INFORME DE LABORATORIO

(Formato Estudiante)

INFORMACIÓN BÁSICA					
ASIGNATURA:	Pruebas de Software				
TÍTULO DE LA PRÁCTICA:	*Pruebas Unitarias y Refactorización de un Cajero Automático (ATM)*				
NÚMERO DE PRÁCTICA:	*02*	**AÑO LECTIVO:**	*2023*	**NRO. SEMESTRE:**	*VII*
FECHA DE PRESENTACIÓN	*19/05/2023*	**HORA DE PRESENTACIÓN**	*07:20*		
INTEGRANTE (s): Hincho Jove, Angel Eduardo Neira Carrasco, Darwin Jesus				**NOTA:**	
DOCENTE(s): *Arisaca Mamani, Robert Edison*					

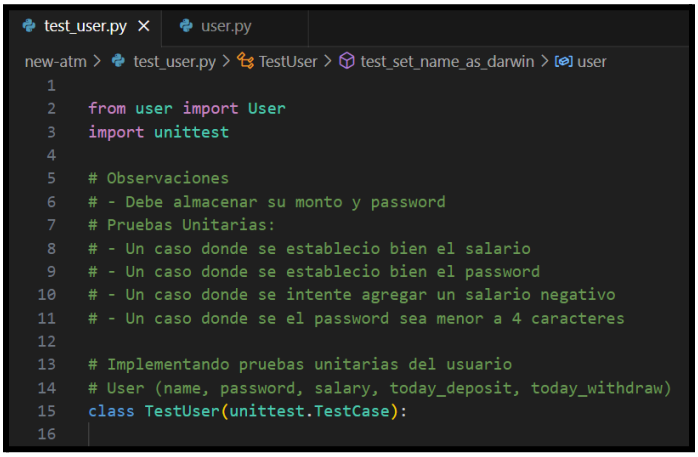
SOLUCIÓN Y RESULTADOS

I. SOLUCIÓN DE EJERCICIOS/PROBLEMAS

PROBLEMAS PROPUESTOS

La solución o programa donde se refactoriza el Cajero Automático (ATM) se encuentra disponible en la plataforma de GitHub bajo el siguiente enlace: https://github.com/ahincho/PrS-TeoA-ATM.git

1. ***Casos de Prueba:***



Siguiendo una metodología Test Driven Development, se especificaron los casos de prueba y contratos antes de realizar la implementación. Para este problema se identificaron dos entidades principales: Usuario y ATM. Estamos viendo los casos de prueba que debe cumplir la implementación de un Usuario.

```
def test_set_invalid_name(self):
    with self.assertRaises(TypeError):
        aUser = User(123, "superPassword", 1500, 0, 0) # Type error
    with self.assertRaises(ValueError):
        bUser = User("abc", "superPassword", 1500, 0, 0) # Too short

def test_set_invalid_password(self):
    with self.assertRaises(TypeError):
        aUser = User("Angel", 1456789, 1500, 0, 0) # Type error
    with self.assertRaises(ValueError):
        bUser = User("Angel", "abc", 1500, 0, 0) # Too short
```

Los dos primeros casos de prueba para la Clase Usuario nos especifican que tanto la contraseña como el nombre del Usuario deben ser valores de tipo String. Así mismo, deben cumplir con alguna condición, en este caso nuestra condición será que tenga más de 4 caracteres.

```
def test_set_invalid_salary(self):
    with self.assertRaises(TypeError):
        aUser = User("Angel", "superPwd", "TypeError", 0, 0) # Type error
    with self.assertRaises(ValueError):
        bUser = User("Angel", "superPwd", -5000, 0, 0) # Negative value

def test_set_invalid_today_deposit(self):
    with self.assertRaises(TypeError):
        aUser = User("Angel", "superPwd", 5000, "abc", 0) # Type error
    with self.assertRaises(ValueError):
        bUser = User("Angel", "superPwd", 5000, -500, 0) # Negative value

def test_set_invalid_today_withdraw(self):
    with self.assertRaises(TypeError):
        aUser = User("Angel", "superPwd", 5000, 0, ["hello", "world"]) # Type error
    with self.assertRaises(ValueError):
        bUser = User("Angel", "superPwd", 5000, 0, -1000) # Negative value
```

Los tres siguientes casos de pruebas o pruebas unitarias se refieren a la parte económica. Tanto el salario, total de depósitos y retiros realizados en el día deben ser valores enteros o flotantes mayores a 0.

```python
def test_set_name_as_darwin(self):
    user = User("Darwin", "superPwd", 5000, 0, 0)
    self.assertEqual(user.get_name(), "Darwin")

def test_set_password_as_1234(self):
    user = User("Darwin", "1234", 2500, 0, 0)
    self.assertEqual(user.get_password(), "1234")

def test_set_salary_in_1000(self):
    user = User("Angel", "1234", 1000, 0, 0)
    self.assertEqual(user.get_salary(), 1000)

def test_set_today_deposit_in_500(self):
    user = User("Angel", "1234", 1000, 500, 0)
    self.assertEqual(user.get_today_deposit(), 500)

def test_set_today_withdraw_in_750(self):
    user = User("Angel", "1234", 1000, 0, 750)
    self.assertEqual(user.get_today_withdraw(), 750)
```

Las pruebas unitarias restantes nos ayudarán a comprobar el correcto funcionamiento de la asignación de atributos y propiedades de la clase Usuario. Revisaremos una correcta inicialización del nombre, contraseña, cantidad de dinero, retiro y abono.

test_atm.py

new-atm > test_atm.py > ...

```python
import unittest
from user import User
from new_atm import NewATM

# Observaciones
# - Abstraer una entidad Usuario
# - Corregir el Hardcode: Contrasenia e Intentos
# - No poder retirar o depositar mas de S./3000 en un dia
# - Agregar alguna medida de segurida a la contrasenia
# - Agregar una clase Enum con las opciones del menu
# - A pesar de que falle 3 veces la contrasenia se muestra el menu para hacer operaciones
# - No permitir cantidades negativas
# Pruebas Unitarias:
# - Crear un nuevo Cajero con un Usuario que tenga S./5000 y verificar saldo
# - Al menos una prueba donde no te permita retirar mas de tu saldo
# - Al menos una prueba donde no te permita retirar mas de S./3000
# - Al menos una prueba donde no te permita depositar mas de S./3000
# - Al menos una prueba donde no te permita depositar cantidades negativas
# - Al menos una prueba donde no te permita retirar cantidades negativas

```

Ahora crearemos las pruebas unitarias para el Cajero Automático o ATM. En este caso necesitaremos verificar que recibe un Usuario ha atender, los movimientos bancarios deben ser cantidades positivas y no debe superarse el límite impuesto de 3000 unidades durante un solo día tanto en abonos como retiros.

```python
class TestNewAtm(unittest.TestCase):

    def test_set_invalid_type_user(self):
        with self.assertRaises(TypeError):
            atm = NewATM("MyUser")

    def test_set_invalid_value_user(self):
        with self.assertRaises(ValueError):
            user = User("Angel", "12", 1500, 0, 0)
            atm = NewATM(user)
        with self.assertRaises(ValueError):
            atm = NewATM(None)

    def test_set_an_user_salary_5000(self):
        user = User("Juan", "1234", 5000, 0, 0)
        atm = NewATM(user)
        self.assertEqual(atm.user.get_salary(), 5000)

    def test_invalid_type_on_withdraw(self):
        with self.assertRaises(TypeError): # Type Error on amount
            rUser = User("Rodrigo", "superPwd", 2000, 0, 0)
            atm = NewATM(rUser)
            atm.withdraw("amount")

    def test_invalid_value_on_withdraw(self):
        with self.assertRaises(ValueError): # Value Error on amount
            dUser = User("Darwin", "superPwd", 2000, 0, 0)
            atm = NewATM(dUser)
            atm.withdraw(-5000)
```

En los primeros 5 casos de prueba verificaremos algunas excepciones y errores de asignación que podrían darse al crear el Usuario del ATM así como al realizar un movimiento. Los movimientos bancarios o transaccionales deben ser enteros o flotantes positivos. Pruebas unitarias para el retiro.

```python
    def test_invalid_value_on_withdraw(self):...

    def test_withdraw_moreThan_limit(self):
        with self.assertRaises(Exception):
            user = user = User("Juan", "1234", 1000, 0, 2500)
            atm = NewATM(user)
            atm.withdraw(1000)

    def test_withdraw_moreThan_salary(self):
        with self.assertRaises(Exception):
            user = User("Juan", "1234", 1000, 0, 0)
            atm = NewATM(user)
            atm.withdraw(1500)

    def test_withdraw_negative_value(self):
        with self.assertRaises(ValueError):
            user = User("Juan", "1234", 2500, 0, 0)
            atm = NewATM(user)
            atm.withdraw(-1500)
```

Siguiendo con las pruebas unitarias para los retiros. No deben exceder el límite corporativo impuesto de 3000 unidades monetarias. Tampoco deben exceder el salario disponible por el Usuario y tampoco deben tomar valores negativos.

```python
def test_invalid_type_on_deposit(self):
    with self.assertRaises(TypeError): # Type Error on amount
        rUser = User("Rodrigo", "superPwd", 2000, 0, 0)
        atm = NewATM(rUser)
        atm.deposit("amount")

def test_invalid_value_on_deposit(self):
    with self.assertRaises(ValueError): # Value Error on amount
        dUser = User("Darwin", "superPwd", 2000, 0, 0)
        atm = NewATM(dUser)
        atm.deposit(-5000)

def test_deposit_moreThan_limit(self):
    with self.assertRaises(Exception):
        user = user = User("Juan", "1234", 1000, 2500, 0)
        atm = NewATM(user)
        atm.deposit(1000)

def test_deposit_negative_value(self):
    with self.assertRaises(ValueError):
        user = User("Juan", "1234", 2500, 0, 0)
        atm = NewATM(user)
        atm.deposit(-1500)
```

De igual manera se crean pruebas unitarias para la operación de depósito. Donde se verifica el tipo de dato ingresado así como no superar el límite establecido en 3000 unidades. Tampoco se aceptarán negativos.

2. *Implementación:*

user.py

new-atm > user.py > ...

```python
class User():

    # Constructor of User class

    def __init__(self, name, password, salary, today_deposit, today_withdraw):
        self.set_name(name)
        self.set_password(password)
        self.set_salary(salary)
        self.set_today_deposit(today_deposit)
        self.set_today_withdraw(today_withdraw)
```

En la implementación de la clase Usuario, delegamos la asignación de los parámetros o atributos a otros métodos setters que contemplarán algunos errores y excepciones que podrían presentarse.

user.py

new-atm > user.py > ...

```python
    def set_name(self, name):
        if not isinstance(name, str):
            raise isinstance("Password must be a set of characters.")
        if len(name) < 4:
            raise ValueError("User name must be greater than 4 characters.")
        self.name = name

    def set_password(self, password):
        if not isinstance(password, str):
            raise TypeError("Password must be set of characters.")
        if len(password) < 4:
            raise ValueError("Password must be greater than 4 characters.")
        self.password = password

    def set_salary(self, salary):
        if not isinstance(salary, float) and not isinstance(salary, int):
            raise TypeError("Salary must be an integer or float.")
        if salary < 0:
            raise ValueError("Salary must be positive.")
        self.salary = salary
```

Los métodos setters para el nombre y contraseña verificarán que el tipo de dato ingresado sea un string o conjunto de caracteres sino devolverá un error de tipado. Así mismo si no tienen una longitud mayor a 4 caracteres entonces devolverá un error de valor. El salario debe ser un entero o flotante positivo.

```python
def set_today_deposit(self, today_deposit):
    if not isinstance(today_deposit, int) and not isinstance(today_deposit, float):
        raise TypeError("Todays deposit amount must be an integer or float.")
    if today_deposit < 0:
        raise ValueError("Todays deposit amount must be positive.")
    self.today_deposit = today_deposit

def set_today_withdraw(self, today_withdraw):
    if not isinstance(today_withdraw, int) and not isinstance(today_withdraw, float):
        raise TypeError("Todays deposit amount must be an integer or float.")
    if today_withdraw < 0:
        raise ValueError("Todays withdraw amount must be positive.")
    self.today_withdraw = today_withdraw
```

También verificamos los valores que ingresan como valores iniciales para el monto de retiro y depósito diario. Tienen que tratarse de valores enteros o flotantes positivos.

atm_options.py

new-atm > atm_options.py > ...

```python
# -*- coding: utf-8 -*-
"""
@author: ahincho
@author: dneira
"""

from enum import Enum

class ATM_Options():
    DEPOSIT = 1
    WITHDRAW = 2
    SHOW_STATUS = 3
    EXIT = 4

```

Se hace uso de una clase Enumerador ATM_Options para especificar de mejor manera las opciones disponibles dentro del Cajero Automático y que sea más entendible en código.

atm_config.py

new-atm > atm_config.py > ...

```python
# -*- coding: utf-8 -*-
"""
@author: ahincho
@author: dneira
"""

from enum import Enum

class ATM_Config():
    BREAKLINE_CHARS = 25
    MAX_CHANCES = 3
    USERS_DB = "users.json"
    CURRENCY = "$"
    MAX_DEPOSIT = 3000
    MAX_WITHDRAW = 3000

```

También se crea una clase Enumeradora ATM_Config para guardar algunos valores útiles en la configuración inicial del Cajero y tener un mantenimiento a futuro más cómodo y flexible.

```python
import getpass as gp # We see some problems in differents IDEs
import json # To use and recover JSON files and data
from atm_options import ATM_Options as op
from atm_config import ATM_Config as cfg
from user import User

def breakLine():
    print("*" * cfg.BREAKLINE_CHARS)

class NewATM:

    def __init__(self, user):
        self.set_user(user)

    def set_user(self, user):
        if not isinstance(user, User):
            raise TypeError("You must add an User Type object to the ATM.")
        if user is None:
            raise ValueError("You must add a valid User.")
        try:
            self.user = user
        except TypeError:
            print("There is something wrong with the types of the User attributes. Please check it out.")
        except ValueError:
            print("There is something wrong with the values of the User attritubes. Please check it out.")

    def get_user(self):
        return self.user
```

Para la clase NewATM utilizaremos un método auxiliar que imprimirá un salto de línea formado por asteriscos. El Cajero trabajará con un Usuario y se tendrá que asignar previa evaluación de tipo y valor.

```python
def withdraw(self, amount_withdraw):
    if not isinstance(amount_withdraw, float) and not isinstance(amount_withdraw, int): # Type Error
        raise TypeError("The amount to withdraw must be an integer or a float.")
    if amount_withdraw < 0: # Value error
        raise ValueError("The amount to withdraw must be positive.")
    if (self.user.get_today_withdraw() + amount_withdraw) > cfg.MAX_WITHDRAW:
        print(f"You have withdraw {cfg.CURRENCY}{self.user.get_today_withdraw()} this day.")
        raise Exception(f"You cant withdraw more than {cfg.CURRENCY}{cfg.MAX_WITHDRAW} in the same day")
    if self.user.get_salary() < amount_withdraw:
        raise Exception("Amount to withdraw must be equals or less than your salary.")
    self.user.set_salary(self.user.get_salary() - amount_withdraw) # Removing the amount to withdraw
    self.user.set_today_withdraw(self.user.get_today_withdraw() + amount_withdraw) # Adding the amount to the withdraw day limit
    return self.user.get_salary()
```

Creamos el método withdraw() o retiro() que recibirá como argumento el valor o monto a retirar y verificará su tipo de dato y si se trata de un valor positivo. Luego verificará si no se excede el monto diario a retirar así como que se cuente con saldo suficiente para la operación. Finalmente hace el retiro.

```python
def deposit(self, amount_deposit):
    if not isinstance(amount_deposit, float) and not isinstance(amount_deposit, int): # Type Error
        raise TypeError("The amount to deposit must be an integer or a float.")
    if amount_deposit < 0: # Value error
        raise ValueError("The amount to deposit must be positive.")
    if (self.user.get_today_deposit() + amount_deposit) > cfg.MAX_DEPOSIT:
        print(f"You have deposit {cfg.CURRENCY}{self.user.get_today_deposit()} this day.")
        raise Exception(f"You cant desposit more than {cfg.CURRENCY}{cfg.MAX_DEPOSIT} in the same day")
    self.user.set_salary(self.user.get_salary() + amount_deposit) # Adding the amount to deposit
    self.user.set_today_deposit(self.user.get_today_deposit() + amount_deposit) # Adding the amount to the deposit day limit
    return self.user.get_salary()
```

De igual manera, para los abonos o depósitos verificaremos el tipo de dato así como su valor positivo. Luego revisaremos si se encuentra dentro del límite corporativo. Finalmente se hace el abono a la cuenta.

```python
def show_account_info(self):
    salary_ifo = f"Salary: {cfg.CURRENCY}{self.user.get_salary()}\n"
    today_deposit_info = f"Today deposit: {cfg.CURRENCY}{self.user.get_today_deposit()} amount\n"
    today_withdraw_info = f"Today withdraw: {cfg.CURRENCY}{self.user.get_today_withdraw()} amount"
    return salary_ifo + today_deposit_info + today_withdraw_info
```

Finalmente tenemos el método para mostrar el estado de la cuenta. Devolverá el salario total que posee la cuenta así como la cantidad depositada y retirada el día de hoy.

```python
def print_options(self):
    breakLine()
    deposit_label = f"{op.DEPOSIT}. Deposit.\n"
    withdraw_label = f"{op.WITHDRAW}. Withdraw.\n"
    status_label = f"{op.SHOW_STATUS}. Show Account Status.\n"
    exit_label = f"{op.EXIT}. Exit."
    print(deposit_label + withdraw_label + status_label + exit_label)
    breakLine()
```

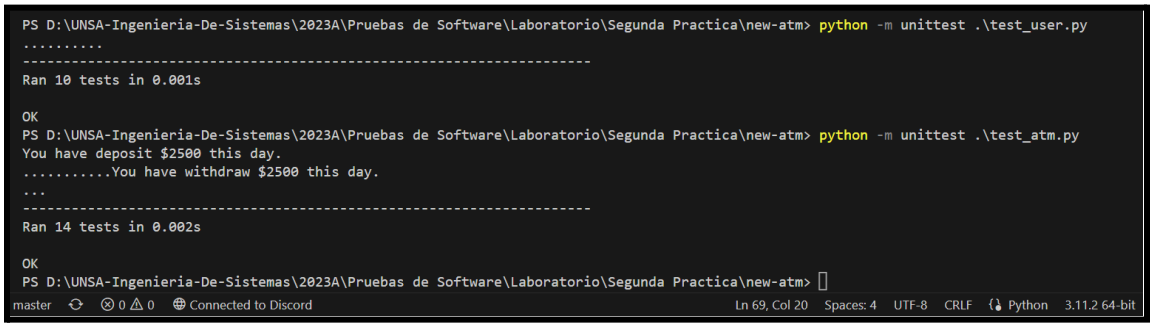
Adicionalmente vamos a imprimir el menú con las opciones disponibles dentro del Cajero Automático.

```python
def show_menu(self):
    while True:
        self.print_options()
        try:
            option = int(input("Select an Option: "))
            breakLine()
            match option:
                case op.DEPOSIT:
                    try:
                        amount_deposit = float(input(f"Amount to pay: {cfg.CURRENCY}"))
                        self.deposit(amount_deposit)
                    except Exception as e:
                        print("You have to write an postive amount to pay.")
                case op.WITHDRAW:
                    try:
                        amount_withdraw = float(input(f"Amount to withdraw: {cfg.CURRENCY}"))
                        self.withdraw(amount_withdraw)
                    except Exception as e:
```

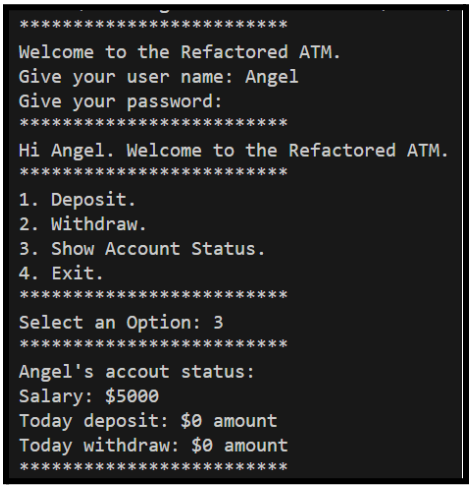
Finalmente generamos un método para mostrar el menú general en el cual el usuario podrá seleccionar una opción entre las implementadas para revisar su cuenta, depositar o retirar.

Se crea un archivo JSON para guardar datos relevantes sobre nuestros usuarios y poder dar una configuración inicial a nuestro Cajero Automático ATM.

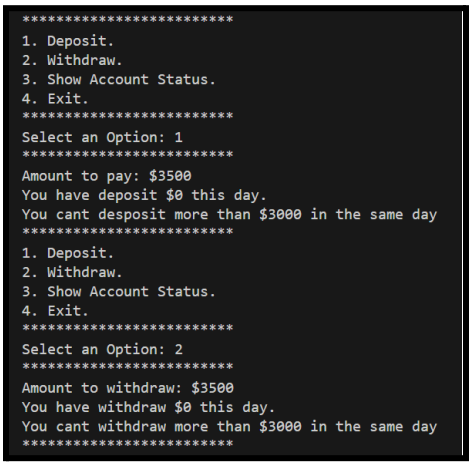
3. Ejecución:



Ejecutamos los casos de prueba y podemos ver como todos los casos de pruebas han sido superados con éxito. Esto nos da una primera barrera de seguridad y confianza en nuestro sistema.



Ingresando al Cajero Automático bajo el Usuario Angel. Revisamos el estado de la cuenta.



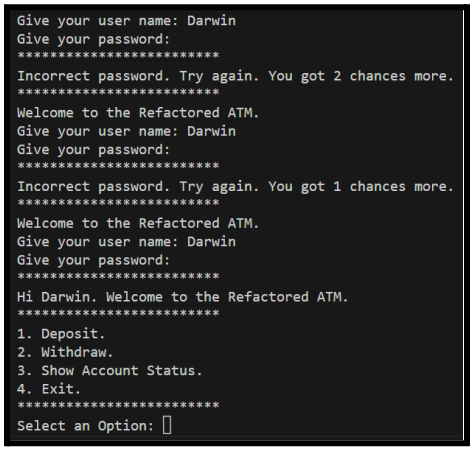
Tratamos de depositar y abonar más de 3000 unidades monetarias que es el límite corporativo establecido. Podemos ver como el sistema actúa de manera adecuada indicando que no es posible.

```
************************
1. Deposit.
2. Withdraw.
3. Show Account Status.
4. Exit.
************************
Select an Option: 1
************************
Amount to pay: $500
************************
1. Deposit.
2. Withdraw.
3. Show Account Status.
4. Exit.
************************
Select an Option: 3
************************
Angel's accout status:
Salary: $5500.0
Today deposit: $500.0 amount
Today withdraw: $0 amount
************************
```

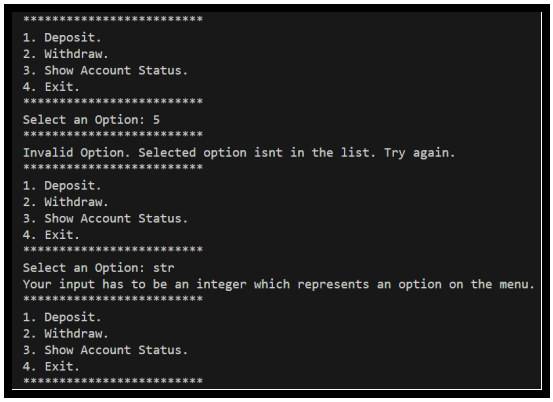
Realizamos un depósito válido y revisamos el estado de la cuenta.

```
************************
1. Deposit.
2. Withdraw.
3. Show Account Status.
4. Exit.
************************
Select an Option: 3
************************
Angel's accout status:
Salary: $4500.0
Today deposit: $500.0 amount
Today withdraw: $1000.0 amount
************************
1. Deposit.
2. Withdraw.
3. Show Account Status.
4. Exit.
************************
Select an Option: 4
************************
Goodbye Angel. See ya.
************************
Welcome to the Refactored ATM.
```

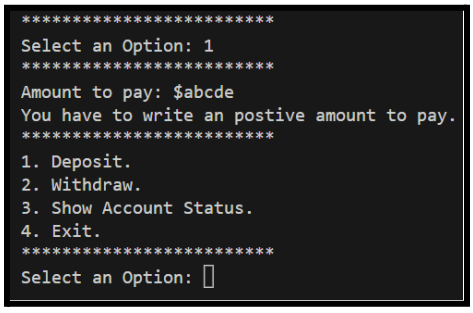
Realizamos un retiro válido y revisamos el estado de la cuenta.



Ingresando el Usuario Darwin con credenciales incorrectas hasta el último intento donde nos logueamos correctamente con las credenciales correspondientes.



En caso se ingrese un valor no entero en el menú entonces vamos a mostrar un mensaje que indique que solo aceptamos valores enteros mostrados en el menú.



Ahora intentamos abonar un valor no permitido, por ejemplo, escribimos un valor de tipo string. Vemos como el sistema se da cuenta de ello y muestra un mensaje concordante.

```
**********
Select an Option: 2
**********
Amount to withdraw: $-1000
You have to write an positive amount to withdraw.
**********
1. Deposit.
2. Withdraw.
3. Show Account Status.
4. Exit.
**********
```

Intentando ingresar un monto a retirar negativo. Esto es imposible y se muestra el mensaje adecuado.

```
**********
Select an Option: str
Your input has to be an integer which represents an option on the menu.
**********
1. Deposit.
2. Withdraw.
3. Show Account Status.
4. Exit.
**********
```

Intentamos ingresar una cadena de caracteres en el menú del Cajero Automático y vemos cómo responde correctamente manejando la excepción o error.

```
**********
Welcome to the Refactored ATM.
Give your user name: 123
Give your password:
**********
Incorrect password. Try again. You got 1 chances more.
**********
Welcome to the Refactored ATM.
Give your user name: 123
Give your password:
**********
Incorrect password. Try again. You got 0 chances more.
**********
You lost all the chances. We are sending a bank staff!
**********
```

Finalmente, cuando fallemos 3 veces al intentar ingresar un Usuario y contraseña el sistema se cerrará y se notificará que se ha enviado a un personal del banco para dar el soporte necesario.

II. SOLUCIÓN DEL CUESTIONARIO

No se compartió o publicó un Cuestionario para el presente Laboratorio.

III. CONCLUSIONES

Conclusión 1: *Durante el desarrollo del Laboratorio se utilizó TDD y se consideró varios casos de prueba de acuerdo al código base brindado para el Cajero Automático, estos casos sirvieron para hacer un mejor uso de la herramienta de unittest así como ver su utilidad al momento del desarrollo y especificación.*

Conclusión 2: *Utilizar unittest para automatizar pruebas unitarias es una excelente práctica de desarrollo. Permite verificar que cada componente de código funcione correctamente de forma aislada y que siga funcionando correctamente a medida que realizamos cambios en el código. Esto brinda confianza en la calidad y la estabilidad de tu código.*

Conclusión 3: *El manejo adecuado de errores y excepciones es crucial para garantizar la robustez y la confiabilidad del código. Es importante asegurarse que las pruebas unitarias cubran la mayor cantidad de casos posibles, incluyendo escenarios de éxito y situaciones de manejo de errores.*

RETROALIMENTACIÓN GENERAL

REFERENCIAS Y BIBLIOGRAFÍA

[1] "Unittest Unit testing framework". Python. https://docs.python.org/3/library/unittest.html .