Guided practice 2: Group 22 report

By

Jose Luis Mejía Acuña - 100472712
Aamani Lakshmi Gurajada - 100559677

Index

Introduction

This guided exercise focuses on the principles of **Test-Driven Development (TDD)** in the context of a **banking management component**. The objective is to design, automate, and execute unit tests to validate core functionalities such as **transfer requests, deposits, and balance calculations**.

Beyond implementing these features, the exercise also emphasizes **understanding how to build and manage software projects using tools like PyBuilder**. Through PyBuilder, we will automate test execution, generate reports, and ensure proper test coverage, reinforcing best practices in **software development and testing**.

Additionally, a strong focus is placed on **coding standards**, including adherence to **PEP8**, ensuring **code quality, maintainability, and readability**. By applying different testing techniques such as **equivalence classes, syntax analysis, and structural testing** we will be able to practice their implementation and see their differences and how some may be better suited for certain things.

Overview Pybuilder

What is Pybuilder

PyBuilder is a **build automation tool** for Python projects that helps manage tasks such as **compiling code, running tests, and generating reports**. It follows a **declarative approach**, where configurations and dependencies are defined in a structured way, allowing for streamlined project execution.

How PyBuilder Works:

For this build we use Pybuilder entirely through the terminal so we start our project with:

```
pyb --start-project my_project
```

After this we will see how the entire project's structure changes so that now:

- **src:**
    - **main**
    - **unittest**
- **target:**
    - **reports**

This is a simplified version but highlights the important new directories built that will be essential for our practice.

src is where our code will be stored, so in main we will store our scripts and their json files, then in unittest we store the testing scripts that **HAVE** to follow a naming convention specified by Pybuilder so that when we run pyb or pyb -v (for added verbose) these tests are automatically run.

PyBuilder will also check how many flows of our code in main is being used by our tests so in our reports in **target** we will see both if we have passed our created tests and how much of our project are we testing.

Benefits for the Project:

Having explained how Pybuilder works it is clear that it is a powerful and useful tool. It automatizes the way we run our tests whilst simultaneously checking for different programming flows.

Pybuider is also useful to create a project with different directories to separate everything clearly.

From all of this we can conclude that Pybuilder and similar tools are essential in the software development process and thus this practice is a great opportunity to gain experience with it and it's work flows whilst building a solid software project with good praxis.

Function 1: Transfer request

Description

This function records bank transfer requests between two Spanish bank accounts. It ensures that all input data is correctly formatted and follows validation rules before storing the transaction. Additionally, it generates a unique transfer code using the MD5 hashing algorithm and saves the transfer details in a JSON file for future reference.

Inputs:

- **from_iban**: The sender's Spanish IBAN (must be valid).

- **to_iban**: The recipient's Spanish IBAN (must be valid).

- **concept**: A short description of the transaction (10-30 characters, containing at least two words).

- **type**: The type of transfer, which must be one of the following: **ORDINARY**, **URGENT**, or **IMMEDIATE**.

- **date**: The transaction date in **DD/MM/YYYY** format (must be between 2025 and 2050 and cannot be earlier than the current date).

- **amount**: The amount to transfer in EUR (must be between 10.00 and 10,000.00, with a maximum of two decimal places).

Main Processes:

- **Validation**: The function verifies that all input values conform to the required formats and constraints. If any value is invalid, an exception is raised.

- **Transfer Code Generation**: If the input data is valid, the function generates a unique MD5 hash based on the transfer details. This hash acts as the transaction identifier.

- **Data Storage**: The transfer request is stored in a JSON file, ensuring that duplicate entries are not recorded.

Outputs:

- A unique **transfer code** (MD5 hash) representing the transaction.

- A **JSON file** containing the stored transfer details.

- An **exception** if any of the validation checks fail or if a duplicate transfer request is detected.

Testing Approach

For function 1 we have used the Equivalence Classes and Boundary Values Analysis i.e we have analyzed the inputs of the function and set the possible ways it can behave, when are they valid when they are not, what limits do they have.

From this we build a table specifying it such as this:

Variable	Criteria	Valid Class	Invalid Class
from_iban	String not iban format	ECV1: String in Iban spanish format	ECNV1: String not iban format, ECNV2: String in not spanish Iban format
to_iban	String not iban format	ECV2: String in Iban spanish format	ECNV3: String not iban format, ECNV4: String in not spanish Iban format
concept	at least 2 strings sep by " "	ECV3: 2 strings separated by " "	ECNV5: words not separeted by " ", BNVV1: hello BNVV2: 12_hello
concept	10 < |concept| < 30	ECV4: string > 10 BVV1: hello worl (10 chars), BVV2: hello world (11 chars) BVV3: instant blockchain payment xy (30 chars) BVV4: instant blockchain payment x (29 chars)	ECNV6: chars less than 10 or over 30, BNVV3: hello, BNVV4: instant blockchain payment xyz
type	Possible values	ECV5: Possible values, BVV5: ORDINARY, BVV6: URGENT, BVV7: IMMEDIATE	ECNV7: not the possible values, BNVV5: , BNVV6: HELLOWORLD
date	DD/MM/YYYY	ECV6: Follows date format, DD is between 01 and 31, MM is between 01 and 12 and YYYY is between 2025 and 2051, BVV8: 01/01/2025, BVV9: 02/02/2026, BVV10: 31/12/2051, BVV11: 30/11/2050 whilst they are equal to or greater than current date	ECNV8: not DD/MM/YYYY format, BNV7: hello, ECNV9: DD/MM/YYYY format but out of ranges, BNV10: 00/00/2024, BNV11: 32/13/2052, BNV12: DD/MM/YYYY(date before current date)
Amount	data_type: float	ECV7: float number.	ECNV10: not a float BNV13: HELLO
Amount	max 2 decimals	ECV8: float number with 2 decimals	ECNV11: float with more than 2 decimals,BNV13: 12.101010

Amou nt	10.00<=q uantity<= 10,000.00	ECV9: float number with 2 decimals between 10.00 and 10,000.00, BVV12: 10.00, BVV13: 10.01, BVV14: 10,000.00, BVV15: 9,999.99	ECNV12: float less than 10.00 BNV14: 9.99, ECNV13: float bigger than 10,000.00, ECNV14: 10,000.01
	json file containin g transactio n or not	ECV10: JSON file doesn't contain the transaction already	ECNV12: JSON file contains the transactino

Table 1: Function 1 ECs and BVs

From this table we have obtained the test cases enumerated in the excel and used in our tests through a json file.

Once they were built we **THEN** started coding the function so that it would pass the tests (passing all of them).

The only problem found is checking repetition. From the code's logic the transfer code is generated using several things but the problematic one is the **timestamp**. A specific variable given at the moment of conception.

We could simply generate a case where we force an specific value for it and add such a case into the output json file but we would be changing artificially our code for a thing that is not possible and even then the code already has an exception prepared for such an scenario thus we decided to not test that part.

Function 2: Deposit into account

Description

Inputs:

- A JSON file containing:

    - A valid IBAN for the recipient account.

    - A deposit amount, formatted as "EUR ####.##".

Processes:

- Validation:

    - Check if the JSON file exists and is correctly formatted.

    - Verify that the IBAN is a valid Spanish account number.

    - Ensure the deposit amount is properly formatted and within acceptable numerical limits.

- Deposit Signature Generation:

    - Use SHA-256 hashing to create a unique deposit signature based on the transaction details.

    - The signature acts as a transaction identifier.

- Data Storage:

    - Record the deposit details in a JSON file, including the IBAN, amount, timestamp, and generated signature.

    - Ensure no duplicate transactions are stored.

Outputs:

- A SHA-256 deposit signature in hexadecimal format.

- A JSON file containing the recorded deposit details.

- An exception if any validation fails (e.g., invalid IBAN, incorrect format, missing file).

Testing Approach:

For function 2 we have used Syntax analysis for our test generation this means that from our needed inputs (We considered both the json file and the output json file as it is needed to perform the code) we create a Grammar that generates the needed inputs, from it we create the derivation tree and via reaching nodes and eliminating, duplicating… , nodes we generate our test cases although to make it more complete we added some more via the method use in function 1 to make the testing more complete.

Here is the grammar:

```
<Input> ::= <Input_Json><Output_Json>
      <Input_Json>::=<IBAN><AMOUNT>
           <IBAN>::=<key_1><iban_res>
                 <key_1>::= "IBAN: "
                 <iban_res>::= <ES><NUMBER>
                       <ES>::= "ES"
                       <NUMBER>::= [0-9]{24}
           <AMOUNT>::=<key2><eur><num1><dot><num2>
                 key2::= "Amount: "
                 eur::= "EUR"
                 num1::=[0-9]{0,4}
                 dot::=".""
                 num2::=[0-9]{2}
      <Output_Json>::=Existence
```

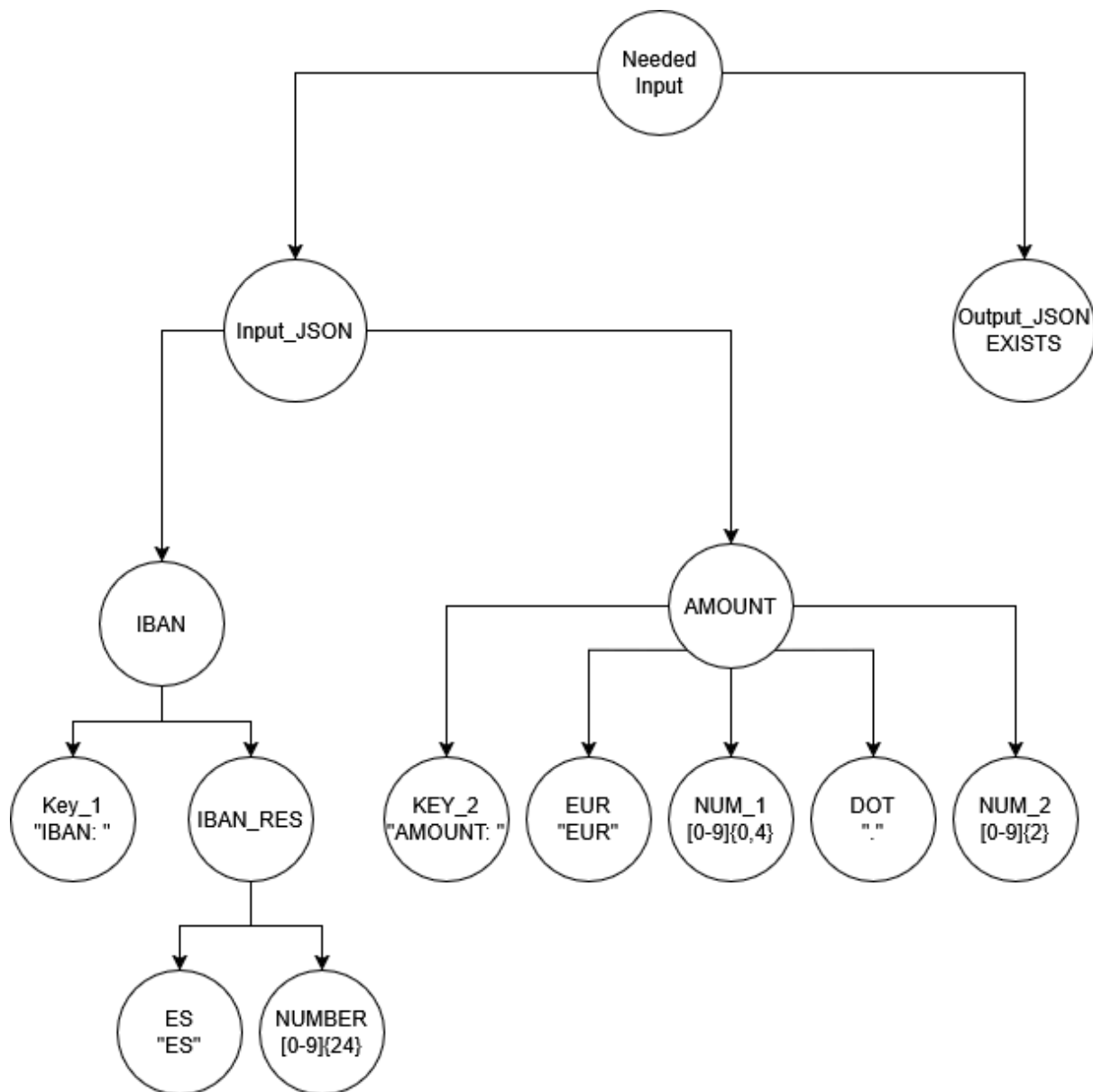And from the grammar using draw.io we got:

Image 1: Function 2 grammar derivation tree

Lastly from this and some other extra cases from the previous method we were able to use the techniques in class to obtain the test cases seen in the excel.

Function 3: Calculate balance

Description

Inputs:

- **IBAN**: A valid Spanish IBAN for which the balance must be calculated.

Processes:

- **Validation**:

    - Verify that the provided IBAN is valid and appears in the transactions data.

- **Aggregation**:

    - Retrieve all transactions associated with the IBAN.

    - Sum the amounts (which may be positive or negative) to obtain the net balance.

- **Storage**:

    - Record the computed balance, along with the IBAN and the timestamp, in a JSON file for reference and auditing.

Outputs:

- A **boolean value** indicating successful calculation and storage.

- A **JSON file** containing the balance details for the specified IBAN.

- An **exception** if the IBAN is invalid, not found, or if any file-related errors occur.

Testing Approach

In this function we have used the easiest and quickest, at least until now and for our team, methodology, which is a structural testing methodology, meaning we directly create the code to complete the task. We analyze the code and its work flows and create a workflow tree and from it we create our test cases which are less redundant and more specific in their functionality.

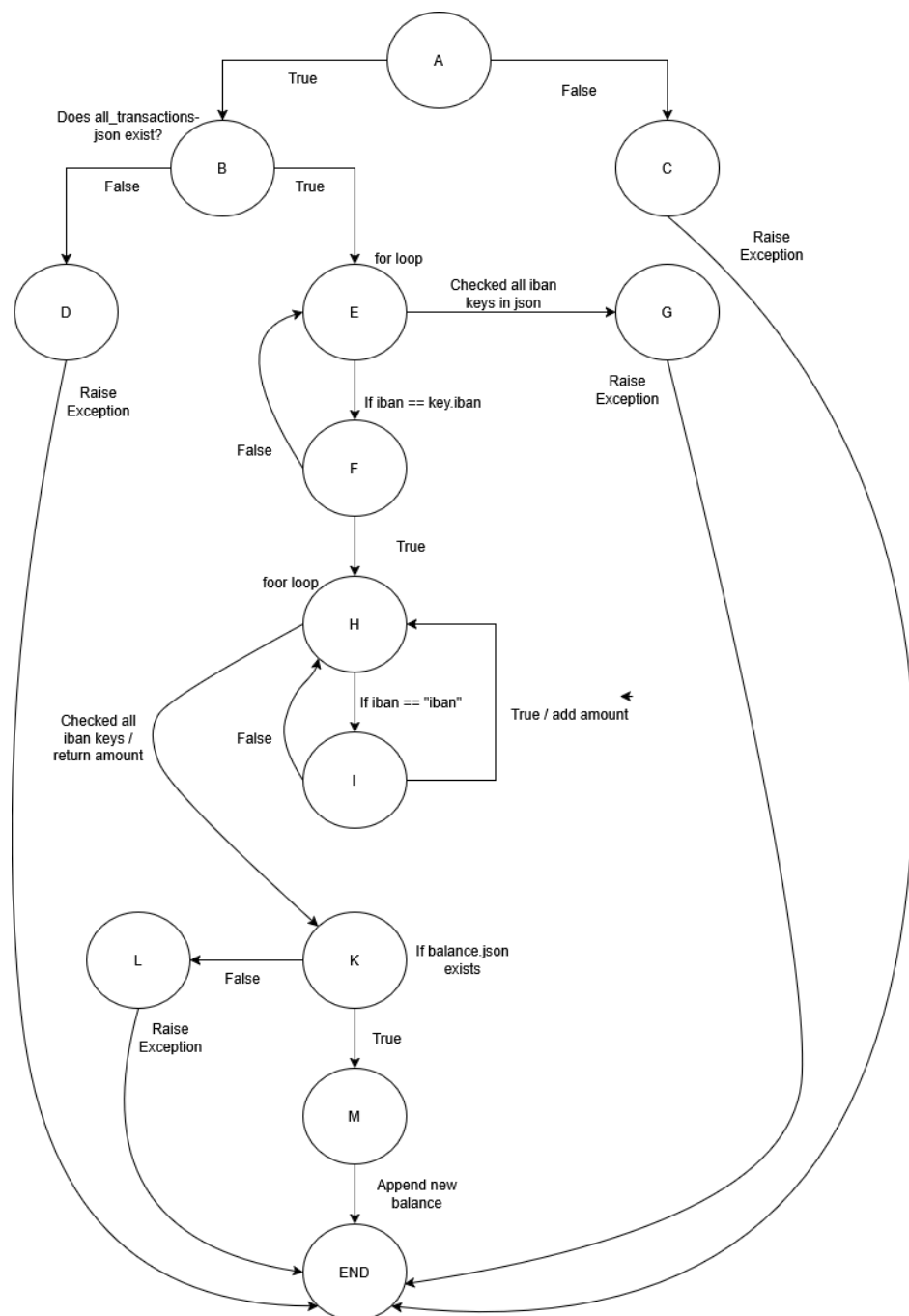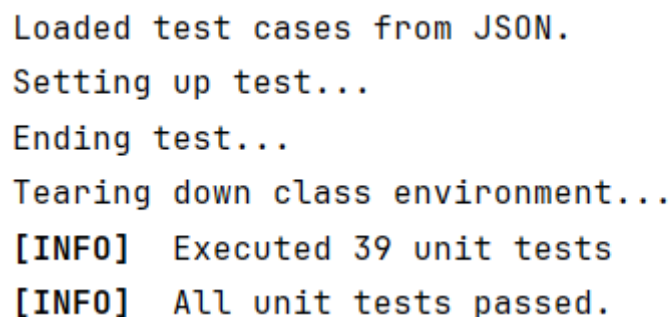From our created script we obtained this tree:

Image 2: workflow tree for function 3

This tree allowed us to create test cases by using paths that reached every possible node meaning we tested all the paths that got to the end node, also it's complexity is 8 which seeing the task in function 3 seems like a nice obtained value.

Analysis of Pybuilder results

Test Execution Results

Running **pyb** in our terminal should return how our tests create and tear down test environments several times and lastly if we have done it right all our tests (39) should pass.
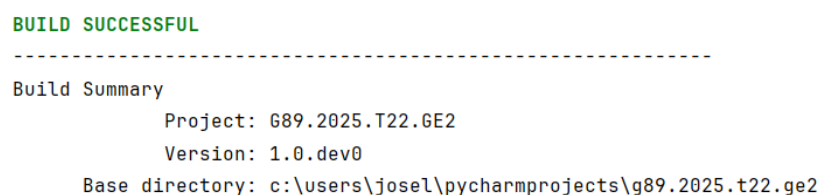


```
Loaded test cases from JSON.
Setting up test...
Ending test...
Tearing down class environment...
[INFO]  Executed 39 unit tests
[INFO]  All unit tests passed.
```

Image 3: Screenshot of result of pyb

This is a very good result, our code indeed passes all the tests which means that if we have made no mistakes in the process we have correctly built all three functions.

Test Coverage Report

Correct usage of Pybuilder also includes checking that we have made sure to tests as many workflows as possible with our tests. Pylint will not allow a successful build if any module has less than a 70% coverage.


```
BUILD SUCCESSFUL
------------------------------------------------------------
Build Summary
            Project: G89.2025.T22.GE2
            Version: 1.0.dev0
     Base directory: c:\users\josel\pycharmprojects\g89.2025.t22.ge2
```

Image 4: Screenshot of successful pybuild build

Not only this but after having built our project reports will be created in our target folder which we can check to see the coverage in each module.

G89.2025.T22.GE2 coverage: 93%

[Files] [Functions] [Classes]

coverage.py v7.7.1, created at 2025-03-25 14:25 +0100

File ▲	statements	missing	excluded	branches	partial	coverage
uc3m_money__init__.py	4	0	0	0	0	100%
uc3m_money\account_balance.py	43	0	0	16	0	100%
uc3m_money\account_deposit.py	75	0	0	26	1	99%
uc3m_money\account_management_exception.py	7	0	0	0	0	100%
uc3m_money\account_manager.py	6	0	0	2	0	100%
uc3m_money\transfer_request.py	109	18	0	26	3	84%
Total	**244**	**18**	**0**	**70**	**4**	**93%**

Image 5: Screenshot of Pybuilder coverage report.

These results showcase how our code exceeds the minimum requirements of Pybuilder of 70% coverage per module and in all modules but the one of function 1 (which lacks the specified case mentioned in it's part in this report).

Conclusions

From our results we can conclude that we have been successful in creating a software project that includes the necessary modules (functions 1 to 3). These modules perform the minimum tasks required and are tested correspondingly with a 93% total coverage.

Pybuilder is a very powerful tool that has allowed us to create a software project easily and test it with different implementations of Test driven development for each function.

In this project we have learnt a lot and been able to see the advantages of test driven development and the possibility to test each other's tests and code to see if it is correct.