Enunciado:
Tu tarea es desarrollar una aplicación informática utilizando la técnica TDD para gestionar una cuenta bancaria. La aplicación debe permitir a los usuarios abrir una cuenta, realizar depósitos, hacer retiros y transferir fondos entre cuentas. A continuación se detallan las etapas de desarrollo utilizando TDD:

Etapa 1: Especificación y prueba inicial

1. Especifica los requisitos básicos del sistema y las funcionalidades clave, como la apertura de cuenta, depósito de fondos, retiro de fondos y transferencia de fondos.

   Requisitos básicos del sistema
   La aplicación de gestión de cuenta bancaria debe cumplir con los siguientes requisitos:
   1) Apertura de cuenta:
      a) Debe ser posible abrir una nueva cuenta bancaria.
      b) Cada cuenta debe tener un saldo inicial de 0.
   2) Depósito de fondos:
      a) Debe ser posible realizar depósitos en la cuenta bancaria.
      b) El saldo de la cuenta debe actualizarse correctamente después del depósito.
   3) Retiro de fondos:
      a) Debe ser posible realizar retiros de la cuenta bancaria.
      b) El saldo de la cuenta debe actualizarse correctamente después del retiro.
      c) No debe ser posible retirar más fondos de los disponibles en la cuenta.
   4) Transferencia de fondos:
      a) Debe ser posible transferir fondos entre dos cuentas bancarias.
      b) Los saldos de ambas cuentas deben actualizarse correctamente después de la transferencia.
      c) No debe ser posible transferir más fondos de los disponibles en la cuenta origen.

2. Escribe una prueba inicial que verifique si el sistema puede crear una instancia de una cuenta bancaria y obtener su saldo inicial.

   Prueba inicial
   Para empezar con el desarrollo utilizando TDD, primero escribo una prueba que verifique si el sistema puede crear una instancia de una cuenta bancaria y obtener su saldo inicial.

   Implementación de la prueba inicial
   Primero, creo una prueba que falla, ya que aún no he implementado la funcionalidad.

```
import unittest
```

```
class CuentaBancaria:
    def __init__(self):
        self.saldo = 0

    def obtener_saldo(self):
        return self.saldo

class TestCuentaBancaria(unittest.TestCase):
    def test_crear_cuenta(self):
        cuenta = CuentaBancaria()
        self.assertEqual(cuenta.obtener_saldo(), 0)

if __name__ == '__main__':
    unittest.main()
```

Próximos pasos
Después de esta etapa inicial, continuaré con la implementación de las siguientes funcionalidades y sus respectivas pruebas:

- Implementar y probar depósitos de fondos.
- Implementar y probar retiros de fondos.
- Implementar y probar transferencias de fondos entre cuentas.

Etapa 2: Desarrollo de las funcionalidades básicas

3. Implementa la funcionalidad para abrir una cuenta bancaria, asegurándote de que secumplan los requisitos especificados. Ejecuta la prueba y verifica que pase correctamente.

```
import unittest

class CuentaBancaria:
    def __init__(self):
        self.saldo = 0

    def obtener_saldo(self):
        return self.saldo

class TestCuentaBancaria(unittest.TestCase):
    def test_crear_cuenta(self):
        cuenta = CuentaBancaria()
```

```
        self.assertEqual(cuenta.obtener_saldo(), 0)

if __name__ == '__main__':
    unittest.main()
```

4. Implementa la funcionalidad para realizar depósitos en una cuenta bancaria. Ejecuta las pruebas y verifica que pasen correctamente.

Modifico la clase CuentaBancaria para incluir un método depositar y añado una nueva prueba:

```
import unittest

class CuentaBancaria:
    def __init__(self):
        self.saldo = 0

    def obtener_saldo(self):
        return self.saldo

    def depositar(self, monto):
        self.saldo += monto

class TestCuentaBancaria(unittest.TestCase):
    def test_crear_cuenta(self):
        cuenta = CuentaBancaria()
        self.assertEqual(cuenta.obtener_saldo(), 0)

    def test_depositar_fondos(self):
        cuenta = CuentaBancaria()
        cuenta.depositar(100)
        self.assertEqual(cuenta.obtener_saldo(), 100)

if __name__ == '__main__':
    unittest.main()
```

5. Implementa la funcionalidad para realizar retiros de una cuenta bancaria. Ejecuta laspruebas y verifica que pasen correctamente.

Modifico la clase CuentaBancaria para incluir un método retirar y añado una nueva prueba:

```
import unittest
```

```python
class CuentaBancaria:
    def __init__(self):
        self.saldo = 0

    def obtener_saldo(self):
        return self.saldo

    def depositar(self, monto):
        self.saldo += monto

    def retirar(self, monto):
        if monto <= self.saldo:
            self.saldo -= monto
            return True
        else:
            return False

class TestCuentaBancaria(unittest.TestCase):
    def test_crear_cuenta(self):
        cuenta = CuentaBancaria()
        self.assertEqual(cuenta.obtener_saldo(), 0)

    def test_depositar_fondos(self):
        cuenta = CuentaBancaria()
        cuenta.depositar(100)
        self.assertEqual(cuenta.obtener_saldo(), 100)

    def test_retirar_fondos(self):
        cuenta = CuentaBancaria()
        cuenta.depositar(100)
        result = cuenta.retirar(50)
        self.assertTrue(result)
        self.assertEqual(cuenta.obtener_saldo(), 50)
        result = cuenta.retirar(60)
        self.assertFalse(result)
        self.assertEqual(cuenta.obtener_saldo(), 50)

if __name__ == '__main__':
    unittest.main()
```

6. Implementa la funcionalidad para transferir fondos entre cuentas bancarias. Ejecuta laspruebas y verifica que pasen correctamente.

Modifico la clase CuentaBancaria para incluir un método transferir y añado una nueva prueba:

```
import unittest

class CuentaBancaria:
    def __init__(self):
        self.saldo = 0

    def obtener_saldo(self):
        return self.saldo

    def depositar(self, monto):
        self.saldo += monto

    def retirar(self, monto):
        if monto <= self.saldo:
            self.saldo -= monto
            return True
        else:
            return False

    def transferir(self, monto, cuenta_destino):
        if self.retirar(monto):
            cuenta_destino.depositar(monto)
            return True
        else:
            return False

class TestCuentaBancaria(unittest.TestCase):
    def test_crear_cuenta(self):
        cuenta = CuentaBancaria()
        self.assertEqual(cuenta.obtener_saldo(), 0)

    def test_depositar_fondos(self):
        cuenta = CuentaBancaria()
        cuenta.depositar(100)
        self.assertEqual(cuenta.obtener_saldo(), 100)

    def test_retirar_fondos(self):
        cuenta = CuentaBancaria()
```

```
        cuenta.depositar(100)
        result = cuenta.retirar(50)
        self.assertTrue(result)
        self.assertEqual(cuenta.obtener_saldo(), 50)
        result = cuenta.retirar(60)
        self.assertFalse(result)
        self.assertEqual(cuenta.obtener_saldo(), 50)

    def test_transferir_fondos(self):
        cuenta_origen = CuentaBancaria()
        cuenta_destino = CuentaBancaria()
        cuenta_origen.depositar(100)
        result = cuenta_origen.transferir(50, cuenta_destino)
        self.assertTrue(result)
        self.assertEqual(cuenta_origen.obtener_saldo(), 50)
        self.assertEqual(cuenta_destino.obtener_saldo(), 50)
        result = cuenta_origen.transferir(60, cuenta_destino)
        self.assertFalse(result)
        self.assertEqual(cuenta_origen.obtener_saldo(), 50)
        self.assertEqual(cuenta_destino.obtener_saldo(), 50)

if __name__ == '__main__':
    unittest.main()
```

Etapa 3: Pruebas adicionales y mejoras

7. Escribe pruebas adicionales para cubrir casos de prueba específicos, como intentarretirar más dinero del disponible en una cuenta o transferir fondos a una cuentainexistente.

Modifico la clase CuentaBancaria y las pruebas para incluir estos casos adicionales:

```
import unittest

class CuentaBancaria:
    def __init__(self):
        self.saldo = 0

    def obtener_saldo(self):
        return self.saldo

    def depositar(self, monto):
```

```
        if monto > 0:
            self.saldo += monto

    def retirar(self, monto):
        if monto <= self.saldo and monto > 0:
            self.saldo -= monto
            return True
        else:
            return False

    def transferir(self, monto, cuenta_destino):
        if self.retirar(monto):
            cuenta_destino.depositar(monto)
            return True
        else:
            return False

class TestCuentaBancaria(unittest.TestCase):
    def test_crear_cuenta(self):
        cuenta = CuentaBancaria()
        self.assertEqual(cuenta.obtener_saldo(), 0)

    def test_depositar_fondos(self):
        cuenta = CuentaBancaria()
        cuenta.depositar(100)
        self.assertEqual(cuenta.obtener_saldo(), 100)

    def test_retirar_fondos(self):
        cuenta = CuentaBancaria()
        cuenta.depositar(100)
        result = cuenta.retirar(50)
        self.assertTrue(result)
        self.assertEqual(cuenta.obtener_saldo(), 50)
        result = cuenta.retirar(60)
        self.assertFalse(result)
        self.assertEqual(cuenta.obtener_saldo(), 50)

    def test_transferir_fondos(self):
        cuenta_origen = CuentaBancaria()
        cuenta_destino = CuentaBancaria()
        cuenta_origen.depositar(100)
        result = cuenta_origen.transferir(50, cuenta_destino)
```

```
        self.assertTrue(result)
        self.assertEqual(cuenta_origen.obtener_saldo(), 50)
        self.assertEqual(cuenta_destino.obtener_saldo(), 50)
        result = cuenta_origen.transferir(60, cuenta_destino)
        self.assertFalse(result)
        self.assertEqual(cuenta_origen.obtener_saldo(), 50)
        self.assertEqual(cuenta_destino.obtener_saldo(), 50)

    def test_retirar_monto_negativo(self):
        cuenta = CuentaBancaria()
        cuenta.depositar(100)
        result = cuenta.retirar(-50)
        self.assertFalse(result)
        self.assertEqual(cuenta.obtener_saldo(), 100)

    def test_depositar_monto_negativo(self):
        cuenta = CuentaBancaria()
        cuenta.depositar(-100)
        self.assertEqual(cuenta.obtener_saldo(), 0)

    def test_transferir_a_cuenta_inexistente(self):
        cuenta_origen = CuentaBancaria()
        cuenta_origen.depositar(100)
        result = cuenta_origen.transferir(50, None)
        self.assertFalse(result)
        self.assertEqual(cuenta_origen.obtener_saldo(), 100)

if __name__ == '__main__':
    unittest.main()
```

8. Ejecuta todas las pruebas y verifica que pasen correctamente.

```
python -m unittest test_Bank.py
```

9. Refactoriza tu código si es necesario para mejorar su estructura, legibilidad y eficiencia.

```
import unittest
```

```
class CuentaBancaria:
    def __init__(self):
        self.saldo = 0

    def obtener_saldo(self):
        return self.saldo

    def depositar(self, monto):
        if monto > 0:
            self.saldo += monto
        else:
            raise ValueError("El monto a depositar debe ser
positivo")

    def retirar(self, monto):
        if monto <= self.saldo and monto > 0:
            self.saldo -= monto
            return True
        elif monto <= 0:
            raise ValueError("El monto a retirar debe ser
positivo")
        else:
            return False

    def transferir(self, monto, cuenta_destino):
        if cuenta_destino is None:
            raise ValueError("La cuenta de destino no puede ser
None")
        if self.retirar(monto):
            cuenta_destino.depositar(monto)
            return True
        else:
            return False

class TestCuentaBancaria(unittest.TestCase):
    def test_crear_cuenta(self):
        cuenta = CuentaBancaria()
        self.assertEqual(cuenta.obtener_saldo(), 0)

    def test_depositar_fondos(self):
        cuenta = CuentaBancaria()
        cuenta.depositar(100)
```

```
        self.assertEqual(cuenta.obtener_saldo(), 100)

    def test_retirar_fondos(self):
        cuenta = CuentaBancaria()
        cuenta.depositar(100)
        result = cuenta.retirar(50)
        self.assertTrue(result)
        self.assertEqual(cuenta.obtener_saldo(), 50)
        result = cuenta.retirar(60)
        self.assertFalse(result)
        self.assertEqual(cuenta.obtener_saldo(), 50)

    def test_transferir_fondos(self):
        cuenta_origen = CuentaBancaria()
        cuenta_destino = CuentaBancaria()
        cuenta_origen.depositar(100)
        result = cuenta_origen.transferir(50, cuenta_destino)
        self.assertTrue(result)
        self.assertEqual(cuenta_origen.obtener_saldo(), 50)
        self.assertEqual(cuenta_destino.obtener_saldo(), 50)
        result = cuenta_origen.transferir(60, cuenta_destino)
        self.assertFalse(result)
        self.assertEqual(cuenta_origen.obtener_saldo(), 50)
        self.assertEqual(cuenta_destino.obtener_saldo(), 50)

    def test_retirar_monto_negativo(self):
        cuenta = CuentaBancaria()
        cuenta.depositar(100)
        with self.assertRaises(ValueError):
            cuenta.retirar(-50)

    def test_depositar_monto_negativo(self):
        cuenta = CuentaBancaria()
        with self.assertRaises(ValueError):
            cuenta.depositar(-100)

    def test_transferir_a_cuenta_inexistente(self):
        cuenta_origen = CuentaBancaria()
        cuenta_origen.depositar(100)
        with self.assertRaises(ValueError):
            cuenta_origen.transferir(50, None)
```

```
if __name__ == '__main__':
    unittest.main()
```

10. Ejecuta todas las pruebas nuevamente para asegurarte de que el código refactorizadono haya introducido errores.

```
python -m unittest test_Bank.py
```

Etapa 4: Cobertura completa de pruebas

11. Asegúrate de que todas las funcionalidades del sistema estén cubiertas por pruebasautomatizadas.

tengo las siguientes pruebas :

- Apertura de cuentas
- Depósitos,
- Retiros
- Transferencias.

12. Examina los casos límite y situaciones excepcionales para garantizar que el sistema secomporte correctamente en todos los escenarios.

Código con cobertura completa de pruebas

```
import unittest

class CuentaBancaria:
    def __init__(self):
        self.saldo = 0

    def obtener_saldo(self):
        return self.saldo

    def depositar(self, monto):
        if monto > 0:
            self.saldo += monto
        else:
            raise ValueError("El monto a depositar debe ser
positivo")

    def retirar(self, monto):
```

```
        if monto <= self.saldo and monto > 0:
            self.saldo -= monto
            return True
        elif monto <= 0:
            raise ValueError("El monto a retirar debe ser
positivo")
        else:
            return False

    def transferir(self, monto, cuenta_destino):
        if cuenta_destino is None:
            raise ValueError("La cuenta de destino no puede ser
None")
        if self.retirar(monto):
            cuenta_destino.depositar(monto)
            return True
        else:
            return False

class TestCuentaBancaria(unittest.TestCase):
    def test_crear_cuenta(self):
        cuenta = CuentaBancaria()
        self.assertEqual(cuenta.obtener_saldo(), 0)

    def test_depositar_fondos(self):
        cuenta = CuentaBancaria()
        cuenta.depositar(100)
        self.assertEqual(cuenta.obtener_saldo(), 100)

    def test_depositar_fondos_negativos(self):
        cuenta = CuentaBancaria()
        with self.assertRaises(ValueError):
            cuenta.depositar(-50)

    def test_retirar_fondos(self):
        cuenta = CuentaBancaria()
        cuenta.depositar(100)
        result = cuenta.retirar(50)
        self.assertTrue(result)
        self.assertEqual(cuenta.obtener_saldo(), 50)
        result = cuenta.retirar(60)
        self.assertFalse(result)
```

```
        self.assertEqual(cuenta.obtener_saldo(), 50)

    def test_retirar_fondos_negativos(self):
        cuenta = CuentaBancaria()
        with self.assertRaises(ValueError):
            cuenta.retirar(-50)

    def test_retirar_fondos_excesivos(self):
        cuenta = CuentaBancaria()
        cuenta.depositar(50)
        result = cuenta.retirar(100)
        self.assertFalse(result)
        self.assertEqual(cuenta.obtener_saldo(), 50)

    def test_transferir_fondos(self):
        cuenta_origen = CuentaBancaria()
        cuenta_destino = CuentaBancaria()
        cuenta_origen.depositar(100)
        result = cuenta_origen.transferir(50, cuenta_destino)
        self.assertTrue(result)
        self.assertEqual(cuenta_origen.obtener_saldo(), 50)
        self.assertEqual(cuenta_destino.obtener_saldo(), 50)
        result = cuenta_origen.transferir(60, cuenta_destino)
        self.assertFalse(result)
        self.assertEqual(cuenta_origen.obtener_saldo(), 50)
        self.assertEqual(cuenta_destino.obtener_saldo(), 50)

    def test_transferir_fondos_a_cuenta_inexistente(self):
        cuenta_origen = CuentaBancaria()
        cuenta_origen.depositar(100)
        with self.assertRaises(ValueError):
            cuenta_origen.transferir(50, None)

    def test_depositar_cero(self):
        cuenta = CuentaBancaria()
        with self.assertRaises(ValueError):
            cuenta.depositar(0)

    def test_retirar_cero(self):
        cuenta = CuentaBancaria()
        with self.assertRaises(ValueError):
            cuenta.retirar(0)
```

```
    def test_transferir_fondos_cero(self):
        cuenta_origen = CuentaBancaria()
        cuenta_destino = CuentaBancaria()
        cuenta_origen.depositar(100)
        result = cuenta_origen.transferir(0, cuenta_destino)
        self.assertFalse(result)
        self.assertEqual(cuenta_origen.obtener_saldo(), 100)
        self.assertEqual(cuenta_destino.obtener_saldo(), 0)

    def test_transferir_fondos_monto_negativo(self):
        cuenta_origen = CuentaBancaria()
        cuenta_destino = CuentaBancaria()
        cuenta_origen.depositar(100)
        with self.assertRaises(ValueError):
            cuenta_origen.transferir(-50, cuenta_destino)

if __name__ == '__main__':
    unittest.main()
```

13. Ejecuta todas las pruebas y verifica que pasen correctamente.
codigo completo de las pruebas:

```
import unittest

class TestCuentaBancaria(unittest.TestCase):
    def test_crear_cuenta(self):
        cuenta = CuentaBancaria()
        self.assertEqual(cuenta.obtener_saldo(), 0)

    def test_depositar_fondos(self):
        cuenta = CuentaBancaria()
        cuenta.depositar(100)
        self.assertEqual(cuenta.obtener_saldo(), 100)

    def test_depositar_fondos_negativos(self):
        cuenta = CuentaBancaria()
        with self.assertRaises(ValueError):
            cuenta.depositar(-50)
```

```
    def test_retirar_fondos(self):
        cuenta = CuentaBancaria()
        cuenta.depositar(100)
        result = cuenta.retirar(50)
        self.assertTrue(result)
        self.assertEqual(cuenta.obtener_saldo(), 50)
        result = cuenta.retirar(60)
        self.assertFalse(result)
        self.assertEqual(cuenta.obtener_saldo(), 50)

    def test_retirar_fondos_negativos(self):
        cuenta = CuentaBancaria()
        with self.assertRaises(ValueError):
            cuenta.retirar(-50)

    def test_retirar_fondos_excesivos(self):
        cuenta = CuentaBancaria()
        cuenta.depositar(50)
        result = cuenta.retirar(100)
        self.assertFalse(result)
        self.assertEqual(cuenta.obtener_saldo(), 50)

    def test_transferir_fondos(self):
        cuenta_origen = CuentaBancaria()
        cuenta_destino = CuentaBancaria()
        cuenta_origen.depositar(100)
        result = cuenta_origen.transferir(50, cuenta_destino)
        self.assertTrue(result)
        self.assertEqual(cuenta_origen.obtener_saldo(), 50)
        self.assertEqual(cuenta_destino.obtener_saldo(), 50)
        result = cuenta_origen.transferir(60, cuenta_destino)
        self.assertFalse(result)
        self.assertEqual(cuenta_origen.obtener_saldo(), 50)
        self.assertEqual(cuenta_destino.obtener_saldo(), 50)

    def test_transferir_fondos_a_cuenta_inexistente(self):
        cuenta_origen = CuentaBancaria()
        cuenta_origen.depositar(100)
        with self.assertRaises(ValueError):
            cuenta_origen.transferir(50, None)

    def test_depositar_cero(self):
```

```
        cuenta = CuentaBancaria()
        with self.assertRaises(ValueError):
            cuenta.depositar(0)

    def test_retirar_cero(self):
        cuenta = CuentaBancaria()
        with self.assertRaises(ValueError):
            cuenta.retirar(0)

    def test_transferir_fondos_cero(self):
        cuenta_origen = CuentaBancaria()
        cuenta_destino = CuentaBancaria()
        cuenta_origen.depositar(100)
        result = cuenta_origen.transferir(0, cuenta_destino)
        self.assertFalse(result)
        self.assertEqual(cuenta_origen.obtener_saldo(), 100)
        self.assertEqual(cuenta_destino.obtener_saldo(), 0)

    def test_transferir_fondos_monto_negativo(self):
        cuenta_origen = CuentaBancaria()
        cuenta_destino = CuentaBancaria()
        cuenta_origen.depositar(100)
        with self.assertRaises(ValueError):
            cuenta_origen.transferir(-50, cuenta_destino)

if __name__ == '__main__':
    unittest.main()
```

ejecucion de prueba:

```
python -m unittest test_Bank.py
```

Recuerda seguir el enfoque TDD, donde agregarás una prueba antes de implementar cadafuncionalidad y verificarás que todas las pruebas pasen antes de pasar a la siguiente etapa.Esto te ayudará a desarrollar una aplicación confiable, mantenible y que cumpla con losrequisitos establecidos.