

Explicación del Código de la Prueba Unitaria

1. Referencias y Librerías

El código comienza con la inclusión de varias librerías y dependencias necesarias para la prueba unitaria. Entre ellas se encuentran:

- **AutoMapper**: Biblioteca para el mapeo de objetos entre diferentes modelos.
- **Moq**: Framework de mocking que permite simular comportamientos de objetos para realizar pruebas unitarias.
- **SIGESPROC**: Espacio de nombres que contiene la lógica de negocio y acceso a datos de la aplicación.
- **Microsoft.VisualStudio.TestTools.UnitTesting**: Proporciona las herramientas necesarias para la ejecución de pruebas unitarias en Visual Studio.

2. Clase PlanillaUnitTest

La clase `PlanillaUnitTest` está decorada con el atributo `[TestClass]`, lo que indica que contiene pruebas unitarias. La clase tiene las siguientes propiedades y métodos clave:

2.1 Constructor de la Clase

El constructor `PlanillaUnitTest` inicializa el servicio `PlanillaService` y realiza la configuración de mocks necesarios. El uso de `Mock<PlanillaRepository>` permite simular el comportamiento del repositorio de planillas para pruebas. Además, se configura el mapeador utilizando AutoMapper, y se asignan las dependencias necesarias para el servicio.

2.2 Método PlanillaCreate

El método `PlanillaCreate` está decorado con `[TestMethod]`, lo que indica que es una prueba unitaria. Este método prueba la funcionalidad de la inserción de una nueva planilla mediante el servicio. Los pasos son:

- Se utiliza `MockPlanillaRepository.Setup()` para simular la inserción de una planilla.
- El servicio `InsertarPlanilla` es llamado con un objeto planilla simulado.
- Finalmente, se verifican los resultados utilizando `Assert.IsInstanceOfType` y `Assert.IsNotNull` para asegurarse de que el resultado sea válido.

2.3 Método PlanillaListar

El método `PlanillaListar` también está decorado con `[TestMethod]` y prueba la funcionalidad de listar planillas. Los pasos son los siguientes:

- Se crea una lista simulada de planillas usando `List<tbPlanillas>()` con algunos datos de

prueba.

- Se utiliza `MockPlanillaRepository.Setup()` para simular el retorno de esa lista cuando se llame al método `ListarPlanilla` del repositorio.
- El servicio es llamado con un parámetro (en este caso, `3`), y se validan los resultados con `Assert.IsInstanceOfType` y `Assert.IsNotNull`.

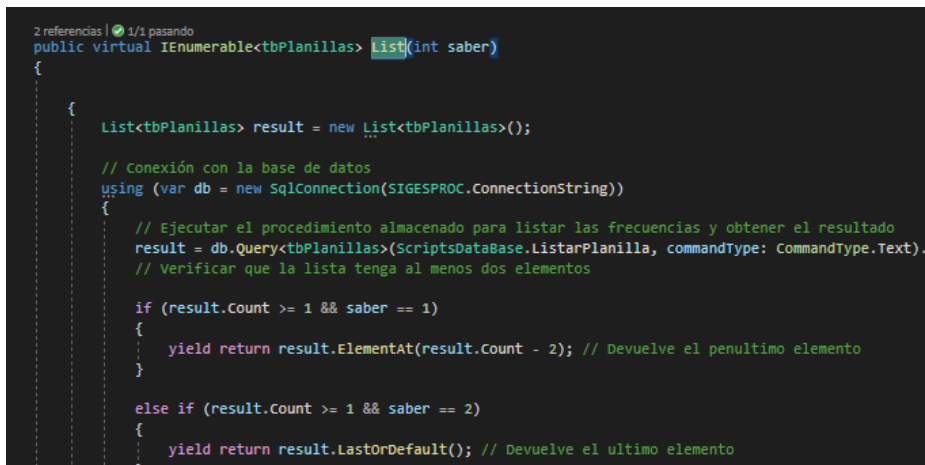
Conclusión

El código presentado es un ejemplo típico de cómo realizar pruebas unitarias con dependencias simuladas en .NET utilizando Moq. Se verifican correctamente las funciones de inserción y listado de planillas mediante la validación de los resultados esperados. Estas pruebas aseguran que la lógica de negocio dentro de `PlanillaService` funcione correctamente sin depender directamente de una base de datos real.

Nota

Si estoy probando nada mas el servicio como el servicio depende de un mi metodo que esta en mi repositorio ocupo simular la respuesta que me daría el repositorio sin necesidad de que entre al repositorio y para esto se usan los mock pero para que esto funcione el metodo del repositorio en este caso necesita ser un virtual.

Ejemplo:



```
2 referencias | 1/1 pasando
public virtual IEnumerable<tbPlanillas> List(int saber)
{
    {
        List<tbPlanillas> result = new List<tbPlanillas>();

        // Conexión con la base de datos
        using (var db = new SqlConnection(SIGESPROC.ConnectionString))
        {
            // Ejecutar el procedimiento almacenado para listar las frecuencias y obtener el resultado
            result = db.Query<tbPlanillas>(ScriptsDataBase.ListarPlanilla, CommandType.Text);
            // Verificar que la lista tenga al menos dos elementos

            if (result.Count >= 1 && saber == 1)
            {
                yield return result.ElementAt(result.Count - 2); // Devuelve el penultimo elemento
            }

            else if (result.Count >= 1 && saber == 2)
            {
                yield return result.LastOrDefault(); // Devuelve el ultimo elemento
            }
        }
    }
}
```

En este caso el metodo del repositorio es virtual pq estoy probando el servicio pero en caso de que pruebe el controlador como mi controlador depende de mi servicio ocuparia tambien simular la respuesta de mi servicio con los mocks y para eso ocuparia pasar mi servicio a virtual.

Ejemplo:

4 referencias | 1/1 pasando

```
public virtual ServiceResult ListarPlanilla(int saber)
{
    var result = new ServiceResult();
    try
    {
        var list = _planillaRepository.List(saber);
        return result.Ok(list);
    }
    catch (Exception ex)
    {
        return result.Error(ex.Message);
    }
}
```