11 Unit testing anti-patterns

This chapter covers

- Unit testing private methods
- Exposing private state to enable unit testing
- Leaking domain knowledge to tests
- Mocking concrete classes

This chapter is an aggregation of lesser related topics (mostly anti-patterns) that didn't fit in earlier in the book and are better served on their own. An *anti-pattern* is a common solution to a recurring problem that looks appropriate on the surface but leads to problems further down the road.

You will learn how to work with time in tests, how to identify and avoid such anti-patterns as unit testing of private methods, code pollution, mocking concrete classes, and more. Most of these topics follow from the first principles described in part 2. Still, they are well worth spelling out explicitly. You've probably heard of at least some of these anti-patterns in the past, but this chapter will help you connect the dots, so to speak, and see the foundations they are based on.

11.1 Unit testing private methods

When it comes to unit testing, one of the most commonly asked questions is how to test a private method. The short answer is that you shouldn't do so at all, but there's quite a bit of nuance to this topic.

11.1.1 Private methods and test fragility

Exposing methods that you would otherwise keep private just to enable unit testing violates one of the foundational principles we discussed in chapter 5: testing observable behavior only. Exposing private methods leads to coupling tests to implementation details and, ultimately, damaging your tests' resistance to refactoring—the most important metric of the four. (All four metrics, once again, are protection against regressions, resistance to refactoring, fast feedback, and maintainability.) Instead of testing private methods directly, test them indirectly, as part of the overarching observable behavior.

11.1.2 Private methods and insufficient coverage

Sometimes, the private method is too complex, and testing it as part of the observable behavior doesn't provide sufficient coverage. Assuming the observable behavior already has reasonable test coverage, there can be two issues at play:

- *This is dead code.* If the uncovered code isn't being used, this is likely some extraneous code left after a refactoring. It's best to delete this code.
- *There's a missing abstraction.* If the private method is too complex (and thus is hard to test via the class's public API), it's an indication of a missing abstraction that should be extracted into a separate class.

Let's illustrate the second issue with an example.

Listing 11.1 A class with a complex private method

```
public class Order
{
    private Customer _customer;
    private List<Product> _products;

    public string GenerateDescription()
    {
        return $"Customer name: {_customer.Name}, " +
            $"total number of products: {_products.Count}, " +
            $"total price: {GetPrice()}";
    }

    private decimal GetPrice()
    {
        decimal basePrice = /* Calculate based on _products */;
        decimal discounts = /* Calculate based on _customer */;
        decimal taxes = /* Calculate based on _products */;
```

The complex private method is used by a much simpler public method.

Complex private method

```
        return basePrice - discounts + taxes;
    }
}
```

The `GenerateDescription()` method is quite simple: it returns a generic description of the order. But it uses the private `GetPrice()` method, which is much more complex: it contains important business logic and needs to be thoroughly tested. That logic is a *missing abstraction*. Instead of exposing the `GetPrice` method, make this abstraction explicit by extracting it into a separate class, as shown in the next listing.

Listing 11.2 Extracting the complex private method

```
public class Order
{
    private Customer _customer;
    private List<Product> _products;

    public string GenerateDescription()
    {
        var calc = new PriceCalculator();

        return $"Customer name: {_customer.Name}, " +
            $"total number of products: {_products.Count}, " +
            $"total price: {calc.Calculate(_customer, _products)}";
    }
}

public class PriceCalculator
{
    public decimal Calculate(Customer customer, List<Product> products)
    {
        decimal basePrice = /* Calculate based on products */;
        decimal discounts = /* Calculate based on customer */;
        decimal taxes = /* Calculate based on products */;
        return basePrice - discounts + taxes;
    }
}
```

Now you can test `PriceCalculator` independently of `Order`. You can also use the output-based (functional) style of unit testing, because `PriceCalculator` doesn't have any hidden inputs or outputs. See chapter 6 for more information about styles of unit testing.

11.1.3 When testing private methods is acceptable

There are exceptions to the rule of never testing private methods. To understand those exceptions, we need to revisit the relationship between the code's publicity and purpose from chapter 5. Table 11.1 sums up that relationship (you already saw this table in chapter 5; I'm copying it here for convenience).

Table 11.1 The relationship between the code's publicity and purpose

	Observable behavior	Implementation detail
Public	Good	Bad
Private	N/A	Good

As you might remember from chapter 5, making the observable behavior public and implementation details private results in a well-designed API. On the other hand, leaking implementation details damages the code's encapsulation. The intersection of observable behavior and private methods is marked `N/A` in the table because for a method to become part of observable behavior, it has to be used by the client code, which is impossible if that method is private.

Note that testing private methods isn't bad in and of itself. It's only bad because those private methods are a proxy for implementation details. Testing implementation details is what ultimately leads to test brittleness. Having that said, there are rare cases where a method is both private and part of observable behavior (and thus the `N/A` marking in table 11.1 isn't entirely correct).

Let's take a system that manages credit inquiries as an example. New inquiries are bulk-loaded directly into the database once a day. Administrators then review those inquiries one by one and decide whether to approve them. Here's how the `Inquiry` class might look in that system.

Listing 11.3 A class with a private constructor

```
public class Inquiry
{
    public bool IsApproved { get; private set; }
    public DateTime? TimeApproved { get; private set; }

    private Inquiry(                                          Private
        bool isApproved, DateTime? timeApproved)              constructor
    {
        if (isApproved && !timeApproved.HasValue)
            throw new Exception();

        IsApproved = isApproved;
        TimeApproved = timeApproved;
    }

    public void Approve(DateTime now)
    {
        if (IsApproved)
            return;

        IsApproved = true;
        TimeApproved = now;
    }
}
```

The private constructor is private because the class is restored from the database by an object-relational mapping (ORM) library. That ORM doesn't need a public constructor; it may well work with a private one. At the same time, our system doesn't need a constructor, either, because it's not responsible for the creation of those inquiries.

How do you test the `Inquiry` class given that you can't instantiate its objects? On the one hand, the approval logic is clearly important and thus should be unit tested. But on the other, making the constructor public would violate the rule of not exposing private methods.

`Inquiry`'s constructor is an example of a method that is both private and part of the observable behavior. This constructor fulfills the contract with the ORM, and the fact that it's private doesn't make that contract less important: the ORM wouldn't be able to restore inquiries from the database without it.

And so, making `Inquiry`'s constructor public won't lead to test brittleness in this particular case. In fact, it will arguably bring the class's API closer to being well-designed. Just make sure the constructor contains all the preconditions required to maintain its encapsulation. In listing 11.3, such a precondition is the requirement to have the approval time in all approved inquiries.

Alternatively, if you prefer to keep the class's public API surface as small as possible, you can instantiate `Inquiry` via reflection in tests. Although this looks like a hack, you are just following the ORM, which also uses reflection behind the scenes.

11.2 Exposing private state

Another common anti-pattern is exposing private state for the sole purpose of unit testing. The guideline here is the same as with private methods: don't expose state that you would otherwise keep private—test observable behavior only. Let's take a look at the following listing.

Listing 11.4 A class with private state

```
public class Customer
{
    private CustomerStatus _status =          Private
        CustomerStatus.Regular;               state

    public void Promote()
    {
        _status = CustomerStatus.Preferred;
    }

    public decimal GetDiscount()
    {
        return _status == CustomerStatus.Preferred ? 0.05m : 0m;
    }
}

public enum CustomerStatus
{
```

```
    Regular,
    Preferred
}
```

This example shows a `Customer` class. Each customer is created in the `Regular` status and then can be promoted to `Preferred`, at which point they get a 5% discount on everything.

How would you test the `Promote()` method? This method's side effect is a change of the `_status` field, but the field itself is private and thus not available in tests. A tempting solution would be to make this field public. After all, isn't the change of status the ultimate goal of calling `Promote()`?

That would be an anti-pattern, however. Remember, *your tests should interact with the system under test (SUT) exactly the same way as the production code and shouldn't have any special privileges*. In listing 11.4, the `_status` field is hidden from the production code and thus is not part of the SUT's observable behavior. Exposing that field would result in coupling tests to implementation details. How to test `Promote()`, then?

What you should do, instead, is look at how the production code uses this class. In this particular example, the production code doesn't care about the customer's status; otherwise, that field would be public. The only information the production code does care about is the discount the customer gets after the promotion. And so that's what you need to verify in tests. You need to check that

- A newly created customer has no discount.
- Once the customer is promoted, the discount becomes 5%.

Later, if the production code starts using the customer status field, you'd be able to couple to that field in tests too, because it would officially become part of the SUT's observable behavior.

NOTE Widening the public API surface for the sake of testability is a bad practice.

11.3 Leaking domain knowledge to tests

Leaking domain knowledge to tests is another quite common anti-pattern. It usually takes place in tests that cover complex algorithms. Let's take the following (admittedly, not that complex) calculation algorithm as an example:

```
public static class Calculator
{
    public static int Add(int value1, int value2)
    {
        return value1 + value2;
    }
}
```

This listing shows an *incorrect* way to test it.

Listing 11.5 Leaking algorithm implementation

```
public class CalculatorTests
{
    [Fact]
    public void Adding_two_numbers()
    {
        int value1 = 1;
        int value2 = 3;
        int expected = value1 + value2;     <-- The leakage

        int actual = Calculator.Add(value1, value2);

        Assert.Equal(expected, actual);
    }
}
```

You could also parameterize the test to throw in a couple more test cases at almost no additional cost.

Listing 11.6 A parameterized version of the same test

```
public class CalculatorTests
{
    [Theory]
    [InlineData(1, 3)]
    [InlineData(11, 33)]
    [InlineData(100, 500)]
    public void Adding_two_numbers(int value1, int value2)
    {
        int expected = value1 + value2;     <-- The leakage

        int actual = Calculator.Add(value1, value2);

        Assert.Equal(expected, actual);
    }
}
```

Listings 11.5 and 11.6 look fine at first, but they are, in fact, examples of the anti-pattern: these tests duplicate the algorithm implementation from the production code. Of course, it might not seem like a big deal. After all, it's just one line. But that's only because the example is rather simplified. I've seen tests that covered complex algorithms and did nothing but reimplement those algorithms in the arrange part. They were basically a copy-paste from the production code.

These tests are another example of coupling to implementation details. They score almost zero on the metric of resistance to refactoring and are worthless as a result. Such tests don't have a chance of differentiating legitimate failures from false positives. Should a change in the algorithm make those tests fail, the team would most likely just copy the new version of that algorithm to the test without even trying to

identify the root cause (which is understandable, because the tests were a mere duplication of the algorithm in the first place).

How to test the algorithm properly, then? *Don't imply any specific implementation when writing tests.* Instead of duplicating the algorithm, hard-code its results into the test, as shown in the following listing.

Listing 11.7 Test with no domain knowledge

```
public class CalculatorTests
{
    [Theory]
    [InlineData(1, 3, 4)]
    [InlineData(11, 33, 44)]
    [InlineData(100, 500, 600)]
    public void Adding_two_numbers(int value1, int value2, int expected)
    {
        int actual = Calculator.Add(value1, value2);
        Assert.Equal(expected, actual);
    }
}
```

It can seem counterintuitive at first, but hardcoding the expected result is a good practice when it comes to unit testing. The important part with the hardcoded values is to precalculate them using something other than the SUT, ideally with the help of a domain expert. Of course, that's only if the algorithm is complex enough (we are all experts at summing up two numbers). Alternatively, if you refactor a legacy application, you can have the legacy code produce those results and then use them as expected values in tests.

11.4 Code pollution

The next anti-pattern is code pollution.

DEFINITION *Code pollution* is adding production code that's only needed for testing.

Code pollution often takes the form of various types of switches. Let's take a logger as an example.

Listing 11.8 Logger with a Boolean switch

```
public class Logger
{
    private readonly bool _isTestEnvironment;

    public Logger(bool isTestEnvironment)      <-- The switch
    {
        _isTestEnvironment = isTestEnvironment;
    }
```

```
    public void Log(string text)
    {
        if (_isTestEnvironment)      <-- The switch
            return;

        /* Log the text */
    }
}

public class Controller
{
    public void SomeMethod(Logger logger)
    {
        logger.Log("SomeMethod is called");
    }
}
```

In this example, `Logger` has a constructor parameter that indicates whether the class runs in production. If so, the logger records the message into the file; otherwise, it does nothing. With such a Boolean switch, you can disable the logger during test runs, as shown in the following listing.

Listing 11.9 A test using the Boolean switch

```
[Fact]
public void Some_test()
{
    var logger = new Logger(true);     <-- Sets the parameter to true to indicate the test environment
    var sut = new Controller();

    sut.SomeMethod(logger);

    /* assert */
}
```

The problem with code pollution is that it mixes up test and production code and thereby increases the maintenance costs of the latter. To avoid this anti-pattern, keep the test code out of the production code base.

In the example with `Logger`, introduce an `ILogger` interface and create two implementations of it: a real one for production and a fake one for testing purposes. After that, re-target `Controller` to accept the interface instead of the concrete class, as shown in the following listing.

Listing 11.10 A version without the switch

```
public interface ILogger
{
    void Log(string text);
}
```

```
public class Logger : ILogger
{
    public void Log(string text)
    {
        /* Log the text */
    }
}
```

Belongs in the production code

```
public class FakeLogger : ILogger
{
    public void Log(string text)
    {
        /* Do nothing */
    }
}
```

Belongs in the test code

```
public class Controller
{
    public void SomeMethod(ILogger logger)
    {
        logger.Log("SomeMethod is called");
    }
}
```

Such a separation helps keep the production logger simple because it no longer has to account for different environments. Note that `ILogger` itself is arguably a form of code pollution: it resides in the production code base but is only needed for testing. So how is the new implementation better?

The kind of pollution `ILogger` introduces is less damaging and easier to deal with. Unlike the initial `Logger` implementation, with the new version, you can't accidentally invoke a code path that isn't intended for production use. You can't have bugs in interfaces, either, because they are just contracts with no code in them. In contrast to Boolean switches, interfaces don't introduce additional surface area for potential bugs.

11.5 Mocking concrete classes

So far, this book has shown mocking examples using interfaces, but there's an alternative approach: you can mock concrete classes instead and thus preserve part of the original classes' functionality, which can be useful at times. This alternative has a significant drawback, though: it violates the Single Responsibility principle. The next listing illustrates this idea.

Listing 11.11 A class that calculates statistics

```
public class StatisticsCalculator
{
    public (double totalWeight, double totalCost) Calculate(
        int customerId)
    {
        List<DeliveryRecord> records = GetDeliveries(customerId);
```

```
        double totalWeight = records.Sum(x => x.Weight);
        double totalCost = records.Sum(x => x.Cost);

        return (totalWeight, totalCost);
    }

    public List<DeliveryRecord> GetDeliveries(int customerId)
    {
        /* Call an out-of-process dependency
        to get the list of deliveries */
    }
}
```

`StatisticsCalculator` gathers and calculates customer statistics: the weight and cost of all deliveries sent to a particular customer. The class does the calculation based on the list of deliveries retrieved from an external service (the `GetDeliveries` method). Let's also say there's a controller that uses `StatisticsCalculator`, as shown in the following listing.

Listing 11.12 A controller using `StatisticsCalculator`

```
public class CustomerController
{
    private readonly StatisticsCalculator _calculator;

    public CustomerController(StatisticsCalculator calculator)
    {
        _calculator = calculator;
    }

    public string GetStatistics(int customerId)
    {
        (double totalWeight, double totalCost) = _calculator
            .Calculate(customerId);

        return
            $"Total weight delivered: {totalWeight}. " +
            $"Total cost: {totalCost}";
    }
}
```

How would you test this controller? You can't supply it with a real `StatisticsCalculator` instance, because that instance refers to an unmanaged out-of-process dependency. The unmanaged dependency has to be substituted with a stub. At the same time, you don't want to replace `StatisticsCalculator` entirely, either. This class contains important calculation functionality, which needs to be left intact.

One way to overcome this dilemma is to mock the `StatisticsCalculator` class and override only the `GetDeliveries()` method, which can be done by making that method virtual, as shown in the following listing.

Listing 11.13 Test that mocks the concrete class

```
[Fact]
public void Customer_with_no_deliveries()
{
    // Arrange
    var stub = new Mock<StatisticsCalculator> { CallBase = true };
    stub.Setup(x => x.GetDeliveries(1))
        .Returns(new List<DeliveryRecord>());
    var sut = new CustomerController(stub.Object);

    // Act
    string result = sut.GetStatistics(1);

    // Assert
    Assert.Equal("Total weight delivered: 0. Total cost: 0", result);
}
```

GetDeliveries() must be made virtual.

The `CallBase = true` setting tells the mock to preserve the base class's behavior unless it's explicitly overridden. With this approach, you can substitute only a part of the class while keeping the rest as-is. As I mentioned earlier, this is an anti-pattern.

NOTE The necessity to mock a concrete class in order to preserve part of its functionality is a result of violating the Single Responsibility principle.

`StatisticsCalculator` combines two unrelated responsibilities: communicating with the unmanaged dependency and calculating statistics. Look at listing 11.11 again. The `Calculate()` method is where the domain logic lies. `GetDeliveries()` just gathers the inputs for that logic. Instead of mocking `StatisticsCalculator`, split this class in two, as the following listing shows.

Listing 11.14 Splitting `StatisticsCalculator` into two classes

```
public class DeliveryGateway : IDeliveryGateway
{
    public List<DeliveryRecord> GetDeliveries(int customerId)
    {
        /* Call an out-of-process dependency
        to get the list of deliveries */
    }
}

public class StatisticsCalculator
{
    public (double totalWeight, double totalCost) Calculate(
        List<DeliveryRecord> records)
    {
        double totalWeight = records.Sum(x => x.Weight);
        double totalCost = records.Sum(x => x.Cost);

        return (totalWeight, totalCost);
    }
}
```

The next listing shows the controller after the refactoring.

Listing 11.15 Controller after the refactoring

```
public class CustomerController
{
    private readonly StatisticsCalculator _calculator;
    private readonly IDeliveryGateway _gateway;

    public CustomerController(
        StatisticsCalculator calculator,
        IDeliveryGateway gateway)
    {
        _calculator = calculator;
        _gateway = gateway;
    }

    public string GetStatistics(int customerId)
    {
        var records = _gateway.GetDeliveries(customerId);
        (double totalWeight, double totalCost) = _calculator
            .Calculate(records);

        return
            $"Total weight delivered: {totalWeight}. " +
            $"Total cost: {totalCost}";
    }
}
```

Two separate dependencies

The responsibility of communicating with the unmanaged dependency has transitioned to `DeliveryGateway`. Notice how this gateway is backed by an interface, which you can now use for mocking instead of the concrete class. The code in listing 11.15 is an example of the Humble Object design pattern in action. Refer to chapter 7 to learn more about this pattern.

11.6 Working with time

Many application features require access to the current date and time. Testing functionality that depends on time can result in false positives, though: the time during the act phase might not be the same as in the assert. There are three options for stabilizing this dependency. One of these options is an anti-pattern; and of the other two, one is preferable to the other.

11.6.1 Time as an ambient context

The first option is to use the *ambient context* pattern. You already saw this pattern in chapter 8 in the section about testing loggers. In the context of time, the ambient context would be a custom class that you'd use in code instead of the framework's built-in `DateTime.Now`, as shown in the next listing.

Listing 11.16 Current date and time as an ambient context

```
public static class DateTimeServer
{
    private static Func<DateTime> _func;
    public static DateTime Now => _func();

    public static void Init(Func<DateTime> func)
    {
        _func = func;
    }
}

DateTimeServer.Init(() => DateTime.Now);

DateTimeServer.Init(() => new DateTime(2020, 1, 1));
```

Initialization code for production

Initialization code for unit tests

Just as with the logger functionality, using an ambient context for time is also an anti-pattern. The ambient context *pollutes* the production code and makes testing more difficult. Also, the static field introduces a dependency shared between tests, thus transitioning those tests into the sphere of integration testing.

11.6.2 Time as an explicit dependency

A better approach is to inject the time dependency explicitly (instead of referring to it via a static method in an ambient context), either as a service or as a plain value, as shown in the following listing.

Listing 11.17 Current date and time as an explicit dependency

```
public interface IDateTimeServer
{
    DateTime Now { get; }
}

public class DateTimeServer : IDateTimeServer
{
    public DateTime Now => DateTime.Now;
}

public class InquiryController
{
    private readonly DateTimeServer _dateTimeServer;

    public InquiryController(
        DateTimeServer dateTimeServer)
    {
        _dateTimeServer = dateTimeServer;
    }

    public void ApproveInquiry(int id)
    {
        Inquiry inquiry = GetById(id);
```

Injects time as a service

```
        inquiry.Approve(_dateTimeServer.Now);
        SaveInquiry(inquiry);
    }
}
```

Injects time as a plain value

Of these two options, prefer injecting the time as a value rather than as a service. It's easier to work with plain values in production code, and it's also easier to stub those values in tests.

Most likely, you won't be able to always inject the time as a plain value, because dependency injection frameworks don't play well with value objects. A good compromise is to inject the time as a service at the start of a business operation and then pass it as a value in the remainder of that operation. You can see this approach in listing 11.17: the controller accepts `DateTimeServer` (the service) but then passes a `DateTime` value to the `Inquiry` domain class.

11.7 Conclusion

In this chapter, we looked at some of the most prominent real-world unit testing use cases and analyzed them using the four attributes of a good test. I understand that it may be overwhelming to start applying all the ideas and guidelines from this book at once. Also, your situation might not be as clear-cut. I publish reviews of other people's code and answer questions (related to unit testing and code design in general) on my blog at https://enterprisecraftsmanship.com. You can also submit your own question at https://enterprisecraftsmanship.com/about. You might also be interested in taking my online course, where I show how to build an application from the ground up, applying all the principles described in this book in practice, at https://unittesting-course.com.

You can always catch me on twitter at @vkhorikov, or contact me directly through https://enterprisecraftsmanship.com/about. I look forward to hearing from you!

Summary

- Exposing private methods to enable unit testing leads to coupling tests to implementation and, ultimately, damaging the tests' resistance to refactoring. Instead of testing private methods directly, test them indirectly as part of the overarching observable behavior.
- If the private method is too complex to be tested as part of the public API that uses it, that's an indication of a missing abstraction. Extract this abstraction into a separate class instead of making the private method public.
- In rare cases, private methods do belong to the class's observable behavior. Such methods usually implement a non-public contract between the class and an ORM or a factory.
- Don't expose state that you would otherwise keep private for the sole purpose of unit testing. Your tests should interact with the system under test exactly the same way as the production code; they shouldn't have any special privileges.

- Don't imply any specific implementation when writing tests. Verify the production code from a black-box perspective; avoid leaking domain knowledge to tests (see chapter 4 for more details about black-box and white-box testing).
- *Code pollution* is adding production code that's only needed for testing. It's an anti-pattern because it mixes up test and production code and increases the maintenance costs of the latter.
- The necessity to mock a concrete class in order to preserve part of its functionality is a result of violating the Single Responsibility principle. Separate that class into two classes: one with the domain logic, and the other one communicating with the out-of-process dependency.
- Representing the current time as an ambient context pollutes the production code and makes testing more difficult. Inject time as an explicit dependency—either as a service or as a plain value. Prefer the plain value whenever possible.