Reflection and Assembly - Project 1

Your task is to create your very own lightweight unit test framework from scratch.

It should contain 2 components:

- **MiniTest** library - containing test attributes allowing users to mark classes and methods as test containers and assertion methods.
- **MiniTestRunner** executable - an application that **dynamically** loads assemblies containing tests, searches for test containers, executes found tests, and presents the test results in the console.

MiniTest

Test Attributes

The library should provide the following attributes used to mark classes and methods as test containers and manage the test lifecycle.

1. **TestClassAttribute** marks a class as a container for test methods.
2. **TestMethodAttribute** marks a method as a unit test to be executed.
3. **BeforeEachAttribute** defines a method to be executed before each test method.
4. **AfterEachAttribute** defines a method to be executed after each test method.
5. **PriorityAttribute** sets a priority (integer) for test prioritization, with lower numerical values indicating higher priority.
6. **DataRowAttribute** enables parameterized testing by supplying data to test methods.
   - it should accept an array of objects (`object?[]`) representing test data.
   - optionally takes a string parameter that documents the test data set.
7. **DescriptionAttribute** allows the inclusion of additional description to a test or a test class.

Assertions

The library should also provide methods to verify test success or failure conditions. It should be handled by a static class `Assert`, which includes methods that will handle assertions.

1. `ThrowsException<TException>(Action action, string message = "")`: Confirms that a specific exception type is thrown during a given operation.
2. `AreEqual<T>(T? expected, T? actual, string message = "")`: Verifies equality between expected and actual values.
3. `AreNotEqual<T>(T? notExpected, T? actual, string message = "")`: Ensures that the expected and actual values are distinct.
4. `IsTrue(bool condition, string message = "")`: Confirms that a boolean condition is true.
5. `IsFalse(bool condition, string message = "")`: Confirms that a boolean condition is false.
6. `Fail(string message = "")`: Explicitly fails a test with a custom error message.

All methods should throw an exception when the condition is not met.
Additionally, all methods should take an optional message string that describes the purpose of the assertion.
An example implementation of `IsTrue`:

```
public static void IsTrue(bool condition, string message = "")
{
    if (!condition)
    {
        throw new AssertionException(message);
    }
}
```

Exception Handling

Implement a custom exception, `AssertionException`, a dedicated exception for failed assertions.

Each assertion method must clearly describe the failure. You should use these strings to format a final message:

```
// ThrowsException<TException>:
$"Expected exception type:<{typeof(TException)}>. Actual exception type:<{ex.GetType()}>. {message}"
$"Expected exception type:<{typeof(TException)}> but no exception was thrown. {message}"
// AreEqual<T>:
$"Expected: {expected?.ToString() ?? "null"}. Actual: {actual?.ToString() ?? "null"}. {message}"
// AreNotEqual<T>:
$"Expected any value except: {notExpected?.ToString() ?? "null"}. Actual: {actual?.ToString() ?? "nu
```

Remarks

`AuthenticationService.Tests` already include tests marked with attributes described earlier. Make sure the tests defined there are compatible with the library attributes and assertions you are writing.

MiniTestRunner

The `MiniTestRunner` is a console application that handles the discovery and execution of tests, providing output and summaries upon completion.

Input

The application takes the assembly file paths as command-line arguments. These assemblies should contain the test classes and methods defined using the `MiniTest` framework.

Example:

```
MiniTestRunner path/to/test-assembly1.dll path/to/test-assembly2.dll
```

Assembly Loading

Use `AssemblyLoadContext` for **dynamic loading** of test assemblies without affecting the primary runtime context. Make sure contexts are unloadable (`isCollectible`) for efficient memory use.

Test Discovery

Search provided assemblies for all classes decorated with `TestClassAttribute`.

In each test class:

- Discover test methods (`TestMethodAttribute`).
- Find parameterized tests (`DataRowAttribute`) for execution with multiple inputs.
- Identify `BeforeEach` and `AfterEach` methods for setup/teardown logic.
  - Late bind `BeforeEach` and `AfterEach` methods to a delegate.

Gracefully skip test classes without a parameterless constructor. Ignore test methods or attributes with incompatible configurations (e.g., parameter mismatch for `DataRow`).
Write a warning message to the console in case of such configuration incompatibilities.

Test Execution

Methods should be executed in order based on the `PriorityAttribute`, where a lower numerical value indicates a higher priority. Methods without the attribute have a priority of `0`. Methods with the same priority should be executed in alphabetical order based on the method name.

For each test class, for each test, execute:

1. `BeforeEach` method through the previously bound delegate.
2. The test method itself
3. `AfterEach` method through the previously bound delegate.

In parameterized tests, each data row provided via `DataRowAttribute` is executed separately (counts as a separate test), passing the specified parameters to the test method.

Test counts as failed if, during the execution of the test method, an unhandled exception was thrown.

Output and Formating

Display individual test results:

- Test status: `PASSED` or `FAILED`.
- Failure reasons and exception messages for failed tests.
- Output the description (if provided) for the class or test method.

Display summary after each test class:

- Total number of tests in a class.
- Number of passed and failed tests in a class.

After running all tests from an assembly, summarize the number of tests (total, failed, passed).

Colorize console output for better readability:

- Green for passed tests.
- Red for failed tests.
- Yellow for test-related warnings (e.g., incompatible configuration, no parameterless constructor).

An example output is included in `example.output.txt`

Scoring

- 2p. - MiniTest library containing attributes and assertions
- 1p. - Dynamically loading and unloading assemblies
- 2p. - Tests discovery
- 3p. - Execution of the tests and output reporting
- 2p. - Code quality (maximum of 25% of points scored from other requirements)