Desarrollo de Software

Práctica 3

Luis Alberto Mejía Troya
Antonio Pancorbo Morales
Manuel Jesús Junquera Lobón
18 de Mayo del 2025

UNIVERSIDAD DE GRANADA

Escuela Técnica Superior de Ingeniería Informática y de Telecomunicaciones

Índice general

Capítulo 1

Diseño

1.1. Diagrama UML y patrones de diseño

En primer lugar, tenemos una clase abstracta llamada `Transaction` que será la clase padre de la cual, heredan las operaciones bancarias. Estas operaciones son tres:

- `DepositTransaction`: Se utilizará para que una cuenta realice un deposito en dicha cuenta.
- `WithdrawalTransaction`: Esta clase se encarga de realizar un reintegro sobre el saldo actual de la cuenta.
- `TransferTransaction`: Esta clase se encarga de realizar una transferencia desde una cuenta a otra.

Tendremos una clase `Account` que especificara una cuenta concreta. Esta clase, tendrá dos métodos principales:

- `deposit`
- `withdraw`

Estos métodos nos permitirá realizar operaciones sobre la misma cuenta.

Por otro lado, tenemos una clase llamada `BankService` que será la encargada de realizar todas las acciones y tendrá una diccionario donde guardaremos todas las cuentas que vayamos creando. Además, tendremos un diccionario, donde guardaremos el listado de cada transferencia que se ha realizado sobre una cuenta concreta.

Por lo tanto, hemos implantado el método fachada en esta clase ya que es la encargada de realizar todas las operaciones necesarias. Además, hemos implementado el patrón Singleton para que solo exista una instancia de esta clase y no puedan crearse nuevas instancias.

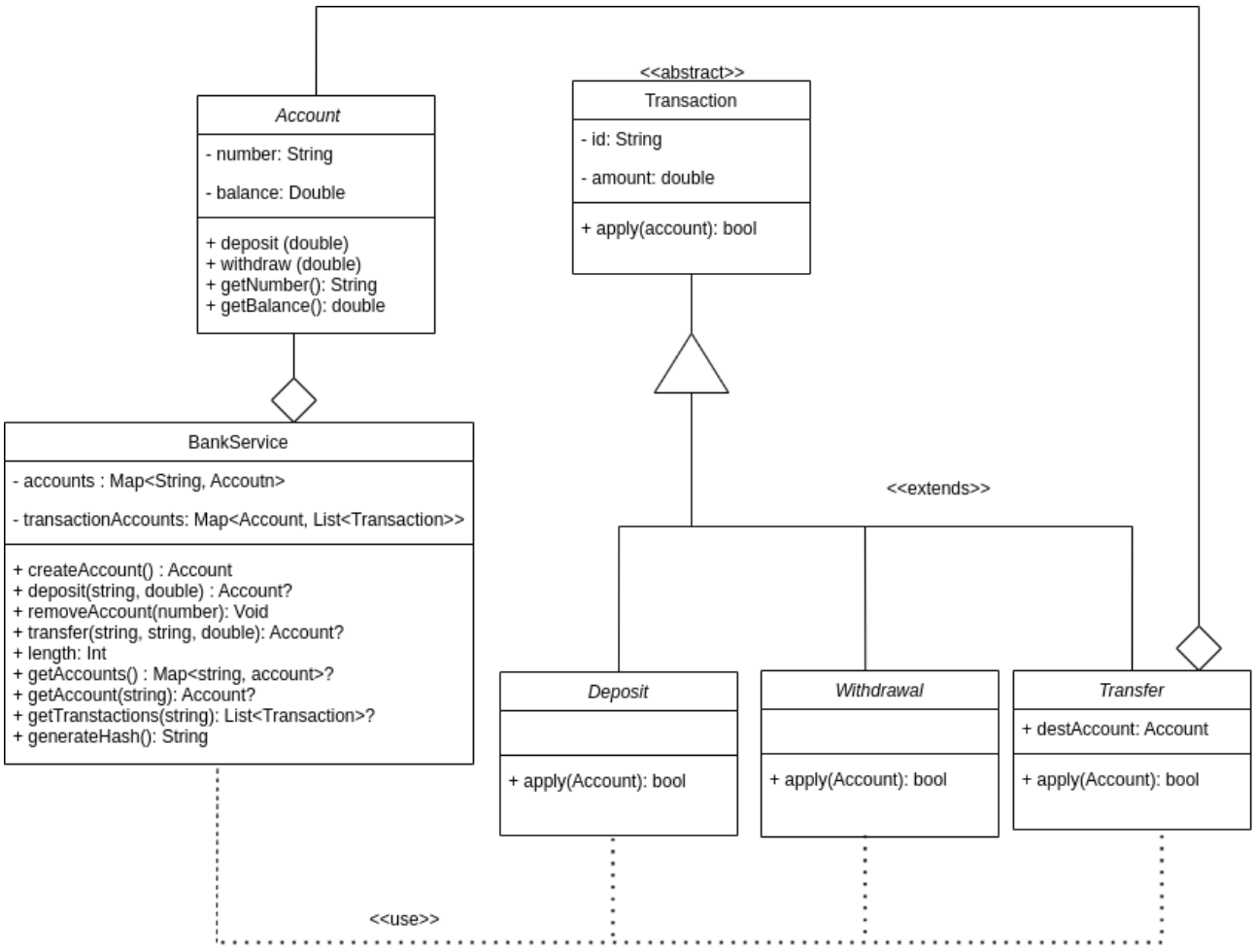

Figura 1.1: Diagrama UML

Capítulo 2

Análisis y pruebas

2.1. Introducción

Hemos realizado un proceso conocido como `TDD` para la realización del código y la implementación de las pruebas unitarias.

Esta técnica consiste en realizar primero cada test unitario y, a partir de ahí, obtener el código correspondiente a ese test. Además, una vez obtenido el resultado correcto, tendremos que refactorizar el código.

Este proceso nos ha llevado más tiempo, pero nos asegura que el código es correcto y que se cumplen los casos de uso necesarios.

2.2. Account

Los test que hemos implementado son los siguiente:

- **Balance inicial**: Se comprueba que al instanciar una cuenta, el saldo es 0. Para ello, hemos hecho uso de la función `setUp` dentro de los test para inicializar una cuenta. Por lo tanto, el atributo `balance` de la clase `Account` tiene que ser inicializado a 0. Por otro lado, tenemos que tener un método que nos devuelva el saldo de la cuenta.

A continuación, se validan los casos para el método `deposit`:

- **No se permite depositar una cantidad negativa**: Hemos creado un test donde comprobamos que el objeto de clase `Account` devuelve `false` si se intenta depositar una cantidad negativa. Por lo tanto, para la implementación del método `deposit`, si la cantidad a ingresar es una cantidad negativa, devolverá directamente el valor de `false`.
- **No se permite depositar una cantidad igual a 0**: También debe devolver `false`. Por lo que hemos implementado una condición adicional el código para que se cumple este caso de uso. Una vez implementado y comprobado que pasan ambos test hemos refactorizado el código en una única condición.
- **Se ha depositado una cantidad positiva**: Para ello, comprobamos que el saldo inicial de la cuenta es 0 y que una vez depositado una cantidad positiva, el método `deposit` nos devuelva un valor `true` y que finalmente aumente el saldo de la cuenta. Por lo tanto, en la implementación del código, si no se cumple los dos test anteriores, añdimos la cantidad solicitada a nuestro saldo actual y finalemente retornamos el valor `true`

Posteriormente, se comprueba el comportamiento del método `withdraw`:

- **No se permite retirar una cantidad negativa**: El método debe devolver `false`. Y para ello, realizamos un procedimiento similar que en el método `deposit`.

- **No se permite retirar una cantidad igual a 0**: El método debe devolver `false`.
- **No se puede retirar una cantidad mayor al saldo actual**: Se comprueba que no es posible extraer más dinero del que hay disponible. Si se intenta realizar, el método devolverá el valor `false`.
- **Se retira una cantidad válida**: Se comprueba que el reintegro es exitoso si la cantidad es menor o igual al saldo, y que este se actualiza el saldo de la cuenta.

Con test unitarios, comprobamos que los casos usos asociados a los test unitarios son correcto y es lo que se espera que haga el programa relacionado con esta parte del código.

2.3. DepositTransaction

Las pruebas unitarias realizadas sobre `DepositTransaction` nos permiten validar que las operaciones de ingreso se comportan correctamente ante distintos escenarios:

- **No se puede depositar una cantidad negativa**: Se comprueba si la cantidad introducida para realizar el depósito es negativa, en cuyo caso se lanza una excepción del tipo `StateError`.
- **Se aumenta el saldo correctamente**: Se crea una transacción con una cantidad válida, se considera que un transacción es valida si se puede realizar la operacion `withdraw` sobre la cuenta que quiere mandar dinero a la otra cuenta. Por otro lado, la cuenta que recibe el dinero se le aplica la operación `deposit` si la operación `withdraw` se ha podido llevar a cabo.

2.4. WithdrawalTransaction

Las pruebas unitarias de `WithdrawalTransaction` comprueban que las retiradas de dinero desde una cuenta se gestionan de forma segura:

- **No se puede retirar una cantidad negativa**: El método `apply` debe retornar `false` y el saldo debe quedar intacto.
- **No se puede retirar una cantidad mayor al balance total**: Se intenta realizar un reintegro de una cantidad superior al balance total de la cuenta, lo cual debe lanzar una excepción `StateError`.
- **Se realiza la retirada correctamente**: Se realiza una reintegro válido, es decir, se retira una cantidad inferior al balance total de la cuenta y se verifica que el saldo disminuye de forma adecuada.

2.5. TransferTransaction

Las pruebas sobre `TransferTransaction` aseguran que las transferencias entre cuentas se ejecutan correctamente y manejan bien los errores:

- **Se lanza excepción al intentar transferir cantidad negativa**: Una transferencia con cantidad negativa debe lanzar una excepción del tipo `StateError`.
- **Transferencia realizada correctamente**: Se comprueba que se transfiere correctamente una cantidad válida desde la cuenta origen a la cuenta destino, y que los saldos se actualizan correctamente.
- **La transferencia no se ha podido realizar por falta de fondos**: Si el saldo de la cuenta origen es insuficiente, se debe lanzar una excepción y los saldos de ambas cuentas deben mantenerse sin cambios.

2.6. BankService

La clase `BankService` actúa como fachada y orquestadora del sistema bancario, centralizando la creación de cuentas y la gestión de transacciones. Las pruebas realizadas validan su correcto funcionamiento en distintos escenarios:

- **Lista vacía**: Se comprueba que, al instanciar por primera vez el objeto, no existe ninguna cuenta creada.
- **Se intenta aumentar el saldo de una cuenta que no existe**: Si se intenta hacer un depósito a un número de cuenta inexistente, el método debe devolver `null`.
- **Aumento del saldo de una cuenta** : Se crea una cuenta, se realiza un depósito y se verifica que el saldo aumenta correctamente.
- **Retirada de dinero con saldo insuficiente**: Se espera que se lance una excepción `StateError` si se intenta retirar más dinero del disponible.
- **Retirada de dinero de forma exitosa**: Se comprueba que al realizar una retirada válida, el saldo se actualiza correctamente.
- **No existen las dos cuentas**: Al intentar realizar una transferencia entre dos cuentas inexistentes, el método debe devolver `null`.
- **No existe la cuenta fuente**: Si la cuenta de origen no existe, la transferencia no debe realizarse.
- **No existe la cuenta destino**: Si la cuenta destino no existe, la transferencia tampoco debe completarse.
- **Saldo insuficiente**: Al intentar transferir una cantidad mayor que el saldo disponible, debe lanzarse una excepción `StateError`.
- **Transacción realizada de forma exitosa**: Se prueba una transferencia válida entre dos cuentas y se comprueba que los saldos se actualizan correctamente.
- **Los identificadores de las transacciones son diferentes**: Se asegura que cada transacción (como un depósito y una retirada) genera un identificador único.

Capítulo 3

Interfaz gráfica

3.1. Introducción

Hemos realizado un proceso modular, donde hemos en componente y pantallas las diferentes interfaces para que el código quede más limpio y sea fácil de mantener.

Una vez inicializado el programa, aparece la siguiente la siguiente interfaz.

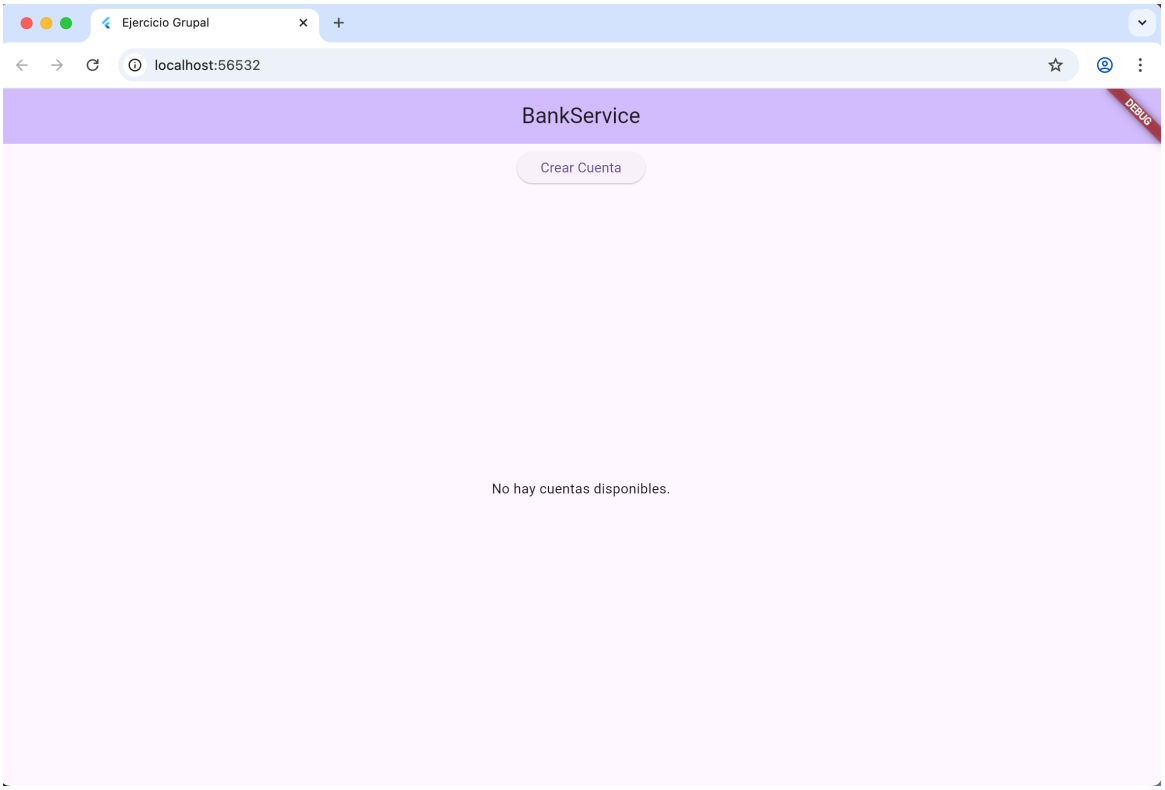

Figura 3.1: Interfaz al iniciar el programa

Una vez que le damos al botón `CrearCuenta` se creará una cuenta donde aparece el número de la cuenta asociado y a la derecha aparecerá un botón para eliminar dicha cuenta.

El saldo de la cuenta no aparece en la pantalla siguiendo el modelo de ocultación de información que se consigue con la programación orientada a objetos ya que es un atributo que debe ser privado.

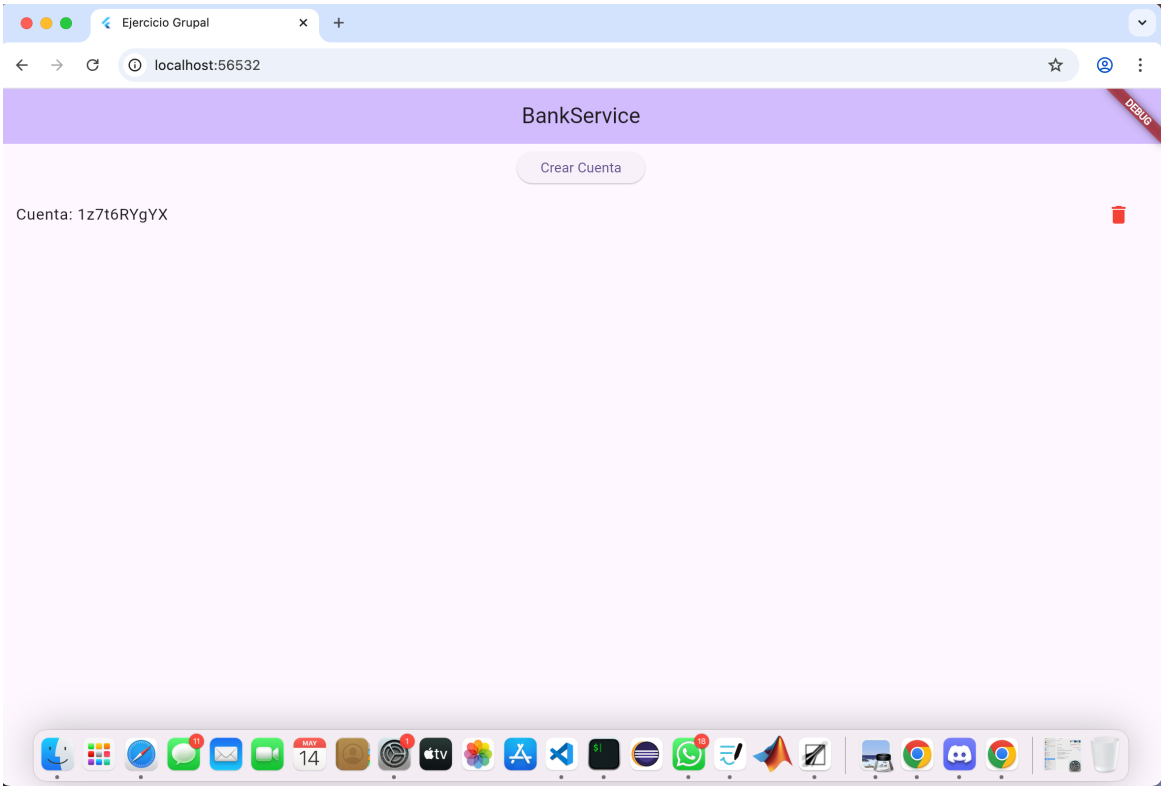

Figura 3.2: Creación de una cuenta

Si hacemos click en una cuenta nos aparecerá el saldo de esa cuenta y también las acciones que podremos realizar en dicha cuenta como `Depositar`, `Retirar` y `Hacer transacción`. Además, las últimas dos operaciones solo son válidas si tenemos saldo suficiente.

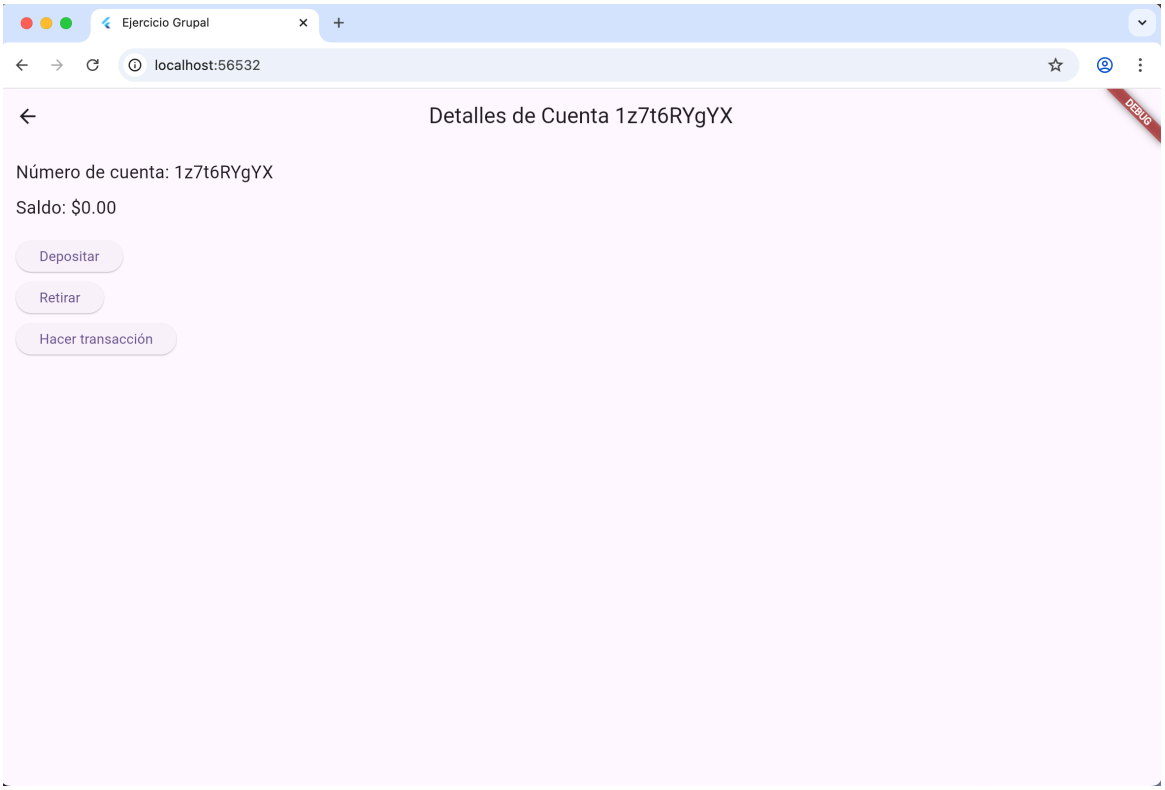

Figura 3.3: Operaciones sobre una cuenta concreta y saldo disponible disponible