Programming 3 - Advanced

Laboratory 8 - Unit Tests and Exceptions

Agenda:

- Unit testing in Software Development
- Test Driven Development (TDD) approach
- C# testing frameworks: MSTest, NUnit, xUnit
- Unit Test in VS
- UnitTest tasks:
    - Task 1: Authorization Service (TDD)
    - Task 2: Roman Numbers Converter
    - Task 3: Array sorting

Unit testing in Software Development

Unit testing is a fundamental practice in software development that focuses on verifying the functionality of individual components in isolation. Such unit is typically the smallest testable part of an application, such as a function or method. The primary goal is to ensure each unit behaves as expected, catching bugs early and improving code quality.

Importance of Unit Testing

Unit tests play a crucial role in early bug detection. By isolating small pieces of code and verifying their behavior, developers can identify and resolve issues before they escalate into larger, more complex problems. This early intervention reduces the overall cost and time spent on debugging and improves the reliability of the codebase.

Beyond bug detection, unit testing enhances code quality. Writing tests encourages developers to think carefully about their code's functionality and structure, leading to more modular and maintainable solutions. Tests also serve as documentation, illustrating how different parts of the code are intended to behave. New team members can review these tests to quickly understand the system's functionality.

Best Practices for Unit Tests

Effective unit testing involves covering a wide range of scenarios, including edge cases and invalid inputs. Proper unit tests should:

- test small, independent units - each test should focus on a specific behavior or scenario,
- clearly describe what the test is verifying with meaningful test names,
- test edge cases,
- be simple and easy to understand, maintain, and extend.

While unit testing offers significant benefits, it’s important to balance thorough testing with practicality. Overly complex or redundant tests can become a maintenance burden, while insufficient testing leaves critical functionality unverified.

Unit Tests vs. Integration Tests

When unit testing software, it is important to test only single, independent units to ensure each component behaves as expected in isolation. However, when classes or modules are intertwined, achieving this separation can be challenging. This is why it's vital to distinguish between unit tests and integration tests. Unit tests focus on verifying individual functions or components in isolation, ensuring they behave correctly under various conditions; they are fast, small in scope, and avoid external dependencies. In contrast, integration tests evaluate how multiple components interact, often involving real or simulated dependencies like databases or APIs, and provide a broader view of system functionality. Together, unit and integration tests provide comprehensive coverage. Unit tests catch low-level bugs, while integration tests ensure the entire system functions cohesively. Balancing both types of tests is essential for building reliable and maintainable software.

Designing Unit Tests

Designing a Test Suite

A well-designed test suite ensures comprehensive validation of software behavior by including a variety of test cases. These test cases should cover the following scenarios:

- Typical Correct Input Data: Verify that the system handles common and expected inputs correctly.
- Boundary Values: Test inputs at the edge of allowable ranges, such as zero, maximum (MaxValue), and minimum (MinValue) values for a given data type.
- Special Boundary Data: Include cases such as empty inputs, very large datasets, or collections where all elements are equal.
- Null References: For reference types, ensure tests handle null inputs gracefully.
- Exception Cases: Validate that methods throw the correct exceptions for invalid or unexpected inputs.
- Randomly Generated Data: For some problems, using randomized inputs can help uncover edge cases and improve test robustness.

In general, each test suite should include at least one representative from each category of input data, covering both valid and invalid cases.

Test code coverage

To measure the effectiveness of a test suite, developers use test code coverage, which indicates the percentage of source code executed during testing. Higher code coverage suggests a lower likelihood of undetected errors. Ideally, a well-designed test suite should aim for at least 70% code coverage, ensuring that most of the codebase is thoroughly tested.

Designing individual tests

Each test should follow a structured approach, often referred to as the Arrange-Act-Assert (AAA) pattern:

1. Arrange: Set up the test environment and prepare the necessary data or configuration.
2. Act: Execute the action or method being tested.
3. Assert: Verify that the outcome meets the expected results.
4. Clean Up: Reset or clean the test environment (optional, but often necessary for certain setups).

Best Practices for Unit Test Design:

- Each test case should focus on a single aspect of the system. If multiple components are tested together, it becomes an integration test rather than a unit test.
- Tests should not depend on one another. While a failure in one test might logically impact another (such as subcases), the execution order of tests should not affect their outcomes.
- Avoid invoking the method under test directly within an assertion statement, except when testing exception handling. Steps 2 (Act) and 3 (Assert) should remain distinct.
- Tests should confirm that methods perform the expected actions and do not inadvertently alter data that should remain unchanged.

Test Driven Development (TDD) approach

Test-Driven Development (TDD) is a software development methodology that emphasizes writing tests before implementing the actual code. This approach ensures that development is driven by clearly defined requirements and expected behavior. TDD follows a simple cycle known as Red-Green-Refactor:

1. Red – Write a failing test: The process begins by writing a test that defines the desired functionality. Since the code to fulfill the requirement doesn't exist yet, the test is expected to fail. This "red" state highlights the need for implementation.

2. Green – Write minimal code to pass the test: Next, you write just enough code to make the test pass. The goal is not to perfect the solution but to create a working implementation that meets the test's requirements. Once the test passes, the code is in the "green" state.

3. Refactor – Improve the code: With a passing test, you can now safely refactor the code to improve its structure, readability, or efficiency. Refactoring ensures that the code remains clean and maintainable without changing its behavior. The test continues to verify that the functionality remains intact.

TDD offers several key benefits that contribute to building high-quality software. By writing tests before the code, developers clarify requirements early, reducing ambiguity and ensuring a clear understanding of the desired functionality. This process encourages modular, testable code, leading to better design and architecture. TDD also minimizes debugging time by catching issues early in development, allowing for quicker and more focused problem resolution. Additionally, with a comprehensive suite of tests in place, developers can confidently refactor code, knowing that any unintended changes will be detected.

C# testing frameworks: MSTest, NUnit, xUnit

In C#, several popular testing frameworks are available, each with its unique features, advantages, and philosophies. The three most widely used frameworks are MSTest, NUnit, and xUnit.

MSTest is Microsoft's built-in testing framework and integrates seamlessly with Visual Studio. It is straightforward to set up, especially for projects already using the Microsoft ecosystem.

NUnit is a popular open-source framework known for its rich feature set and flexibility. It provides a wide range of attributes and assertions, enabling developers to write more complex and expressive tests.

xUnit is a modern, lightweight testing framework designed to align with contemporary testing practices. It emphasizes simplicity and code readability, using conventions over configuration. xUnit promotes the use of constructor injection for setup and teardown logic instead of attributes like [Setup] or [TearDown].

There are fans of each framework, and in our case, at this point of learning C#, the choice dean't matter as all these frameworks are similar. Because of that we will stick to the framework designed for Microsoft ecosystem - MSTest.

Unit Test in VS

Automatically Generating Unit Tests

To automatically generate unit tests for a class, follow these steps:

1. Open the solution containing the project for which you want to create unit tests.
2. Navigate to the file with the source code of the class you wish to test.
3. Right-click on the class name to open the context menu.
4. Select "Create Unit Tests" from the context menu.

This process creates a new Unit Test Project within the solution. The new project's name matches the original project's name with the Tests suffix added. Inside this project, a test class is generated, named after the class being tested with the same Tests suffix. For each public method in the original class, the generated test class includes an empty test method named after the original method, with Test appended to the name.

You can also generate unit tests for individual methods:

1. Right-click on the method you want to test.
2. Select "Create Unit Tests" from the context menu.

Note that automatic test generation is only available for classes, their public methods, and public constructors. It does not support generating tests for structures or non-public class members, including properties, operators, or indexers, even if they are public.

Manually Creating Unit Tests

You can also create unit tests manually, which offers greater flexibility. Follow these steps to create a test project manually:

1. Open the solution containing the project you want to test.
2. From the main menu, select File → New → Project → Installed → Visual C# → Test → Unit Test Project (.NET Framework).
3. In the Solution field, select "Add to solution."
4. Add a reference to the assembly you want to test from the new unit test project.

The manually created test project is initially empty. You need to create test classes and methods yourself. These classes and methods can be named freely, but adopting the Tests suffix convention (as used in auto-generated tests) is recommended for clarity and consistency. You can also add new classes or methods to an automatically generated test project and modify their names as needed.

Running Unit Tests

Running Individual Tests

To run a specific test method:

1. Right-click on the method's name in the code editor.
2. Select "Run Test(s)" from the context menu.

Running All Tests in a Class

To execute all test methods within a specific class:

1. Right-click on the class name in the code editor.
2. Select "Run Test(s)" from the context menu.

Using the Test Explorer

A more convenient way to manage and run tests is through the Test Explorer window:

1. Open the Test Explorer by selecting Test → Windows → Test Explorer from the main menu.
2. In the Test Explorer, you can view, select, and run individual or multiple tests.

Running Tests from the Main Menu

You can also run tests directly from the main menu:

1. Go to Test → Run.
2. Choose the appropriate test group to execute.

Organizing Tests with Playlists

Tests can be organized into playlists, allowing you to run specific sets of tests:

1. Open the Test Explorer window.
2. Right-click on the test you want to add.
3. Select "Add to Playlist" and choose an existing playlist or create a new one.

Playlists provide a convenient way to group related tests, enabling you to focus on specific parts of your test suite during development or debugging.

Unit Tests attributes

There are several attributes associated with unit tests. They are used by the unit test engine to automatically run tests. The list of basic attributes is presented below (it isn't a complete list of unit test attributes):

- TestClassAttribute

This attribute should precede a testing class declaration.

- TestMethodAttribute

This attribute should precede a testing method definition. It informs the unit test engine that this method should be automatically invoked by the engine.

- DataRowAttribute

Attribute to define inline data for a testing method

- TestInitializeAttribute

The method which is preceded by this attribute is automatically invoked by the unit test engine before each testing method (i.e. a method preceded by TestMethodAttribute) of a given testing class. This method is to prepare the test environment before each test.

- TestCleanupAttribute

The method which is preceded by this attribute is automatically invoked by the unit test engine after each testing method (i.e. a method preceded by TestMethodAttribute) of a given testing class. This method is to clean-up the test environment after each test.

- ClassInitializeAttribute

The method which is preceded by this attribute is automatically invoked by the unit test engine once, before invoking all testing methods (i.e. methods preceded by TestMethodAttribute) of a given testing class. This method is to prepare common test environment before all tests from a given testing class (it is invoked before the method preparing the environment for a single test).

- ClassCleanupAttribute

The method which is preceded by this attribute is automatically invoked by the unit test engine once, after invoking all testing methods (i.e. methods preceded by TestMethodAttribute) of a given testing class. This method is to clean-up common test environment after all tests from a given testing class (it is invoked after the method cleaning-up the environment for a single test).

- AssemblyInitializeAttribute

The meaning of this attribute is similar to the meaning of ClassInitializeAttribute attribute, but it is related to the whole testing assembly not to a specified testing class. This attribute is rarely used.

- AssemblyCleanupAttribute

The meaning of this attribute is similar to the meaning of ClassCleanupAttribute attribute, but it is related to the whole testing assembly not to a specified testing class. This attribute is rarely used.

- ExpectedExceptionAttribute

The constructor of this attribute has to specify the exception type. This attribute should precede testing methods which are expected to throw exception of the specified type. Note: Nowadays it is preferred to specify expected exceptions by ThrowsException methods of Assert class.

- IgnoreAttribute

Testing methods or classes preceded by this attribute are ignored by the unit test engine (tests aren't run). It is possible (but not necessary) to pass to the attribute constructor a string parameter which describes the reason for ignoring the tests.

- TimeoutAttribute

This attribute sets timeout for a given test. Timeout value (in miliseconds) is passed to the attribute constructor as parameter. Note: If the test is broken by timeout the test clean-up method isn't executed.

Unit Tests asserts

The Assert class provides a set of static methods to verify various conditions in unit tests. These methods play a crucial role in validating expected outcomes, and they throw an AssertFailedException if the specified condition is not met. Below are some of the most commonly used assertions:

- AreEqual: Verifies that two objects are equal; the assertion fails if they are not.
- AreNotEqual: Verifies that two objects are not equal; the assertion fails if they are equal.
- IsTrue: Checks if a specified condition is true; the assertion fails if it is false.
- IsFalse: Checks if a specified condition is false; the assertion fails if it is true.
- IsNull: Tests if a specified object is null; the assertion fails if the object is not null.
- IsNotNull: Tests if a specified object is not null; the assertion fails if it is null.
- Fail: Causes the test to fail unconditionally; useful for marking incomplete tests or verifying unreachable code.
- ThrowsException: Verifies that a delegate throws an expected type of exception; the assertion fails if no exception or a different type of exception is thrown.

In addition to the Assert class, there are specialized classes like CollectionAssert and StringAssert for validating collections and strings, respectively. These classes provide methods to check conditions related to the elements within collections or specific characteristics of strings.

Most assertion methods offer multiple overloaded versions, allowing you to pass additional parameters such as a custom message. This message helps describe the assertion and is displayed in the test framework's output, aiding in debugging and clarifying test failures.

Task 1 - Authentication Service (TDD)

In this task, we want to implement authentication service that will will have simple functionality for:

1. registering new users,
2. logging in,

and will store registered users and their password hash, as well as the list of currently logged in users.

In this task we will constrain valid usernames with the requirements as given below:

- between 8 to 32 characters
- starting with the letter
- only letters, numbers and underscore "_" are allowed

Valid passwords should fulfill below requirements:

- between 8 to 16 characters
- at least one special character (?=.*[!@#$%^&*()_+-=[]{}|\,.<>?])
- at least one small letter, one upper letter
- at least one number

We will do this task with the TDD approach.

Creating Tests Project

1. Create the solution with a template for class library (or console application, it doesn't matter). Name the project "AuthenticationService". In created project we will write our authentication service. Add one empty class "AuthService".
2. Right click the solution name and choose "Add -> New Project".
3. In the templates find "MSTest Test Project". Name it "AuthenticationService.Tests"
4. In order to be able to test classes from another project we have to add references to this project. Right click on Tests project, choose "Add-> Project Reference" and choose the "AuthenticationService" project.
5. Rename the test file to "AuthServiceTests.cs"

In this file we will design the tests for the authentication service from our main project.

First we will write tests for registering functionality.

Register - tests and implementation

1. First we will add new function to the AuthService.

In "AuthService.cs" add:

```
public class AuthService
{
    Dictionary<string, string> _users = new();

    public int GetRegisteredUsersCount() => _users.Count;

    public bool Register(string username, string password)
    {
        throw new NotImplementedException();
    }
}
```

In the `_users` dictionary we will store usernames as the keys and hashed passwords as the values.

The `GetRegisteredUsersCount()` method will help us keep track on the number of registered users.

2. As in proper TDD approach, we will first design few tests for `Register()` function.

We can think about the different scenarios, few of them are:

- a user with valid username and valid password tries to register,
- a user with invalid username tries to register,
- a user with invalid password tries to register,
- a user with username already existing in database tries to register,
- a user with username non-existing in database tries to register.

There can be much more scenarios, we encourage you to think about other cases and write you own custom scenarios.

The above scenarios can be written as tests shown below:

```
[TestClass]
public class AuthServiceTests
{
    [TestMethod]
    public void Register_NewUsername_ShouldAddNewUser()
    {
        var authService = new AuthService();
        var result = authService.Register("SuperUser", "Kpsv#1Au9");
        Assert.IsTrue(result);
        Assert.AreEqual(authService.GetRegisteredUsersCount(), 1);
    }

    [TestMethod]
    public void Register_InvalidUsername_ShouldRejectNewUser()
    {
        var authService = new AuthService();
        var result = authService.Register("superme", "StrongPass_123");
        Assert.IsFalse(result);
        Assert.AreEqual(authService.GetRegisteredUsersCount(), 0);
    }

    [TestMethod]
    public void Register_InvalidPassword_ShouldRejectNewUser()
    {
        var authService = new AuthService();
        var result = authService.Register("SuperUser", "StrongPass123");
        Assert.IsFalse(result);
        Assert.AreEqual(authService.GetRegisteredUsersCount(), 0);
    }

    [TestMethod]
    public void Register_ExistingUsername_ShouldRejectExistingUser()
    {
        var authService = new AuthService();
        authService.Register("SuperUser", "StrongPass_123");
        var result = authService.Register("SuperUser", "StrongerPass_123");
        Assert.IsFalse(result);
        Assert.AreEqual(authService.GetRegisteredUsersCount(), 1);
    }

    [TestMethod]
    public void Register_TwoDifferentUsernames_ShouldAddBothUsers()
    {
        var authService = new AuthService();
        var registerResult1 = authService.Register("SuperUser", "StrongPass_123");
        var registerResult2 = authService.Register("YetAnotherSuperUser",
"StrongerPass_123");
```

```
        Assert.IsTrue(registerResult1);
        Assert.IsTrue(registerResult2);
        Assert.AreEqual(authService.GetRegisteredUsersCount(), 2);
    }
}
```

When naming the tests, there are many conventions. In this, we first name the tested function, then we name the scenario/input type, and as the last part we specify the expected output.

When you try to run the tests, all will fail.

Username/password validation - tests and implementation

1. In order to check the usernames and passwords if they are correct, we will add new class `CredentialsValidator` in "AuthenticationService" project.

```
public static class CredentialsValidator
{
    public static bool ValidateUsername(string username)
    {
        throw new NotImplementedException();
    }

    public static bool ValidatePassword(string password)
    {
        throw new NotImplementedException();
    }
}
```

These methods are perfect to write unit tests for.

2. Create CredentialsValidatorTests.cs file in "AuthenticationService.Tests" project.

We can write several passwords and usernames that are valid/invalid given the restrictions in our task. For each such scenario we can add custom test, e.g.

```
[TestMethod]
public void CredentialsValidator_UsernameStartsWithNumber_ShouldFail()
{
    Assert.IsFalse(CredentialsValidator.ValidateUsername("123abc456"));
}

[TestMethod]
public void CredentialsValidator_UsernameStartsWithUnderscore_ShouldFail()
{
    Assert.IsFalse(CredentialsValidator.ValidateUsername("_username"));
}
```

... or we can add one method for batch checking different invalid usernames scenarios. While the first approach is more descriptive, the second one is muuch shorter:

```
[DataRow("123abc456")]
[DataRow("_username")]
[DataRow("user")]
[DataRow("\\(^o^)/")]
[DataRow("User@name")]
[DataRow("SuperLongUsernameNobodysGoingToRepeat")]
[TestMethod]
public void CredentialsValidator_InvalidUsername_ShouldFail(string s)
{
    Assert.IsFalse(CredentialsValidator.ValidateUsername(s));
}
```

3. Add test scenarios for valid username as well. Add additional methods for valid/invalid passwords.

Example tests:

```
[DataRow("UserName")]
[DataRow("MarioTheStrong_01")]
[DataRow("hello_kitty_")]
[DataRow("smiley2137")]
[DataRow("qwertyqwerty")]
[DataRow("ONLY_CAPSLOCK")]
[TestMethod]
public void CredentialsValidator_ValidUsername_ShouldSucceed(string s)
{
    Assert.IsTrue(CredentialsValidator.ValidateUsername(s));
}

[DataRow("QWErtyASDfgh_123!@#")]
[DataRow("password")]
[DataRow("1234567Aa")]
[DataRow("PaSsWoRd!")]
[DataRow("STRONG_PASS!")]
[DataRow("\\(^o^)/")]
[TestMethod]
public void CredentialsValidator_InvalidPassword_ShouldFail(string s)
{
    Assert.IsFalse(CredentialsValidator.ValidatePassword(s));
}


[DataRow("o\\(O_0)/o")]
[DataRow("!Nic3Password*")]
[DataRow("(modnaRlat0t")]
[DataRow("PJDs6a!q")]
[TestMethod]
public void CredentialsValidator_ValidPassword_ShouldSucceed(string s)
```

```
{
    Assert.IsTrue(CredentialsValidator.ValidatePassword(s));
}
```

4. Now, with given tests, implement `CredentialValidator` methods. It can be done using Regex/string checking/however you like.

5. Implement `Register()` method of the `AuthService` class. With credentials validator it should be now pretty straight-forward. For now you don't have to worry about password hashing, we will create module for that.

GetUserData - tests and exceptions

To check the actual content of the `_users` dictionary we will write method for getting user data by given username. In order to do that we will write helper `User` class that will store output value.

1. Create `User` class. It should store both user's username and password.

```
public class User
{
    public User(string username, string passwordHash)
    {
        Username = username;
        PasswordHash = passwordHash;
    }

    public string Username { get; set; }
    public string PasswordHash { get; set; }
}
```

Add method `GetUserData(string username)` in `AuthService` class. If no user was found, we will want to throw custom (defined by us) exception. For now it can throw `NotImplementedException`.

Custom Exception: UserNotFoundException

In this workshop, you will be working with a custom exception called `UserNotFoundException`. This exception is a simple example of how to define your own exceptions in C#.

The `UserNotFoundException` class is derived from the built-in `Exception` class, which is the base class for all exceptions in C#. By inheriting from `Exception`, `UserNotFoundException` can be used like any other exception in C#, and it includes all the functionality of the base `Exception` class.

It's implementation is shown below:

```
public class UserNotFoundException : Exception
{
    public UserNotFoundException(string message) : base(message) {}
}
```

The class inherits from `Exception`, meaning it is a custom exception that behaves like the standard exceptions in .NET. This allows you to throw and catch this exception just like any other exception.

The constructor accepts a string parameter called message, which is used to provide a detailed description of the error. This message is passed to the base `Exception` class using the `base(message)` call. The base keyword is used to invoke the constructor of the parent class (`Exception`), ensuring that the message is stored in the exception object.

2. Add tests for `GetUserData(string username)` methodin AuthServiceTests.cs in "AuthenticationService.Tests" project.

We want to check whether getting non-existing user data will throw an error, and existing user should return correct data. Now we will also test wheather password was hashed. If it was, then returned User data `PasswordHash` should be different from passed as the input.

```
[TestMethod]
public void GetRegisteredUserData_NonExistingUsername_ShouldThrowError()
{
    var authService = new AuthService();
    authService.Register("SuperUser", "StrongPass_123");
    Assert.ThrowsException<UserNotFoundException>(() =>
authService.GetRegisteredUserData("superuser"));
}

[TestMethod]
public void GetRegisteredUserData_ExistingUsername_ShouldThrowError()
{
    var authService = new AuthService();
    authService.Register("SuperUser", "StrongPass_123");
    var user = authService.GetRegisteredUserData("SuperUser");
    Assert.AreEqual(user.Username, "SuperUser");
    Assert.AreNotEqual(user.PasswordHash, "StrongPass_123");
}
```

3. Implement method `GetUserData(string username)` in `AuthService` class.

```
public User GetRegisteredUserData(string username)
{
    if(!_users.ContainsKey(username))
        throw new UserNotFoundException("User not found!");
    return new User(username, _users[username]);
}
```

Now only (probably) the test where existing user data is acquired should fail, as most probably you don't have password hashing method implemented yet.

Password hashing - tests and implementation

1. Create new `PasswordHasher` class. Create static method `string HashPassword(string)`, that will create password hash based on the password input. For now it can throw `NonImplementedException`.

2. Create tests for `PasswordHasher` class. In new file `PasswordHasherTests.cs` create class for tests.

If the two passwords are the same, then their hashes will also be the same, with the same salt and seed given. In this task we will not worry about storing the salt or seeds, we will use the same hashing function for all passwords.

If two passwords are not the same, then hashes will be different. As a matter of fact, in typical hashing, when one character is differing, whole hash is entirely different. We will setup such test, so that you will be able to debug such code later and see for youselves.

In the "same password case" we will test 3 random passwords with `DataRow` attribute, in different password we will change only one character:

```
[DataRow("!Nic3Password*")]
[DataRow("(modnaRlat0t")]
[DataRow("PJDs6a!q")]
[TestMethod]
public void PasswordHasher_SamePassword_ShouldHaveSameHash(string s)
{
    var passHash1 = PasswordHasher.HashPassword(s);
    var passHash2 = PasswordHasher.HashPassword(s);
    Assert.AreEqual(passHash1, passHash2);
}

[TestMethod]
public void PasswordHasher_DifferentPasswords_ShouldHaveDifferentHashes()
{
    var passHash1 = PasswordHasher.HashPassword("KaWaRaNo_FTW!");
    var passHash2 = PasswordHasher.HashPassword("KaWaRaNo_FTW?");
    Assert.AreNotEqual(passHash1, passHash2);
}
```

3. Implement `PasswordHasher` hashing method. If you struggle with hash implementation, here is complete implementation using methods from Cryptograhy modules, but we encourage you to look about that in the internet.

```
public static class PasswordHasher
{
    const int keySize = 64;
    const int iterations = 350000;
    static byte[] salt = RandomNumberGenerator.GetBytes(keySize);
    public static string HashPassword(string password)
    {
        var hash = Rfc2898DeriveBytes.Pbkdf2(
```

```
            Encoding.UTF8.GetBytes(password),
            salt,
            iterations,
            HashAlgorithmName.SHA512,
            keySize);
            return Convert.ToHexString(hash);
    }
}
```

The `GetRegisteredUserData_ExistingUsername_ShouldThrowError` test will still fail, if you didn't change `Register` function implementation. You can even refactor the test - to see whether user.PasswordHash is returning the same value as the `PasswordHasher.HashPassword` function output.

Login function - test and implementation

1. Add `bool Login(string username, string password)` methods to AuthService. Add `List<string> _loggedUsers` field to the AuthService class. It will store all logged currently users in the service. Add `GetLoggedUsersCount()` method to get number of logged users.

The class should look like this:

```
public class AuthService
{
    private readonly Dictionary<string, string> _users = new();
    private readonly List<string> _loggedUsersNames = new();

    public int GetRegisteredUsersCount() => _users.Count;

    public int GetLoggedUsersCount() => _loggedUsersNames.Count;

    public bool Register(string username, string password)
    {
        ...
    }
    public bool Login(string username, string password)
    {
        throw new NotImplementedException();
    }

    public User GetRegisteredUserData(string username)
    {
        if(!_users.ContainsKey(username))
            throw new UserNotFoundException("User not found!");
        return new User(username, _users[username]);
    }
}
```

2. Design the tests for `Login` method.

You should consder scenarios such as:

- passing non-existing username
- passing invalid password
- passing valid password
- passing valid password for the second time?

Write your own tests for `Login` method. This time we will not provide the implementation. Remember to use `GetLoggedUsersCount()` method to check whether number of logged users is correct. You can consider adding accesor to the class for logged users, you check for logged user info as well.

This time you can design the tests as you wish.

3. Implement `Login()` method, trying to pass all written tests.

Task 2 - Roman Numbers Converter

You are provided with RomanNumberTests.cs and RomanNumbers.cs files. Add the needed implementation so that all the tests will succeed.

Task 3 - Array Sorting

You are provided with SortingTests.cs and Sorting.cs files. Add the sorting algorithms implementation, so that the tests will succeed.