Unit Testing

Don't Unit Test Private Methods in C# – Do This Instead

Kristijan Kralj

So you want to test private methods in your C# code?

Good, you are at the right place.

The best possible way to test private methods is not to test them at all.

In this post, I will explain why you shouldn't test private methods, what refactoring to perform if you want to test logic in them. And finally, what hacks are available if you insist on testing (https://methodpoet.com/unit-testing-in-c/) a private method in your code.

Table of Contents

1. Why you shouldn't unit testing private methods in C#
2. How to change the code to test the private method
3. What hacks are available if you want to test private methods as-is?
    3.1. Reflection
    3.2. PrivateObject
    3.3. InternalsVisibleToAttribute
4. Conclusion

Why you shouldn't unit testing private methods in C#

You may not know this, but the quality of your code can actually predict your company's success in the future. Software is a crucial component of our lives now, so it is important to make sure that you write the best code you can.

Code quality is measured by several metrics, but a few of the more important ones are the number of bugs, the number of tests, and how decoupled your design is. To improve the software design, you may want to split large methods into smaller methods, where a public method calls other private methods.

Private methods are made private for a reason. And in your unit tests, you should only test the public methods. But why should you avoid testing private methods? Well, first of all, there's the problem of access. Just like other classes can interact only with the public methods of your class, the same should apply to a unit test. The unit test should only test the public interface (https://methodpoet.com/unit-testing-best-practices/). Each private method is an implementation detail. They contain the logic necessary to implement a piece of functionality. Private methods exist due to code reusability and to avoid having large public methods that do everything.

By testing private methods, your tests will become more fragile because and you'll tend to break them every time you change code in and around those private methods. On the other hand, by testing public methods only, you'll reduce the number of unit tests you need, and you'll make tests less brittle.

But what if it's hard to test private behavior through the public method? This can happen if the public method does a lot of interaction with the database and/or performs a lot of network requests. In other words, the public method does too much, it breaks the single responsibility principle (https://methodpoet.com/single-responsibility-principle/), and it's not easy to change the production code to write a unit test that will execute the private method.

The best option is to move the method to its own object.

How to change the code to test the private method

If your private method contains much valuable logic that you think should be covered with tests, it is best to extract it into a new class.

How?

By applying Replace Method with Method Object refactoring (https://refactoring.guru/replace-method-with-method-object). You can use it to convert a private method to a separate class. With this refactoring, the object is created to provide the same behavior as the original method. This can be a tricky refactoring, and you need to follow the next steps:

1. Create a new empty class and name it based on the purpose of the method.

2. In the new class, create a private field for storing the original class. There is a possibility that you will need to call some method from the original class in the new class.
3. Create a private field for each local variable.
4. Create a constructor that takes the same parameters as your private method.
5. Create a new public method, copy the body of the original method to it, and replace the local variables with the private fields of your new class.
6. Call the method from the new class in the original class.

Let's say you have the following God class and the private method IsLifeBeautiful.

```
public class God
{
    private bool IsLifeBeautiful(string yourName)
    {
        if (yourName == "God")
        {
            return true;
        }
        return false;
    }
}
```

After applying the refactoring, the code would look like this:

```
public class LifeDecider
{
    private string _name;

    public LifeDecider(string name)
    {
        _name = name;
    }

    public bool IsBeautiful()
    {
        if (_name == "God")
        {
            return true;
        }
        return false;
    }
}
```

And you can call the new class.

```
var result = new LifeDecider("God").IsBeautiful();
```

Of course, this is a very simple example, but you get the idea.

The upside of this refactoring is that it forces better modularization of code. It also makes it easier to test code because you can easily find what the new class needs to do.

What hacks are available if you want to test private methods as-is?

If you still want to proceed and write a test for your private method, there are a couple of options. The easiest thing you can do is to change the access modifier from private to public. However, this is not the best thing to do since you expose your class's internals to the outside world.

Reflection

The first option you can do is to use reflection to call the private method. The next example shows how to do it. Let's say you have the following God class and the private method IsLifeBeautiful.

You can write the following test method to call the private method.

```
1  [Fact]
2  public void IsLifeBeautiful_returns_true_when_your_name_is_God()
3  {
4      God sut = new God();
5
6      MethodInfo methodInfo = typeof(God)
7          .GetMethod("IsLifeBeautiful",
8                     BindingFlags.NonPublic | BindingFlags.Instance);
9      object[] parameters = { "God" };
10
11     object result = methodInfo.Invoke(sut, parameters);
12
13     Assert.True((bool)result);
14 }
```

In this case, you use MethodInfo to invoke the private method. You can also pass the method parameters as an array. Not the prettiest solution, but it works.

PrivateObject

If you use the MSTest unit testing framework, you can use PrivateObject class (https://docs.microsoft.com/en-us/dotnet/api/microsoft.visualstudio.testtools.unittesting.privateobject?view=visualstudiosdk-2019) to call the private method. One note here is that this is only available in the .NET framework. This is not available in the .NET Core version.

The next test code sample shows how to use it.

```
[TestMethod]
public void IsLifeBeautiful_returns_true_when_your_name_is_God()
{
    God sut = new God();
    object[] parameters = { "God" };
    PrivateObject po = new PrivateObject(sut);

    var returnValue = po.Invoke("IsLifeBeautiful", parameters);

    Assert.IsTrue((bool)returnValue);
}
```

InternalsVisibleToAttribute

The last trick you can do is change the method modifier from private to internal and then use the InternalsVisibleTo attribute. You then define which project can see internal methods from your original project.

This is the change you need to make.

```
[assembly: InternalsVisibleTo("TestPrivateMethods")]

namespace GodLibrary
{
    public class God
    {
        internal bool IsLifeBeautiful(string yourName)
        {
            if (yourName == "God")
            {
                return true;
            }
            return false;
        }
    }
}
```

In this case, the IsLifeBeautiful has become an internal method. And now it’s possible to call it in your test method.

Conclusion

Private methods are usually not designed to be called externally. Therefore, it is not an efficient use of your time, and tests will be brittle. You will be wasting time testing private methods because they contain the implementation of your object rather than its public interface.

Write unit tests only for the public methods. When you test a public method, you are testing the public API between the object and the caller. If the public method changes, any tests for that method will fail. What your public interface communicates to the outside world is not what your private method communicates.