



## INFORME DE GUÍA PRÁCTICA

### 1. Portada

<b>Tema:</b>	Implementación del método Factorial con TDD y MSTest
<b>Unidad de Organización Curricular:</b>	Profesional.
<b>Nivel y Paralelo:</b>	Cuarto - .A”
<b>Alumnos participantes:</b>	Quisaguano Molina Dennis Javier Jurado Jacome Cristian Ernesto .....
<b>Asignatura:</b>	Metodología Ágiles
<b>Docente:</b>	Ing. Hernan Fabricio Naranjo Ávalos, Mg.
<b>Fecha:</b>	20 de junio de 2025

### 2. Informe de guía práctica

#### 2.1. Objetivos

##### Objetivo general

Implementar funcionalidades en C# mediante el enfoque de Desarrollo Guiado por Pruebas (TDD), utilizando frameworks como MSTest y NUnit para validar métodos con pruebas unitarias.

##### Objetivos específicos

- Desarrollar pruebas unitarias con MSTest y NUnit.
- Implementar el ciclo TDD en ejercicios prácticos.
- Comprobar el comportamiento de las funciones desde consola.
- Refactorizar el código para mejorar su legibilidad.

#### 2.2. Modalidad

Presencial

#### 2.3. Tiempo de duración

Presenciales: 3

No presenciales: 0

#### 2.4. Instrucciones

- Crear un proyecto de consola en C# para resolver los ejercicios propuestos.
- Crear dos proyectos de prueba: uno usando MSTest y otro usando NUnit.
- Implementar cada ejercicio utilizando la metodología TDD (Desarrollo Guiado por Pruebas), lo cual implica los siguientes pasos:



1. Escribir primero las pruebas unitarias.
  2. Ejecutar las pruebas, esperando que fallen inicialmente.
  3. Implementar el código mínimo necesario para que las pruebas pasen.
  4. Refactorizar si es necesario.
- Para cada ejercicio, se debe mostrar evidencia del proceso TDD (capturas o logs de pruebas).
  - Subir el proyecto completo con las carpetas organizadas.

## 2.5. Listado de equipos, materiales y recursos

Listado de equipos y materiales generales empleados en la guía práctica:

- TAC
- Computadora.
- Herramientas Ofimáticas.
- Aula virtual.
- Internet.

TAC (Tecnologías para el Aprendizaje y Conocimiento) empleados en la guía práctica:

- Plataformas educativas
- ☐ Simuladores y laboratorios virtuales
- Aplicaciones educativas (Visual Studio Code, .NET SDK)
- ☐ Recursos audiovisuales
- ☐ Gamificación
- Inteligencia Artificial
- Otros (Especifique): \_\_\_\_\_

## 2.6. Actividades por desarrollar

- Configuración del entorno de desarrollo para C# y MSTest.
- Creación de la estructura de proyectos para la lógica y las pruebas.
- Implementación del método Factorial siguiendo el ciclo TDD:
  1. Escribir una prueba que falle (Fase Roja).
  2. Escribir el código mínimo para que la prueba pase (Fase Verde).
  3. Refactorizar el código, asegurándose de que todas las pruebas sigan pasando (Fase Azul/-Refactorización).
- Documentación de cada fase y los resultados obtenidos.



## 2.7. Resultados Obtenidos

# 3. Ejercicio 1: FactorialNUnitTDD — Pruebas con NUnit

## Enunciado

Implementar un programa en C# que calcule el factorial de un número entero no negativo utilizando desarrollo guiado por pruebas (TDD) y el framework de pruebas **NUnit**.

## Historia de implementación

Se creó la solución **FactorialNUnitApp** con dos proyectos: uno para la lógica (**FactorialNUnitApp**) y otro para pruebas unitarias con **NUnit**. El desarrollo se guió mediante el ciclo TDD (fallo → éxito → refactorización), con ejecución progresiva de pruebas y validación desde consola.

## ¿Cuántas pruebas se realizaron y qué resultados se obtuvieron?

Se realizaron un total de 5 pruebas unitarias usando NUnit:

Cuadro 1: Pruebas unitarias realizadas con NUnit para el cálculo del factorial

N°	Nombre de la Prueba	Objetivo de la Prueba
1	Factorial_Cero_EsUno	Verificar que <code>factorial(0)</code> retorna 1.
2	Factorial_Uno_EsUno	Verificar que <code>factorial(1)</code> retorna 1.
3	Factorial_Cinco_Es120	Verificar que <code>factorial(5)</code> retorna 120.
4	Factorial_Diez_Es3628800	Verificar que <code>factorial(10)</code> retorna 3628800.
5	Factorial_Negativo_Excepcion	Verificar que se lanza una excepción para $n < 0$ .

## ¿Qué se realizó?

- Se preparó el entorno con una solución de consola y un proyecto de pruebas con NUnit.
- Se implementaron las pruebas una por una, comenzando con entradas triviales como 0 y 1.
- Se verificó el comportamiento del método **Factorial** para entradas válidas y negativas.
- Se refactorizó el código aplicando una versión más elegante con LINQ.

## ¿Qué debe cumplir internamente el desarrollo?

1. **Preparar el entorno:** Crear la solución, proyecto de lógica y de pruebas con sus referencias.
2. **Ejecutar pruebas:** Ejecutar cada prueba después de escribirla y ver el fallo esperado.
3. **Validar resultados:** Usar aserciones con `Assert.AreEqual` y `Assert.Throws`.



## Proceso y capturas de evidencia: FactorialNUnitTDD

```
PS C:\Users\Cristian Jurado\Documents\EjerciciosPrueba\FactorialNUnitTDD> dotnet test
Restauración completada (2,8s)
FactorialNUnitApp realizado correctamente (2,7s) + FactorialNUnitApp\bin\Debug\net9.0\FactorialNUnitApp.dll
FactorialNUnitApp.Tests error con 4 errores (0,7s)
C:\Users\Cristian Jurado\Documents\EjerciciosPrueba\FactorialNUnitTDD\FactorialNUnitApp.Tests\FactorialTests.cs(12,20): error CS017: 'Assert' no contiene
una definición para 'AreEqual'
C:\Users\Cristian Jurado\Documents\EjerciciosPrueba\FactorialNUnitTDD\FactorialNUnitApp.Tests\FactorialTests.cs(18,20): error CS017: 'Assert' no contiene
una definición para 'AreEqual'
C:\Users\Cristian Jurado\Documents\EjerciciosPrueba\FactorialNUnitTDD\FactorialNUnitApp.Tests\FactorialTests.cs(24,20): error CS017: 'Assert' no contiene
una definición para 'AreEqual'
C:\Users\Cristian Jurado\Documents\EjerciciosPrueba\FactorialNUnitTDD\FactorialNUnitApp.Tests\FactorialTests.cs(30,20): error CS017: 'Assert' no contiene
una definición para 'AreEqual'
Compilación error con 4 errores en 6,6s
PS C:\Users\Cristian Jurado\Documents\EjerciciosPrueba\FactorialNUnitTDD>
```

Figura 1: Fase de fallo: se crea la primera prueba unitaria con NUnit para validar que Factorial(0) devuelva 1. Al no existir aún el método, se produce un error esperado.

```
PS C:\Users\Cristian Jurado\Documents\EjerciciosPrueba\FactorialNUnitTDD> dotnet test
Restauración completada (0,7s)
FactorialNUnitApp realizado correctamente (0,3s) + FactorialNUnitApp\bin\Debug\net9.0\FactorialNUnitApp.dll
FactorialNUnitApp.Tests realizado correctamente (0,5s) + FactorialNUnitApp.Tests\bin\Debug\net9.0\FactorialNUnitApp.Tests.dll
NUnit Adapter 4.6.0.0: Test execution started
Running all tests in C:\Users\Cristian Jurado\Documents\EjerciciosPrueba\FactorialNUnitTDD\FactorialNUnitApp.Tests\bin\Debug\net9.0\FactorialNUnitApp.Tests.dll
NUnit Adapter 4.6.0.0: discovered 6 of 6 NUnit test cases using Current Discovery mode, Non-Explicit run
FactorialNUnitApp.Tests prueba realizado correctamente (2,4s)
Resumen de pruebas: total: 6; con errores: 0; correcto: 6; omitido: 0; duración: 2,4 s
Compilación realizado correctamente en 4,5s
PS C:\Users\Cristian Jurado\Documents\EjerciciosPrueba\FactorialNUnitTDD>
```

Figura 2: Fase de éxito inicial: se implementa la lógica mínima para pasar las pruebas de Factorial(0) y Factorial(1), ambas devolviendo 1 correctamente.



```
Factorial.cs  FactorialTests.cs x
FactorialUnitTDD > FactorialUnitApp.Tests > FactorialTests.cs > ...
1  using NUnit.Framework;
2  using FactorialUnitApp;
3  using System;
4
5  namespace FactorialUnitApp.Tests
6  {
7      0 referencias | Ejecutar todas las pruebas | Depurar todas las pruebas
8      public class FactorialTests
9      {
10         [Test]
11         0 referencias | Ejecutar prueba | Depurar prueba
12         public void Factorial_De_Cero_Es_Uno()
13         {
14             Assert.AreEqual(1, Calculadora.Factorial(0));
15         }
16
17         [Test]
18         0 referencias | Ejecutar prueba | Depurar prueba
19         public void Factorial_De_Uno_Es_Uno()
20         {
21             Assert.AreEqual(1, Calculadora.Factorial(1));
22         }
23
24         [Test]
25         0 referencias | Ejecutar prueba | Depurar prueba
26         public void Factorial_De_Cinco_Es_120()
27         {
28             Assert.AreEqual(120, Calculadora.Factorial(5));
29         }
30     }
```

Figura 3: Fase de éxito ampliado: se agregan nuevas pruebas para valores como 5 y 10, y también se valida el lanzamiento de excepción al ingresar valores negativos.

```
using System;
using System.Linq;

0 referencias
public static class Calculadora
{
    0 referencias
    public static long Factorial(int n)
    {
        ValidarEntrada(n);

        return (n == 0 || n == 1)
            ? 1
            : Enumerable.Range(2, n - 1).Aggregate(1L, (acc, val) => acc * val);
    }

    1 referencia
    private static void ValidarEntrada(int n)
    {
        if (n < 0)
            throw new ArgumentException("No se puede calcular el factorial de un número negativo.", nameof(n));
    }
}
```

Figura 4: Refactorización avanzada: se implementa una versión alternativa usando LINQ con Enumerable.Range y Aggregate para calcular el factorial de forma declarativa.



```
Factorial.cs  FactorialTests.cs  Program.cs X
FactorialNUnitTDD > FactorialNUnitApp > Program.cs > ...
1  using System;
2
3  namespace FactorialNUnitApp
4  {
5      0 referencias
6      class Program
7      {
8          0 referencias
9          static void Main()
10         {
11             Console.WriteLine("=== Cálculo del Factorial ===");
12
13             Console.Write("Ingrese un número entero no negativo: ");
14             if (!int.TryParse(Console.ReadLine(), out int numero) || numero < 0)
15             {
16                 Console.WriteLine("Entrada inválida. Debe ingresar un número entero no negativo.");
17                 return;
18             }
19
20             try
21             {
22                 long resultado = Calculadora.Factorial(numero);
23                 Console.WriteLine($"{numero}! = {resultado}");
24             }
25             catch (Exception ex)
26             {
27                 Console.WriteLine($"Error: {ex.Message}");
28             }
29         }
30     }
31 }
```

Figura 5: Ejecución final desde consola: el usuario ingresa un número por teclado y el programa muestra el factorial calculado utilizando la lógica validada por pruebas unitarias.

## 4. Ejercicio 2: SombraSolarTDD — Pruebas con MSTest

### Enunciado

Implementar un programa en C# que calcule la longitud de la sombra proyectada por un árbol en función de la hora del día. Se usa el ángulo solar interpolado entre tres referencias:

- 06:00 → 0°
- 12:00 → 90°
- 18:00 → 180°

El desarrollo se guía mediante TDD, empleando el framework de pruebas **MSTest**.

### Historia de implementación

Se construyó la solución **SombraSolarApp** aplicando el ciclo TDD. Se validaron casos límite como cuando el sol está en el horizonte (ángulo 0° o 180°) y cuando se encuentra en el punto más alto (90°). El código se refactorizó para mayor claridad, y se probó desde consola ingresando altura y hora.

### ¿Cuántas pruebas se realizaron y qué resultados se obtuvieron?

Se desarrollaron y ejecutaron un total de **4 pruebas unitarias** utilizando **MSTest**, orientadas a verificar tanto la interpolación del ángulo solar como el cálculo correcto de la sombra proyectada. Todas las pruebas pasaron exitosamente, validando el comportamiento esperado del sistema bajo distintos escenarios.



Cuadro 2: Pruebas unitarias realizadas para el cálculo de la sombra solar.

N°	Nombre de la Prueba	Objetivo de la Prueba
1	Angulo_A_Las_6AM_EsCero	Validar que el ángulo solar calculado a las 06:00 sea exactamente 0 grados.
2	Angulo_A_Las_12PM_Es90	Confirmar que a las 12:00 el ángulo solar corresponde a 90 grados, posición cenital.
3	Angulo_A_Las_9AM_Es45	Verificar que la interpolación a las 09:00 produce un ángulo de 45 grados.
4	Sombra_Correcta_Para90	Comprobar que, al tener un ángulo solar de 90°, la longitud de la sombra calculada sea 0 metros.

### ¿Qué se realizó?

- Se creó el proyecto de consola y pruebas MSTest.
- Se implementaron pruebas para validar el ángulo solar en distintas horas.
- Se desarrolló la lógica de interpolación de ángulos y cálculo de tangente.
- Se refactorizó el código y se probó desde consola con entradas dinámicas.

### ¿Qué debe cumplir internamente el desarrollo?

1. **Preparar el entorno:** Crear los proyectos y agregar referencias necesarias.
2. **Ejecutar pruebas:** Usar `dotnet test` para verificar cada validación.
3. **Validar resultados:** Comprobar valores calculados mediante `Assert.AreEqual`.

### Proceso y capturas de evidencia: SombraSolarTDD

```
PS C:\Users\Cristian Jurado\Documents\EjerciciosPrueba> mkdir SombraSolarTDD

Directorio: C:\Users\Cristian Jurado\Documents\EjerciciosPrueba

Mode                LastWriteTime         Length Name
----                -
d-----          19/6/2025   20:26             SombraSolarTDD

PS C:\Users\Cristian Jurado\Documents\EjerciciosPrueba> cd SombraSolarTDD
La plantilla "Aplicación de consola" se creó correctamente.
Restaurando C:\Users\Cristian Jurado\Documents\EjerciciosPrueba\SombraSolarTDD\SombraSolarApp\SombraSolarApp.csproj:
Restauración realizada correctamente.
```

Figura 6: Inicio del proyecto: creación de la solución en Visual Studio Code, junto con los proyectos de consola y de pruebas MSTest.



```
PS C:\Users\Cristian Jurado\Documents\EjerciciosPrueba\SombraSolarTDD> dotnet new mstest -o SombraSolarApp.Tests
La plantilla "Proyecto de prueba de MSTest" se creó correctamente.

Procesando acciones posteriores a la creación...
La plantilla "Proyecto de prueba de MSTest" se creó correctamente.

Procesando acciones posteriores a la creación...
Restaurando C:\Users\Cristian Jurado\Documents\EjerciciosPrueba\SombraSolarTDD\SombraSolarApp.Tests\SombraSolarApp.Tests.csproj:
Restauración realizada correctamente.
Restauración realizada correctamente.
```

Figura 7: Se agrega cada proyecto a la solución usando `dotnet sln add` y se enlaza correctamente el proyecto de pruebas con el proyecto principal.

```
PS C:\Users\Cristian Jurado\Documents\EjerciciosPrueba\SombraSolarTDD> dotnet new sln -n SombraSolarTDD
La plantilla "Archivo de la solución" se creó correctamente.

PS C:\Users\Cristian Jurado\Documents\EjerciciosPrueba\SombraSolarTDD> dotnet sln add SombraSolarApp\SombraSolarApp.csproj
Se ha agregado el proyecto "SombraSolarApp\SombraSolarApp.csproj" a la solución.

PS C:\Users\Cristian Jurado\Documents\EjerciciosPrueba\SombraSolarTDD> dotnet sln add SombraSolarApp.Tests\SombraSolarApp.Tests.csproj
Se ha agregado el proyecto "SombraSolarApp.Tests\SombraSolarApp.Tests.csproj" a la solución.

PS C:\Users\Cristian Jurado\Documents\EjerciciosPrueba\SombraSolarTDD> dotnet add SombraSolarApp.Tests reference SombraSolarApp\SombraSolarApp.csproj
Se ha agregado la referencia "...SombraSolarApp\SombraSolarApp.csproj" al proyecto.
```

Figura 8: Confirmación del enlace entre proyectos. Se estructura correctamente la solución con los archivos `.csproj` vinculados.

```
using System;

namespace SombraSolarApp
{
    0 referencias
    public class Sombra
    {
        1 referencia
        public static double CalcularAngulo(TimeSpan hora)
        {
            if (hora < new TimeSpan(6, 0, 0) || hora > new TimeSpan(18, 0, 0))
                throw new ArgumentException("Hora fuera del rango permitido (06:00 a 18:00)");

            double minutosDesde6AM = hora.TotalMinutes - 360;
            return (minutosDesde6AM / 720) * 180;
        }

        0 referencias
        public static double CalcularSombra(double altura, TimeSpan hora)
        {
            if (altura <= 0)
                throw new ArgumentException("La altura debe ser mayor a cero.");

            double angulo = CalcularAngulo(hora);
            if (angulo == 0 || angulo == 180)
                return double.PositiveInfinity;

            double rad = angulo * Math.PI / 180;
            return altura / Math.Tan(rad);
        }
    }
}
```

Figura 9: Primera prueba: se implementa el test `Angulo_A_Las_6AM_EsCero` para verificar que el ángulo solar a las 06:00 sea 0°. La prueba aún no se ejecuta.





```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System;
using SombraSolarApp;

namespace SombraSolarApp.Tests
{
    [TestClass]
    0 referencias | Ejecutar todas las pruebas | Depurar todas las pruebas
    public class SombraTests
    {
        [TestMethod]
        0 referencias | Ejecutar prueba | Depurar prueba
        public void Sombra_A_Las_09_00_Es_Igual_Altura()
        {
            double altura = 10;
            TimeSpan hora = new TimeSpan(9, 0, 0);
            double sombra = Sombra.CalcularSombra(altura, hora);
            Assert.AreEqual(10, sombra, 0.1);
        }

        [TestMethod]
        0 referencias | Ejecutar prueba | Depurar prueba
        public void Angulo_A_Las_12_Es_90()
        {
            TimeSpan hora = new TimeSpan(12, 0, 0);
            double angulo = Sombra.CalcularAngulo(hora);
            Assert.AreEqual(90, angulo, 0.1);
        }
    }
}
```

Figura 10: Se ejecuta la primera prueba sin errores. En esta fase ya se ha implementado la función `CalcularAnguloSolar`, devolviendo correctamente 0° para las 06:00.



```
[TestMethod]
0 referencias | Ejecutar prueba | Depurar prueba
public void Sombra_A_Las_12_Es_0()
{
    double altura = 10;
    TimeSpan hora = new TimeSpan(12, 0, 0);
    double sombra = Sombra.CalcularSombra(altura, hora);
    Assert.IsTrue(sombra < 0.1);
}

[TestMethod]
[ExpectedException(typeof(ArgumentException))]
0 referencias | Ejecutar prueba | Depurar prueba
public void Altura_Negativa_Lanza_Excepcion()
{
    Sombra.CalcularSombra(-5, new TimeSpan(9, 0, 0));
}

[TestMethod]
0 referencias | Ejecutar prueba | Depurar prueba
public void Sombra_A_Las_06_Es_Infinita()
{
    double altura = 10;
    TimeSpan hora = new TimeSpan(6, 0, 0);
    double sombra = Sombra.CalcularSombra(altura, hora);
    Assert.IsTrue(double.IsPositiveInfinity(sombra));
}
}
```

Figura 11: Se agregan nuevas pruebas: por ejemplo, que a las 12:00 el ángulo sea 90° y a las 09:00 sea 45°. Todas las pruebas unitarias pasan correctamente.

```
PS C:\Users\Gustian Jurado\Documents\EjerciciosPrueba\SombraSolar\TD> dotnet test
Restauración completada (0,7s)
SombraSolarApp realizado correctamente (0,3s) - SombraSolarApp\bin\Debug\net9.0\SombraSolarApp.dll
SombraSolarApp.Tests realizado correctamente (0,3s) - SombraSolarApp.Tests\bin\Debug\net9.0\SombraSolarApp.Tests.dll
Probar paralelización habilitada para C:\Users\Gustian Jurado\Documents\EjerciciosPrueba\SombraSolar\TD\SombraSolarApp.Tests\bin\Debug\net9.0\SombraSolarApp.Tests.dll (trabajos: 8, ámbito: MethodLevel)
SombraSolarApp.Tests prueba realizado correctamente (1,0s)

Resumen de pruebas: total: 6; con errores: 0; correcto: 6; omitido: 0; duración: 1,0 s
Compilación realizado correctamente en 2,7s
```

Figura 12: Verificación de cálculo de sombra: se implementa prueba para asegurar que a 90° el árbol no proyecta sombra (longitud igual a 0).



```
using System;

namespace SombraSolarApp
{
    0 referencias
    public class Sombra
    {
        1 referencia
        public static double CalcularAngulo(TimeSpan hora)
        {
            if (hora < new TimeSpan(6, 0, 0) || hora > new TimeSpan(18, 0, 0))
            {
                throw new ArgumentException("Hora fuera del rango permitido (06:00 a 18:00)", nameof(hora));
            }

            double minutosDesde6AM = hora.TotalMinutes - 360;
            return (minutosDesde6AM / 720) * 180;
        }

        0 referencias
        public static double CalcularSombra(double altura, TimeSpan hora)
        {
            if (altura <= 0)
            {
                throw new ArgumentException("La altura debe ser mayor a cero.", nameof(altura));
            }

            double angulo = CalcularAngulo(hora);

            if (angulo == 0 || angulo == 180)
            {
                return double.PositiveInfinity;
            }

            double radianes = GradosARadianes(angulo);
            return altura / Math.Tan(radianes);
        }
    }
}
```

Figura 13: Se refactoriza el método para incluir validaciones de entrada: se verifica que la hora ingresada esté dentro del rango permitido (06:00 a 18:00).

```
1 referencia
private static double GradosARadianes(double grados) => grados * Math.PI / 180;
}
```

Figura 14: Validación adicional: el usuario debe ingresar una altura positiva para el árbol. Se asegura que el valor ingresado sea mayor que 0.

```
PS C:\Users\Cristian Jurado\Documents\UjerciciosPrueba\SombraSolar\TDD> dotnet test
Restauración completada (0,7s)
SombraSolarApp realizado correctamente (0,4s) - SombraSolarApp\bin\Debug\net9.0\SombraSolarApp.dll
SombraSolarApp.Tests realizado correctamente (0,1s) - SombraSolarApp.Tests\bin\Debug\net9.0\SombraSolarApp.Tests.dll
Probar paralelización habilitada para C:\Users\Cristian Jurado\Documents\UjerciciosPrueba\SombraSolar\TDD\SombraSolarApp.Tests\bin\Debug\net9.0\SombraSolarApp.Tests.dll (Trabajos: 8, Ámbito: MethodLevel)
SombraSolarApp.Tests prueba realizado correctamente (1,8s)

Resumen de pruebas: total: 6; con errores: 0; correcto: 6; omitido: 0; duración: 1,0 s
Compilación realizado correctamente en 2,5s
```

Figura 15: Refactorización del cálculo angular: se mejora el uso de TimeSpan para controlar correctamente los minutos desde las 6:00 AM.



```
using System;
using SombraSolarApp;

0 referencias
class Program
{
    0 referencias
    static void Main()
    {
        Console.WriteLine("Ingrese la altura del árbol en metros: ");
        if (!double.TryParse(Console.ReadLine(), out double altura) || altura <= 0)
        {
            Console.WriteLine("Altura inválida.");
            return;
        }

        Console.WriteLine("Ingrese la hora (formato HH:mm, entre 06:00 y 18:00): ");
        if (!TimeSpan.TryParse(Console.ReadLine(), out TimeSpan hora) ||
            hora < new TimeSpan(6, 0, 0) || hora > new TimeSpan(18, 0, 0))
        {
            Console.WriteLine("Hora fuera del rango permitido.");
            return;
        }

        try
        {
            double sombra = Sombra.CalcularSombra(altura, hora);
            double angulo = Sombra.CalcularAngulo(hora);
        }
    }
}
```

Figura 16: El código se organiza y documenta adecuadamente. Se muestra el mensaje correspondiente si el ángulo solar es 0° o 180°, indicando sombra infinita.

```
        if (double.IsInfinity(sombra))
            Console.WriteLine("La sombra es infinita (el sol está en el horizonte).");
        else
        {
            Console.WriteLine($"Ángulo solar: {angulo:F2}°");
            Console.WriteLine($"Longitud de la sombra: {sombra:F2} metros.");
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Error: {ex.Message}");
    }
}
```

Figura 17: Se encapsula el cálculo de radianes para separar responsabilidades y mejorar la reutilización del código.

```
PS C:\Users\Cristian Jurado\Documents\EjerciciosPrueba\SombraSolarTDD> dotnet run --project SombraSolarApp
Ingrese la altura del árbol en metros: 10
Ingrese la hora (formato HH:mm, entre 06:00 y 18:00): 09:00
Ángulo solar: 45,00°
Longitud de la sombra: 10,00 metros.
```

Figura 18: Refactorización final: se simplifica el cálculo usando expresiones limpias y se ajusta el formato de los mensajes para el usuario.

## Comparación entre MSTest y NUnit



Cuadro 3: Diferencias entre MSTest y NUnit utilizadas en el desarrollo de pruebas.

Criterio	MSTest	NUnit
Origen y soporte	Framework oficial de Microsoft para pruebas en .NET. Está totalmente integrado con Visual Studio.	Framework de pruebas independiente y de código abierto. Muy usado en entornos multiplataforma.
Instalación	Se incluye por defecto al crear proyectos de prueba en Visual Studio.	Requiere instalar el paquete NUnit y NUnit3TestAdapter.
Atributos principales	Usa atributos como <code>[TestMethod]</code> , <code>[TestClass]</code> , <code>[ExpectedException]</code> .	Usa atributos como <code>[Test]</code> , <code>[TestFixture]</code> , <code>Assert.Throws()</code> .
Flexibilidad en aserciones	Más limitada, principalmente <code>Assert.AreEqual</code> , <code>Assert.IsTrue</code> , etc.	Más expresivo con <code>Assert.That(..., Is.EqualTo(...))</code> , múltiples opciones para validación fluida.
Curva de aprendizaje	Muy simple para principiantes; ideal para empezar.	Requiere entender su sintaxis, pero brinda mayor control a mediano/largo plazo.
Integración con herramientas	Excelente integración con herramientas de Microsoft (Azure, DevOps).	Compatible con muchas herramientas externas y runners personalizados.
Velocidad de ejecución	Muy rápida y estable en entornos de Visual Studio.	Igualmente rápida, pero puede requerir configuración adicional.
Licencia y comunidad	Licencia Microsoft. Comunidad amplia pero menos abierta.	Código abierto, gran comunidad activa y evolución continua.
Experiencia en este proyecto	Se usó en el ejercicio Sombra Solar. Muy fácil de configurar y ejecutar.	Se usó en el ejercicio Factorial. Muy potente y flexible al momento de validar excepciones.

## Opinión Grupal sobre MSTest vs NUnit

Tras aplicar ambas herramientas en ejercicios prácticos, considero que:

- **MSTest** es ideal para comenzar, gracias a su integración automática con Visual Studio, su sencillez y su curva de aprendizaje accesible. Lo recomiendo para pruebas directas y proyectos institucionales simples.
- **NUnit**, por otro lado, ofrece mayor expresividad, control y posibilidades de configuración. Me pareció especialmente útil al validar excepciones con precisión, como en el ejercicio del factorial.

**Conclusión:** Tras comparar ambos frameworks, se concluye que **NUnit ofrece una mayor robustez y flexibilidad** en la definición y gestión de pruebas unitarias. Aunque **MSTest** fue adecuado para ejercicios más directos como el de la sombra solar, **NUnit** demostró ser más completo y expresivo, especialmente en escenarios donde se requiere un control detallado de las condiciones y aserciones. Por tanto, preferimos **NUnit** para proyectos que demandan pruebas más complejas o personalizadas.



#### 4.1. Conclusiones

- La metodología TDD permitió mejorar la calidad del código al enfocarse primero en los requerimientos esperados antes de desarrollar la lógica.
- Las pruebas unitarias realizadas con MSTest y NUnit fueron fundamentales para verificar el correcto funcionamiento de cada caso implementado.
- El ciclo Rojo-Verde-Refactor fue evidente en ambas implementaciones (factorial y sombra solar), asegurando que el código evolucionara con mejoras constantes sin perder fiabilidad.
- La ejecución de los programas desde consola evidenció que la funcionalidad implementada responde adecuadamente a entradas dinámicas, incluyendo validaciones y respuestas a casos extremos.
- El uso de LINQ y buenas prácticas en la refactorización ayudó a obtener código más limpio, mantenible y robusto.

#### 4.2. Recomendaciones

- Mantener la práctica del enfoque TDD para el desarrollo de nuevas funcionalidades, ya que fomenta la claridad en los requisitos y la prevención de errores futuros.
- Documentar cada etapa del proceso (fallo, éxito y refactorización), lo cual facilita el análisis y comprensión del desarrollo por parte de otros colaboradores.
- Utilizar tanto MSTest como NUnit en distintos escenarios, para conocer las fortalezas de cada framework y tener mayor flexibilidad en proyectos reales.
- Reforzar el conocimiento de expresiones lambda, LINQ y manejo de excepciones, ya que son recursos clave en el desarrollo moderno con C#.
- Integrar estas prácticas en proyectos grupales para mejorar la colaboración, trazabilidad y calidad del software desde las primeras etapas.

#### 4.3. Anexos

<https://drive.google.com/file/d/1z1DrZHZ-jgEvFW8VWGinS3D5q7ZphpS3/view?usp=sharing>