Chekhov's gun principle for testing

BY PAWEŁ ŚWIĄTKOWSKI
01 Nov 2024

It’s not an uncommon notion that writing tests is more of a storytelling task than a technical one. Most recently I encountered it in The Bike Shed podcast, but you can find blog posts and conference talks about it as well. And if it is a storytelling act, perhaps we should look into narrative principles to make our tests better?

One of the first rules that comes to my mind when thinking about storytelling is **Chekhov’s gun rule**. What’s it about? In Anton Chekhov’s own words (I’m quoting after Britannica):

> One must never place a loaded rifle on the stage if it isn’t going to go off. It’s wrong to make promises you don’t mean to keep.

Aforementioned Britannica also provides a definition as follows:

> principle in drama, literature, and other narrative forms asserting that every element introduced in a story should be necessary to the plot

How does that apply to our tests-storytelling? Imagine you are writing a test for a piece of e-commerce functionality. You have products in categories, and these categories have some kind of constraints on the buyer. The most obvious example would be that alcohol cannot be sold to people under the age of eighteen. In our case, such people cannot add such products to the cart.

Let's see an example test code - I'm using Elixir here, but this applies to most languages:

```
est "don't allow adding products to cart when age constraint is not met" do
 buyer = %Person{
   name: "John Smith",
   age: 17,
   country: :uk,
   registered_on: ~U[2023-09-16T18:17:22Z]
 }

 category = %Category{
   name: "Alcohol",
   external_id: 3242,
   constraints: [
     %AgeConstraint{min: 18}
   ]
 }

 product = %Product{
   name: "Triple Hazy IPA",
   category: category,
   sku: "TRI-557",
   added_at: ~U[2022-01-01T12:16:54Z]
 }

 cart = Cart.init(buyer)

 assert Cart.add(cart, product, quantity: 2) == {:error, :constraint_violated}
nd
```

This is not exactly a bad test, but it violates Chekhov's gun principle in multiple places. Every scalar introduced in our test "arrange" part is a gun in Chekhov's terms. It's there on the stage. The reader might expect it to go off. We have 10 scalars here, 11 guns on stage. What does going-off mean? That by changing this value to some other, the test should start failing. Let's examine our scalars:

- `name: "John Smith"` : no matter to what we change it, it won't make the test fail
- `age: 17` : if we change it to 18 or 22, the test will fail
- `country: :uk` : will not fail (unless we implement country-based constraints, but we do not for now)
- `registered_on: <date>` : does not matter
- `name: "Alcohol"` : does not matter

- `external_id: 3242` : what's even that? does not matter
- `min: 18` : changing it to `15` will make the test fail
- The remaining free from product, name, sku and date of adding do not affect the test
- `quantity: 2` : again, does not matter

Summing up, only two out of our eleven guns can go off here. About 18%. The rest is a pure distraction, leading the reader astray if they try to understand the test dynamics.

How do we fix it? By using abstractions! Tests need their abstractions just like your regular code does. And just like with regular code, you should make sure your abstractions are right in a given context. Here one potential abstraction is a factory. Let's see how the improved version could look like:

```
est "don't allow adding products to cart when age constraint is not met" do
 buyer = person_factory(age: 17)
 category = category_factory(constraints: [%AgeConstraint{min: 18}])
 product = product_factory(category: category)

 cart = Cart.init(buyer)

 assert Cart.add(cart, product) == {:error, :constraint_violated}
nd
```

The test is visibly shorter now. It contains just 5 non-empty lines. Every scalar here matters and there are just two of them. This way we don't overwhelm the reader and they can likely quickly draw a conclusion: aha, so the buyer is 17 years old, but the constraint is that they need to be at least 18, so the `:constraint_violated` error is returned. Makes sense.

Corollary: Make your guns visible

This is just my addition to the narrative principles:

> If something is to go off, show it before.

People rather don't like these *deus ex machina* moments. If there is something that is important for the test to pass, show it explicitly. Don't hide it under the abstraction.

Imagine that in the test above the first line just says `buyer = person_factory()`. Asked during the code review about it, the developer says that in the factory default age is actually 17, so there's no need to repeat it. **This is a misuse of the abstraction**. Don't rely on what's hidden. In theory everyone should be able to go into the factory and change the values - and tests should still pass. Some people even swear that defaults in factories should be random because of that.

Summary

To make your tests better and more readable (telling story in a better way), eliminate all the data that is irrelevant for the test flow. Leave only things that are important - values that, when changed in a certain way, will make the test fail. On the other hand, don't hide important stuff in the abstractions. Make the narrative flow without readers scratching their heads.

END OF THE ARTICLE

Tags: TESTING

This article was written by me – **Paweł Świątkowski** – on **01 Nov 2024**. I'm on Fediverse (Ruby-flavoured account, Elixir-flavoured account). Let's talk.

Loading comments…