segundo caso usaremos una sobrecarga del mismo método, al cual enviaremos también la deuda a recibir y un 1, como referencia de que será el mes uno a pagar, este método sobrecargado toma en cuenta el caso en el cual no se pagarán intereses en el primer mes y obtendremos así la cuota del mismo. Además, según sea la opción se asignará *true* o *false* según sea el caso al atributo *primerMensualidad* de la cuenta asociada a la *vistaPrevia*, que almacena justamente la decisión ya descrita. A continuación se explican en detalle los métodos mencionados anteriormente.

retornoCuotaMensual (double): Este método de instancia de la clase Corriente que tiene como objetivo principal hacer el cálculo de los valores a pagar en una cuota con la condición de que en el primer mes sí se hace el pago de intereses correspondientes. Es un método que recibe como parámetro un double que almacena la DeudaActual con la que llega el usuario para el cálculo de la cuota independiente del mes en cuestión. (Notar que el hecho de que la DeudaActual sea la DeudaTotal significa que se habla del primer mes de pago de cuota), y que devuelve la información de la cuota correspondiente. Este inicia creando un Array de tipo double, Array llamado *cuotaMensual* donde se añadirán los valores y la información a devolver por parte del método. Luego de esto, se llama al método estático de la clase *Corriente* llamado *calculoInteresNominalMensual*, al cual se le envían

los intereses de la cuenta desde la cual se llama el método en el que estamos. Este método me devuelve el interés nominal mensual calculado a partir de los intereses de la cuenta (que hacen acolación al atributo *intereses* pero que referencia la tasa efectiva de interés anual de la cuenta. Este método se explica en detalle más adelante. Luego de tener la tasa de interés a usar se hacen cálculos de *interes*, *abono_capital* y *cuotaMensualFinal* de la cuota en cuestión, agregándolos en orden a las posiciones del Array a devolver y haciendo uso de los atributos de la cuenta de instancia correspondiente al método. Una vez hecho esto, se devuelve este Array con la información de la cuotaMensual.

retornoCuotaMensual(double, int): Este método de instancia es una réplica del método anterior pero en la condición en que en el primer mes no pagaremos intereses. Para esto, hacemos uso del segundo atributo que es ingresado como parámetro, en donde hacemos un condicional que cubre tres casos, el primero de ellos en donde estamos en el mes 1 y no pagamos intereses, el segundo en el cual se sumarán los intereses del mes 1 con los del mes 2 y el tercero para el mes 3 en adelante, en donde se copia el del método anterior.

calculoInteresNominalMensual(double): Método estático de la clase *Corriente* que se encarga de hacer la conversión de la tasa efectiva anual (recibido por parámetro) por la tasa de interés nominal mensual. Son cálculos matemáticos basados en la realidad del cálculo de este tipo de datos. Una vez realizadas las operaciones algebraicas correspondientes, se hace uso del método *redondeoDecimal* que se encargará de redondear el decimal para retornar un porcentaje con dos cifras decimales.

redondeoDecimal(double, int): Método estático de la clase Cuenta que tiene como objetivo principal el de redondear los valores de las cantidades monetarias y de los porcentajes usados para la impresión de los datos. Recibe un valor double a redondear y un entero que indica la cantidad de decimales que deseamos (se hace así para generalizarlo, pero su utilidad principal es la descrita). Luego de ser recibido, realiza la operación algebraica necesaria para hacer el redondeo con la ayuda de la librería *Math*, para devolver el valor que resulta de tal operación.

Luego de recibir cuál será la primer cuota a pagar por parte del usuario según su elección, se procede a realizar la impresión de la *vistaPrevia* de la cuenta y de la primer cuota con el fin de que el usuario tenga claridad en su decisión y escoger lo mejor para su proyección monetaria. Además de esto, se brinda la posibilidad de que el usuario obtenga un resumen completo de las cuotas que tendrá que pagar con el fin de saldar la deuda en el mes correspondiente. Esto se realizará con base el la decisión del pago de intereses de la cuenta auxiliar de *vistaPrevia*, almacenada en el atributo *PrimerMensualidad*, según este llamaremos al método sobrecargado correspondiente *calculadoraCuotas* que nos devolverá un Array con Arrays de tipo double que almacenarán cada una de las cuotas de todos los meses necesarios para saldar la deuda. A continuación una descripción detallada de los métodos usados para esto.

calculadoraCuotas(Cuotas, double, double): Este método estático de la clase Corriente tiene como objetivo principal realizar el cálculo y retorno de cada una de las cuotas necesarias para hacer el pago de una deuda. Para este fin, el método recibe un parámetro de tipo Cuotas, del cual se obtiene la cantidad de cuotas en que se hace el pago, dos parámetros de tipo double, que reciben la deuda y los intereses a cobrar respectivamente. Este método hace uso de un ciclo que emula el método *retornoCuotaMensual* ya descrito, simplemente se hace uso de un ciclo for con el fin de realizar dicho proceso para cada uno de los meses y agregándolo al atributo *cuota* de tipo double[][] que será devuelto al final del método. En este caso se toma el caso en el que se pagan los intereses todos los meses.

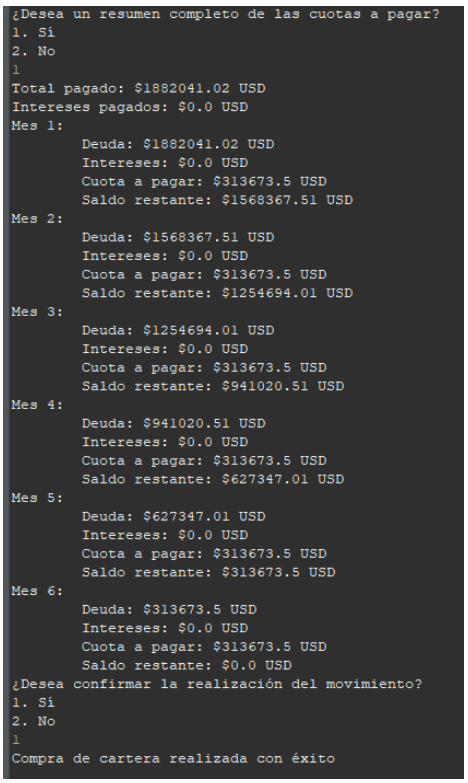
calculadoraCuotas(Cuotas, double, double, boolean): Este método estático de la clase Corriente tiene como objetivo principal realizar el cálculo y retorno de cada una de las cuotas necesarias para hacer el pago de una deuda, teniendo en cuenta en este caso que no se realizará el pago de intereses en el primer mes de cuota. Para este fin, el método recibe un parámetro de tipo Cuotas, del cual se obtiene la cantidad de cuotas en que se hace el pago, dos parámetros de tipo double, que reciben la deuda y los intereses a cobrar respectivamente. Finalmente se recibe un booleano auxiliar que nos indica que método usar y específica que no se pagarán intereses en el primer mes de cuota. Este método hace uso de un ciclo que emula el método *retornoCuotaMensual* sobrecargado ya descrito, simplemente se hace uso de un ciclo for con el fin de realizar dicho proceso y teniendo en cuenta el mes de la cuota en que se realiza, para cada uno de los meses y agregándole al atributo *cuota* de tipo double[][] que será devuelto al final del método. En este caso se toma el caso en el que se pagan los intereses todos los meses.

Luego de obtener las cuotas a pagar que serán almacenadas en el atributo *cuotaCalculadora*, para luego enviarlo al método *informacionAdicionalCalculadora* y recibir la información adicional correspondiente al pago total cuando se finalice el pago de la deuda que será almacenada en el atributo *infoAdicional*. Más adelante se detalla bien el método.

informacionAdicionalCalculadora(double[][], double): Este método estático de la clase Corriente tiene como objetivo principal retornar información de interés al momento de pagar una deuda en su totalidad, estos harán referencia a la deuda total, al total pagado real y a los intereses pagados. Para esto se reciben las cuotas y sus valores, junto con la deuda inicial y se realiza el recorrido del primer parámetro con el fin de hacer la suma de todas las cuotas para obtener los intereses pagados. Además, notar que los intereses se recibirán de la diferencia entre el total pagado y la deuda inicial. En cada uno de estos casos se hace el uso del método *redondeoDecimal* con el fin de realizar una devolución de valores más adecuada, manejando dos decimales. Cada uno de los valores recibidos se agrega a un Array de tipo double y se retorna al programa.

Tanto *cuotaCalculadora* como *infoAdicional* se envían al método estático del Main llamado *calculadoraCuotas* sobrecargado, este mismo se encargará de realizar la impresión de las cuotas en el orden deseado. Luego de este proceso, se procede con la validación de la realización del movimiento, para esto se da la opción al usuario de confirmar el deseo de realizar el movimiento, de ser afirmativa la respuesta, con los setters se reiniciarán los datos necesarios de la cuenta origen (disponible y plazo_Pago), además se elimina de las cuentas asociadas al Usuario y de las cuentas Totales la cuenta destino escogida con el fin de que sea reemplazada por la cuenta Corriente de *vistaPrevia*, que será la que entrará a hacer parte de las cuentas del usuario (se usa el *asociarCuenta*) de la clase Usuario y se imprime el mensaje de éxito.

De ser la respuesta negativa, se elimina la *vistaPrevia* creada en el método de la lista con las cuentas totales, con el fin de que desaparezca y se dejan las cuentas que tenía el usuario tal cual como las tenía en un inicio. Una vez realizada la confirmación finaliza la funcionalidad.



4.5. Funcionalidad de Cambio de Divisa:

Planteamiento:
La funcionalidad de cambio de divisa se desarrolló con el propósito de solventar las disparidades de valor que existen entre diversas monedas durante un proceso de intercambio monetario. En un entorno económico, donde las monedas poseen valores fluctuantes, resulta esencial contar con una solución que permita equilibrar estas diferencias. El cambio de divisa proporciona la capacidad de convertir una moneda en otra, lo cual simplifica y agiliza las transacciones. Esta funcionalidad busca minimizar las barreras y dificultades asociadas con las disparidades de valor, lo que a su vez promueve una mayor eficiencia y estabilidad en los intercambios monetarios del programa. Además, el cambio de divisa también es una herramienta, al permitir a los individuos y empresas diversificar sus inversiones y activos en diferentes monedas.

Funcionamiento:

El método principal es “CambioDivisa()”. Al ser llamado, muestra un mensaje de bienvenida al servicio de cambio de divisa. Si la variable booleana novato es verdadera, se muestran explicaciones adicionales sobre los tipos de cambio disponibles: convencional y exacto.

A continuación, se solicita al usuario que elija el tipo de cambio deseado, en caso de escoger exacta se establece la variable booleana “exacta” como verdadera. Luego se solicita al usuario que seleccione la divisa desde la cual desea hacer el cambio. Se muestra la lista de divisas disponibles en el sistema y se lee la elección del usuario. La divisa seleccionada se almacena en la variable “divisaA”. A continuación, se buscan las cuentas de ahorro asociadas al usuario que tengan la misma divisa que “divisaA” y se almacenan en la lista “ahorrosPosibles”. Se muestra nuevamente la lista de divisas disponibles, excluyendo la previamente escogida, y se solicita al usuario que seleccione la divisa del destino. La divisa correspondiente se guarda en “divisaB”.

Se verifica si el cambio es exacto o convencional, y se muestra un mensaje para solicitar el monto a cambiar. El valor ingresado por el usuario se almacena en la variable “monto”.

A continuación, se crea un objeto de la clase Movimientos llamado “cambioDiv”, que representa el cambio de divisa a realizar. Se invoca el método estático “facilitarInformación()” de la clase Movimientos para buscar bancos que permitan realizar el cambio de divisa especificado. El resultado se almacena en la lista “existeCambio”.
facilitarInformación() recibe un objeto movimiento con la divisa de origen, la de destino y el usuario. De los bancos asociados al usuario, les cambia el atributo asociado a true. Luego para los bancos asociados al usuario, recorre los tipos de cambio de divisa del atributo “Dic”, si encuentra un tipo de cambio igual al pedido, anota el banco en la lista “existeCambio”.

Se verifica si existe al menos un banco disponible para el cambio en la lista “existeCambio”. Si no hay bancos, se muestra un mensaje y se termina el proceso. Si hay bancos, se muestra la cantidad y se continúa. Luego se muestra una lista de las cuentas de ahorro disponibles en “ahorrosPosibles”, es decir, las cuentas del usuario con la divisa como la desde donde se va a hacer el cambio . Si no hay cuentas disponibles, se muestra un mensaje y se termina el proceso.

Se muestra una lista de cotizaciones posibles para el cambio de divisa y se utiliza el método estático “cotizarTaza()” de la clase Banco para obtener la información de cotización de los bancos disponibles. Las cotizaciones se almacenan en la lista “imprimir”.
cotizarTasa() recibe el usuario, la lista “existeCambio”, “cadena” que es la concatenación de los nombres de las divisas escogidas, y la lista “ahorrosPosibles”. Para cada ahorro de “ahorrosPosibles”, para cada banco de “existe cambio” se obtiene el valor de la tasa de cambio y se le hacen descuentos si es en el propio banco y se se está asociado.La cuota de manejo depende de la suscripción. Todos estos datos se almacenan en “imprimir” que es también un objeto de tipo movimientos, a modo de cotización.

A continuación, se muestra la lista de cotizaciones y se solicita al usuario que elija una opción. La cotización seleccionada se almacena en la variable “escogencia” que es un objeto de la clase Movimientos.

Se solicita al usuario que confirme si desea continuar con el proceso. Si la respuesta es negativa, se termina el proceso. Luego se solicita al usuario que indique si la cuenta receptora del dinero es de su propiedad o no. Si la respuesta es negativa, se solicita el nombre o correo electrónico del destinatario y se busca el objeto Usuario correspondiente. De cualquier forma, el usuario se almacena en la variable de tipo Usuario “usuarioB”.

A continuación, se solicita al usuario que elija la cuenta receptora del dinero en la divisa objetivo, mostrando una lista de opciones. La cuenta seleccionada se guarda en la variable “cuentaB”.

Si el cambio es exacto o no, se verifica si el usuario tiene suficientes fondos en la cuenta de origen para realizar el cambio, esto se hace mediante el método sobrecargado “comprobarSaldo()” . Si no hay suficientes fondos, se muestra un mensaje y se termina el proceso. Si hay suficientes fondos, se llama al método sobrecargado, para “exacta” o no, “hacerCambio()” de la clase Cuenta para realizar el cambio de divisa con el monto especificado.
hacerCambio() recibe la escogencia, el monto del cambio, la cuenta que lo recibe y el usuario. su multiplica por los valores escogidos en la cotización de “escogencia” y ser crea el objeto movimiento “m”, que esta vez sí efectúa cambios en los saldos de las dos cuentas implicadas.

Finalmente, se muestra una tabla indicando cómo quedan las cuentas después del cambio de divisa.

Diagrama Funcionalidad (Necesita de una cuenta de Lucidchart).

5. ***Manual de usuario con los elementos necesarios para poder evaluar el correcto funcionamiento del sistema (nombres, contraseñas, etc):***

Importante: Lo que está en cursiva son las preguntas que se le hacen al usuario y que debe responder ingresando datos por consola.

Instrucciones para Iniciar Sesión:

Correr el programa. Aparecerá un menú de inicio de sesión:

Bienvenido al gestor de dinero:

1. Escribir uno (1) para ingresar un usuario:
   - **1.1. *Nombre de usuario:* Jaime Guzman**
   - **1.2. *Contraseña:* 12345**
   - 1.3. (Los demás usuarios que están cargados en el programa están con el propósito de participar en las funcionalidades del programa o con propósitos de pruebas).
2. Escribir dos (2) para crear un usuario:
   - 2.1. Si se crea un usuario desde cero se tendrán que crear los diferentes objetos necesarios para el funcionamiento de algunas funcionalidades manualmente mediante la consola.
3. Escribir tres (3) para el acceso administrativo:
   - 3.1. Con este acceso se podrán hacer cosas adicionales que un acceso normal no te permiten.
4. Escribir 4 para cerrar el programa:

Instrucciones del menú Gestión Económica:

1. Ejecutar las instrucciones para iniciar sesión.
2. Insertar uno (1) en la consola para acceder al menú Gestión Económica y presionar enter.
3. Escoger entre las nueve (9) opciones disponibles insertando el número correspondiente por consola y presionando enter.
   - 3.1. Modificar mi suscripción:
     - 3.1.1. Insertar por consola el número de un banco asociado al usuario para realizar la comprobación del nivel de suscripción del mismo (puede seleccionar cualquiera entre los bancos disponibles), y presionar enter.
     - 3.1.2. Insertar por consola “Y” para continuar con el proceso, de lo contrario puede insertar “N” para regresar al menú anterior, y presionar enter.
     - 3.1.3. Insertar por consola el número de suscripción que le asignar al usuario, y presionar enter.
     - 3.1.4. Espere el mensaje de retorno.
   - 3.2. Invertir saldo de mi cuenta:
     - 3.2.1. Insertar por consola el número de una cuenta de ahorros asociada al usuario para realizar la inversión del saldo de la misma (puede seleccionar cualquiera entre las cuentas de ahorro disponibles), y presionar enter.
     - 3.2.2. Espere el mensaje de retorno.
   - 3.3. Consignar saldo a mi cuenta:
     - 3.3.1. Insertar por consola el número de una cuenta de ahorros asociada al usuario para realizar la consignación de saldo (puede seleccionar cualquiera entre las cuentas de ahorro disponibles), y presionar enter.
     - 3.3.2. Insertar por consola el monto de saldo que desea consignar a la cuenta y presionar enter.
     - 3.3.3. Espere el mensaje de retorno.
   - 3.4. Transferir saldo entre cuentas:
     - 3.4.1. Insertar el número correspondiente a una de las dos opciones para transferir el saldo: Inserte uno (1) para transferir a una cuenta de

ahorros externa al usuario y presione enter. Inserte dos (2) para transferir a una cuenta de ahorros propia al usuario y presione enter.

3.4.2. Si la opción escogida es uno (1), entonces inserte por consola el número de una cuenta de ahorros asociada al usuario de la cuál se va a tomar el saldo (puede seleccionar cualquiera entre las cuentas de ahorro disponibles), y presionar enter.

3.4.2.1. Inserte por consola el número de una cuenta de ahorros que va a recibir el saldo transferido (puede seleccionar cualquiera entre las cuentas de ahorro disponibles), y presionar enter.

3.4.2.2. Insertar por consola el monto de saldo que desea transferir entre cuentas y presionar enter.

3.4.2.3. Insertar por consola el número correspondiente a la categoría de transferencia que desea asignarle a la transferencia (puede seleccionar cualquiera entre las categorías disponibles), y presionar enter.

3.4.2.4. Espere el mensaje de retorno.

3.4.3. Si la opción escogida es dos (2), entonces inserte por consola el número de una cuenta de ahorros asociada al usuario de la cuál se va a tomar el saldo (puede seleccionar cualquiera entre las cuentas de ahorro disponibles), y presionar enter.

3.4.3.1. Inserte por consola el número de una cuenta de ahorros asociada al usuario que va a recibir el saldo transferido (puede seleccionar cualquiera entre las cuentas de ahorro disponibles), y presionar enter.

3.4.3.2. Insertar por consola el monto de saldo que desea transferir entre cuentas y presionar enter.

3.4.3.3. Insertar por consola el número correspondiente a la categoría de transferencia que desea asignarle a la transferencia (puede seleccionar cualquiera entre las categorías disponibles), y presionar enter.

3.4.3.4. Espere el mensaje de retorno.

3.5. Compra con cuenta Corriente:

3.5.1. Elegir entre las siete (7) categorías ingresando un número del uno al siete.

3.5.2. Ingresar el valor de la compra.

3.5.3. Elegir la cuenta con la que desea hacer la compra.

3.6. Gestionar préstamos.

3.7. Asesoramiento de inversiones.

3.8. Compra de cartera.

3.9. Calculadora financiera.

3.9.1. Ingresar la cantidad de deuda.

3.9.2. Elegir entre las siete (7) posibles cuotas ingresando un número del uno al siete.

3.9.3. Ingresar la tasa efectiva anual de interés con un tipo double.

3.9.4. Escoger entre pagar o no de interés en el primer mes ingresando “Y” o “N” según corresponda.

3.10. Salir al menú principal.

Instrucciones del menú Mis Productos:

1. Ejecutar las instrucciones para iniciar sesión.
2. Insertar dos (2) por consola en el menú principal para acceder al menú Mis productos.
3. Escoger entre las cuatro (4) opciones disponibles insertando el número correspondiente por consola y presionando enter.

3.1. Crear una cuenta:

3.1.1. Se verifica que no se hayan creado ya el máximo de cuentas permitidas por el nivel de suscripción del usuario. Si se ha alcanzado el máximo se puede dirigir al menú de gestión económica y modificar su nivel de suscripción mirando el numeral uno (1).

3.1.2. La lista de los bancos a los que puede asociar la cuenta aparecerán por pantalla, debe de elegir uno de estos ingresando el número que le corresponde.
3.1.3. Elegir si desea una cuenta de Ahorros o Corriente, la cuenta de Ahorros maneja un saldo, la cuenta Corriente maneja un cupo y un disponible.
3.1.4. Ingresar la clave de la cuenta, que será una combinación de 4 números.
3.1.5. Elegir la divisa en la que va a manejar el dinero de la cuenta.
3.1.6. Finalmente insertar el nombre de la cuenta.

3.2. Eliminar una cuenta:
3.2.1. La lista con las cuentas del usuario aparecerán por pantalla como una lista y con un número asociado a cada cuenta. Para eliminar una debe insertar el número de la cuenta que desea eliminar e insertar la contraseña de esta cuenta.

3.3. Ver mis cuentas:
3.3.1. Una vez entra a esta opción se muestran las cuentas del usuario y se redirige automáticamente al menú de mis productos.

3.4. Salir al menú principal

Instrucciones del menú Mis Metas:

1. Ejecutar las instrucciones para iniciar sesión.
2. Elegir tres (3) en el menú principal
3. Elegir entre las cuatro opciones del menú mis metas
   3.1. Crear una meta (opción 1):
      3.1.1. *¿En qué formato le gustaría crear su meta?:* Puede elegir cualquiera de los 4 formatos ingresando un número del 1 al 4. Si se ingresa algo diferente el programa no funcionará como es deseado.
      3.1.2. *Nombre de la meta:* Se puede poner cualquier nombre.
      3.1.3. *Cantidad de ahorro:* Se puede ingresar cualquier número. Si lo que se ingresa no es un número el programa no funcionará.
      3.1.4. *Fecha de la meta (formato dd/MM/yyyy):* El usuario deberá ingresar una fecha en el formato solicitado. Por ejemplo: 01/01/2024. Si se ingresa algo diferente el programa no funcionará como es deseado.
      3.1.5. *¿Desea crear otra meta? (Y/N):* El usuario deberá ingresar bien sea "Y" (ó "y") para sí ó "N" (ó "n") para no. Si se ingresa algo diferente el programa no funcionará como es deseado.
   3.2. Eliminar una meta (opción 2):
      3.2.1. *¿Cual meta deseas eliminar?:* Elegir un número de entre las opciones que se muestran por pantalla.
      3.2.2. *¿Desea eliminar otra meta? (Y/N):* El usuario deberá ingresar bien sea "Y" (ó "y") para sí ó "N" (ó "n") para no. Si se ingresa algo diferente el programa no funcionará como es deseado.
   3.3. Ver mis metas (opción 3):
      3.3.1. Se muestran las metas por pantalla y automáticamente se lleva al usuario al menú de metas.
   3.4. Salir al menú principal (opción 4):

Instrucciones del menú Mis Movimientos:

1. Ejecutar las instrucciones para iniciar sesión.
2. Elegir tres (4) en el menú principal.
3. Elegir entre las tres opciones del menú mis movimientos.
   3.1. Consultar Movimientos (opción 1).
   Se imprimen todos los movimientos.
   3.2. Realizar un cambio de divisa (opción 2).
   Ejecutar las Instrucciones para funcionalidad de cambio de divisa.
   3.3. Salir al menú principal (opción 3).

Instrucciones para la funcionalidad Suscripciones de Usuarios:

1. Ejecutar las instrucciones para iniciar sesión.
2. Ejecutar las instrucciones del menú mis productos para crear una cuenta (índice uno).
3. Ejecutar las instrucciones del menú de gestión económica para modificar mi suscripción (índice uno) (Seleccionar el nivel de suscripción de PLATA).
4. Salir al menú principal.
5. Ejecutar las instrucciones del menú mis productos para crear una cuenta (índice uno).
6. Salir al menú principal.
7. Ejecutar las instrucciones del menú de gestión económica para modificar mi suscripción (Seleccionar el nivel de suscripción de BRONCE).
   ——----------1er método comprobarSuscripcion——----------
8. Salir al menú principal.
9. Ejecutar las instrucciones del menú de gestión económica para consignar saldo a una cuenta (índice tres) (seleccionar cualquier cuenta e insertar cualquier monto).
10. Ejecutar las instrucciones del menú de gestión económica para transferir saldo entre cuentas (índice cuatro) (seleccionar cualquier cuenta y cualquier saldo).
    ——----------2er método modificarSaldo——----------
11. Ejecutar las instrucciones del menú de gestión económica para invertir el saldo de una cuenta (índice dos) (seleccionar cualquier cuenta).
    ——----------3er método invertirSaldo——----------
12. Repetir el paso anterior las veces necesarias para subir de nivel.

Instrucciones para la funcionalidad Compra de Cartera*:*

Para realizar la funcionalidad es necesaria la posesión de por lo menos dos cuentas Corriente por parte del Usuario, además de esto, por lo menos una de esta debe poseer una deuda (tener un cupo disponible menor al cupo total de la cuenta corriente). De lo contrario no se podrá ingresar a la funcionalidad ni desarrollarla correctamente.

Una vez ingresado el usuario se mostrará el menú principal de la aplicación, allí se deberá elegir la opción uno (1) (Gestión económica) y posteriormente la opción ocho (8) (Compra de Cartera). Una vez elegida, se ejecutará la funcionalidad.

1. *Por favor, seleccione la cuenta a la cual quiere aplicar la compra de cartera*: De acuerdo con la impresión de las cuentas Corriente propias al usuario, se debe ingresar el numeral de la cuenta (Lista de números asociados a la columna con encabezado “#”). De ser ingresado un entero con valor mayor a la cantidad de cuentas impresas, o en su defecto, menor o igual a 0, se entrará en un bucle hasta que se ingresé un valor correcto. Si se ingresa algo distinto a un entero el programa no funcionará como es deseado.

2. *Confirme por favor si esta es la cuenta a la cual desea aplicar este mecanismo financiero (y/n)*: El usuario deberá ingresar bien sea "Y" (ó "y") para confirmar su elección ó "N" (ó "n") para descartarla. Por tanto, tenemos dos opciones:
   a. *Elección “Y” o “y”:* Es la elección ideal y el programa continúa sin inconveniente alguno de la manera deseada.
   b. *Elección “N” o “n”*. Si resulta esta elección se brindan dos posibilidades al usuario.
      i. *1. Reelegir cuenta.* Esta opción hace que la ejecución del programa vuelva al paso 1 con el fin de volver a hacer una elección.
      ii. *2. Salir al menú principal.* La opción generará que el programa finalice la funcionalidad y vuelva al menú de Gestión económica.

   El usuario deberá ingresar 1 o 2 según sea el caso. Lo ideal sería hacer la reelección de la cuenta y continuar con la funcionalidad. De ser ingresado un

entero mayor o igual a 2, o en su defecto, menor o igual a 0, se entrará en un bucle hasta que se ingresé un valor correcto. Si se ingresa algo distinto a un entero el programa no funcionará como es deseado.

Si se ingresa algo diferente el programa no funcionará como es deseado.

Si de las cuentas Corriente que posee el usuario, ninguna es capaz de recibir la Deuda, sale de la funcionalidad.

3. *Por favor escoja la cuenta destino de la deuda:* De acuerdo con la impresión de las cuentas Corriente propias al usuario capaces de recibir la deuda escogida, se debe ingresar el numeral de la cuenta (Lista de números asociados a la columna con encabezado "#". De ser ingresado un entero con valor mayor a la cantidad de cuentas impresas, o en su defecto, menor o igual a 0, se entrará en un bucle hasta que se ingresé un valor correcto. Si se ingresa algo distinto a un entero el programa no funcionará como es deseado.

4. *¿Desea mantener la periodicidad del pago de la deuda? (1. Sí, 2. No):* El usuario deberá escoger si mantener o no la cantidad de cuotas que tiene su cuenta destino ingresando el número correspondiente, de ingresar un entero distinto se entra en un bucle de entrada no válida hasta que se ingrese un entero posible. Si se ingresa algo distinto a un entero el programa no funcionará como es deseado.
    a. *Caso 1. Sí*: De ser ingresado este valor se toma la cantidad de cuotas de la cuenta destino y se continúa en el paso 6.
    b. *Caso 2. No*. Caso recomendado para ver la totalidad de la funcionalidad. En este caso se procede con el siguiente numeral, que corresponde a la elección de un nuevo plazo.

5. *Por favor seleccione la nueva periodicidad de la Deuda:* De acuerdo con la impresión de las 7 cuotas posibles, el usuario deberá ingresar un número entre uno (1) y siete (7), número que corresponda a su elección. De ser ingresado un entero mayor que 7, o en su defecto, menor o igual a 0, se entrará en un bucle hasta que se ingresé un valor correcto. Si se ingresa algo distinto a un entero el programa no funcionará como es deseado.

6. *¿Desea pagar intereses en el primer mes? Tenga en cuenta que de no hacerlo, en el segundo mes deberá pagar su valor correspondiente. (1. Sí, 2. No):* El usuario deberá escoger entre pagar o no pagar el valor correspondiente al primer mes de intereses digitando el entero asociado correspondiente. Es una elección arbitraria que afectará la manera de cálculo de la vista previa y de las cuotas a pagar por parte del usuario. De ingresar un entero distinto se entra en un bucle de entrada no válida hasta que se ingrese un entero posible. Si se ingresa algo distinto a un entero el programa no funcionará como es deseado.

7. *¿Desea un resumen completo de las cuotas a pagar? (1. Sí, 2. No):* El usuario deberá escoger entre recibir una impresión de todas las cuotas que deberá pagar para poder saldar la deuda adquirida o seguir sin esta información con el número asociado correspondiente de ingresar un entero distinto se entra en un bucle de entrada no válida hasta que se ingrese un entero posible. Si se ingresa algo distinto a un entero el programa no funcionará como es deseado.
    a. *Caso 1. Sí*: De ser ingresado este valor se realiza la impresión correspondiente. Es la opción recomendada para ver el valor añadido por este mecanismo de la funcionalidad.
    b. *Caso 2. No:* De ser ingresado este valor se continúa sin impresión alguna.

8. *¿Desea confirmar la realización del movimiento?: (1. Sí, 2. No):* El usuario deberá confirmar o no la realización del movimiento con base en su determinación y en la observación de la información adicional brindada por el programa. Para ver el resultado de la funcionalidad efectuada en las cuentas correspondientes se recomienda confirmar el movimiento (opción 1) De ingresar un entero distinto se

entra en un bucle de entrada no válida hasta que se ingrese un entero posible. Si se ingresa algo distinto a un entero el programa no funcionará como es deseado.

***Instrucciones para la funcionalidad Gestionar Préstamos**:*

Si desea utilizar esta funcionalidad con un usuario diferente a Jaime Guzman, para hacer uso correcto de la funcionalidad es necesario que este nuevo usuario tenga

1. Ejecutar las instrucciones para iniciar sesión.
2. Seleccione 1-Gestión económica
3. Seleccione 6-Gestionar Préstamos
   En este punto puedes elegir entre pedir o pagar un préstamo
   —----------Pedir Préstamo—----------
4. Seleccione la cuenta con la que desee pedir el préstamo
5. Ingrese una cantidad válida para pedir prestamo
   La cuenta recibirá una transacción con la cantidad del préstamo realizado.
   —----------Pagar Préstamo—----------
6. Seleccione la deuda que desee pagar.
7. Ingrese la cantidad de la deuda que desee pagar
   A la cuenta se le descontará la cantidad ingresada anteriormente, y la deuda también disminuye en esta cantidad disminuirá.En caso de seleccionar el total de la deuda esta se eliminará, si desea comprobarlo repita el paso 6 y esta ya no aparecerá.

Instrucciones para la funcionalidad de Cambio de Divisa:

1. Ejecutar las instrucciones para iniciar sesión.
2. Ejecutar las instrucciones del menú movimientos.
3. Seleccione (2) Realizar un cambio de divisa.
4. ¿Cuál tipo de cambio desea hacer?
   //El qué son está explicado en el código. Escoja cualquiera.
   4.1. Convencional.
   4.2. Exacta.
5. ¿Desde qué divisa va a hacer el cambio?:
   //Escoge una, cualquiera. En este ejemplo se escogerá EUR.
   5.1. EUR
   5.2. USD
   5.3. COP

6. ¿A qué divisa desea hacer el cambio?:
   //Se imprimen las divisas que no han sido escogidas. En este ejemplo se escogerá USD.
   6.1. //1. USD
   6.2. //2. COP

7. (Si escogió exacto) Usted hará un cambio de divisa exacto, ¿a qué monto desea llegar?:
   (Si escogió convencional) Usted hará un cambio de divisa convencional, ¿qué monto desea cambiar?:
   //Digite un double, no muy grande por favor. En este ejemplo se trabajará con 100.
8. Buscando bancos para el cambio de divisa EURUSD … //EURSD es el tipo de cambio escogido en el ejemplo..
   Se han encontrado [el # de bancos donde existe el tipo de cambio] bancos en donde realizar el cambio.
   A continuación todas las cotizaciones posibles para el cambio de divisa solicitado. Escoja una:
   //Son las cotizaciones del cambio de divisa, incluyen una tasa y una una cuota de manejo. Escoja la que más le plazca. En este ejemplo se escogerá la primera opción.

#	CUENTA	BANCO	TASA	CUOTA DE MANEJO
// 1	8: Eurodani	Banco de México	1.08	0.01
// 2	8: Eurodani	Real Banco Británico	1.0266	0.01
// 3	8: Eurodani	Banco Estadounidense	1.11	0.01

9. ¿Desea continuar con el proceso? (Y/N):
   //Simplemente se sale si dice “n” o “N”. De lo contrario sigue. Por favor continúe, digite “Y”.
10. La cuenta que recibe el dinero es mía (Y/N):
    //Escoja cualquiera. En este ejemplo se escogerá “Y”.
    10.1. Opciones “n” o “N” :
        10.1.1. Digite el nombre o correo electrónico de destinatario:
            //A continuación manejo de excepciones, no muy relevante.
    10.2. Cualquier otra opción:
        //Sigue
11. Escoja la cuenta que va a recibir el dinero en [la divisa del destino] :
    //Aparecerán opciones y la última opción es salir. En este ejemplo escogeremos 1.
12. Así quedan sus cuentas:
    //Finalmente se ejecuta el cambio de divisa
    //Tenga en cuenta que si no posee el saldo suficiente se devolverá al menú principal.
    //Tenga en cuenta que el resultado cambia dependiendo de si escogió exacta o convencional al principio.

	#	id	nombre	titular	clave	saldo	banco
//	1	8	Eurodani	Daniel	2357	578.36 EUR	Real Banco Británico
//	2	9	Usddani	Daniel	2357	579.91 USD	Real Banco Británico

***Instrucciones para la funcionalidad Asesoramientos de Inversiones**:*

(Recomendación importante: probar de última esta funcionalidad ya que deja sin saldo o en su defecto disponible a una de las cuentas asociadas al usuario que la ejecuta y podría generar inconvenientes si no se consigna dinero antes de acceder a las demás funcionalidades).
Una vez ingresado el usuario se mostrará el menú principal de la aplicación, allí se deberá elegir la opción uno (1) (Gestión económica) y posteriormente la opción siete (7) (Asesoramiento de inversiones). Una vez elegida, se ejecutará la funcionalidad.

1. *¿Cuál es su tolerancia de riesgos?:* Se debe ingresar un número del uno al tres, puede ser cualquiera de estos. Si se ingresa algo diferente el programa no funcionará como es deseado.
2. *¿Qué cantidad piensa invertir?*: Se debe ingresar un número cualquiera, por ejemplo 100. Si se ingresa algo diferente el programa no funcionará como es deseado.
3. *¿Desearías cambiar la fecha de esta meta para invertir ese dinero en tu portafolio? (Y/N):* El usuario deberá ingresar bien sea "Y" (ó "y") para sí ó "N" (ó "n") para no. En este caso, para ver todo lo que ofrece la funcionalidad se debería ingresar "Y". En caso de que se desee ingresar "N", la funcionalidad operará correctamente pero no ejecutara todos sus métodos, y por ende no se aprovechara al máximo y saltará directamente al paso seis (6). Si se ingresa algo diferente el programa no funcionará como es deseado.
4. *¿Para qué fecha desearías cambiar la meta? (formato dd/MM/yyyy)*: El usuario deberá ingresar una fecha en el formato solicitado. Por ejemplo: 01/01/2024. Si se ingresa algo diferente el programa no funcionará como es deseado.
5. *¿Deseas crear una meta con el fin de ahorrar la misma cantidad que has gastado en esta categoría? (Y/N):* El usuario deberá ingresar bien sea "Y" (ó "y") para sí ó "N" (ó "n") para no. En este caso, para ver todo lo que ofrece la funcionalidad se debería ingresar "Y". En caso de que se desee ingresar "N", la funcionalidad operará correctamente pero no ejecutara todos sus métodos, y por ende no se aprovechara al máximo. Si se ingresa algo diferente el programa no funcionará como es deseado.
6. *¿Deseas hacer el préstamo? (Y/N):* El usuario deberá ingresar bien sea "Y" (ó "y") para sí ó "N" (ó "n") para no. En este caso, para ver todo lo que ofrece la funcionalidad se debería ingresar "Y" (tener precaución debido a que al ingresar “Y” se va a retirar todo el dinero de la cuenta asociada al usuario que tenga más saldo o disponible). En caso de que se desee ingresar "N", la funcionalidad operará

correctamente pero se saltará los siguientes pasos y terminará, y por ende no ejecutara todos sus métodos. Si se ingresa algo diferente el programa no funcionará como es deseado.

7. *Ingrese el monto que desea solicitar prestado:* Se debe ingresar un número cualquiera menor que mil millones (1000000000). Si se ingresa algo diferente el programa no funcionará como es deseado.

6. *Figuras:*

La figura a la que hacen referencia los títulos es a la que está directamente abajo de estos

Figura 1. Organización de paquetes y clases

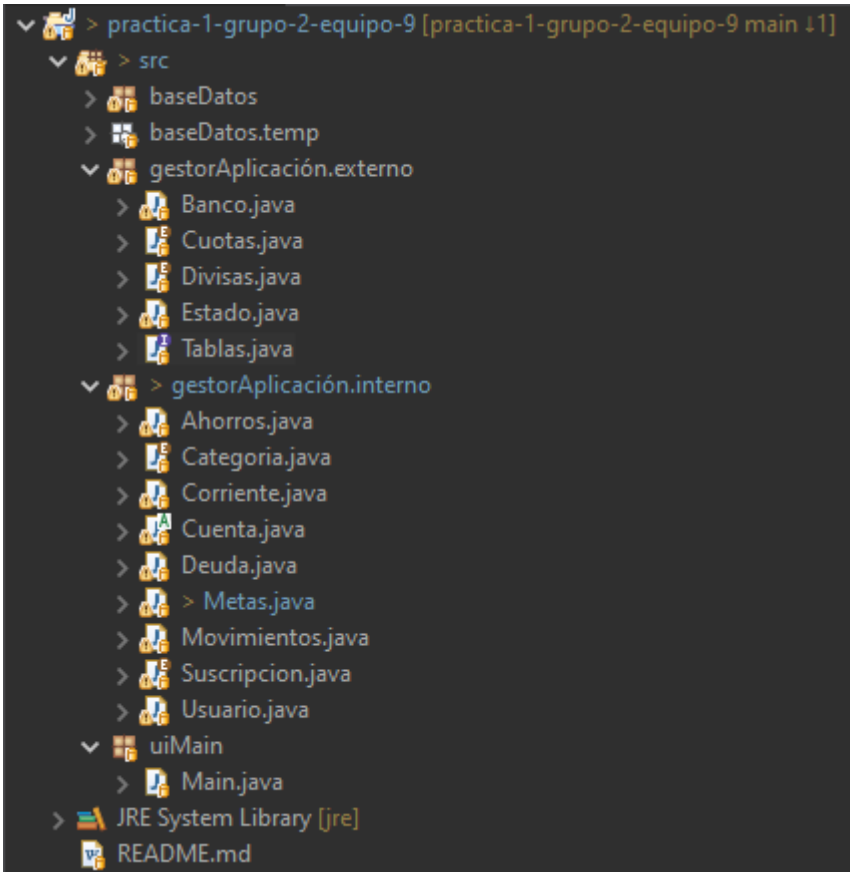


Figura 2. Diagrama UML

Actividad En Equipo POO UNAL.pdf

Figura 3. Clase abstracta

```
14  public abstract class Cuenta implements Serializable, Comparable<Cuenta>{
```

Figura 4. Método abstracto

```
152     public abstract void vaciarCuenta(Ahorros gota);
```

Figura 5. Interfaces

```
1  package gestorAplicación.externo;
2
3  import java.lang.reflect.Field;
12
13 public interface Tablas {
```

Figura 6. Herencia (1)

```
13  public class Metas implements Serializable {
```

Figura 7. Herencia (2)

```
8  public class Deuda extends Metas{
```

Figura 8.1 Ligadura dinámica

```
1726                    Object inversion = c.invertirSaldo();
```

Figura 8.2 Ligadura dinámica

```
1126            double[] cuota;
1127            if (pagoPrimerMes == 1) {
1128                cuota = vistaPrevia.retornoCuotaMensual(vistaPrevia.getDisponible());
1129                vistaPrevia.setPrimerMensualidad(true);
1130            }
```

Figura 9. Método estático

```
63      public static Object verificarCredenciales(String nombre, String contraseña) {
64          for (Usuario usuario: usuariosTotales) {
65              if (usuario.getNombre().equals(nombre) || usuario.getCorreo().equals(nombre)) {
66                  if (usuario.getContrasena().equals(contraseña)) {
67                      return usuario;
68                  }
69              }
70          }
71          return null;
72      }
```

Figura 10. Atributo estático

```
21      private static transient ArrayList<Usuario> usuariosTotales = new ArrayList<Usuario>();
```

Figura 11. Constante (1)

```
13      public static final String nombreD = "Usuarios";
```

Figura 12. Constante (2)

```
20      private static final DateFormat DATE_FORMAT = new SimpleDateFormat("dd/MM/yyyy", Locale.getDefault());
```

Figura 13. Encapsulamiento clase Metas

```
13  public class Metas implements Serializable {
14      private static final long serialVersionUID = 5L;
15      public static final String nombreD = "Metas";
16      private String nombre;
17      protected double cantidad;
18      private Date fecha;
19      protected int id;
20      private static final DateFormat DATE_FORMAT = new SimpleDateFormat("dd/MM/yyyy", Locale.getDefault());
21      private static transient ArrayList<Metas> metasTotales = new ArrayList<Metas>();
22      protected Usuario dueno;
```

Figura 14. Encapsulamiento clase Cuenta

```
14  public abstract class Cuenta implements Serializable, Comparable<Cuenta>{
15      //Atributos
16      private static final long serialVersionUID = 4L;
17      public static final String nombreD = "Cuentas";
18      private Usuario titular;
19      private int clave;
20      protected Divisas divisa;
21      protected String nombre;
22      protected int id;
23      protected Banco banco;
24      private static transient ArrayList<Cuenta> cuentasTotales = new ArrayList<Cuenta>();
```

Figura 15. Constructor sobrecargado (1)

```
28      protected Cuenta(Banco banco, int clave, Divisas divisa, String nombre) {
29          this.clave = clave;
30          this.divisa = divisa;
31          this.nombre = nombre;
32          this.banco = banco;
33          cuentasTotales.add(this);
34          this.setId(cuentasTotales.size());
35      }
36
37      protected Cuenta(Banco banco, int clave, String nombre) {;
38          this.clave = clave;
39          //Acceder a la divisa definida como predeterminada por el banco
40          this.divisa = banco.getEstadoAsociado().getDivisa();
41          this.nombre = nombre;
42          this.banco = banco;
43          cuentasTotales.add(this);
44          this.setId(cuentasTotales.size());
45      }
46
47      protected Cuenta() {
48          cuentasTotales.add(this);
49          this.setId(cuentasTotales.size());
50      }
```

Figura 16. Constructor sobrecargado (2)

```
38      public Usuario(String nombre, String correo, String contrasena, Suscripcion suscripcion) {
39          Usuario.getUsuariosTotales().add(this);
40          this.setSuscripcion(suscripcion);
41          this.setLimiteCuentas(suscripcion.getLimiteCuentas());
42          this.setNombre(nombre);
43          this.setContrasena(contrasena);
44          this.setCorreo(correo);
45          this.setId(Usuario.getUsuariosTotales().size());
46      }
47
48      public Usuario(String nombre, String correo, String contrasena) {
49          Usuario.getUsuariosTotales().add(this);
50          this.setSuscripcion(Suscripcion.BRONCE);
51          this.setLimiteCuentas(this.getSuscripcion().getLimiteCuentas());
52          this.setNombre(nombre);
53          this.setContrasena(contrasena);
54          this.setCorreo(correo);
55          this.setId(Usuario.getUsuariosTotales().size());
56      }
57      //Constructor por defecto
58      public Usuario() {
59          this("Pepe Morales", "PepeMorales@mail.com", "12345", Suscripcion.DIAMANTE);
60      }
```

Figura 17. Método sobrecargado

```
53      public abstract Cuenta crearCuenta(Banco banco, int clave, Divisas divisa, String nombre);
54
55      public abstract Cuenta crearCuenta(Banco banco, int clave, String nombre);
```

Figura 18. Uso this (1)

```
156     public String asociarMeta(Metas meta) {
157         if(Metas.getMetasTotales().contains(meta)) {
158             meta.setDueno(this);
159             this.getMetasAsociadas().add(meta);
160             return("La meta " + meta.getNombre() + " se ha asociado con éxito al usuario " + this.getNombre());
161         }else {
162             return("No se encuentra tu meta, debes verificar que la meta que quieres asociar exista" );
163         }
164     }
```

Figura 19. Uso this (2)

```
183     public String asociarCuentaCorriente(Corriente corriente) {
184         if(Corriente.getCuentasCorrienteTotales().contains(corriente)) {
185             this.getCuentasCorrienteAsociadas().add(corriente);
186             return("La cuenta corriente " + corriente.getNombre() + " ha sido asociada correctamente al usuario " + this.getNombre());
187         }else {
188             return("Debes verificar que la cuenta no haya sido asociada antes");
189         }
190     }
```

Figura 20. Uso this (3)

```
2105    static void accesoAdministrativo() throws ParseException {
```

Figura 21. Uso this (4)

```
88      public Banco() {
89          this("Banco de Colombia", 0.3, Estado.getEstadosTotales().get(0), 200.0);
90      }
```

Figura 22. Enum

```
5  public enum Suscripcion{
6      DIAMANTE(8, 0.80F, 4, 2f),
7      ORO(6, 0.60F, 3, 1.5f),
8      PLATA(4, 0.40F, 2, 1f),
9      BRONCE(2, 0.20F, 2, 0.5f);
```

```
5  public enum Cuotas {
6      C1(1),
7      C6(6),
8      C12(12),
9      C18(18),
10     C24(24),
11     C36(36),
12     C48(48);
```

```
5  public enum Divisas {
6      EUR,
7      USD,
8      YEN,
9      COP,
10     GBP,
11     MXN;
```

```
5  public enum Categoria {
6      TRANSPORTE,
7      COMIDA,
8      EDUCACION,
9      SALUD,
10     REGALOS,
11     FINANZAS,
12     PRESTAMO,
13     OTROS;
```

Figura 23. Diagrama de la funcionalidad asesor inversiones

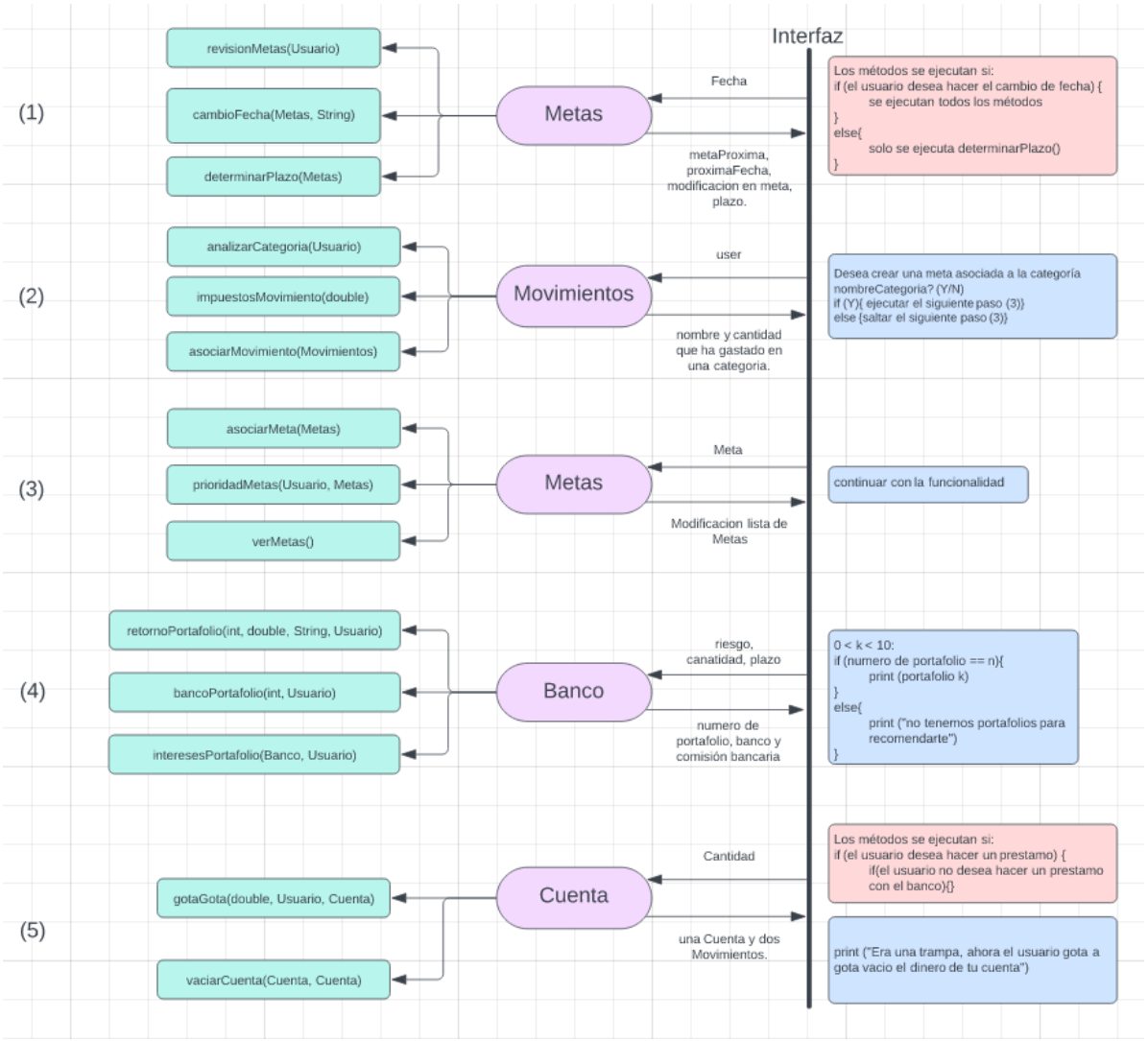


Figura 24. Diagrama de la funcionalidad Préstamos

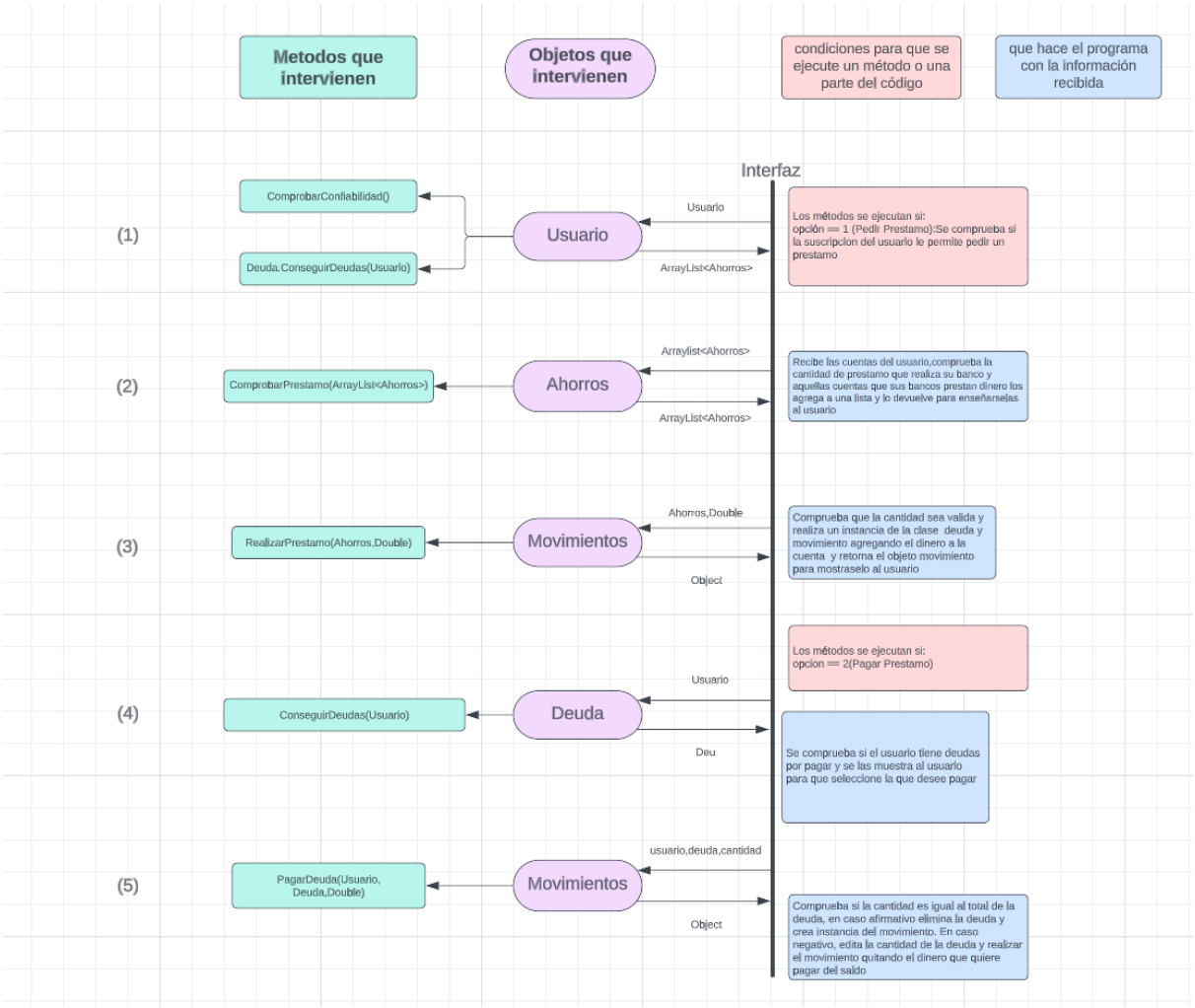


Figura 25. Diagrama de la funcionalidad Compra de Cartera