Jose Andres Matarrita Miranda C04668

Preguntas En su documento de reporte, conteste brevemente las siguientes preguntas.

A) Basado en el tutorial del ejercicio de pruebas unitarias, en la última sección, Retest, rewrite, and reanalyze, explique muy brevemente ¿Por qué se necesita agregar la instrucción Assert.Fail a este caso de prueba?

En el código proporcionado, es necesario agregar la instrucción `Assert.Fail` al método de prueba `Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRange`.La razón de agregar `Assert.Fail` es para que la prueba falle explícitamente si no se lanza la excepción esperada. En este caso, si el método `Debit` no lanza una `ArgumentOutOfRangeException` cuando el monto de débito es mayor que el saldo o menor que cero, el método de prueba debería fallar.

B) Basado en la lectura de conceptos sobre pruebas unitarias, responda muy brevemente ¿cuál es el valor de las pruebas unitarias en el flujo de desarrollo de software?

Las pruebas unitarias desempeñan un papel fundamental en el flujo de desarrollo de software. Proporcionan una detección temprana de errores al permitir identificar y corregir problemas en el código de manera temprana, antes de que se propaguen y se conviertan en problemas más graves en etapas posteriores. Además, las pruebas unitarias mejoran la calidad del código al obligar a los desarrolladores a escribir código modular, bien estructurado y de fácil mantenimiento. Actúan como una documentación ejecutable del comportamiento esperado de las diferentes partes del sistema, lo que facilita la comprensión del código y agiliza el proceso de desarrollo. Las pruebas unitarias también proporcionan una mayor confianza al realizar refactorizaciones, ya que si las pruebas siguen pasando después de realizar cambios, se tiene la seguridad de que no se han introducido errores. En resumen, las pruebas unitarias son esenciales para garantizar la calidad y confiabilidad del software, mejorar la eficiencia del desarrollo y reducir la probabilidad de errores en el producto final.

C) Basado en la lectura de conceptos sobre pruebas unitarias, responda muy brevemente ¿cuáles son las principales partes que componen una prueba unitaria?

Las principales partes que componen una prueba unitaria son:

1. Preparación (Setup): En esta etapa se prepara el entorno necesario para ejecutar la prueba. Esto puede incluir la creación de objetos, la configuración de parámetros y la inicialización de variables.

2. Ejecución: Aquí se llama al método o función que se desea probar con los valores de entrada adecuados. Se realiza la ejecución del código que se quiere evaluar.

3. Verificación (Assertion): En esta parte se verifica si el resultado obtenido de la ejecución coincide con el resultado esperado. Se utilizan aserciones (assertions) para comparar los valores y determinar si la prueba ha sido exitosa o no.

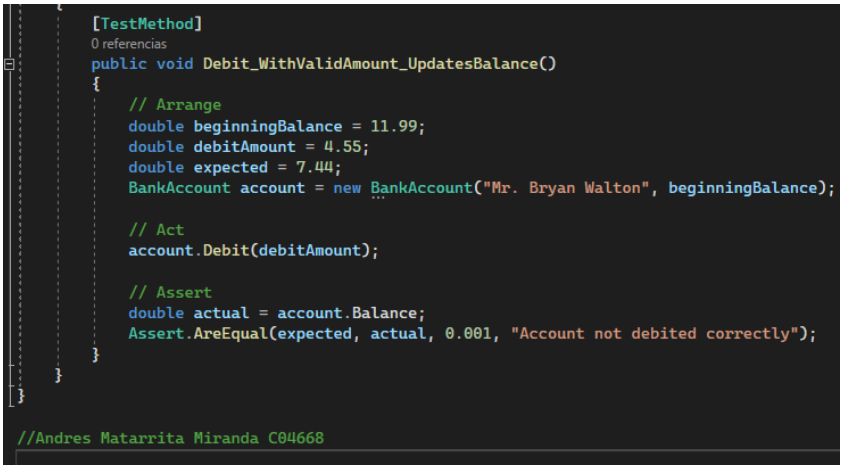
4. Limpieza (Teardown): En esta fase se realiza la limpieza de cualquier recurso utilizado durante la prueba. Puede incluir la liberación de memoria, el cierre de conexiones o la eliminación de objetos temporales.

Estas partes conforman el ciclo básico de una prueba unitaria, donde se prepara el entorno, se ejecuta el código a probar, se verifica el resultado y se realiza la limpieza necesaria. Es importante asegurarse de que cada prueba unitaria sea independiente y no dependa de otras pruebas o del estado global del sistema.

D) Basado en la lectura de conceptos sobre pruebas unitarias, responda muy brevemente ¿Para qué sirve establecer un timeout a un caso de prueba?

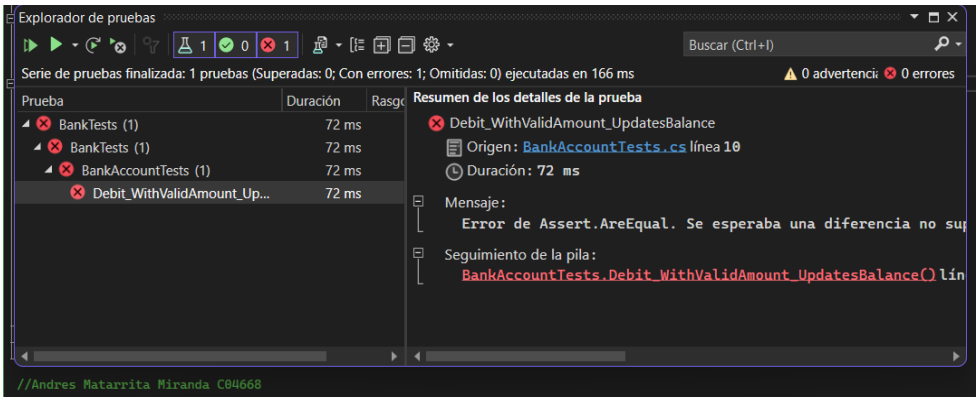
Establecer un timeout en un caso de prueba es útil para evitar bloqueos o ciclos infinitos. Ayuda a controlar el tiempo máximo permitido para la ejecución de la prueba, evitando que consuma demasiado tiempo. Esto asegura un flujo ágil de las pruebas y garantiza respuestas rápidas. Además, permite identificar y solucionar problemas de rendimiento o ineficiencias en el código. Establecer un límite temporal proporciona una mayor eficiencia en el proceso de prueba y evita retrasos innecesarios.

To create a test method

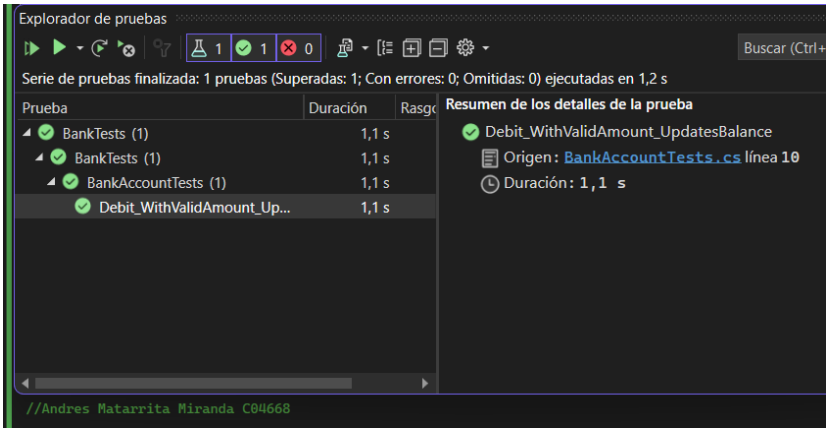


```
    [TestMethod]
    0 referencias
    public void Debit_WithValidAmount_UpdatesBalance()
    {
        // Arrange
        double beginningBalance = 11.99;
        double debitAmount = 4.55;
        double expected = 7.44;
        BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);

        // Act
        account.Debit(debitAmount);

        // Assert
        double actual = account.Balance;
        Assert.AreEqual(expected, actual, 0.001, "Account not debited correctly");
    }
}

//Andres Matarrita Miranda C04668
```

Build and run the test



Rerun the test



Create and run new test methods

```
        [TestMethod]
        0 referencias
        public void Debit_WhenAmountIsLessThanZero_ShouldThrowArgumentOutOfRange()
        {
            // Arrange
            double beginningBalance = 11.99;
            double debitAmount = -100.00;
            BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);

            // Act and assert
            Assert.ThrowsException<System.ArgumentOutOfRangeException>(() => account.Debit(debitAmount));
        }
    }
}

//Andres Matarrita Miranda C04668
```

Refactor the test methods

```
        [TestMethod]
        0 referencias
        public void Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRange()
        {
            // Arrange
            double beginningBalance = 11.99;
            double debitAmount = 20.0;
            BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);

            // Act
            try
            {
                account.Debit(debitAmount);
            }
            catch (System.ArgumentOutOfRangeException e)
            {
                // Assert
                StringAssert.Contains(e.Message, BankAccount.DebitAmountExceedsBalanceMessage);
            }
        }
    }
}

//Andres Matarrita Miranda C04668
```

Retest, rewrite, and reanalyze

```
        [TestMethod]
        0 referencias
        public void Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRange()
        {
            // Arrange
            double beginningBalance = 11.99;
            double debitAmount = 20.0;
            BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);

            // Act
            try
            {
                account.Debit(debitAmount);
            }
            catch (System.ArgumentOutOfRangeException e)
            {
                // Assert
                StringAssert.Contains(e.Message, BankAccount.DebitAmountExceedsBalanceMessage);
                return;
            }

            Assert.Fail("The expected exception was not thrown.");
        }
    }
}

//Andres Matarrita Miranda C04668
```