Taller Lógica-Andrés Mora

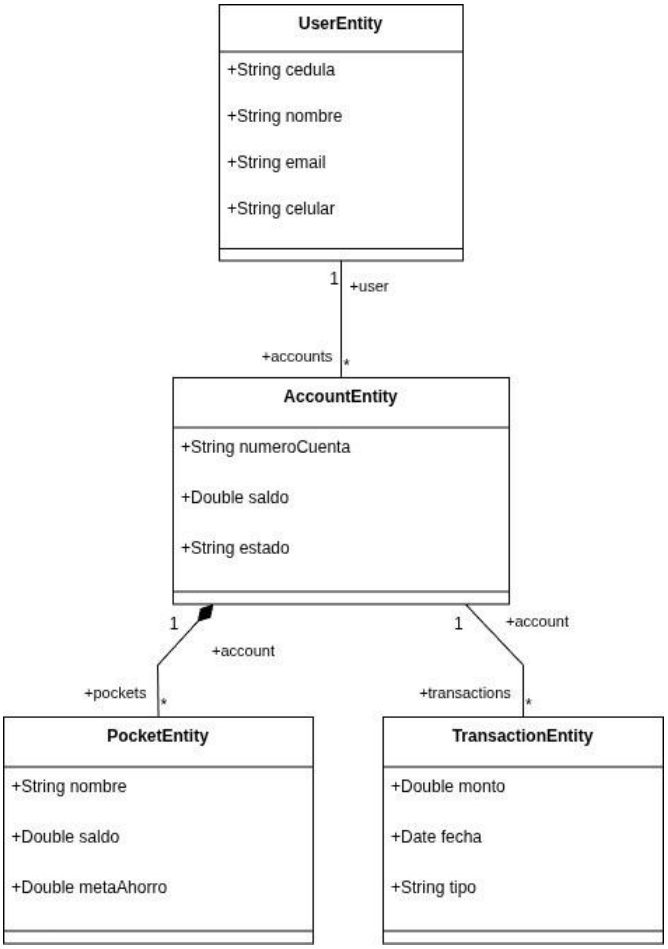
Introducción

Contexto de Negocio

NeoFin Andes es una nueva startup financiera universitaria. Su propuesta de valor es permitir a los estudiantes manejar una Cuenta Principal y dividir su dinero en Bolsillos (ahorros específicos, ej: "Para el semestre", "Para la fiesta", "Arriendo"). Además, permite realizar Transferencias entre usuarios.

A diferencia del taller de persistencia donde el objetivo era solo "guardar datos", en este taller el objetivo es implementar la capa de lógica para cumplir con las reglas de negocio y proteger la integridad de los datos (Por ejemplo, no podemos guardar una transferencia si no hay fondos, ni crear un bolsillo con un nombre duplicado).

Diagrama de Dominio



Para este taller, asumimos que la capa de persistencia (Entities y Repositories) ya existe.

1. **Usuario (UserEntity):** Datos básicos (Cédula, Nombre, Email).
2. **Cuenta (AccountEntity):**
   - Hereda de BaseEntity.
   - Atributos: numeroCuenta (String), saldo (Double), estado (String: ACTIVA/BLOQUEADA).
   - Relación: Un Usuario tiene muchas Cuentas
3. **Bolsillo (PocketEntity):**
   - Atributos: nombre (ej: "Viaje"), saldo (Double), metaAhorro (Double).
   - Relación: Una Cuenta tiene muchos Bolsillos (@OneToMany).
4. **Transaccion (TransactionEntity):**
   - Atributos: monto (Double), fecha (Date), tipo (String: ENTRADA/SALIDA).
   - Relación: Una Cuenta tiene muchas Transacciones.

Objetivos

- Implementar reglas de negocio mediante validaciones de estado y saldo en la capa de servicio.
- Definir escenarios de prueba utilizando el patrón *Given-When-Then* para cubrir flujos de éxito y error.
- Garantizar la atomicidad de las operaciones mediante el uso de @Transactional en procesos de transferencia de fondos.

Preparación

Para la realización de este taller, se utilizará como base el repositorio ISIS2603_Taller_Logica, siga los pasos presentados a continuación para hacer **Fork** y configurar su entorno de trabajo:

1. Ingrese al repositorio base en el siguiente enlace: ISIS2603_Taller_Logica.
2. Localice y presione el botón “Fork” (Bifurcar si está en español) en la esquina superior derecha del repositorio base.
3. En la configuración del nuevo repositorio, créelo con la siguiente estructura “**ISIS2603_202610_T2_<usuario_uniandes_sin_punto>**” por ejemplo: “**ISIS2603_202610_T2_afrodriguezm1**.
4. Una vez creado el fork en su cuenta personal, clone el repositorio en su máquina local.
5. Ejecute la aplicación y valide su correcta ejecución revisando el resultado por consola como se muestra a continuación:

Reglas de Negocio a Implementar

El taller consiste en crear los Servicios (AccountService, PocketService, TransactionService) que validen las siguientes reglas antes de llamar a los repositorios.

- **Regla 0 (ejemplo):** Para crear un Bolsillo, este debe estar asociado obligatoriamente a una Cuenta existente y con estado ACTIVA. Si la cuenta está en estado BLOQUEADA, no se pueden crear bolsillos. Adicionalmente, una misma cuenta no puede tener dos Bolsillos con el mismo nombre. (Ej: No puedo tener dos bolsillos llamados "Ahorro").
    - *Nota:* Dos cuentas diferentes SÍ pueden tener bolsillos con el mismo nombre.
- **Regla 1:** Para mover dinero de la Cuenta a un Bolsillo:
    1. Verificar que saldoCuenta >= monto.
    2. Restar el monto de la Cuenta.
    3. Sumar el monto al Bolsillo. *Si el saldo es insuficiente, la operación falla y no se debe guardar nada.*
- **Regla 2:** Para mover dinero entre dos cuentas
    1. Validar que la cuenta origen y destino existan.
    2. Validar que la cuenta origen no sea la misma que la destino.
    3. Validar fondos suficientes en origen.
    4. Actualizar ambos saldos atómicamente.

Actividades Sincrónicas del Taller (Clase)

Parte A: Identificación de Casos de Prueba

Antes de desarrollar el código de la lógica, es necesario identificar qué escenarios pueden ocurrir al intentar aplicar las reglas de negocio descritas anteriormente.

Los escenarios describen qué situación específica se está probando. No basta con probar que “funciona”, se deben probar los casos de éxito (donde todas las reglas se cumplen) y

los casos de error (donde alguna regla no se cumple y se lanza una excepción). Ejemplo*:* "Crear bolsillo con saldo suficiente" vs. "Crear bolsillo en cuenta bloqueada".

En la siguiente lista, identifique por nombre los casos de prueba necesarios para validar las Reglas 1 y 2. Debe considerar el camino de éxito como los casos de error (excepciones).
Para guiarse, puede tomar como ejemplo los escenarios de prueba de la Regla 0:

Escenarios Regla 0:

1. Éxito: Crear un bolsillo en una cuenta activa
2. Fallo: Intentar crear un bolsillo en una cuenta que no existe
3. Fallo: Intentar crear un bolsillo con un nombre que ya existe en esa cuenta
4. Fallo: Intentar crear un bolsillo en una cuenta bloqueada

Ahora, escriba los escenarios de éxito y fallo para las Reglas 1 y 2 que permitan validar las reglas de negocio

Escenarios Regla 1:

1. Éxito: Mover dinero desde una cuenta a un bolsillo con saldo suficiente
2. Fallo: Intentar mover el dinero cuando el saldo de la cuenta es insuficiente
3. Fallo: intentar mover dinero desde una cuenta que no existe
4. Fallo: intentar mover dinero a un bolsillo que no existe
5. Fallo: Tener un monto invalido

Escenarios Regla 2:

1. Éxito: Transferir dinero entre dos fondos existentes, son distintitas y con fondos suficientes
2. Fallo: intentar trasferir desde una cuenta de origen que no existe
3. Fallo: Intentar trasferir a una cuenta de destino que no existe
4. Fallo: Intentar transferir entre la misma cuenta
5. Fallo: intentar trasferir sin fondos suficientes en la cuenta de origen

Parte B: Definición de Casos de Prueba

Una vez identificados los nombres de los casos, debemos detallar el estado del sistema y el resultado esperado. Se utilizará un patrón estándar de pruebas muy común en el

mundo del desarrollo de software conocido como Given-When-Then (Dado que - Cuando - Entonces). Su propósito es obligar al diseñador de pruebas a pensar en qué debe pasar antes de preocuparse por cómo escribir el código.

Debemos definir los tres estados de la prueba:

Estado Inicial (Given): Define qué existe en la base de datos antes de que se ejecute el código. Es el contexto necesario para que la prueba tenga sentido.

- *Ejemplo:* Si se va a probar la regla "No se pueden crear nombres duplicados", el estado inicial obligatoriamente debe tener ya guardado un bolsillo con ese nombre en la base de datos simulada.

Acción (When): Es la ejecución concreta del método de lógica (Service). Aquí defines qué datos le estás pasando al método.

- *Ejemplo:* Llamar a `pocketService.createPocket(idCuenta, nuevoBolsillo)`.

Resultado Esperado (Then): Define cómo cambia el mundo después de la acción. Aquí es donde se verifica si la lógica funcionó. Se divide en tres verificaciones posibles:

- Retorno: ¿Qué devolvió el método? (el objeto creado).
- Persistencia: ¿Qué cambió en la base de datos? (¿El saldo bajó? ¿Apareció un nuevo registro?).
- Excepciones: Si algo salió mal, ¿el código lanzó el error correcto (ej. `BusinessLogicException`) y detuvo la operación?

A continuación, se presenta la tabla descriptiva del patrón Given-When-Then para la regla 0:

Escenario	**Estado Inicial (BD)**	**Acción (Input)**	**Resultado Esperado (Output/BD)**
Éxito: Crear un bolsillo	Existe Cuenta #123 (Saldo: 1000)	Crear Bolsillo "Ahorro" (Monto: 0) en Cuenta #123	1. Se crea el objeto Bolsillo. 2. Se guarda en la BD. 3. Retorna la entidad creada.

Fallo: Cuenta Inexistente	No existe la cuenta #999	Crear Bolsillo en Cuenta #999	Lanza Excepción: EntityNotFoundException.
Fallo: Cuenta Bloqueada	Existe Cuenta #123 con estado “BLOQUEADA”	Crear Bolsillo "Ahorro" en Cuenta #123	Lanza Excepción: BusinessLogicException ("La cuenta está bloqueada"). No guarda nada nuevo.
Fallo: Duplicado	Existe Cuenta #123 con un Bolsillo "Ahorro"	Crear Bolsillo "Ahorro" en Cuenta #123	Lanza Excepción: BusinessLogicException ("El nombre ya existe"). No guarda nada nuevo.

Ahora, complete la tabla de escenarios para las Reglas 1 y 2:

Regla 1:

Escenario	Estado Inicial (BD)	Acción (Input)	Resultado Esperado (Output/BD)
Éxito: Saldo suficiente	Existe una cuenta #123 con saldo = 1000 y un bolsillo con saldo 100	Se intenta mover 200 de la cuenta “123 al bolsillo	El saldo de la cuenta #123 queda con 800 y el bolsillo con 300
Fallo: Fondos insuficientes	Existe una cuenta #123 con saldo 50 Y un bolsillo con saldo 100	Se intenta mover 200 de la cuenta #123 a el bolsillo	La operación falla, se lanza una excepción, el saldo de la cuenta sigue siendo 50 y la del bolsillo 100

Fallo: Cuenta no existe	La cuenta #999 no existe	Se intenta mover 50 de la cuenta #999 a el bolsillo	La operación falla y se lanza una excepción de entidad no encontrada
Fallo: Bolsillo inexistente	La cuenta #123 existe, pero el bolsillo “inexistente” no existe	Se intenta mover 50	La operación falla y se lanza una excepción de entidad no encontrada
Fallo: Monto invalido	Existe una cuenta y un bolsillo	Se intenta mover 0	La operación falla y se lanza excepción de negocio

Regla 2:

Escenario	Estado Inicial (BD)	Acción (Input)	Resultado Esperado (Output/BD)
Éxito: Transferir dinero entre dos cuentas	Cuenta #123 tiene saldo de 1000 y otra cuenta #456 tiene saldo de 100 y ambas cuentas son diferentes	Se intenta trasferir 200 de la cuenta #123 a la cuenta #456	El saldo de la cuenta #123 queda en 800 y el de la cuenta #456 en 300
Fallo: Cuenta de origen no existe	La cuenta #999 no existe	Se intenta transferir 50 de la cuenta #999 a una cuenta #123	La operación falla y se lanza excepción de entidad no encontrada
Fallo: Cuenta de destino no existe	La cuenta #999 no existe	Se intenta transferir 50 de una cuenta #123 a la cuenta #999	La operación falla y se lanza excepción de entidad no encontrada

Fallo: Misma Cuenta	Existe una cuenta #123	Se intenta trasferir a si misma	La operación falla y se lanza una excepción de negocio
Fallo: Fondos insuficientes	Cuenta #123 con saldo de 50 y otra cuenta #456 con saldo 100	Se intenta transferir 200	La operación falla, se lanza excepción de negocio y ningún saldo cambia

Parte C: Implementación de Servicios

Ahora observe la implementación de la lógica en la clase PocketService.java. La capa de lógica es el "cerebro" de la aplicación; coordina el acceso a los datos y verifica que las reglas se cumplan antes de guardar nada permanentemente.

Siga la guía que se encuentra en la capa de lógica en: ¿

Actividades Asincrónicas del Taller (Casa)

Parte D: Implementación de Pruebas

Una vez implementada la lógica, no necesitamos arrancar toda la aplicación para ver si funciona. Usaremos JUnit, una librería para pruebas unitarias.

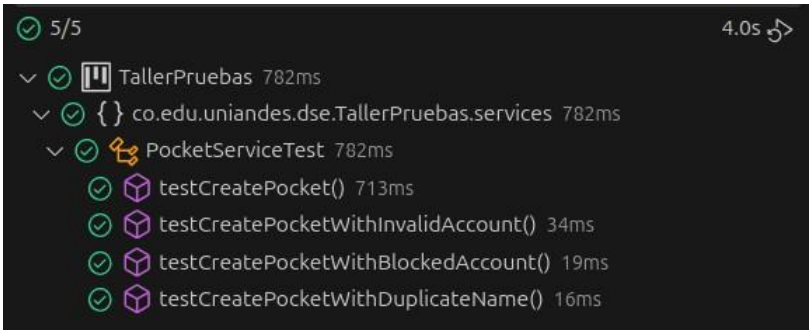
¿Cómo se verifica el código?

1. Busque la carpeta src/test/java.
2. Abra el archivo PocketServiceTest.java.

3. Notará que la clase tiene anotaciones como @DataJpaTest y @Transactional. Esto configura una base de datos en memoria temporal para la prueba, asegura que cada prueba sea independiente (limpia los datos antes de cada test usando @BeforeEach) y permite probar la lógica sin afectar la base de datos real.
4. Dé clic derecho sobre el archivo y selecciona "Run Test" (o use el botón de "Play" verde al lado de la clase).

Interpretación:

- **Verde:** La lógica cumple con los escenarios definidos
- **Rojo:** Algo falló. Revise el mensaje de error ("AssertionFailedError" o "Exception no esperada") y ajuste la lógica en el Servicio.



Nota: En las pruebas, se usa una librería llamada **Podam** (factory.manufacturePojo) para crear datos ficticios automáticamente y no tener que escribirlos a mano.

Parte E: Preguntas Conceptuales

Responda las siguientes preguntas en su documento de entrega para demostrar su comprensión teórica:

1. Sobre @Transactional: En la Regla 1, usted debe restar dinero de una cuenta y sumarlo a un bolsillo. Son dos operaciones de guardado distintas (accountRepository.save y pocketRepository.save).
    a. **Pregunta: ¿Qué pasaría si el servidor se apaga justo después de restar el dinero de la cuenta, pero antes de sumarlo al bolsillo?**
    Si la operación no estuviera protegida por una transacción, podría ocurrir un estado que sea inconsistente en la base de datos. Es decir, el dinero se restaría correctamente de la cuenta, pero como se apagaría el servidor antes de ejecutar la suma del bolsillo, ese segundo cambio no se guardaría. Lo anterior nos provocaría que el sistema pierda coherencia, ya que el dinero desaparecería de la cuenta sin llegar al bolsillo, probando así una inconsistencia.

    b. **Investigación: ¿Cómo ayuda la anotación @Transactional a evitar que ese dinero se "pierda" en el limbo? (Explique el concepto de atomicidad).**
    La anotación de transactional nos garantiza que todas las operaciones dentro del método se ejecuten dentro de una misma transacción. Esto

quiere decir que el proceso es atómico.

Ahora bien, la atomicidad es una de las propiedades que tiene ACID de las bases de datos y tiene un significado como “Todo o nada”, de esta forma, si todas las operaciones se ejecutan correctamente, la transacción hace commit y los cambios se guardan definitivamente. Por otro lado, si ocurre algún error o excepción antes de finalizar, la transacción nos hace rollback y revierte todos los cambios que se realizaron en ese método.

2. Sobre la Inyección de Dependencias:
   a. **En su código usó @Autowired para obtener el AccountRepository dentro de PocketService. Pregunta: ¿Por qué es mejor usar @Autowired en lugar de hacer AccountRepository repo = new AccountRepository() manualmente dentro del servicio? ¿Qué ventaja nos da esto a la hora de hacer pruebas?**

      Aquí es mejor usar @Autowired porque nos permite que Spring gestione automáticamente la inyección de los repositorios. Asimismo, sabemos que los repositorios en spring Data jpa no son clases normales que se instancian de manera manual, sino que son implementaciones dinámicas generadas por el framework y configuradas por el EntityManager, el datasource y el contexto de persistencia.

      De esta forma, si intentáramos crear un repositorio manualmente con un new, perderíamos toda esa configuración automática y el repositorio no funcionaría correctamente.

3. Lógica vs. Controlador:
   a. **Pregunta: Si quisiéramos validar que el monto de una transacción no sea negativo, ¿Por qué se recomienda poner ese if en la clase Service y no directamente en el Controller (API) o en la pantalla del celular (Frontend)?**
      Si queremos validar que el monto de una transacción no sea negativo, esa validación la debemos colocar en el data service y no en el controller ni en el fronted. La razón principal es que el service nos representa la capa de la lógica de negocio, mientras que el controller solo maneja la comunicación con el exterior (API). Por su parte, el fronted tampoco es confiable ya que este puede ser manipulado por el usuario.

      De esta forma, si la validación solo estuviera en el fronted o en el controlador, un usuario podría saltársela usando herramientas como Postman, y podrían existir múltiples interfaces y seria necesario duplicar la validación de cada una.

      Sin embargo, si la regla esta en service, se garantiza que siempre se cumpla, sin importar desde donde se invoque, se mantienen la consistencia del negocio y se facilita la realización de las pruebas unitarias sobre las reglas.

4. Pruebas Unitarias:
    a. **Investigación: Investigue qué es Test Driven Development (TDD) y explique sus principios fundamentales (Red, Green, Refactor). ¿Qué ventajas tiene utilizar esta metodología de desarrollo?**
    Test Driven Development es una metodología de desarrollo en la que primero escribimos las pruebas y luego el código que las hace pasar. Esta se basa en tres pasos principales:
    Red: Se escribe un test que falla porque la funcionalidad aun no está implementada.
    Green: Se implementa el mínimo código necesario para que asi el test pase.
    Refactor: Se mejora la calidad del código sin cambiar su comportamiento, asegurando que los test sigan pasando.

    Este método tiene varias ventajas eus nos permite detectar errores tempranos, tiene un diseño mucho mas limpio, se tiene mayor confianza al refactorizar y hay uan documentación implícita del comportamiento esperado.

    b. **Proponga al menos tres (3) casos de prueba adicionales, más allá de los ya descritos en la Parte A**
    Además de los escenarios ya descritos, se pueden proponer los siguientes casos:
        1. Transferir un monto extremadamente alto. Aquí nos permitiría verificar que el sistema no presente errores de desbordamiento, presión o comportamiento inesperado antes valores extremos
        2. Intentar transferir un monto negativo. Lo anterior, se debe lanzar una excepción de negocio.
        3. Transferir dinero a una cuenta cuyo saldo inicial es null. El sistema debe tratar el saldo como cero y poder sumar correctamente el monto.
5. **Tipos de Assert: En JUnit, la validación final se realiza mediante "Asserts". Investigue y explique brevemente cuáles son y para qué sirven los tipos de aserciones, dando un ejemplo de en qué caso usaría cada uno.**
    R/ En JUnit los asserts nos permiten verificar que el comportamiento real del sistema coincide con lo esperado.
    1. assertEquals(excpected, actual)
    Verifica que los valores sean iguales.
    Se usa en casos cuando se espera un valor exacto, como un saldo especifico después de una operación.
    Ejemplo de uso real:
    Después de trasferir 200 desde una cuenta con saldo 1000, el saldo debe ser 800.
    assertEquals(800.0, cuentaActualizada.getSaldo());

2. **assertNotEquals(expected, actual)**
   Verifica que dos valores no sean iguales.
   Ejemplo de uso real:
   Se usa por ejemplo para comprobar que el saldo cambio después de una operación.
   assertNotEquals(1000.0, cuentaActualizada.getSaldo());
   Esto nos permite saber que ocurrió una modificación.
3. **assertTrue (condition)**
   Verifica que una condición sea verdadera
   La usamos cuando no buscamos un valor exacto, sino que se cumpla una regla lógica.
   Ejemplo de uso real:
   Verificar que ningún saldo sea negativo.
   assertTrue(cuentaActualizada.getSaldo() >= 0);
4. **assertFalse(condition)**
   Nos verifica si una condición es falsa.
   Ejemplo de uso real:
   Si una cuenta no debe estar bloqueada hacemos lo siguiente:
   assertFalse(cuenta.isBloqueada());
   Como vemos, la usamos para confirmar que algo no ocurre.
5. **assertNull**
   Verifica que un objeto sea null
   Ejemplo de uso real:
   Se usaría si buscamos una entidad inexistente y se espera que no esté en la base de datos.
   assertNull(resultadoBusqueda);
6. **assertNotnull**
   Nos verifica que un objeto no sea null
   Ejemplo de uso real:
   La usamos para validar que una entidad fue creada correctamente.
   assertNotNull(cuentaCreada);
7. **assertThrows (Exception.class, executable)**
   Verifica que se lance una excepción especifica.
   Es fundamental para probar las reglas de negocio que deben fallar
   Ejemplo de uso real:
   Si el monto es negativo se debe lanzar una excepción
   assertThrows(BusinessLogicException.class, () -> {
   accountService.transferBetweenAccounts(id1, id2, -100.0);
   });
8. **assertDoesNotThrow(executable)**
   Verifica que una operación no lance excepciones
   Ejemplo de uso real:
   Se usaría para verificar correctamente que una operación valida sea ejecutada correctamente.
   assertDoesNotThrow(() -> {
   accountService.transferBetweenAccounts(id1, id2, 100.0);
   });
9. **assertAll(..)**
   Nos permite agrupar multiples validaciones en un mismo test.

La utilizamos cuando queremos validar varios resultados de una opracion sin que el test se detenga en el primer fallo.

Ejemplo de uso real:

```
assertAll(
()->assertEquals(800.0,cuentaOrigen.getSaldo()),
()->assertEquals(300.0,cuentaDestino.getSaldo())
);
```

Es muy útil cuando una operación modifica varios estados al mismo tiempo.