Universidad de Camagüey “Ignacio Agramonte y Loynaz”

Facultad de Informática y Ciencias Exactas

1er.año Ing. Informática

Título: Sistema de control de cuentas bancarias

Equipo: 2

INFORME DE LA PRÁCTICA LABORAL

Autores:

Marcos Rodríguez Gerra

Alejandra Padrón Rivero

Marcos Fernández Pérez

Jorge Torres Soria

Curso: 2023.

Asignatura: Programación Orientada a Objeto.

Camagüey, diciembre de 2023

**Índice**:

[**Introducción** 1](#_Toc152901347)

[**Enunciado del problema a resolver** 1](#_Toc152901348)

[**Herramientas y tecnologías** 2](#_Toc152901349)

[**Desarrollo** 4](#_Toc152901350)

[**Análisis del problema** 4](#_Toc152901351)

[**Diagrama de clases UML** 4](#_Toc152901352)

[**Tipos de datos** 5](#_Toc152901353)

[**Descripción de métodos** 5](#_Toc152901354)

[**CRUD** 5](#_Toc152901355)

[**Depositar y retirar saldo** 7](#_Toc152901356)

[**Calcular interés de una cuenta** 7](#_Toc152901357)

[**Interés de todas las cuentas de plazo fijo en 5 años** 8](#_Toc152901358)

[**Listar cuentas de plazo fijo de la mas de 10,000 CUP** 8](#_Toc152901359)

[**Pruebas unitarias desarrolladas al modelo** 9](#_Toc152901360)

[**Banco** 9](#_Toc152901361)

[**Cliente y Comercial** 11](#_Toc152901362)

[**Cuentas** 11](#_Toc152901363)

[**Conclusiones:** 14](#_Toc152901364)

[**Recomendaciones:** 15](#_Toc152901365)

# **Introducción**

Un banco es una institución sobre la cual recae una gran responsabilidad debido a que están tratando constantemente con grandes sumas de dinero de muchas personas. El presente informe tiene como objetivo presentar una aplicación para el control de las cuentas bancarias, que tiene como finalidad facilitar la gestión y administración de las cuentas de los clientes. A través de este sistema se busca automatizar procesos, mejorar la eficiencia de la gestión de cuentas y garantizar la seguridad de la información.

**Abstract**

A bank is an institution on which a great responsibility falls because they are constantly dealing with large sums of money from many people. The objective of this report is to present an application for the control of bank accounts, which aims to facilitate the management and administration of customer accounts. Through this system we seek to automate processes, improve the efficiency of account management and guarantee the security of information.

## **Enunciado del problema a resolver**

Un Banco desea automatizar el control de cuentas bancarias. De las cuentas en general se conoce el número de cuenta, los datos del cliente, los datos del comercial que realizó la apertura de la cuenta, el saldo, el tipo de moneda (que puede ser CUP, CUC, USD o EUR), la fecha en que se abrió y la fecha en que se retiró la última vez. Además, se conoce que existen cuentas de formación de fondos de las cuales se conoce que tienen una cuota de incremento mensual que va a la cuenta directamente del salario de la persona; y que existen cuentas de plazo fijo que tienen un plazo dado en años que cuyo interés es superior al resto de las cuentas siempre que no se hagan extracciones en el plazo fijado. Por otro lado, del cliente se registra el nombre, el sexo, el carné de identidad, el centro de trabajo, la ocupación y el salario mensual. De los comerciales se registra el nombre, el sexo, el carné de identidad y los años de experiencia de trabajo en el banco. El sistema cuenta con una lista de cuentas, una lista de clientes y una lista de comerciales. El sistema debe permitir realizar las siguientes funcionalidades:

**a)** Implemente la funcionalidad necesaria para abrir (crear) y cerrar (eliminar) cuentas bancarias; además debe permitir listar los datos de las cuentas de cada tipo de manera independiente, así como gestionar (insertar, actualizar, eliminar y listar) los datos de los clientes y los comerciales.

**b)** Implemente la funcionalidad necesaria para calcular el interés de una cuenta dado su número, teniendo en cuenta que este se calcula de la siguiente forma:

• Para las cuentas simples el interés es de 4% por cada peso por año sin extraer.

• Para las cuentas de formación de fondo es de 6% por cada peso por año sin extraer.

• Para las cuentas de plazo fijo es de 8% por cada peso el primer año y aumenta 2% por cada peso por año de plazo fijo hasta los 5 años, que sería 16%.

**c)** Implemente la funcionalidad necesaria para hacer un depósito en una cuenta dado el número de la cuenta y el valor a depositar actualizando el saldo de la cuenta, tenga en cuenta que antes de cada depósito se calcula el interés a la cuenta y se le suma si no es a plazo fijo.

**d)** Implemente la funcionalidad necesaria para determinar a cuánto asciende el interés de todas las cuentas de plazo fijo que hay creadas en una fecha dada dentro de los próximos 5 años.

**e)** Implemente la funcionalidad necesaria para determinar los datos del propietario de la cuenta de mayor saldo estimado en este momento teniendo en cuenta que para estimar el saldo hay que tener en cuenta el interés acumulado.

**f)** Implemente la funcionalidad necesaria para determinar el listado los datos de las cuentas de plazo fijo de más de 10000 CUP que sean de un plazo dado y que se muestre el interés acumulado, ordenado ascendentemente por el número de la cuenta.

**g)** Pruebe que las operaciones implementadas en el modelo funcionan correctamente según los datos de prueba que usted le entró al programa.

## **Herramientas y tecnologías**

Lenguaje de programación:

* Python 3.11.1: Es un lenguaje de programación de alto nivel, interpretado y de propósito general. Es conocido por su sintaxis clara y legible, lo que lo hace ideal para principiantes en programación. Python es versátil y se utiliza en una amplia gama de aplicaciones, desde desarrollo web hasta inteligencia artificial y análisis de datos. Además, cuenta con una comunidad activa que ofrece una amplia gama de bibliotecas y herramientas que facilitan el desarrollo de software en Python

Herramientas para el desarrollo:

* Visual Studio Code 1.84.1: Es un editor de código fuente desarrollado por Microsoft que es altamente personalizable y ligero, pero potente. Funciona en múltiples plataformas y es muy popular entre los desarrolladores debido a sus múltiples extensiones y soporte para diversos lenguajes de programación. Ofrece características como resaltado de sintaxis, finalización de código, depuración integrada, control de versiones y una integración perfecta con herramientas y servicios de desarrollo populares.
* Qt Designer 5.11.1: Es una herramienta de diseño gráfico que forma parte del framework QT. Se utiliza para crear interfaces de usuario de manera visual para aplicaciones que utilizan QT, un framework de desarrollo de aplicaciones multiplataforma. QT Designer permite a los desarrolladores diseñar interfaces de usuario de forma rápida y sencilla, utilizando una variedad de widgets y elementos gráficos, y luego integrar estos diseños en sus aplicaciones escritas en C++ o Python, entre otros lenguajes compatibles.
* PyQt5: Es un conjunto de enlaces Python para la biblioteca de desarrollo de software QT. Proporciona una forma poderosa y flexible de crear interfaces gráficas de usuario utilizando el lenguaje de programación Python junto con las capacidades de la biblioteca QT. Con PyQt5, los desarrolladores pueden crear aplicaciones de escritorio multiplataforma que incluyan una amplia variedad de widgets y herramientas gráficas, lo que lo convierte en una opción popular para el desarrollo de interfaces de usuario.
* Visual Paradigm Community Edition 1.17.1: Es una herramienta de modelado de software que permite a los desarrolladores y diseñadores visualizar, diseñar y modificar el diseño de sus aplicaciones de manera rápida y efectiva.

# **Desarrollo**

Primero, como vamos a descomponer el problema desde el punto de vista de la programación orientada a objetos y dado que estamos tratando con conceptos como cuentas bancarias, clientes y comerciales, lo lógico es abordar esto primeramente desarrollando un diagrama de clases pues es una herramienta fundamental para visualizar y documentar la estructura estática de un sistema de software, en este caso nuestro Sistema de control de cuentas bancarias.

## **Análisis del problema**

Para resolver este problema, nos proponemos desarrollar un sistema que permita gestionar la información de las cuentas, clientes y comerciales del banco. El sistema debe contar con una interfaz intuitiva y fácil de usar para que los usuarios puedan realizar las operaciones de manera eficiente.

Entre las funcionalidades que se deben implementar en el sistema se encuentran la creación y eliminación de cuentas bancarias, la gestión de los datos de los clientes y comerciales, el cálculo de intereses para diferentes tipos de cuentas, la realización de depósitos y la determinación del saldo estimado y el interés acumulado.

Para probar que el sistema funciona correctamente, se deben realizar pruebas con diferentes conjuntos de datos de prueba que permitan verificar el correcto funcionamiento del sistema en diferentes escenarios y situaciones. De esta manera, se puede asegurar que el sistema es confiable y eficiente en su desempeño.

### **Diagrama de clases UML**

### **Tipos de datos**

En este proyecto se utilizan principalmente 4 tipos de datos:

1. Números: Estos incluyen enteros (int) y números de punto flotante (float). Se utilizan principalmente para representar valores numéricos y realizar cálculos.
2. Cadenas de caracteres (strings): Estos se utilizan para representar texto. En este proyecto, se utilizan para almacenar y manipular datos como nombres, centros de trabajo, ocupaciones, etc.
3. Fechas (date): Es una estructura de datos en Python que representa fechas y horas en el tiempo. Esta estructura se encuentra en el módulo “datetime”, que forma parte de la biblioteca estándar de Python.
4. Objetos: En este proyecto, se utilizan objetos para representar entidades del mundo real como clientes, cuentas, comerciales, etc. Estos objetos contienen atributos y métodos que permiten representar y manipular el estado y el comportamiento de las entidades.

## **Descripción de métodos**



### **CRUD**

#### **Create**

Ingresar clientes y comerciales:

Estas funciones se encargan de ingresar un nuevo cliente o comercial en el repositorio. Verifica que el cliente o comercial no existan en el repositorio antes de ingresarlos, validando que el ci no se repita en sus respectivas listas y que además sea válido.

Ingresar cuentas simples, de plazo fijo y de formación de fondos:

Estas funciones se encargan de ingresar los distintos tipos de cuentas en el repositorio. Verifica que las cuentas no existan en el repositorio antes de ingresarlas, verificando que el cliente y el comercial de las cuentas a ingresar existe en el repositorio, de no existir, lanza un error. Además, también comprueban que el número de la cuenta a ingresar no se repita en ninguno de los tipos de cuentas del repositorio, guardando el resultado de buscar el número de cuenta en cada lista existente, si no devuelve None entonces el número ya existen en el repositorio, y por ultimo verifica que no exista ninguna cuenta con el número de cuenta de la nueva cuenta a ingresar, de existir lanza un error.

#### **Read**

Cada una de estas funciones devuelve una lista que contiene los objetos de los diferentes modelos que se almacenan en el repositorio. Además, estas funciones se convierten en atributos de lectura en la clase **Banco** debido a que el decorador **@property** se coloca antes de cada una de ellas. Esto significa que no se pueden modificar directamente desde fuera de la clase, lo que asegura el principio de encapsulación y evita que la información almacenada en las listas sea modificada incorrectamente

#### **Update**

Actualizar clientes y comerciales:

En este caso las funciones se encargan de actualizar un cliente o comercial del repositorio, a partir de su número de identificación (ci), primero validando ese identificador usando funciones auxiliares, luego busca el índice del cliente o comercial por el “ci”, de no encontrarlo lanza un error, después se busca si el “ci” del nuevo cliente ya existe en el repositorio y si el “ci” del cliente a modificar y el del nuevo cliente son iguales, entonces quiere decir que no se modificó el atributo “ci” y entonces el nuevo cliente o comercial sobrescribe al anterior, pero si el “ci” del nuevo cliente o comercial no existe en el repositorio entonces quiere decir que se modificará el “ci” del anterior, por tanto se actualizan los identificadores de todas las cuentas que contenían al anterior cliente o comercial con el identificador del nuevo, y de existir el “ci” del nuevo cliente o comercial y no ser el mismo que el del anterior, entonces quiere decir que ya existe otro cliente o comercial con el “ci” del nuevo y se lanza un error.

Actualizar cuentas simples, de plazo fijo y formación de fondos:

Las funciones de actualizar cuentas reciben como parámetro el número de la cuenta a modificar y la nueva cuenta, primeramente, valida ese identificador usando funciones auxiliares, para verificar que la cuenta a modificar existe en el repositorio, de no existir lanza un error, verifica que el comercial existe en el repositorio, de no existir lanza un error, y no es necesario comprobar el cliente ya que este no se puede cambiar una vez creada la cuenta, luego se busca si el número de la nueva cuenta se repite en alguna de las listas de cuentas del repositorio, para después comprobar si existencuentas de otro tipo con el mismo número que el de la nueva cuenta, de existir lanza un error, si el número de la cuenta nueva es el mismo que el de la cuenta anterior o no existe ninguna otra cuenta con el mismo número, se sobrescribe la cuenta anterior con la cuenta nueva y por ultimo si existe otra cuenta del mismo tipo con el mismo número que el de la cuenta nueva y no es la cuenta anterior, entonces se lanza un error.

#### **Delete**

Eliminar clientes y comerciales:

Las funciones de eliminar clientes y comerciales del repositorio, lo hacen a partir de su número de identificación (ci) y a través de un filter se sobrescribe su respectiva lista, dejando solo las cuentas que cumplan con el requisito de que su identificador no sea el mismo que el del parámetro.

. Pero la función de eliminar cliente tiene la particularidad de que también elimina todas las cuentas asociadas a dicho cliente

Eliminar cuentas simples, de plazo fijo y de formación de fondos:

Las funciones para eliminar los distintos tipos de cuentas reciben el identificador, en este caso es el número de cuenta es similar a los clientes y comerciales, a través de un filter se sobrescribe su respectiva lista, dejando solo las cuentas que cumplan con el requisito de que su identificador no sea el mismo que el del parámetro.

### **Depositar y retirar saldo**

El método **depositar y retirar** en la clase **Banco**funcionan muy parecidos, y realiza lo siguiente:

1. Se busca el índice de la cuenta cuyo número es igual al parámetro **num** utilizando funciones auxiliares. Este método devuelve el índice de la cuenta simple que cumple con la condición, o **None** si no se encuentra ninguna cuenta.
2. Verifica si el índice obtenido en es diferente de **None**. Esto indica que se encontró una cuenta cuyo número es igual al parámetro **num**.
3. Si se encontró, se suma o resta en dependencia de la función que se ejecute el parámetro **saldo** a la cantidad de efectivo de la cuenta. En caso de la función de Retirar se verifica que el saldo de la cuenta sea mayor o igual que la cantidad que se desea retirar, de lo contrario se lanza un error.
4. Si no se encontró ninguna cuenta que cumpla con el número de parámetro **num**, se lanza una excepción indicando que "La cuenta no existe en el repositorio".

### **Calcular interés de una cuenta**

La función **calcularInteresxNum** calcula el interés que ha generado una cuenta concreta, basándose en el saldo y las condiciones específicas de la cuenta (como el plazo). La función toma como parámetro el número de la cuenta a la que se quiere calcular el interés.

El código de la función realiza los siguientes pasos:

1. Comprueba si el número de cuenta dado corresponde a una cuenta simple, una cuenta de ahorro con plazo fijo (PF) o una cuenta de ahorro con fecha fija (FF). Para ello, busca el índice de la cuenta en las listas correspondientes utilizando funciones auxiliares. En caso de que el número de cuenta no corresponda a ninguna cuenta en las listas del repositorio, la función lanza una excepción indicando que la cuenta no existe en el repositorio.
2. Si se encuentra una cuenta que corresponda al número de cuenta dado, la función utiliza el método **calcularInteres** de la clase correspondiente (CuentaSimple, CuentaPF o CuentaFF) para calcular el interés generado por la cuenta.
3. Finalmente, la función devuelve el interés calculado y el atributo tipo\_moneda de la cuenta.

Esta función permite obtener el interés generado por una cuenta específica de las listas de cuentas del repositorio, sin tener que conocer los detalles de la cuenta ni del cliente.

### **Interés de todas las cuentas de plazo fijo en 5 años**

El método **interesPF5Anios** en la clase **Banco** es responsable de calcular el interés que genera cada cuenta PF de una fecha de creación dada al cabo de 5 años, bajo el supuesto de que solo se aplica el interés de la cuenta en la cantidad de años acordados en el plazo.

1. Este método recibe la fecha de creación de las cuentas y utiliza un bucle **for** para recorrer cada cuenta de plazo fijo en la lista **listaCuentaPF** y obtener solo, las que correspondan con la fecha de creación, sino se encuentra ninguna, se lanza un error.
2. A continuación con un bucle **for** se cambiantodas las fechas de último retiro para simular que han pasado 5 años, entonces el método utiliza la función **map()** y **lambda x** para obtener con el método **interes()** de la clase **CuentaPF**, el interés de cada cuenta en la lista de cuentas PF obtenida en el paso anterior. Luego con la función **list()** se convierten los resultados en una lista.
3. Una vez hecho todo lo anterior, con otro bucle **for** se regresan las fechas de último retiro a la normalidad.
4. Por último el método realiza el **return** de una tupla con la lista de intereses y la lista de cuentas PF obtenida en el paso 1.

### **Listar cuentas de plazo fijo de más de 10,000 CUP**

El método **cuentasPFmas10MilCUP** es responsable de obtener las cuentas de plazo fijo que tienen un saldo mayor a 10,000 CUP.

1. Se define una variable llamada **lista\_cuentas\_pf\_mas\_10\_mil\_pesos** para almacenar las cuentas PF que cumplen con la condición de tener un saldo mayor a 10,000 CUP.
2. Se utiliza un bucle **for** para recorrer cada cuenta PF en la lista **listaCuentaPF** de la clase **Banco**.
3. Para cada cuenta de plazo fijo, se calcula su saldo en CUP y se guarda en el atributo **saldo\_cup** de la cuenta.
4. Se verifica si el saldo de la cuenta en CUP es mayor a 10,000. Si se cumple esta condición, se agrega la cuenta a la lista **lista\_cuentas\_pf\_mas\_10\_mil\_pesos**.
5. Si la lista **lista\_cuentas\_pf\_mas\_10\_mil\_pesos** resulta vacía después de filtrar todas las cuentas de plazo fijo, se lanza una excepción indicando que no existen cuentas de plazo fijo con más de 10,000 CUP en el repositorio.
6. Finalmente, se ordena la lista **lista\_cuentas\_pf\_mas\_10\_mil\_pesos** en orden ascendente utilizando el atributo **num\_cuenta** de las cuentas PF como criterio de ordenamiento con la función **sorted()**.
7. Se devuelve la lista **lista\_cuentas\_pf\_mas\_10\_mil\_pesos**, que contiene las cuentas de plazo fijo filtradas según la condición de tener un saldo mayor a 10,000 CUP.

## **Pruebas unitarias desarrolladas al modelo**

Se deben crear pruebas unitarias para cada función o método del proyecto. Estas pruebas deben incluir casos de prueba positivos (casos en los que se espera que la función funcione correctamente) y casos de prueba negativos (casos en los que se espera que la función no funcione correctamente). Para las pruebas se utilizan 2 funciones en específico, assertEqual, que es una función de prueba que verifica si dos valores son iguales y si no son iguales, se lanza una excepción de error de prueba y assertRaises, que es una función para verificar si el programa arroja la excepción correcta cuando se pasan argumentos no válidos a una función determinada. Todas las siguientes pruebas unitarias ayudan a garantizar que el código esté funcionando correctamente y que pueda ser utilizado en un entorno real.

### **Banco**

Para desarrollar una serie de pruebas unitarias para la clase **Banco**las clases **Cliente**, **Comercial**, **CuentaSimple**, **CuentaFF** y **CuentaPF**. El objetivo principal de las pruebas es verificar que todas las funciones de la clase **Banco** estén funcionando correctamente. Para ello, se realizan pruebas en los siguientes aspectos:

1. **Prueba de propiedades**: Se verifica que todas las listas que forman parte de la clase **Banco** estén vacías al inicio de las pruebas.
2. **Prueba de ingresar**: Se prueba que se puedan ingresar correctamente clientes, comerciales y cuentas en el banco. Se verifica que no se puedan ingresar cuentas ni clientes con el mismo número de identificación o “ci”. Además, se comprueba que no se puedan ingresar cuentas con “ci” que no existen en el repositorio.
3. **Prueba de eliminar**: Se verifica que se puedan eliminar correctamente clientes, comerciales y cuentas del banco. Al eliminar un cliente, también se eliminan las cuentas asociadas a ese cliente.
4. **Prueba de actualizar**: Se prueba que se puedan actualizar correctamente los datos de clientes, comerciales y cuentas en el banco. Se verifica que no se pueda actualizar una cuenta cuyo número ya existe en una cuenta de otro tipo.
5. **Prueba de guardar y cargar BD**: Se prueba que se puedan guardar y cargar correctamente las listas de clientes, comerciales, cuentas simples, cuentas a plazo fijo y cuentas a plazo fijo en los archivos de texto correspondientes.
6. **Prueba de funciones auxiliares**: Se prueban funciones auxiliares como buscar clientes, comerciales y cuentas en el banco, validar los códigos de identificación (CI) y actualizar los CI de comerciales y clientes en cuentas.
7. **Prueba de calcular intereses por número**: Se prueba que se pueda calcular correctamente el interés generado por el número de cuenta en el banco.
8. **Prueba de depositar y retirar:** Se prueban las funcionalidades de depositar y retirar saldo en todos los tipos de cuentas y en las que es necesario se calcula el interés y se suma manualmente al saldo, y por ultimo prueba que lance un error al no encontrar la cuenta.
9. **Prueba de calcular interés a cuentas de plazo fijo en 5 años:** Primero, se evalúa si el método genera una excepción. En este caso, si no hay cuentas de plazo fijo en el banco, se espera que se genere una excepción, lo que demuestre que la prueba está funcionando correctamente. Luego, se crean varias cuentas de distintos tipos y se agregan al banco. También se modifica el valor del atributo **num\_cuenta** de una de las cuentas de plazo fijo para que cumpla con los requisitos. Después, se ejecuta el método **interesPF5Anios()**y se espera que su resultado sea igual a una lista específica de valores, si los resultados coinciden, la prueba ha sido exitosa.
10. **Prueba de buscar la cuenta de mayor saldo:** Al inicio de la prueba, se ejecuta el método **cuentaMayorSaldo()** y se espera que genere una excepción, ya que no existen cuentas en el banco. Si se lanza una excepción, la prueba ha sido exitosa. Luego, se crean varias cuentas de los diferentes tipos y se agregan al banco, para comprobar si el método **cuentaMayorSaldo()** retorna la cuenta con el saldo mayor, se cambia el atributo **tipo\_moneda** de cada cuenta. Se verifica si el número de cuenta de la cuenta retornada por el método **cuentaMayorSaldo()** coincide con el número de cuenta de la cuenta esperada a través del método **assertEqual()**. Si todos los casos de prueba funcionan correctamente, la prueba unitaria ha sido exitosa.
11. **Prueba de listar cuentas de plazo fijo con más de 10,000 CUP:** Primero, se ingresan algunos clientes, cuentas comerciales y cuentas de plazo fijo en el banco. Luego, se verifica si el método **cuentasPFmas10MilCUP** arroja una excepción cuando no hay cuentas de plazo fijo con un saldo superior a 10 mil pesos. A continuación, se modifica una cuenta de plazo fijo para que tenga un saldo superior a 10,000 pesos y se ingrese en el banco. Además, se modifica el tipo de moneda de otra cuenta de plazo fijo para que también cumpla con la condición de tener un saldo superior a 10,000 pesos. Después de ingresar las cuentas de plazo fijo, se llama al método **cuentasPFmas10MilCUP** para obtener las cuentas de plazo fijo con saldo superior a 10 mil pesos y finalmente, se comprueba si las cuentas de plazo fijo obtenidas están ordenadas ascendentemente por su número de cuenta. Para ello, se compara el primer número de cuenta con el segundo y el tercero. Si el primer número de cuenta es menor que el segundo y el tercero, entonces están ordenados correctamente

### **Cliente y Comercial**

Estas pruebas unitarias se realizan sobre el método "property\_and\_setter" de las clases "Cliente" y “Comercial”. El objetivo de estas pruebas es verificar que las propiedades y los setters de las clases funcionen correctamente.

Para hacer esto, se crea una instancia de la clase correspondiente en el método "setUp". Luego, se utilizan los assertEqual para verificar que los valores de las propiedades sean iguales a los valores esperados. Además, se cambian los valores de las propiedades utilizando los setters y se verifica nuevamente que los valores sean correctos.

### **Cuentas**

En estas pruebas unitariasse valida el comportamiento de las clases de los distintos tipos de cuentas.

Las pruebas unitarias se basan en los siguientes conceptos:

1. Configuración: Se crea una instancia de la clase y se definen algunas propiedades básicas en el método **setUp**. Este método se ejecuta antes de cada prueba.
2. Pruebas individuales: Se definen varios métodos de prueba individuales para verificar que el cálculo de intereses, la operación de depósito, la operación de retiro y la conversión de saldo a CUP funcionen correctamente.
3. Aserciones: Las aserciones son las pruebas en sí. En el caso de Python, las aserciones se implementan utilizando el módulo **unittest**. Se utilizan aserciones para verificar que el resultado de cada operación sea el esperado.

El código de la prueba unitaria realiza las siguientes operaciones:

1. Verifica que las propiedades básicas de la cuenta bancaria, como el número de cuenta, el cliente, los datos comerciales, el saldo, la moneda y las fechas, se hayan configurado correctamente en el método **test\_property\_and\_setter**.
2. Verifica que el cálculo de intereses funcione correctamente en el método **test\_calcular\_interes**.
3. Verifica que las operaciones de depósito y retiro funcionen correctamente en el método **test\_depositar\_retirar**. Además, verifica que la fecha del último retiro se actualice correctamente.
4. Verifica que la conversión de saldo a CUP funcione correctamente en el método **test\_calcular\_saldo\_CUP**. Se prueban diferentes conversiones, como la conversión de CUP a CUP, CUC a CUP, EUR a CUP y USD a CUP.
5. En el caso particular de las cuentas de plazo fijo, además, en esta clase se definen nuevas pruebas específicas para la clase, como la prueba **test\_property\_and\_setter** verifica que las propiedades básicas que se hayan en esta cuenta que no están en la clase padre, al igual que la prueba **test\_calcular\_interes** verifica que el cálculo de intereses funcione correctamente ya que en esta clase tiene su variante, y por ultimo con la prueba **test\_depositar\_retirar** verifica que las operaciones de depósito y retiro funcionen correctamente en la clase con la variante que además, en esta prueba, se verifica que si se cumple el plazo establecido para la cuenta, se suma el interés al saldo.
6. Por último también está el caso de las cuentas de formación de fondos, que hereda de la cuenta simple, pero agrega parámetros y modifica métodos que también hay que probar su funcionamiento. En este caso ya cuenta con un método **setUp** que crea una instancia de la clase **CuentaSimple** para las pruebas. Sin embargo, esta clase necesita crear una instancia de su clase en lugar de la clase padre. Por lo tanto, define su propio método **setUp** que crea una instancia, hereda también los métodos de prueba de la clase padre, pero también define sus propios métodos de prueba específicos para la clase como, la prueba **test\_calcular\_interes** verifica que el cálculo de intereses funcione correctamente en esta clase, y la prueba **test\_depositar\_retirar** verifica que las operaciones de depósito y retiro funcionen correctamente en la clase de cuentas de formación de fondos. Además, en esta prueba, se verifica que, si se cumple el plazo establecido para la cuenta, se suma el interés al saldo.

# **Conclusiones:**

Durante el desarrollo de este proyecto, se ha logrado diseñar e implementar un Sistema de Control de Cuentas Bancarias utilizando Programación Orientada a Objetos (POO) en Python. Se han aplicado conceptos clave de POO, como la definición de clases, encapsulamiento, herencia y polimorfismo para representar entidades del mundo real como clientes, comerciales y diferentes tipos de cuentas bancarias. La implementación de pruebas unitarias ha permitido validar el comportamiento de las funciones y métodos desarrollados, garantizando la fiabilidad y precisión del sistema.

Además, se ha empleado un enfoque metodológico sólido que ha permitido descomponer el problema, desarrollar un diagrama de clases UML y definir un conjunto claro de funcionalidades a través de las cuales se ha puesto a prueba la robustez del sistema.

Durante este proceso, se ha adquirido experiencia en el diseño e implementación de sistemas orientados a objetos, así como en el uso de herramientas como Visual Studio Code, Qt Designer y PyQt5 para el desarrollo de interfaces gráficas de usuario.En resumen, el proyecto ha representado un desafío significativo que ha permitido adquirir valiosas habilidades en el desarrollo de sistemas orientados a objetos y la implementación de soluciones en Python.

# **Recomendaciones:**

Como una posible mejora, se sugiere considerar la implementación de un sistema de autenticación y autorización, que permita la gestión de diferentes niveles de permisos de usuario para garantizar la seguridad y la privacidad de la información. Asimismo, se podría explorar la integración con sistemas de reporte y análisis para brindar a los usuarios una visión más detallada y analítica de las operaciones financieras.

Además, la posibilidad de implementar notificaciones automáticas, tanto para los clientes como para los administradores, podría mejorar la interacción y la comunicación entre el sistema y los usuarios.

**Bibliografía**

-Documentación de PyQt5

<https://www.riverbankcomputing.com/static/Docs/PyQt5/>

-Documentación de Python

<https://docs.python.org/3/>

-Qt Assistant versión 5.15.3