Unit Testing

2 Reasons Why Most C# Unit Tests for Abstract Class Fail

Kristijan Kralj

So you've decided to start writing unit tests for your code.

Great decision! But you're struggling with how to test abstract classes.

Unfortunately, testing can be like that. It can be complex to know when to write a test and how and what to test.

The realm of abstract classes, a core feature of several programming languages, such as C#, has remained one common area of confusion.

This post should answer many common questions, including how you can use them and whether you should write unit tests for them.

Table of Contents

The need for abstract classes

Abstract classes can seem incredible. They're great for code reuse, save you from repeating the code, and make sure that the implementation details of classes don't leak out.

Even though their popularity has declined in the last few years, thanks to dependency injection (https://methodpoet.com/dependency-injection) and favoring composition over inheritance (https://en.wikipedia.org/wiki/Composition_over_inheritance), you can still find uses for abstract classes. Not to mention they are often present in the legacy code applications (https://methodpoet.com/legacy-code/).

Let's explore a class hierarchy for a mechanic shop application.

You may begin your implementation like this:

```
public class Truck
{
    public string License { get; set; }

    public string GetLicense()
    {
        return $"License: {License}";
    }
}

public class Bus
{
    public string License { get; set; }

    public string GetLicense()
    {
        return $"License: {License}";
    }
}
```

By now, you may have already realized that both `Bus` and `Truck` classes share a common method, `GetLicense`. Although DRY (https://methodpoet.com/top-software-engineering-principles/) is always a suggested approach, repeating one method it's not a big deal.

However, as you begin to add other types of cars into the mix, such as SUVs, sports cars, sedans, monster trucks, and so on, you may find yourself copy-pasting segments of code for your new classes. Fortunately, C# heavily anticipates this behavior, where the concept of an abstract class plays a key role. In this case, rather than repeating methods and other core pieces of functionality, you may merely create one class that others can inherit from:

```
public abstract class Vehicle
{
    public string License { get; set; }

    protected abstract string GetCarType();

    public string GetLicense()
    {
        return $"License: {License}";
    }
}

public class Truck : Vehicle
{
    protected override string GetCarType() => "Truck";
}

public class Bus: Vehicle
{
    protected override string GetCarType() => "Bus";
}
```

Everything in the code looks great, but how to test the methods inside the `Vehicle` class?

Unit testing abstract class in C#

Unit tests (https://methodpoet.com/unit-testing-in-c) are tests that check if the individual units have errors. The main goal is to provide certainty that your code works. If you know that the tiniest units of your program work efficiently, you can be far more confident in the correctness of the entire application.

Since its inception, unit tests have remained a core part of writing efficient programs. This is mainly because they easily enable you to catch errors during development (https://methodpoet.com/who-performs-unit-testing/), get rid of regression bugs (https://methodpoet.com/unit-tests-vs-regression-tests), and save the time you may spend solving a problem after your app has already gone into production. As a result, unit tests are the greatest decisive factor (https://methodpoet.com/is-unit-testing-a-waste-of-time/) between a buggy and a successful release.

In addition to this, you can also take things further by implementing other forms of testing in your application, such as integration (https://methodpoet.com/integration-tests-vs-unit-tests) and end-to-end testing (https://methodpoet.com/unit-tests-vs-end-to-end-tests/). By creating an effective complete test hierarchy (https://methodpoet.com/testing-pyramid/), you will detect as many errors as possible at all layers of your production code.

However, writing test code for an abstract class isn't always easy.

Should I write C# unit tests for abstract class?

No.

One of the many areas that most beginner testers struggle with is writing a test case for an abstract class method. If you jump back into the mechanic application, the first solution that comes to mind is to write a test for all the classes, including the abstract class. While this may seem logical at first, you may quickly begin to see the cracks in this approach as you explore further.

It is wrong due to two reasons.

#1 Testing abstract classes is wrong because abstract classes are implementation details

An abstract base class is a special type of object. Its main job is to be a helper. That means that it is a way to store and reuse common code. In other words, an abstract class is an implementation detail.

In the above example, the classes such as `Truck`, `Bus`, `Sedan`, and any other class can inherit from `Vehicle` and instantly get access to the `GetLicense` method.

Additionally, you can't instantiate abstract classes in C#. Therefore, you cannot directly create an instance in the test method. This makes it much more difficult to write efficient unit tests. You will have to rely on introducing the fake classes (https://methodpoet.com/stub-vs-mock/) to test the functionality of the abstract class rather than testing it directly.

To test methods that a child class inherits, write a test for the class that is a concrete implementation.

In the above case, does that mean you need to write a unit test for every class that inherits the abstract class and has that method?

No. Write only one test – no need to repeat the same test case in every test class for the same method. Introduce another test only if you change the method implementation in one of the derived classes.

#2 Testing abstract classes is wrong because you don't cover the implementation in the concrete classes

One way to get around the problem of not being able to initialize the abstract class in the test case would be to create a dummy class in the test solution. Then the next step would be to write all the necessary tests for the new dummy class.

Consider the following implementation:

```
public class TestVehicle: Vehicle
{
    protected override string GetCarType() => "TestVehicle";
}
```

That might be a good idea at first, but there is an issue with that approach.

If you only test the `GetLicense` method of the `TestVehicle` , what happens if the `Truck` derived class overrides the default implementation? The functionality will become entirely different from the method defined in Vehicle. So, even if the tests for the `TestVehicle` pass, your application may still have bugs.

In general, writing the C# unit test for abstract class implementations does not bring much good. Therefore, don't write unit tests for them. These classes only focus on implementation details and not the actual implementation that is carried out in the concrete classes. So, even if your abstract class tests pass, your app may still contain bugs.

What to test instead of abstract classes

While the mechanic application only scratched the surface of how abstract classes work and relate to concrete classes, you may likely be dealing with more complex situations in your application. However, in most cases, the way that you set up your tests should remain the same.

Once you have successfully created your concrete classes, you should focus on testing those classes while safely ignoring the base class. And remember, abstract classes provide a way to reuse the implementation.

Conclusion

In conclusion, don't write units test for an abstract class. The abstract class itself is an implementation detail. Therefore, there is no need to test it. Instead, you'll want to write a unit test for the child classes.

Finally, don't create abstract classes if you don't need them. If you need to reuse a method implemented in the abstract class, you can create a separate class and inject (https://methodpoet.com/dependency-injection-benefits/) it into the appropriate classes. That will decrease decoupling, make your code easier to test, and improve maintainability.