Records in Test Builder pattern

The problem

Sometimes we may be in a situation where we need to write tests for a component whose setup is difficult or just takes a long time.

Let's take a look at the following sample of the code which shows an attempt to write a unit test for the address validation service.

The creation of the object is quite quick and simple, the object is created in the same scope as the assertion, so it's quite easy to read and understand the whole test.

The problem

The problem starts to occur when we add more tests that need valid address objects.

Why this is a problem?

The problem

In this example, I have copy-pasted the address from one test to the other. Unfortunately, it's not the best approach.

The more duplication we have, the more expensive it will be to change the type we are dealing with.

We should wisely follow the DRY principle and be careful in how many places we instantiate complex types, because it may badly influence our technical depth and total size of the code.

We can fix this problem in a few ways:

- move variable to the test fixture field, initialize it from a place which guarantees fresh instance per test (constructor / Setup method) and change specific values inside the test if needed
- extract a static helper method and use it to create a new instance when it's needed
- create a builder for the type and use Builder Pattern

Solution 1 - field

It looks like the easiest solution, Address is set to the field. A field has an initializer which is called for every test, so there is no shared state between the tests.

Solution 1 - field - pros & cons

This solution has some pros and cons:

pros:

- simple and easy to understand code

cons:

- lives outside the test scope, so it's quite difficult to understand what is tested and what is the expected result
- field initialization has some limitations and sometimes may not be enough. For example, when we need to setup N fields depending on each other. To overcome this, we may use built-in features of the testing framework, like Setup and Teardown methods

Solution 2 - static helpers

Instead of relying on the shared field, we can create a helper method to create our test object. The method can be used in many test fixtures, which increases reusability.

Solution 2 - static helpers

We can even extend the method to have optional parameters with default values. It will allow us to create instances with different values when needed, for example when testing the bad request scenario.

Solution 2 - helpers - pros & cons

A helper is an improvement and can be all that is necessary in many test cases and scenarios.

pros:

- each test creates its own instance of the address that lives only in the scope of the test
- factory methods can be customizable, for example with optional parameters and default values

cons:

- method syntax can be overwhelming
- a change in the method signature may create a lot of breaking changes and build errors
- not everything can be easily created in such a way, for example, generation of the random ID would have to rely on additional static field

Solution 3 - Builder Pattern

A is a class specialized in creating a single class. It may store information about default values and have a method that can setup custom values. Usually, the builder is built with the fluent syntax, which makes it more readable.

Solution 3 - Builder Pattern

Solution 3 - Builder Pattern - pros & cons

The Builder Pattern is the evolution of the static helper methods and all test data factories. It brings an elegant way of creating objects with their default values and provides a helpful API for customizations.

pros:

- a unified and elegant way of creating a specific entity. For any next entity, we create the next builder
- provides customization points available with fluent API
- can fulfill and handle more complex scenarios, like creating sub-entities (using sub-builders) or holding a state, for example for randomizations
- the object is initialized only in the builder class. Because of that, any future changes to Address and the way how it's constructed will impact only the builder class, not dozens of other tests.

cons:

- requires to write more code write
- may look like overkill for the simple scenarios

Builder Pattern & records

Builder Pattern & records

We can use everything that we learned to redesign our builder pattern class into a record type

Builder Pattern & records

We can also use a clone mechanism to reuse partially prepared builder and customize its copies.

Solution 3 - Builder Pattern on records - pros & cons

Usage of the record type for the Builder Pattern is highly debatable, but it has its pros and cons.

pros:

- no need to write boilerplate methods and underlying fields
- possibility to clone an object if needed, for example when we need two partially pre-customized addresses

cons:

- combination of the and method that returns actual type may look like over-engineering

Materials

- https://raygun.com/blog/unit-testing-patterns/
- https://www.tdddev.com/2015/04/unit-test-builder-pattern.html
- https://ardalis.com/improve-tests-with-the-builder-pattern-for-test-data/