Ejercicio Guiado 2: Desarrollo Dirigido por Pruebas

Desarrollo de Software
Ingeniería Informática - Grupo 85
Grupo de trabajo 3

Paula Díaz Collado 100522368
Hugo Caballero Lino 100495814

ÍNDICE

1. Introducción

El Ejercicio Guiado 2 se enmarca en la asignatura Desarrollo de Software, cuyo objetivo principal es aplicar metodologías ágiles y técnicas de verificación de código en un entorno práctico. En este caso, el ejercicio se centra en el Desarrollo Dirigido por Pruebas (TDD), un enfoque que prioriza la escritura de pruebas automatizadas antes de implementar la funcionalidad, garantizando así un código robusto, mantenible y libre de errores críticos.

Contexto y Relevancia

El sistema bancario requiere altos estándares de fiabilidad y precisión, ya que un error en una transferencia o cálculo de saldo puede tener consecuencias graves. Este ejercicio simula un módulo de gestión de transacciones bancarias, donde se implementan tres funciones clave:

1. Solicitud de transferencias (validación de IBAN, conceptos, montos y fechas).
2. Registro de ingresos (procesamiento de archivos JSON y generación de firmas seguras).
3. Cálculo de saldos (suma de movimientos y manejo de errores).

Metodología Utilizada

- TDD (Test-Driven Development):
    - Se definieron primero los casos de prueba (en un archivo Excel) antes de escribir el código.
    - Cada función se implementó en iteraciones, verificando que las pruebas unitarias fueran exitosas.
- Pruebas Funcionales y Estructurales:
    - Función 1: Análisis de clases de equivalencia y valores límite (ej.: IBANs válidos/inválidos).
    - Función 2: Análisis sintáctico (gramática JSON y árbol de derivación).
    - Función 3: Pruebas estructurales (gráfico de flujo y cobertura de bucles).
- Automatización con PyBuilder:
    - Configuración de un pipeline automatizado para ejecutar pruebas y validar el código.

Herramientas y Estándares

- Lenguaje: Python (por su sintaxis clara y soporte para TDD).
- Control de Versiones: GitHub (trabajo colaborativo bajo propiedad colectiva de código).
- Calidad de Código:
    - PEP8: Verificado con PyLint para garantizar legibilidad y consistencia.
    - PyTest: Framework para ejecución de pruebas unitarias.
- Documentación:
    - Casos de prueba en Excel y reporte en PDF para facilitar la revisión.

Estructura del Proyecto

El desarrollo se organizó en un repositorio GitHub con carpetas dedicadas a:

- Código fuente (src/).
- Pruebas unitarias (test/).
- Datos de ejemplo (data/).
- Documentación técnica (docs/).

---

2. Objetivos del Ejercicio

El Ejercicio Guiado 2 está diseñado para consolidar competencias clave en el desarrollo de software, centrándose en tres pilares fundamentales:

Practicar la Propiedad Colectiva de Código y Estándares de Codificación

- Propiedad colectiva:
    - Fomentar el trabajo colaborativo mediante el uso de GitHub, donde todos los miembros del equipo deben contribuir de manera equilibrada (commits, revisiones, resoluciones de conflictos).

- Estándares PEP8:
    - Aplicar reglas de estilo de Python (indentación, nombrado de variables, longitud de líneas) mediante PyLint.
    - Justificación: Un código estandarizado mejora la mantenibilidad y reduce errores en integraciones futuras.

Dominar el Proceso de Desarrollo Guiado por Pruebas (TDD)

- Ciclo TDD (Red-Green-Refactor):
    - Red: Escribir pruebas unitarias antes del código (basadas en los requisitos funcionales).
    - Green: Implementar el código mínimo para pasar las pruebas.
    - Refactor: Optimizar el código sin romper su funcionalidad.
- Ejemplo práctico:
    - Para la función transfer_request, primero se definieron pruebas para validar IBANs (ej.: ES9121000418450200051332 como válido, ES123 como inválido) antes de implementar la lógica.

Automatización de Pruebas con PyBuilder

- Configuración de un pipeline automatizado:
    - PyBuilder ejecuta pruebas unitarias, verifica cobertura y genera informes en cada cambio (build).
    - Ventaja: Detecta errores temprano y asegura que el código cumple con los requisitos antes de integrarse.

Aplicar Técnicas de Pruebas Funcionales y Estructurales

- Pruebas funcionales (qué hace el sistema):
    - Función 1 (Transferencia):
        - Clases de equivalencia: IBANs válidos/inválidos, montos dentro/fuera de rango (10.00 ≤ amount ≤ 10000.00).
        - Valores límite: amount = 10.00 (mínimo), amount = 10000.00 (máximo).
    - Función 2 (Ingreso):
        - Análisis sintáctico del JSON: Estructura obligatoria (IBAN, AMOUNT), formato de moneda (EUR ####.##).
- Pruebas estructurales (cómo lo hace el sistema):
    - Función 3 (Saldo):
        - Gráfico de flujo: Cobertura de todas las ramas (ej.: si el IBAN no existe en el archivo JSON).
        - Pruebas de bucles: Verificar que suma correctamente múltiples movimientos (ej.: +2483,43 y -3221,12).

Integración de Herramientas Profesionales

- GitHub:
    - Uso de issues, pull requests y code reviews para simular un entorno ágil real.
- PyLint y PyTest:
    - Garantizan que el código no solo funcione, sino que también sea limpio y eficiente.

Documentación Profesional

- Generar informes técnicos en PDF y Excel que:
    - Expliquen los casos de prueba (entradas, salidas esperadas).
    - Incluyan gráficos (árboles de derivación, flujos de control).
    - Sean presentables para evaluación o auditorías futuras.

Impacto en el Aprendizaje

Este ejercicio no solo evalúa la capacidad de codificar, sino también:

- Trabajo en equipo: Coordinación para dividir tareas (ej.: un miembro implementa TDD, otro documenta).
- Pensamiento crítico: Priorizar pruebas para casos edge (ej.: fecha 31/02/2025 en transferencias).
- Adaptabilidad: Uso de nuevas herramientas (PyBuilder) en un contexto estructurado.

Conclusión: Los objetivos trascienden la implementación técnica; buscan formar desarrolladores que integren calidad, colaboración y metodologías ágiles en su flujo de trabajo.

---

3. Casos de Prueba

Función 1: Solicitar Transferencia (transfer_request)

El método transfer_request valida los datos de una transferencia bancaria (IBAN de origen y destino, concepto, tipo, fecha y monto) utilizando la clase AccountManager. Si los datos son correctos, crea un objeto TransferRequest, genera un código único mediante MD5 y lo añade al JSON de la transferencia. Luego, guarda la información en un archivo transfers.json, asegurándose de que el directorio existe y manejando posibles errores de archivo o datos duplicados. Finalmente, retorna el código de la transferencia.

Teniendo así, los casos de prueba realizados con unitest, donde nos encontramos con los siguientes (redactados uno a uno en el excel de casos de prueba).

1. TC1-TC4 (Validaciones exitosas)
    - Se prueban transferencias con datos correctos (IBANs válidos, concepto adecuado, tipo de transferencia válido, fecha correcta y monto correcto).
    - Resultado esperado: Se genera un código de transferencia en MD5 y se almacena en el archivo JSON.
2. TC5-TC12 (Errores en el IBAN)
    - TC5-TC6: from_iban inválido (número erróneo o caracteres no permitidos).
    - TC7-TC8: from_iban con más o menos de 24 caracteres.
    - TC9: from_iban de otro país que no es España.
    - TC10: from_iban con primeros dos caracteres que no son letras.
    - TC11: from_iban con caracteres incorrectos (ejemplo: letras aleatorias).
    - TC12: from_iban no es una cadena de texto válida.
    - Resultado esperado: Se lanza un error indicando que from_iban no es válido.
3. TC13-TC18 (Errores en el to_iban)
    - TC13-TC14: to_iban con más o menos de 24 caracteres.
    - TC15: to_iban de otro país que no es España.
    - TC16: to_iban con primeros dos caracteres que no son letras.

- TC17-TC18: to_iban contiene caracteres erróneos o no es una cadena válida.
- Resultado esperado: Se lanza un error indicando que to_iban no es válido.

4. TC19-TC23 (Errores en el concepto de la transferencia)
    - TC19: Concepto no es una cadena de texto.
    - TC20: Concepto demasiado largo.
    - TC21: Concepto demasiado corto.
    - TC22: Concepto inválido con caracteres especiales.
    - TC23: Concepto compuesto solo por números.
    - Resultado esperado: Se lanza un error indicando que el concepto es inválido.
5. TC24-TC32 (Errores en la fecha de la transferencia)
    - TC24: Fecha no es una cadena de texto.
    - TC25: Formato de fecha incorrecto.
    - TC26: Día (DD) no es numérico.
    - TC27: Día no separado correctamente por barras.
    - TC28: Día mayor a 31.
    - TC29: Día menor a 1.
    - TC30: Mes (MM) mayor a 12.
    - TC31: Mes menor a 1.
    - TC32: Año (YYYY) incorrecto.
    - Resultado esperado: Se lanza un error indicando que la fecha es inválida.
6. TC33-TC37 (Errores en el monto de la transferencia)
    - TC33: Monto contiene texto en vez de números.
    - TC34: Monto superior al permitido.

- TC35: Monto menor al permitido.
- TC36: Monto con formato decimal incorrecto.
- TC37: Monto con separación decimal incorrecta.
- Resultado esperado: Se lanza un error indicando que el monto es inválido.

Cada caso de prueba verifica un escenario distinto para asegurar que el sistema maneje correctamente errores en los datos ingresados.

Función 2: Ingreso en Cuenta (deposit_into_account)

La función deposit_into_account procesa un depósito bancario a partir de un archivo JSON, validando que contenga las claves "IBAN" y "AMOUNT". Comprueba que el IBAN sea español y tenga la longitud correcta, y que el monto esté en euros y sea un número válido. Si alguna validación falla, lanza una excepción AccountManagementException. Si los datos son correctos, genera un registro del depósito con una firma SHA-256 y lo guarda en un archivo JSON llamado deposits.json. Además, maneja errores como archivo no encontrado, formato JSON inválido y excepciones generales.

Teniendo así, los casos de prueba realizados con unitest, donde nos encontramos con los siguientes (redactados uno a uno en el excel de casos de prueba).

Pruebas de casos no válidos (TCNV)

Estas pruebas validan que la función lance una excepción cuando los datos de entrada no son correctos.

1. test_TCNV2

- Propósito: Verificar que la función detecte un JSON con una estructura incorrecta.
- Caso: Se pasa un archivo JSON con un formato inválido.
- Esperado: Se lanza AccountManagementException con el mensaje "Formato JSON incorrecto".

2. test_TCNV1

- Propósito: Verificar que el archivo JSON de entrada contenga obligatoriamente el campo IBAN.

- Caso: El JSON solo tiene el campo AMOUNT, pero falta IBAN.
- Esperado: Se lanza AccountManagementException con el mensaje "Formato JSON incorrecto".

3. test_TCNV3

- Propósito: Validar que el sistema detecte un IBAN incorrecto.
- Caso: Se usa un IBAN del Reino Unido (GB29...), mientras que el sistema solo acepta IBAN españoles.
- Esperado: Se lanza AccountManagementException con el mensaje "IBAN inválido".

4. test_TCNV4

- Propósito: Detectar un IBAN con formato erróneo.
- Caso: Se usa un IBAN español, pero con un número de caracteres incorrecto.
- Esperado: Se lanza AccountManagementException con el mensaje "IBAN inválido".

5. test_TCNV5

- Propósito: Detectar un IBAN con más caracteres de los permitidos.
- Caso: Se introduce un IBAN con demasiados caracteres.
- Esperado: Se lanza AccountManagementException con el mensaje "IBAN inválido".

6. test_TCNV6

- Propósito: Validar que solo se acepten depósitos en euros.
- Caso: Se usa AMOUNT: "USD 100.00".
- Esperado: Se lanza AccountManagementException con el mensaje "Moneda incorrecta".

7. test_TCNV7

- Propósito: Validar que el monto debe ser numérico.

- Caso: Se usa AMOUNT: "EUR one hundred" en lugar de un número.
- Esperado: Se lanza AccountManagementException con el mensaje "Monto no numérico".

8. test_TCNV8

- Propósito: Detectar montos inválidos.
- Caso: Se usa AMOUNT: "EUR 0.00", lo que no tiene sentido para un depósito.
- Esperado: Se lanza AccountManagementException con el mensaje "Monto inválido".

9. test_TCNV9

- Propósito: Detectar montos negativos.
- Caso: Se usa AMOUNT: "EUR -1.00".
- Esperado: Se lanza AccountManagementException con el mensaje "Monto inválido".

10. test_TCNV10

- Propósito: Validar que el JSON no tenga errores sintácticos.
- Caso: Se intenta procesar un JSON mal formado.
- Esperado: Se lanza AccountManagementException con el mensaje "Formato JSON incorrecto".

11. test_TCNV12

- Propósito: Detectar IBAN con caracteres adicionales incorrectos.
- Caso: Se usa un IBAN con texto adicional pegado.
- Esperado: Se lanza AccountManagementException con el mensaje "IBAN inválido".

12. test_TCNV15

- Propósito: Validar que no haya errores en el formato del monto.

- Caso: Se repite el campo AMOUNT en el JSON.
- Esperado: Se lanza AccountManagementException con el mensaje "Monto no numérico".

13. test_TCNV16

- Propósito: Detectar errores en la estructura del JSON.
- Caso: Se introduce un error en la sintaxis del JSON (llaves mal cerradas).
- Esperado: Se lanza AccountManagementException con el mensaje "Monto no numérico".

Pruebas de casos válidos (TCV)

Estas pruebas verifican que la función no lance excepciones y genere una firma de transacción válida.

1. test_TCV1

- Propósito: Validar que un archivo con datos correctos genere una firma válida.
- Caso: Se usa un IBAN válido (ES9121000418450200051332) y un monto correcto.
- Esperado: La función retorna un hash SHA-256 de 64 caracteres.

2. test_TCV2

- Propósito: Verificar que el monto mínimo aceptado (EUR 0.01) sea válido.
- Caso: Se usa AMOUNT: "EUR 0.01".
- Esperado: La función retorna un hash SHA-256 de 64 caracteres.

3. test_TCV3

- Propósito: Verificar que la función acepte montos grandes.
- Caso: Se usa el monto máximo permitido AMOUNT: "EUR 999999999.99".
- Esperado: La función retorna un hash SHA-256 de 64 caracteres.

4. test_TCV4

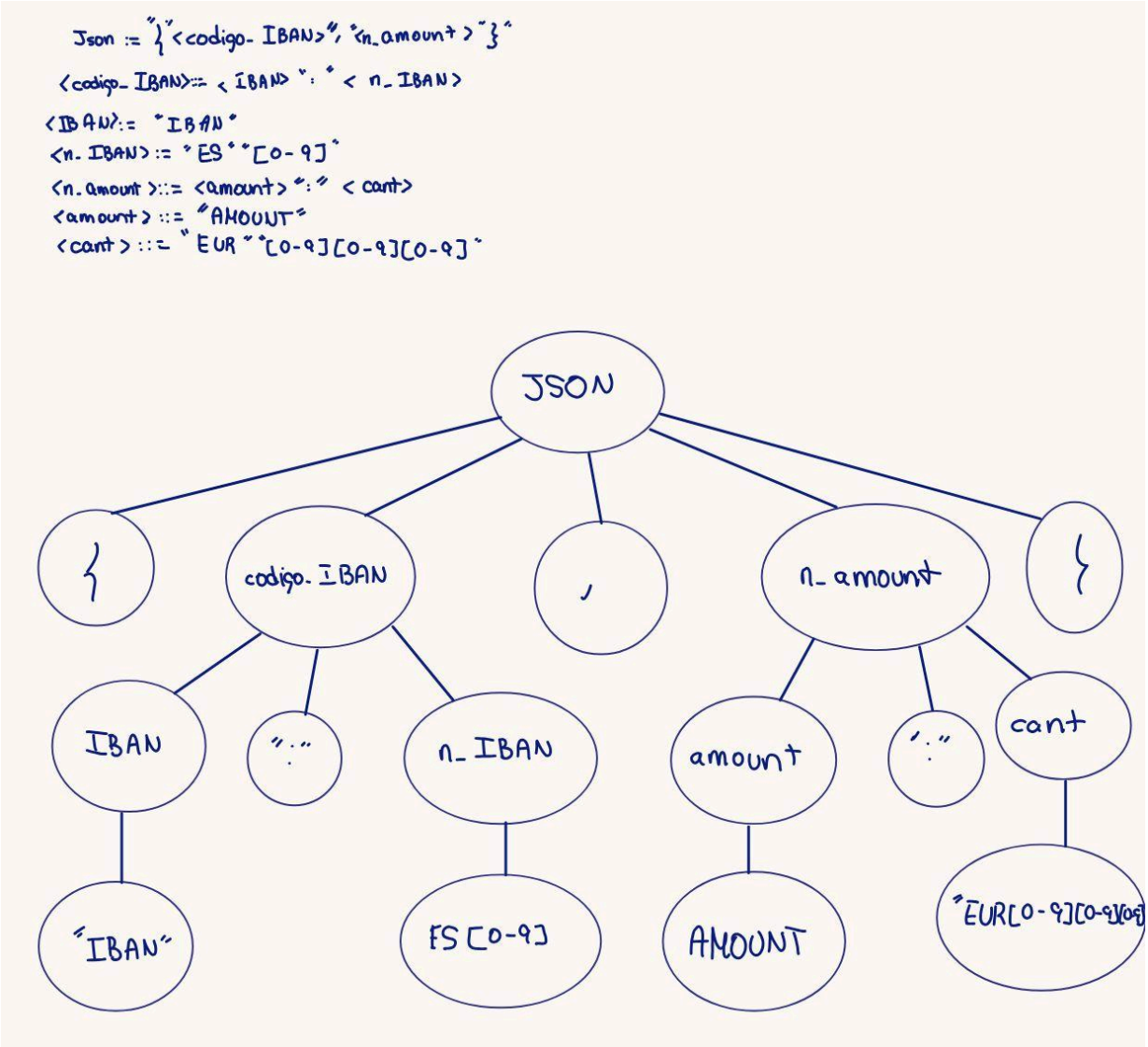
- Propósito: Verificar que la función acepte montos intermedios.
- Caso: Se usa AMOUNT: "EUR 123.45".
- Esperado: La función retorna un hash SHA-256 de 64 caracteres.

5. test_TCV5

- Propósito: Verificar que el formato decimal sea válido.
- Caso: Se usa AMOUNT: "EUR 123.4" (con solo una cifra decimal).
- Esperado: La función retorna un hash SHA-256 de 64 caracteres.

6. test_TCV6

- Propósito: Validar que se permita el mismo monto con diferentes estructuras de JSON.
- Caso: Se usa AMOUNT: "EUR 123.4", igual que test_TCV5.
- Esperado: La función retorna un hash SHA-256 de 64 caracteres.



Todos los test han sido obtenidos mediante la gramática de la función, sacando así los casos de cada nodo terminal/no terminal.

Función 3: Calcular Saldo (calculate_balance)

Propósito:
Sumar movimientos positivos/negativos asociados a un IBAN en un archivo JSON de transacciones.

El programa carga del archivo JSON**: Abre y lee un archivo JSON que contiene los datos del depósito.

Las validaciones:
- Verifica que el JSON tenga la estructura correcta (campos "IBAN" y "AMOUNT")
- Valida el formato del IBAN (debe empezar con "ES" y tener 24 caracteres)
- Valida el monto (debe empezar con "EUR " seguido de un número positivo)

En cuanto a la creación de datos del depósito:
- Construye un objeto con información del depósito incluyendo:
- Algoritmo de hash usado (SHA-256)
- Tipo de operación (DEPOSIT)
- IBAN destino
- Monto del depósito
- Fecha/hora actual (timestamp)

Crea una firma SHA-256 basada en los datos del depósito y guarda los datos del depósito (incluyendo la firma) en un archivo `deposits.json`

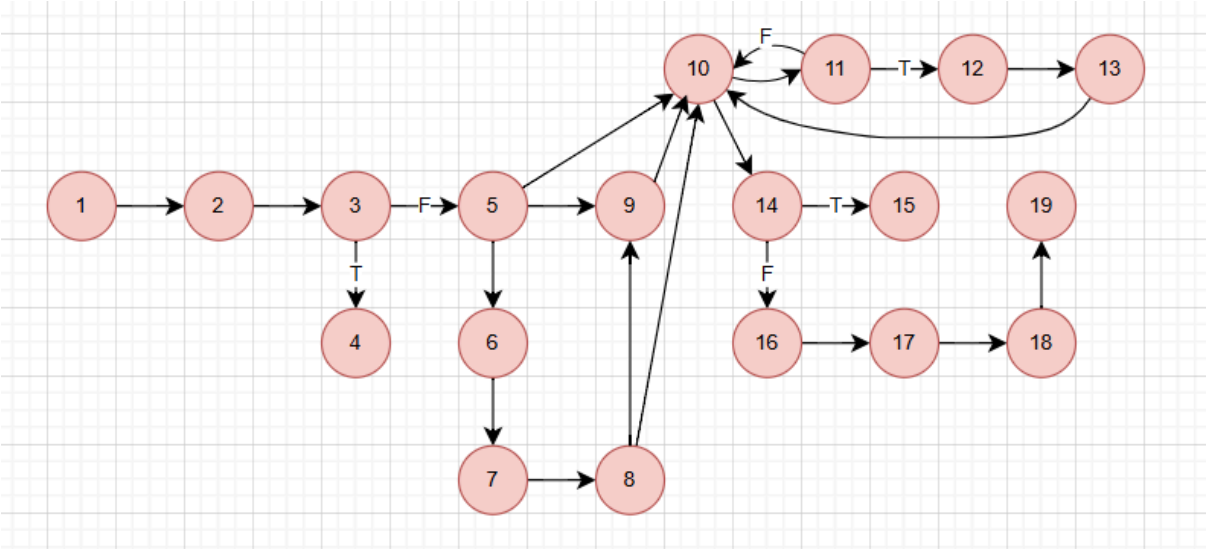
El método captura varios tipos de errores:
- Archivo no encontrado
- JSON inválido
- Errores de validación (IBAN incorrecto, monto inválido)
- Otros errores inesperados

Devuelve la firma digital del depósito como confirmación.

Aquí tenemos los nodos indicados:

```python
def calculate_balance(self, iban):  2 usages  ± 100522368
    """Funcion para calcular el balance"""
1   iban_found = False
2   balance_result = 0
3   path_all_transactions = Path(
        __file__).resolve().parent.parent.parent / "unittest" / "data" / "all_transactions.json"
4   if not self.validate_iban(iban):
        raise AccountManagementException("InvalidIBANCode")
    try:
        with open(path_all_transactions, encoding="utf-8", mode="r") as file:
5           all_transactions = json.load(file)
6   except FileNotFoundError:
7       path_all_transactions.parent.mkdir(parents=True, exist_ok=True)
8       all_transactions = []
        with open(path_all_transactions, encoding="utf-8", mode="w") as file:
9           json.dump(all_transactions, file, indent=4)
10  except json.JSONDecodeError:
        raise AccountManagementException("File all_transactions is not a valid JSON")
11  for transaction in all_transactions:
12      if transaction["IBAN"] == iban:
13          balance_result += float(transaction["amount"])
14          iban_found = True
15  if not iban_found:
        raise AccountManagementException("InvalidIBANCode")
16  path_balance_file = Path(__file__).resolve().parent.parent.parent / "unittest" / "data" / f"{iban}.json"
    with open(path_balance_file, encoding="utf-8", mode="w") as file:
17      balance_data = {"iban": iban, "date": str(datetime.now()), "balance": balance_result}
18      json.dump(balance_data, file, indent=4)
19  return True
```

Este es el grafo:



Los tests verifican el método calculate_balance de la clase AccountManager, probando diferentes casos de validación de IBAN y cálculo de saldo. Los tests están numerados como tc1 a tc10 y cada uno sigue un camino específico en el código (indicado en los comentarios como secuencias de números).

Tests Exitosos:

- `test_calculate_balance_ok_tc1`
  Camino probado: `1_2_3_5_10_11_10_14_16_17_18_19`
  - Prueba un IBAN válido (`ES9121000418450200051332`)
  - Renombra archivos para usar datos de prueba
  - Verifica que el cálculo del saldo sea exitoso (`True`)
  - Comprueba que el saldo calculado sea exactamente `-1280.06`
  - Restaura los archivos originales

- `test_calculate_balance_ok_tc2`
  Camino probado: `1_2_3_5_6_7_8_9_10_11_12_13_10_11_10_14_16_17_18_19`
  - Similar al tc1 pero con otro IBAN válido (`ES8658342045412168772O4`)
  - Sigue un camino más complejo en el código (más validaciones internas)
  - También verifica que el saldo sea `-1280.06`

Tests Fallidos:

- `test_calculate_balance_ko_tc3`
  Camino probado: `1_2_3_5_6_7_8_9_10_11_10_14_15`
  - Prueba con IBAN `ES7620770024003102575766` (inválido)

- `test_calculate_balance_ko_tc4`
  Camino probado: `1_2_3_5_6_7_8_9_10_14_15`
  - IBAN `ES9111112222333344445555` (inválido)

- `test_calculate_balance_ko_tc5`
  Camino probado: `1_2_3_4`
  - IBAN claramente inválido `ES00INVALID00000000000000`

- `test_calculate_balance_ko_tc6`
  Camino probado: `1_2_3_5_6_7_8_10_11_10_14_15`
  - IBAN `ES9111112222333344445555` (inválido)

- `test_calculate_balance_ko_tc7`
  Camino probado: `1_2_3_5_10`
  - IBAN `ES7921000813610123456789` (inválido)

- `test_calculate_balance_ko_tc8`
  Camino probado: `1_2_3_5_10_11_10_14_15`
  - IBAN `ES1234567890123456789012` (inválido)

- `test_calculate_balance_ko_tc9`
  Camino probado: `1_2_3_5_10_14_15`
  - IBAN `ES7121000418450200051332` (inválido)

- `test_calculate_balance_ko_tc10`
  Camino probado: `1_2_3_5_10_11_12_13_10_11_10_14_15`
  - IBAN `ES9521000418450200051332` (inválido)