46. What are mocks?

Brief summary: Mocks are objects that can be used to substitute real dependencies for testing purposes. For example, we don't want to use a real database connection in unit tests. Instead, we will replace the object connecting to a database with a mock that provides the same interface, but returns test data. We can set up what will be the results of the methods called on mocks, as well as verify if a particular method has been called. Mocks are an essential part of unit testing, and it's nearly impossible to test a real-life application without them.

Mocks are objects that "pretend" to be other objects and are used mostly for testing purposes. For example, we don't want to use a real database connection in unit tests, and we will explain why in a minute. Instead, we will replace the object connecting to a database with a mock that provides the same interface, but returns test data.

Let's say we want to unit test this class:

```
class PersonalDataFormatter
{
    0 references
    public string Format()
    {
        var people = ReadPeople();
        return string.Join("\n",
            people.Select(p => $"{p.Name} born in" +
            $" {p.Country} on {p.YearOfBirth}"));
    }

    1 reference
    private IEnumerable<Person> ReadPeople()
    {
        Console.WriteLine("Reading from real database");
        return new List<Person>
        {
            new Person("John", 1982, "USA"),
            new Person("Aja", 1992, "India"),
            new Person("Tom", 1954, "Australia"),
        };
    }
}
```

Imagine the ReadPeople method connects to a real database, performing all necessary steps like opening the database connection, executing some SQL queries, etc.

The tests of the PersonalDataFormatter class could look like this:

```
    [TestFixture]
    0 references
    public class PersonalDataFormatterTests
    {
        private PersonalDataFormatter _cut = new PersonalDataFormatter();

        [Test]
        | 0 references
        public void ShallFormatPersonalDataCorrectly()
        {
            var result = _cut.Format();
            var expectedResult = @"John born in USA on 1982
Aja born in India on 1992
Tom born in Australia on 1954";
            Assert.AreEqual(expectedResult, result);
        }
    }
```

This may even work under some circumstances, but there are numerous **problems** with this approach:

- A test that connects to a database is **not a unit test**. A unit test should test only one piece of functionality. Here we test the class, the database connection, and the database itself.
- Also, unit tests should be fast, and connecting to a database **takes time**.
- This test only reads from the database, but what if other tests would also write to it? If some other test would add a new person to the database, this test would start to fail, as the result would contain one more line. As tests would run, the database **state would change** constantly, affecting the results. Because of that, we would be forced to reset the database to some desired state before each test, which would again take significant time.
- What if the database is **not set up** on the computer of another developer? This test may work for us, but it may not work for others.
- What if the database contains **millions of entries**? Then the expected value in this test would be an enormous string, which would obviously be problematic, especially if the test failed and in this huge string we would try to find the exact part that doesn't match the expected result.

To solve all those issues, we need a mechanism that will allow us to **mock** the database connection. Instead of using an object connecting to a real database, we will use a **fake** one, that will return a predefined set of data used for testing purposes only.

But first, we must refactor this code to use **Dependency Injection**, so we are not tightly coupled with the implementation that connects to a real database:

```
class PersonalDataFormatter
{
    private readonly IPeopleDataReader _peopleDataReader;

    1 reference
    public PersonalDataFormatter(
        IPeopleDataReader peopleDataReader)
    {
        _peopleDataReader = peopleDataReader;
    }

    1 reference | 1/1 passing
    public string Format()
    {
        var people = _peopleDataReader.ReadPeople();
        return string.Join(Environment.NewLine,
            people.Select(p => $"{p.Name} born in" +
            $" {p.Country} on {p.YearOfBirth}"));
    }
}
```

```
public interface IPeopleDataReader
{
    3 references | 1/1 passing
    IEnumerable<Person> ReadPeople();
}

0 references
public class DatabasePeopleDataReader : IPeopleDataReader
{
    3 references | 1/1 passing
    public IEnumerable<Person> ReadPeople()
    {
        Console.WriteLine("Reading from real database");
        return new List<Person>
        {
            new Person("John", 1982, "USA"),
            new Person("Aja", 1992, "India"),
            new Person("Tom", 1954, "Australia"),
        };
    }
}
```

Great. Now, in the production code, we can inject the implementation that connects to a real database:

```
public class Program
{
    0 references
    public static void Main(string[] args)
    {
        var personalDataFormatter = new PersonalDataFormatter(
            new DatabasePeopleDataReader());

        Console.WriteLine(personalDataFormatter.Format());
    }
}
```

But for unit tests, we will use a mock. I will be using the **Moq** library for that, which is one of the most popular mocking libraries for C#. To create a mock of some interface, we can simply use the Mock<T> class:

```
[TestFixture]
0 references
public class PersonalDataFormatterTests
{
    private Mock<IPeopleDataReader> _peopleDataReaderMock;
    private PersonalDataFormatter _cut;

    [SetUp]
    0 references
    public void SetUp()
    {
        _peopleDataReaderMock = new Mock<IPeopleDataReader>();
        _cut = new PersonalDataFormatter(
            _peopleDataReaderMock.Object);
    }
```

As you can see I moved the creation of the _cut object to the SetUp method. This is because I want a brand-new mock for each test, which is a good practice since the mocks have their own state (they can track what methods had been called upon them, which is used for validating mock behavior. We will talk more about it later in the lecture).

Let's now use the mock in the test. I will set it up to return some predefined People objects when the ReadPeople method is called:

```
        [Test]
        0 references
        public void ShallFormatPersonalDataCorrectly()
        {
            _peopleDataReaderMock.Setup(mock => mock.ReadPeople())
                .Returns(new List<Person>
                    {
                        new Person("Person1", 1982, "Country1"),
                        new Person("Person2", 1992, "Country2"),
                        new Person("Person3", 1954, "Country3"),
                    });

            var result = _cut.Format();
            var expectedResult = @"Person1 born in Country1 on 1982
Person2 born in Country2 on 1992
Person3 born in Country3 on 1954";
            Assert.AreEqual(expectedResult, result);
        }
    }
```

Great. Now when the _cut object uses the ReadPeople method from the IPeopleDataReader interface that is its dependency, the mock will be used. It will return the predefined collection of people.

This **solves** all problems mentioned before:

- This test is now a **real unit test**. It tests the PersonalDataFormatter class in isolation.
- It is **fast** because it doesn't connect to a database.
- It has no way of **affecting other tests**, as it doesn't modify any shared state (with the test not using mocks, if the test would write to a database, it would modify its content for all other tests).
- The test will work on any machine, no matter if some database is present on it or not.
- We have full control over the data. We can define a small set of people that is enough for testing the PersonalDataFormatter. We won't be affected by the fact that there are millions of people in the database.

All right. Please notice that mocks have one more powerful ability - we can **verify** if some methods have been called upon them as part of the test verification. Let's consider this class.

```
public class EnthusiasticGreeter
{
    0 references
    public void PrintHelloNTimes(int n)
    {
        for(int i = 0; i < n; i++)
        {
            Console.WriteLine("Hello!");
        }
    }
}
```

This class is quite simple, but unfortunately, it is not easy to test. The PrintHelloNTimes method is void, so there is no result to be compared with the expected result.

The test that validates this class should basically have a way of checking if the "Hello!" was printed to the console given count of times. It could possibly be done by actually running the program (which would make this test non-unit) and

somehow intercepting the output printed to the console. But this would be complex, tricky, and non-unitary. After all, we would be testing the Console class as much as the EnthusiasticGreeter class.

The solution is again, to use mock. But what to mock here, exactly? Well, ideally it would be to mock the Console class, but this is impossible since it's static. In most frameworks, including Moq, the mocking mechanism is based on inheritance or interface implementations, so a mock object is basically a derived type from the type we want to mock or it implements the mocked interface. We can't have classes derived from static classes. Again, we will need to use Dependency Injection:

```
public class EnthusiasticGreeter
{
    readonly Action<string> _printToConsole;

    0 references
    public EnthusiasticGreeter(Action<string> printToConsole)
    {
        _printToConsole = printToConsole;
    }

    0 references
    public void PrintHelloNTimes(int n)
    {
        for(int i = 0; i < n; i++)
        {
            _printToConsole("Hello!");
        }
    }
}
```

In the production code, we will simply inject an action that uses Console.Writeline:

```
var enthusiasticGreeter = new EnthusiasticGreeter(
    message => Console.WriteLine(message));

enthusiasticGreeter.PrintHelloNTimes(5);
```

But for testing purposes, we will use a mock of the Action object:

```
[TestFixture]
0 references
public class EnthusiasticGreeterTests
{
    private Mock<Action<string>> _printToConsoleMock;
    private EnthusiasticGreeter _cut;

    [SetUp]
    0 references
    public void SetUp()
    {
        _printToConsoleMock = new Mock<Action<string>>();
        _cut = new EnthusiasticGreeter(
            _printToConsoleMock.Object);
    }
```

We can now write a test that checks that “Hello!” has been printed as many times as the number provided with the parameter:

```
[Test]
| 0 references
public void ShallPrintHello5Times_WhenCalledPrintHello5Times()
{
    _cut.PrintHelloNTimes(5);
    _printToConsoleMock.Verify(
        mock => mock("Hello!"), Times.Exactly(5));
}
```

As you can see, using mocks allowed us to test code that doesn’t return a value. Instead, we tested that a specific method was called with a given parameter and a given number of times.

Let’s summarize. Mocks are objects that can be used to substitute real dependencies for testing purposes. For example, we don’t want to use a real database connection in unit tests. Instead, we will replace the object connecting to a database with a mock that provides the same interface, but returns test data. We can set up what will be the results of the methods called on mocks, as well as verify if a particular method has been called. Mocks are an essential part of unit testing, and it’s nearly impossible to test a real-life application without them.

Bonus questions:

- "**What is Moq?**"
  Moq is a popular mocking library for C#. It allows us to easily create mocks of interfaces, classes, Funcs, or Actions. It gives us the ability to decide what result

will be returned from the mocked functions, as well as validate if some function has been called, how many times, and with what parameters.

- "**What is the relation between mocking and Dependency Injection?**"
  Mocking is hard to implement without the Dependency Injection. Dependency Injection allows us to inject some dependencies to a class, so we can choose whether we inject real implementations or mocks. If the dependency of the class would not be injected but rather created right in the class, we could not switch it to a mock implementation for testing purposes.