**\*\*I. Introduction to Unit Testing and Mocking\*\***

**- The importance and purpose of unit testing**

unit testing plays a crucial role in programming. It's a method used to validate that individual units of software code function correctly.

იუნით ტესტირებას მნიშვნელოვანი როლი უკავია პროგრამირებაში. მისი ძირითადი დანიშნულებაა კოდის კონკრეტული ნაწილების გამართულ ფუნქციონირებაში დარწმუნება.

1. \*\*Ensure Code Correctness:\*\* The primary purpose of unit testing is to ensure that each unit of the software (typically functions or methods) operates as intended. By writing tests that confirm the desired behavior, developers can be confident that the code they write is functioning correctly.

უპირველეს ყოვლისა Unit ტესტების პირველი მიზანია ის რომ ჩვენი კოდის სისწორის რაღაც დონის გარანტია შექმნას.

3. \*\*Documentation:\*\* Unit tests can serve as a form of documentation. By looking at the unit tests, one can understand what functionality is expected from the unit of the code. This is helpful not only to the author but to anyone else who might need to understand or maintain the code in the future.

იუნით ტესტები ერთგვარ დოკუმენტაციასაც წარმოადგენენ, ვინაიდან ჩვენ მათში აღვწერთ იმას თუ როგორ უნდა იმუშავოს ჩვენმა კონკრეტულმა unit მა.

4. \*\*Design Aid:\*\* Writing unit tests can help in designing the software. It can lead to better modularization and more decoupled code since each unit of the code should be testable independently of others. This also often leads to better architecture, as it encourages the separation of concerns.

Unit ტესტების წერამ შეიძლება ხელი შეგვიწყოს უკეთესი დიზაინის შექმნაშიც. როდესაც ტესტების წერისას გავაანალიზებთ, რომ გვაწუხებს კონკრეტულ unit ში კოდის რაოდენობა, ან დამოკიდებულებები ან სხვა რამ, საუკეთესო შემთხვევაში ამ პრობლემეის მოგვარებასაც დავიწყებთ ან მიზნად დავისახავთ ამას.

5. \*\*Refactoring Support:\*\* When modifying or refactoring code, having a suite of unit tests is invaluable. The tests can verify that the new code behaves the same way as the old code, ensuring that no functionality is lost or inadvertently changed during the process.

Refactoring ში დაგვეხმარება. ვინაიდან unit ტესტებში აღწერილი გვექნება თუ როგორ უნდა იმუშავოს ჩვენმა აპლიკაციამ, რეფაქტორინგი აღარ შეგვაშინებს, ვინაიდან ტესტები ჩვენი ცვლილების ვალიდაციაში დაგვეხმარება.

6. \*\*Integration Ease:\*\* Unit testing helps in integration by ensuring that individual units of code work correctly before they are assembled together. This can prevent bugs from arising due to the interaction of different units of code.

7. \*\*Quality Assurance:\*\* Unit tests contribute to overall quality assurance. By making sure each individual component of the software works as expected, it improves the overall reliability of the software.

ზოგადად unit ტესტები გვეხმარება ნაკლებად ხარვეზიანი კოდის წერაში და შესაბამისად ამით ეხმარება QA პროცესსაც.

To maximize the effectiveness of unit testing, it's important to write tests that are comprehensive and cover a variety of different conditions, including edge cases. It's also important to maintain and update your unit tests as the software evolves, to ensure that they continue to provide value.

**- Explanation of the concept of Mocking**[What are the benefits and drawbacks of using a mocking framework for unit testing? (linkedin.com)](https://www.linkedin.com/advice/0/what-benefits-drawbacks-using-mocking-framework)

Mock, stub, fake, spy, and test double are all types of "test doubles" used in unit testing to replace real dependencies of the unit being tested. They help to ensure that the unit tests are isolated and don't inadvertently test the behaviors of other parts of the system. Here's a brief summary of each:

1. \*\*Test Double\*\*: This is the generic term for any kind of pretend object used in place of a real object for testing purposes. The name comes from the concept of a stunt double in movies. All of the following are types of test doubles.

2. \*\*Mock\*\*: A mock object is a dummy object used for testing that also contains pre-programmed behavior (like a return value) as well as expectations about calls (how many times a method will be called, what arguments it will be called with, etc.). The test will fail if the expectations are not met.

3. \*\*Stub\*\*: Stubs provide canned answers to calls made during the test. Unlike mocks, they do not throw an exception when an unexpected invocation is made. Stubs are typically used when the testing system doesn't care about the interaction but needs a specific response from the interaction.

4. \*\*Fake\*\*: A fake is a more functional implementation of an object that takes shortcuts for the sake of the test. Fakes have working implementations, but usually take some shortcut which makes them not suitable for production (an in memory „database is a good example).

5. \*\*Spy\*\*: Spies are objects that record information about how they are called. They can be set up to only record information (like how many times methods are called), or they can also be set up to replace methods similar to a stub. They provide more information than stubs because they can tell you if a method was called, how many times it was called, and what arguments it was called with.

These concepts allow you to isolate the unit of code under test and ensure that the tests run quickly, in a controlled environment, and can provide useful feedback, whether they pass or fail.

Moq features

საშუალებას გვაძლევს linq ის დახმარებით მარტივად შევქმნათ მოკები და გამოვიყენოთ მათი სრული პოტენციალი. შეგვიძლია დავთრექოთ მომხდარი ქოლები, დავაბრუნოთ სასურველი მნიშვნელობები, ვისროლოთ ექსეფშენები, გავაკეთოთ კონკრეტული გამოძახებების სექვენსები და ა.შ.

NSubstitute

პრეზენტაციაში ჩავყაროთ პირდაპირ თითო ფიჩერზე თითო სქრინი დოკუმენტაციიდან მერე ვაჩვენოთ პრაქტიკაში რაც გამოგვადგა (ასევე ყველა სხვა ბიბლიოთეკაზე)

practical examples:

Setup Return +

Return for specific args +

Throw exception +

Assert call +

Multiple returns +

**\*\*II. Overview of Mocking Libraries in C#\*\***

**- Introduction to different Mocking libraries: Moq, NSubstitute, Rhino Mocks, etc.**

**- Comparison of different libraries (features, ease of use, etc.)**

**\*\*III. Deep Dive into Moq\*\***

**- Setup and installation of Moq**

**- Basic concepts: Mocks, Stubs, and Fakes**

**- Creating a mock object with Moq**

**- Setting up expectations and verifying behaviour**

**- Mocking methods and properties**

**- Sequential and conditional mocking**

**- Mocking event-related behaviour**

**- Limitations and best practices**

**\*\*IV. Deep Dive into NSubstitute\*\***

**- Setup and installation of NSubstitute**

**- Differences between NSubstitute and Moq**

**- Creating substitutes for interfaces and classes**

**- Checking received calls**

**- Returning results and configuring behaviour**

**- Argument matching**

**- Using NSubstitute with async code**

**- Limitations and best practices**

**\*\*V. Choosing the Right Mocking Framework\*\***

**- Factors to consider when choosing a mocking framework**

**- Pros and cons of using Moq, NSubstitute and other libraries**

**\*\*VI. Advanced Topics and Best Practices\*\***

**- Auto mocking containers**

**- Partial mocks and spies**

**- Best practices in mocking and unit testing**

**- Pitfalls to avoid in mocking**

**\*\*VII. Practical Examples and Case Studies\*\***

**- Walkthrough of simple unit tests using mocking**

**- Case studies showcasing real-world use of mocking in projects**

**\*\*VIII. Q&A and Discussion\*\***

**- Open the floor to questions**

**- Discuss students' experiences with mocking and unit testing**