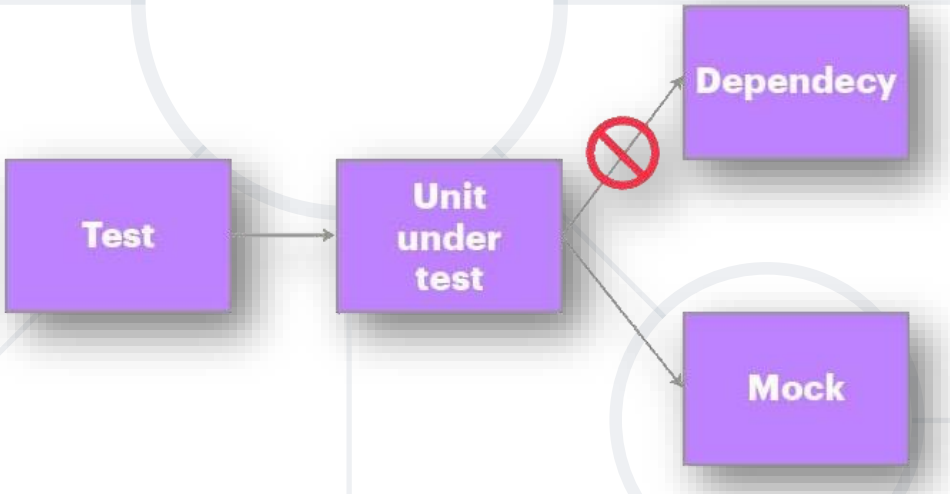
# Unit Testing with Mocking

## Code Reliability Through Isolated Testing




**SoftUni Team**

**Technical Trainers**

Software University

SoftUni

Software University

https://softuni.bg

# You Have Questions?

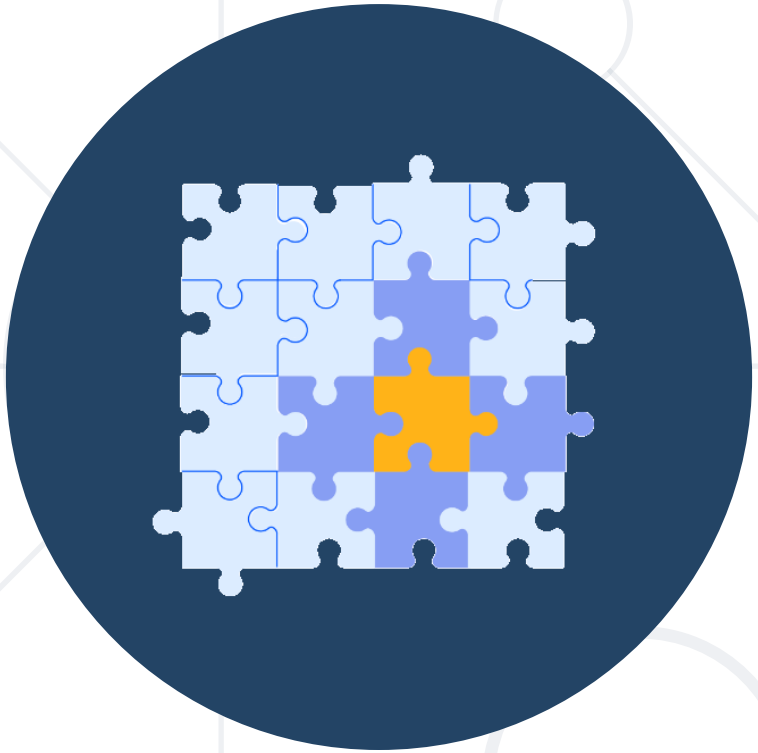
Software University

## sli.do

# #QA-Auto-BackEnd

# Table of Contents





# Unit Testing and Dependencies

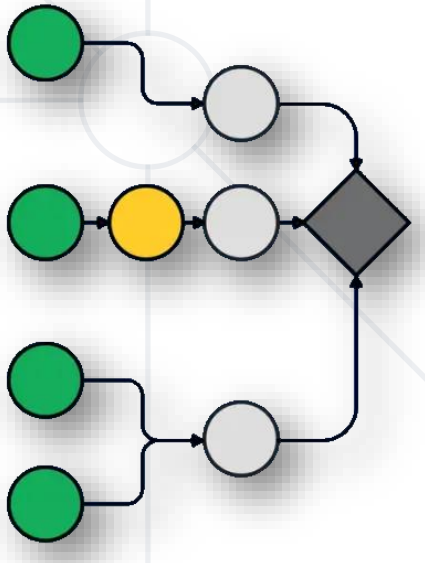
## Components Interaction

# Unit Testing

- Unit tests assess the **smallest testable parts** of an application (typically a **method** or a **function**)
- Tests are designed to **run in isolation**
- Evaluate **single unit's behavior** independently of others
- Each test can be **executed repeatedly** under the **same conditions** and have the **same results**

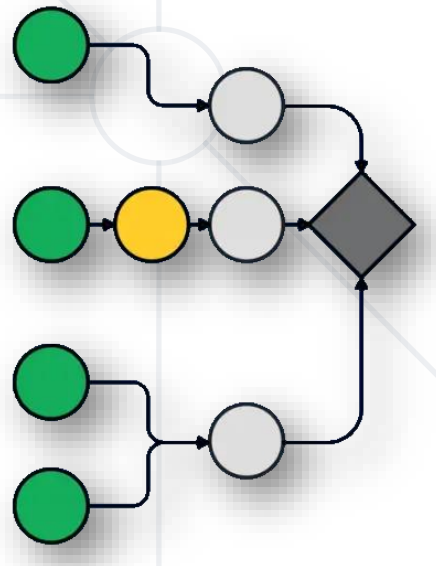
# Understanding Dependencies

- In software, a **dependency** is when a **piece of code requires another component** to function
- While dependencies are vital, they also **complicate unit testing**, as the test relies on external factors
- **Units** under test often **interact** with **databases**, **services**, or other **systems**



# Coupling

- These dependencies between components of a system are known as '**coupling**'
- Coupling refers to how **closely connected different units** of code are within an application
- **High coupling** can make unit tests challenging, because it can tie the behavior of the code you want to test to the behavior of the external dependencies



# Coupling and Testing

- Consider testing an **application** that provides **different greetings** based on the **time of day**
  - **"Good morning!"** is returned from 5 AM to before 12 PM
  - **"Good afternoon!"** - from 12 PM to before 6 PM
  - **"Good evening!"** - from 6 PM to before 10 PM
  - **"Good night!"** - 10 PM to before 5 AM

```
public class GreetingProvider
{
    public string GetGreeting()
    {
        var hour = DateTime.Now.Hour;
        if (hour >= 5 && hour < 12)
        {
            return "Good morning!";
        }
// Code logic…
    }
}
```

# Coupling and Testing

Software University

- **Tests** for the 'Get Greeting' app will **not** be **consistent**
- They **depend on the system time**, which changes throughout the day
- The application is **tightly coupled** with the system clock
- **Direct dependence** on the system's current time
- This close relationship with the system clock makes it **difficult to test** the greeting functionality **in isolation**
- For reliable and predictable testing, it's important to have control over the testing environment

# Decoupling

- The process of **reducing dependencies** between system components
- It allows **individual components** to **change without affecting others**, simplifying tests
- Decoupling with **Interfaces** and **Dependency Injection**:
  - **Using Interfaces**: Define contracts with interfaces to **reduce direct dependencies** on concrete implementations
  - **Implementing Dependency Injection**: Inject dependencies, like interfaces, at runtime to maintain **loose coupling**

# Decoupling

- Modify the code so that **GreetingProvider** doesn't directly depend on the system clock
- Creating an Interface **ITimeProvider**, which has a method to get the current time
- This way, **GreetingProvider** will not directly call **DateTime.Now**
- Modify **GreetingProvider** to accept an **ITimeProvider** through its constructor (dependency injection)
- Pass any implementation of **ITimeProvider**, including a mock one for testing

# Interfaces

```
public interface ITimeProvider
{
    DateTime GetCurrentTime();
}

public class SystemTimeProvider : ITimeProvider
{
    public DateTime GetCurrentTime()
    {
        return DateTime.Now; // Real implementation
    }
}
```

# Implementation

```
public class GreetingProvider
{
    private readonly ITimeProvider _timeProvider;

    public GreetingProvider(ITimeProvider timeProvider)
    {
        _timeProvider = timeProvider;
    }

    public string GetGreeting()
    {
        var hour = _timeProvider.GetCurrentTime().Hour;
        // Remaining logic for returning greetings...
    }
}
```

# Explanation

- With this design, we can:
  - **Easily mock** the ITimeProvider when **writing unit tests** for GreetingProvider, creating a **mock time provider** that returns a specific time
  - **Test** all greeting scenarios reliably **at any time of day**
  - **Consistent test results** regardless of the actual time of day
  - So what is **mocking**?

# Mocking

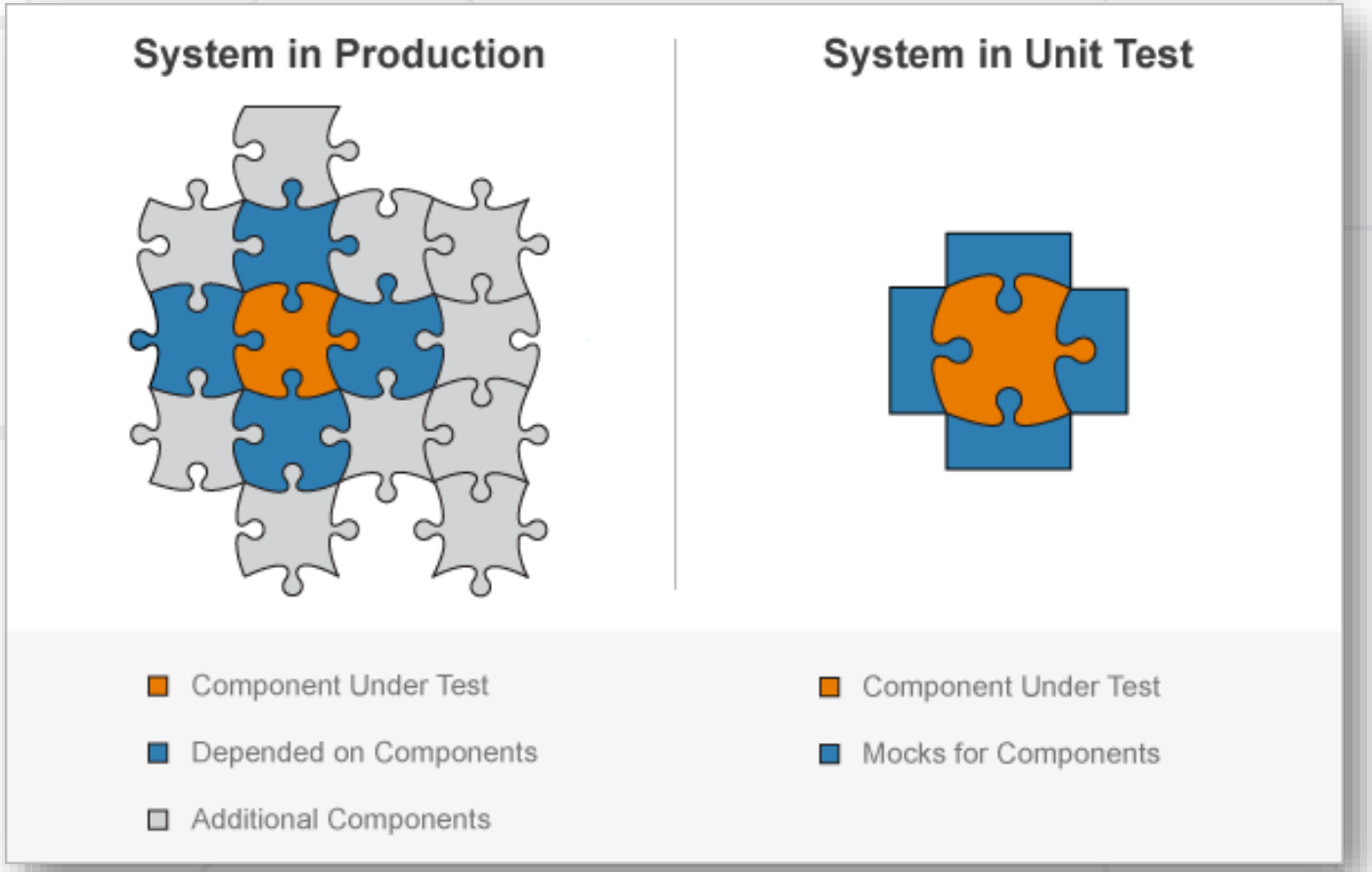
## Simulating External Dependencies in Unit Tests

# What is Mocking?

- **Mocking** – something made as an imitation
- Software practice, primarily used in **Unit Testing**
- When an object under test have **dependencies** on other objects
- To **isolate** the behavior, the other objects are replaced
- The replacements are **mocked objects**
- Mocked objects **simulate** the behavior of the **real objects**
- Basically, **Mocking** is creating objects that **simulate behavior**

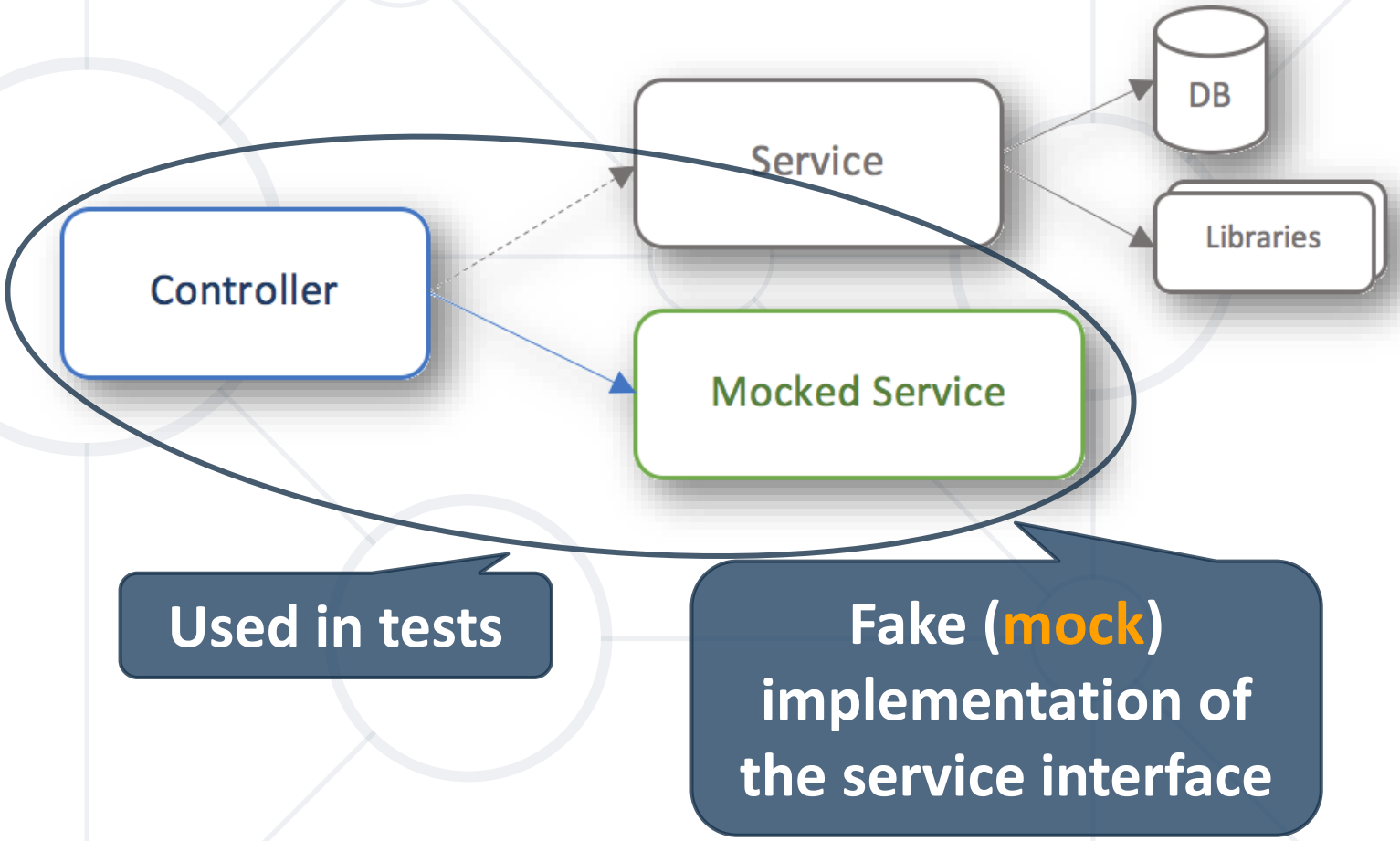
# Why Mocking?

- **Unit testing** should **test a single component**
  - Isolated from all the others
- **External dependencies** should be **mocked** (faked, simulated)



# Mocking

- **Mocking simulates the behavior** of real objects
  - Usually done through interfaces
  - Real implementation (e.g., with a database)
  - Fake implementation (used for the unit tests)



# What to Mock?

- Replace components that are **beyond the scope of the test** or **impractical to include**:
  - **Databases** and data storage systems
  - **External APIs**, such as payment gateways or email services
  - **Network requests** and **responses**
  - **Cloud resources** and **services**
  - **Hardware interfaces** that interact with the software
  - **Authentication APIs** and user **verification systems**


# Moq Framework

## Introduction

# Moq Framework

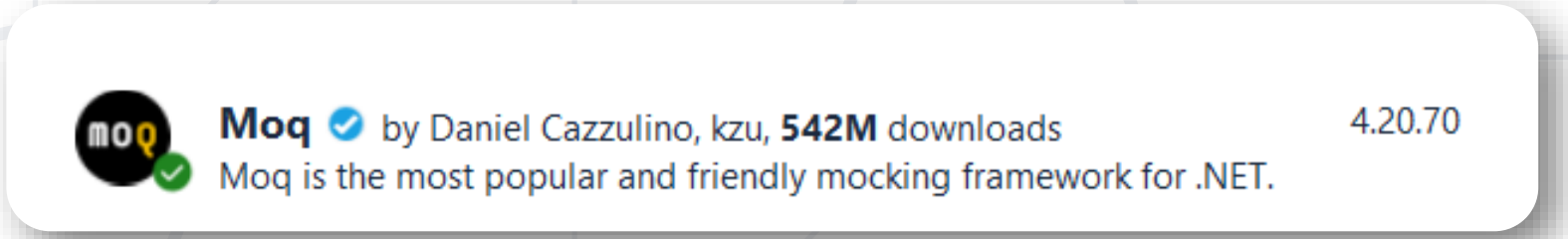
- **Moq** is a .NET mocking framework for **crafting mock objects**
- Enables **simulating behaviors** for unit testing scenarios
- Aids in **isolating unit tests** from real object dependencies
- Used for **imitating object behaviors** and **responses**
- Essential for testing units in **isolation** from databases or APIs
- Key to **confirming object behaviors** in test conditions

# Why Moq?

- Mocking library for .NET developed **from scratch** to **fully leverage**
- **Extremely simple** and does not require any prior knowledge or experience with mocking concepts
- Designed to **quickly set up dependencies** for tests in a very practical manner
- **Clear** and **concise setup** of mocks
- Unique feature called "**Linq to Mocks**"
- **Most popular** mocking library for .NET

# Installing Moq

- Right-click on the test project in the **Solution Explorer**
- Select "**Manage NuGet Packages**"
- Search for the **Moq** package in the NuGet Package Manager
- Install Moq by clicking on the "**Install**" button for the package



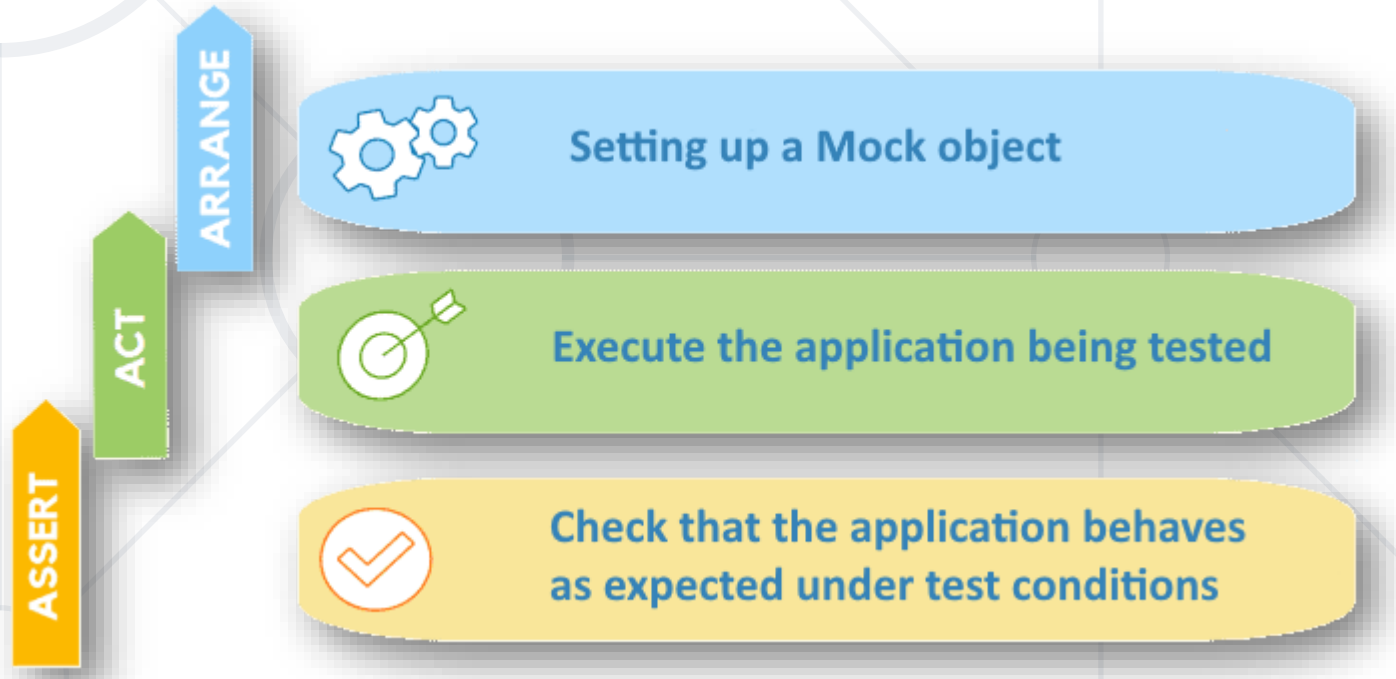


# First Steps with Moq

## Writing Your First Test

# AAA Pattern Recap

- Crafting a basic Moq test, follows the well known AAA pattern
- It involves:
  - Arrange (Create)
  - Act (Test)
  - Assert (Verify)



# Setup Your Testing Project

- **Set Up Your Testing Environment**
- Initialize **Mock** and **GreetingProvider**:
  - Decare a **mock object** for the **ITimeProvider** interface
  - Declare an **instance** of **GreetingProvider**, which will use the **mocked ITimeProvider**

```
 [TestFixture]
    public class GreetingTests
    {
        private Mock<ITimeProvider> mockTimeProvider;
        private GreetingProvider greetingProvider;
//...
```

# Setup Your Testing Project

- In the [**SetUp**] method
  - Initialize the **mock object** for **ITimeProvider before each test**
  - Create an **instance of GreetingProvider**, injecting the **mock ITimeProvider object**

```
[SetUp]
    public void Setup()
    {
        mockTimeProvider = new Mock<ITimeProvider>();
                    greetingProvider = new
                    GreetingProvider(mockTimeProvider.Object);
        }
```

# First Test with Moq

Software University

```
[Test]
    // Test method to verify the greeting at 9 AM
    public void GreetingAt9AmShouldBeGoodMorning()
    { // Set up the mock to return 9 AM when
        GetCurrentTime() is called
        mockTimeProvider.Setup(tp =>
                    tp.GetCurrentTime()).Returns
                    (new DateTime(2024, 1, 1, 9, 0, 0));
     // Call the method under test
        string result = greetingProvider.GetGreeting();
    // Check that the result is as expected for 9 AM
        Assert.That(result, Is.EqualTo("Good morning!"));
      }
```

# Run Your Test

Software University

- **Execute** the test method
  - It **should pass** regardless of the actual system time
  - You've created a **test** that's **independent** of the actual time, thanks to Moq
  - This way, your **test's behavior** remains **predictable** and under your control
- Continue with the **other tests** for the **different greetings** in a similar manner


# Mocking Database Repository

## Simulating Database Interactions with Moq

# The Purpose of a Service Layer

- It contains the business **logic**
- Responsible for **processing data**, enforcing **business rules**, and **making decisions**
- The service layer **interacts** with the **repository** to **fetch or persist data**, but it doesn't need to know **how the data** is actually **stored** or **retrieved**
- This **separation** makes the code more modular and **easier to maintain**

# The Role of a Repository

Software University

- Layer or class that **sits between** the **business logic** and the **data source** (like a database)
- Handles **data access**:
  - **Retrieves data** from the database
  - Performs **CRUD operations**
  - Persists **any changes back** to the database
- **Abstracts** the complexity **of data access**, providing a **cleaner** and more **focused interface** for the business logic to interact with data

# Why Mock the Repository?

- Unit testing the service layer focuses on **isolating** and **testing business logic**
- **Tests** may **fail** due to **external factors** unrelated to the business logic, such as database or network issues
- **Mocking** is used to create a **simulated version** of the **repository** to **mimic** its **behavior**
- A **mock repository allows** to define **expected data returns** for specific calls, **eliminating** the need for a **real database**
- It enables **assess to the service layer's responses** with **various data conditions**, ensuring our service layer reacts correctly

# Problem ItemManagement

- The **ItemManagement** application is a simple **console-based system** designed to **manage items**
- It allows users to **perform basic CRUD operations** on items
- The application **consists of**:
  - **ItemService:** A service layer containing business logic to handle item operations
  - **IItemRepository:** An interface for the data repository, responsible for direct data manipulation in the database

# Problem ItemManagement

- **Setup Provided**
  - The ItemService and IItemRepository are already implemented

```
public interface IItemRepository
{
    void AddItem(Item item);
    Item GetItemById(int id);
    IEnumerable<Item> GetAllItems();
    void UpdateItem(Item item);
    void DeleteItem(int id);
}
```

```
public class ItemService
{
    private readonly IItemRepository _itemRepository;
    public ItemService(IItemRepository itemRepository)
    {
        _itemRepository = itemRepository;
    }
//More Code
}
```

# Problem ItemManagement

- A **test project is set up** with **Moq** and **NUnit frameworks**, ready for writing and running tests
- **Write unit tests** for the **ItemService** class using the **Moq framework**
- The tests should **ensure** that **ItemService correctly interacts** with the **IItemRepository** and adheres to the expected business logic **without relying** on a **real database**
- **Next section** points to some **useful methods** to help you with the task


# Key methods in Moq

## Mimic Any Behavior

# Mock<T> Constructor

- Used to create a mock object of the **specified type T**
- **Example:**

```
var mockRepository = new Mock<IRepository>();
```

- This creates a **mock instance** of the **IRepository interface**

# Setup Method

- **Configures** a **method** of the **mocked object** to **perform** a specific **action** or **return** a specific **value**
- **Example:**

```
mockRepository.Setup(repo =>
                                repo.FindById(1)).Returns(new Item());
```

- This sets up the **FindById method** so that when it's called with the **argument 1**, it **returns** a **new Item object**

# Returns Method

- Used in **conjunction with Setup** to specify the **value** that a mocked method should **return**
- **Example:**

```
mockRepository.Setup(repo => repo.GetAll()).Returns(new
List<Item>());
```

- This specifies that **calling GetAll** on the mock object should **return** a **new list of Item** objects

# Verify Method

- Used to **ensure** that a **specific interaction** with the mock object **occurred**
- **Example:**

```
mockRepository.Verify(repo => repo.Save(), Times.Once());
```

- This **verifies** that the **Save** method was **called exactly once**

# It.IsAny<T>()

- Used in **Setup** and **Verify** to indicate that **any value of type T** is **acceptable** as an **argument**
- **Example:**

```
mockRepository.Setup(repo =>
            repo.FindById(It.IsAny<int>())).Returns(new Item());
```

- This sets up **FindById** to **return** a **new Item** object regardless of the integer value passed to it

# Callback Method

- Allows you to **capture** the **arguments** passed to a method and **perform actions** when a method is called
- **Example:**

```
mockRepository.Setup(repo =>
                repo.Add(It.IsAny<Item>())).Callback<Item>(item =>
                itemList.Add(item));
```

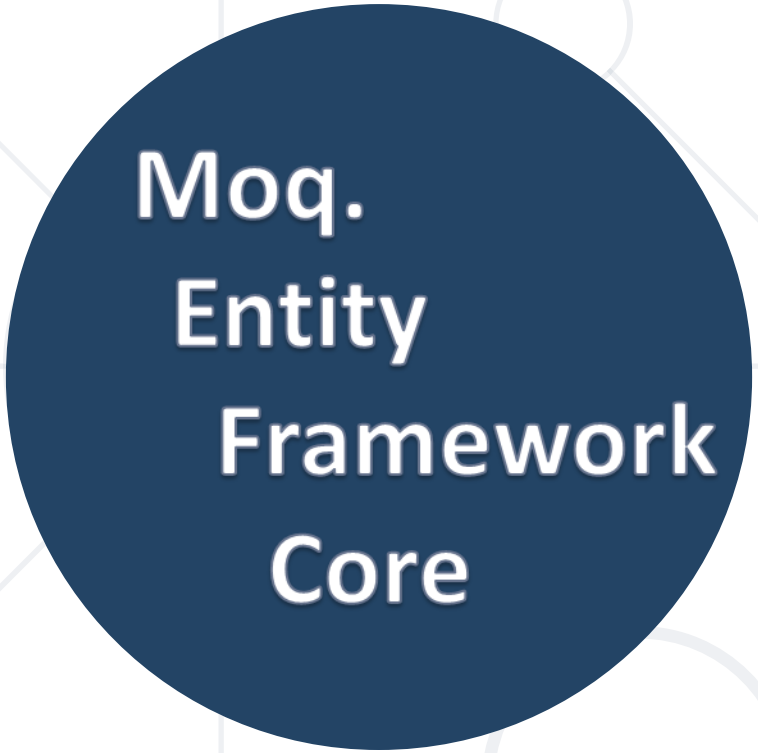
- This sets up the **Add method** so that whenever **it's called**, the **passed Item** is **added** to **itemList**

# Throws Method

- Configures a method to **throw** an **exception** when called
- **Example:**

```
mockRepository.Setup(repo => repo.Delete(It.IsAny<int>()))
                             .Throws<InvalidOperationException>();
```

- This setup causes the **Delete** method to throw an **InvalidOperationException** when called




# Moq.EntityFrameworkCore

Basic Overview

# Moq.EntityFrameworkCore

Software University

- An **extension** for the Moq library, designed to enhance testing with EF Core
- Simplifies creating and managing mock **DbSet<T> objects**, allowing for simulation of **database contexts**
- Provides the ability to perform LINQ queries on mock sets, **mirroring real database** operations
- **Reduces complexity** in setting up mocks for EF Core contexts
- Enables testing of **various data interactions**
- Focuses on verifying the **correctness of business logic** by mocking out the data layer

# When to Use Moq.EntityFrameworkCore

Software University

- **Queryable DbSet**:
  - Utilize Moq.EntityFrameworkCore when complex LINQ queries on a mocked DbSet are required, surpassing basic Add or Find functionalities
- **Testing Behavior with Queryable Data**:
  - If your service layer performs complex data manipulations using LINQ and you need to test these behaviors as they would interact with a DbSet
- https://www.nuget.org/packages/Moq.EntityFrameworkCore/

# Summary



- What are Dependencies
- Coupling vs Decoupling
- What is Mocking: Fake implementation
- What to Mock: Dependencies, Behaviors, Interactions
- Moq Framework - Mocking, Testing, Isolating Dependencies
- Using Moq Framework in C#
- Few words about Moq.EntityFrameworkCore

# Questions?

SoftUni

Software University | SoftUni Creative | SoftUni Digital | SoftUni Foundation | SoftUni Kids | Finance Academy

# SoftUni Diamond Partners

Software University












# Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, about.softuni.bg
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity

# License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – https://about.softuni.bg/
- © Software University – https://softuni.bg