Unit testing in C#

Software testing

2020/21 II. semester

Fazekas Levente

INZV77

Contents

Abstract

For this, I have used a well-known method called Unit Testing, designated as a white-box test, meaning we know what the source code is. For said source, we write individual tests, separating each component, in this case, methods. By testing the individual components in a computer program, we can increase its reliability, have bugs discovered ahead of time, fix issues promptly.

Aspects fo testing

In unit testing, we have to consider a number of aspects:

- A test must only examine one component.
- Tests must not bypass the boundaries of their modules.
- Tests should run independently.
- Tests must not depend on the environment they run in.
- Tests should not have any side effects.
- Tests should be able to run at compile time.

Unit testing tools and conventions

Mocking

In C# we have a library called mock which handles any Mocking task. The idea is to create a concrete implementation of an interface and control how specific methods on that interface responds when called, allowing us to test all of the paths through code.

Example

Creating the code

```
namespace api.Controllers
{
    [ApiController]
    [Route("[controller]")]
    public class BankController
    {
        private readonly IBankService _bankService;

        public BankController(
            IBankService bankService
        )
        {
            _bankService = bankService;
        }

        [HttpPost]
        public string Transfer(ICard sender, ICard recipient, double
            amount)
        {
            if (!_bankService.Withdraw(sender, amount)) {
                return "Cannot withdraw";
            }
            if (!_bankService.Deposit(recipient, amount)) {
                return "Cannot deposit";
            }

            return "Transfered";
        }
    }
}
```

Here we have created a BankController class with a dependency, which is a BankService. Note how we will not create any concrete implementations of our services at this point. We are more interested in establishing and testing the behavior of our code. That means that concrete service implementations can come later.

```
namespace Services
{
    public interface ICard
    {
        public string CardNumber { get; set; }
        public string Name { get; set; }
        public DateTime ValidTo { get; set; }
    }
    public interface IBankService
    {
        bool Deposit(double total, ICard card);
        bool Withdraw(double total, ICard card);
    }
}
```

Creating our Mock

In our test project we can create the following:

```
var bankServiceMock = new Mock<IBankService>();
```

The above is not a concrete implementation but a Mock object. A Mock can be:

- Instructed, you can tell a mock that if a certain method is called then it can answer with a certain response.
- Verified, verification is something you carry out after your production code has been called. You carry this out to verify that a certain method has been called with specific arguments.

Instructing our Mock

Now we have a Mock object that we can instruct. To instruct it we use the method `Setup()` like so:

```
bankServiceMock.Setup(b => b.Withdraw()).Returns(true);
bankServiceMock.Setup(b => b.Deposit()).Returns(true);
```

Of course, the above won't compile, we need to give the `Deposit() and Withdraw()` methods the arguments. There are two ways we can give the `Deposit() and Withdraw()` methods the arguments they need:

```
var card = new Card("owner", "number", "CVV number");

bankServiceMock.Setup(b => b.Deposit(114, card)).Returns(true);
bankServiceMock.Setup(b => b.Withdraw(114, card)).Returns(true);
```

```
bankServiceMock.Setup(b => b.Withdraw(It.IsAny<double>(),card)).Returns(
    true);
bankServiceMock.Setup(b => b.Deposit(It.IsAny<double>(),card)).Returns(
    true);
```

Accessing our implementation

General arguments We will need to pass an implementation of our Mock when we call the actual production code. So how do we do that? There's an Object property on the Mock that represents the concrete implementation. Below we are using just that. We first construct `cardMock` and then we pass `cardMock.Object` to the `Deposit() and Withdraw()` methods.

```
cardMock = new Mock<ICard>();

bankServiceMock.Setup(b => b.Withdraw(It.IsAny<double>(),cardMock.Object()
    )).Returns(true);
bankServiceMock.Setup(b => b.Deposit(It.IsAny<double>(),cardMock.Object())
    ).Returns(true);
```

Add unit tests

```
[Test]
public void ShouldReturnTransfered()
{
    // arrange
    bankServiceMock.Setup(b => b.Withdraw(It.IsAny<double>(),cardMock.
        Object())).Returns(true);
    bankServiceMock.Setup(b => b.Deposit(It.IsAny<double>(),cardMock.
        Object())).Returns(true);

    // act
    var result = controller.Transfer(cardMock.Object, cardMock.Object, 1);

    // assert
    bankServiceMock.Verify(b => b.Withdraw(It.IsAny<double>(),cardMock.
        Object()), Times.Once());
    bankServiceMock.Verify(b => b.Deposit(It.IsAny<double>(),cardMock.
        Object()), Times.Once());

    Assert.AreEqual("Transfered", result);
}
```

AAA methodology

In Unit testing we divide all test methods into three parts or phases. These are: Arrange, Act, Assert.

So using our previous example:

Arrange

Here we set up the necessary conditions to run our test. We create all the mocks, setups, and other classes in this phase.

```
bankServiceMock.Setup(b => b.Withdraw(It.IsAny<double>(),cardMock.Object()
    )).Returns(true);
bankServiceMock.Setup(b => b.Deposit(It.IsAny<double>(),cardMock.Object())
    ).Returns(true);
```

Act

In this phase we call the actual code:

```
var result = controller.Transfer(cardMock.Object, cardMock.Object, 1);
```

Assert

```
bankServiceMock.Verify(b => b.Withdraw(It.IsAny<double>(),cardMock.Object
    ()), Times.Once());
bankServiceMock.Verify(b => b.Deposit(It.IsAny<double>(),cardMock.Object()
    ), Times.Once());

Assert.AreEqual("Transfered", result);
```

Now, there are two pieces of assertions that take place here. First, we have a Mock assertion. We see that as we are calling the method `Verify()` that essentially says: I expect the `Deposit() and Withdraw()` methods to get called once each.

Next, we have a normal assertion. It says out `result` variable should contain the value `Transfered`.

Testing a project

Next we will be looking at actual production code from a project I wrote. The project had very little budget, hence the lack of mocking, but some testing was required. Due to the fact, that I do not fully own the project, I cannot share implementation details, but I will provide the necessary documentation comments.