

# Testing in an Elixir world

---

rafaelrochasilva@gmail.com

---

@RocRafael

# Rafael Rocha

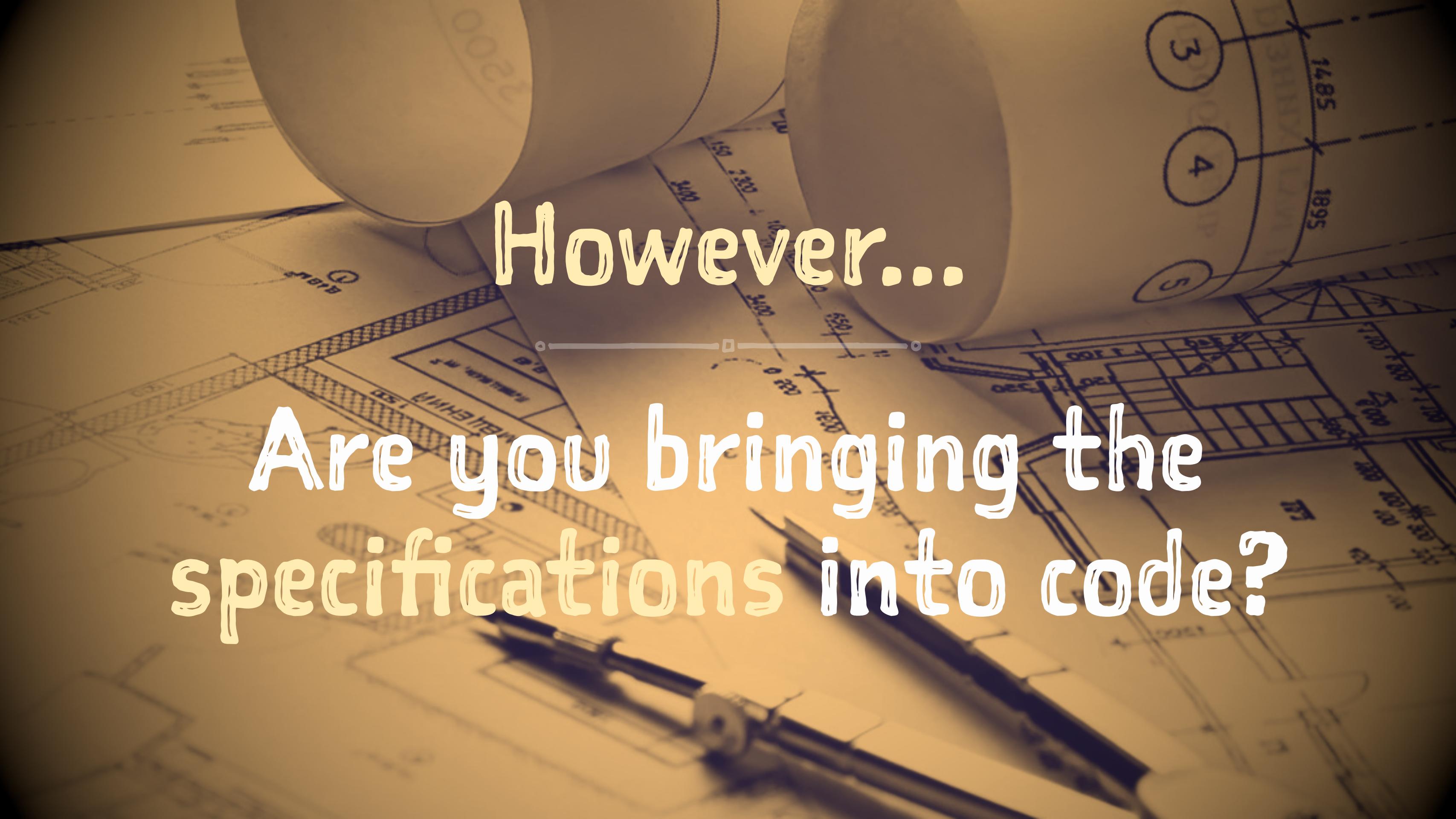
- Full Stack developer at  
*Plataformatec / The RealReal*
- Master degree in *Electrical Engineering*
- Bachelor of *Computer Science*
- Father 🧑‍🤝‍🧑 ❤️ & Cooker 🥘



# *Agenda*

- Specifications and software development
- Why testing?
- Base **test** concepts
  - Types of **test**
  - Test Pyramid
  - Test Clarity
- Use Case with **\*\*Elixir\*\***
  - Outside-in approach
  - Refactoring code with tests
  - Test double with fake clients
  - Doctests

When we start a user story, read the  
description, the acceptance criteria,  
and start coding

A sepia-toned photograph showing a close-up view of aeronautical charts and navigation equipment. In the center, a circular dial or compass rose is visible, with numbers like 300, 220, 140, and 340. To the right, there's a map with various flight routes, altitude levels, and geographical features. A prominent grid of latitude and longitude lines covers the background. In the foreground, a pen lies diagonally across the chart.

However...

Are you bringing the  
specifications into code?

Are you confident about your  
deliverables?



Before answering those  
questions we need some  
base testing concepts

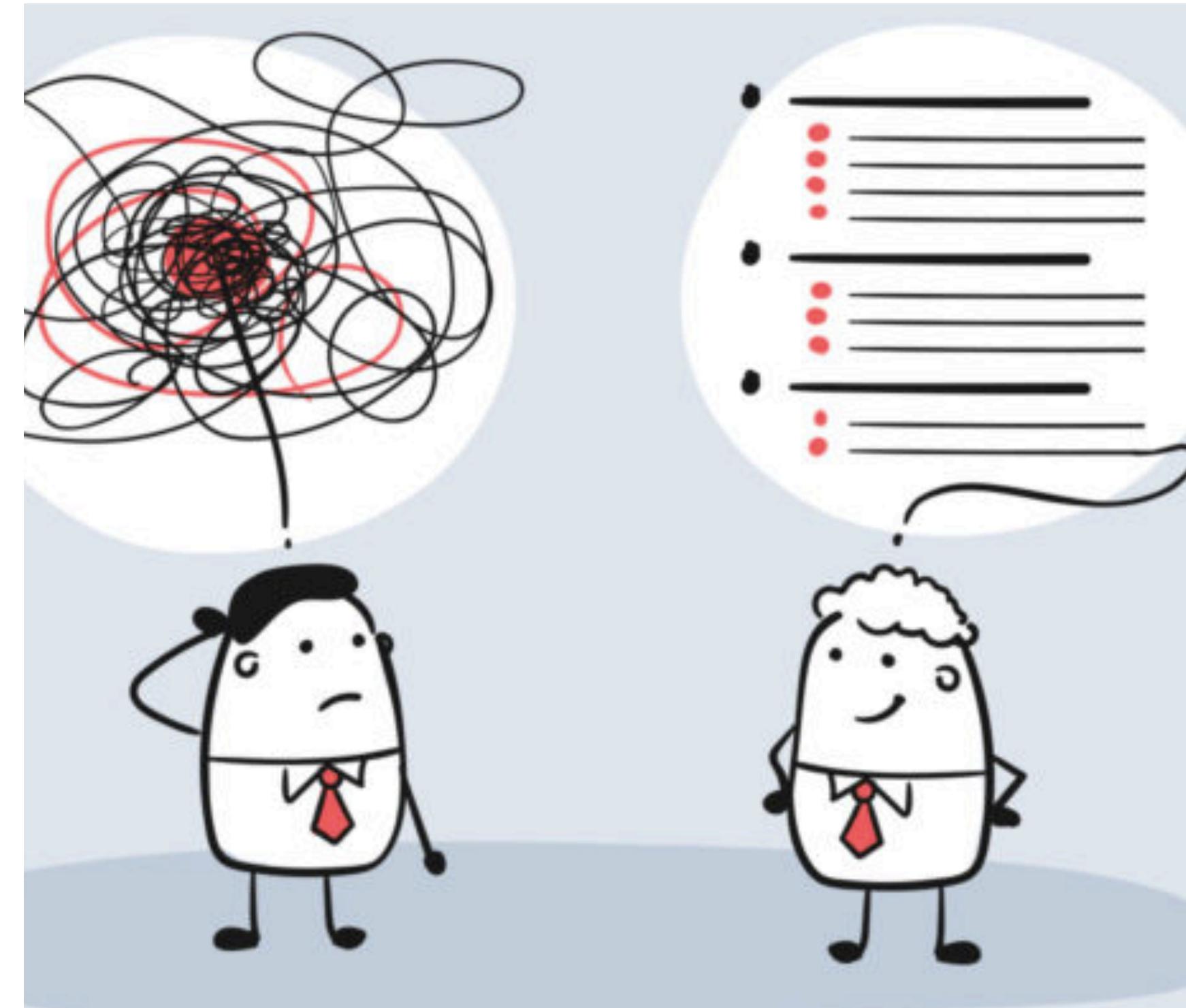
# Why testing?

[x] Being self-confident



# Why testing?

- [x] Being self-confident
- [x] Organizing thoughts



# Why testing?

- [x] Being **self-confident**
- [x] Organizing thoughts
- [x] Keeping the **costs low**



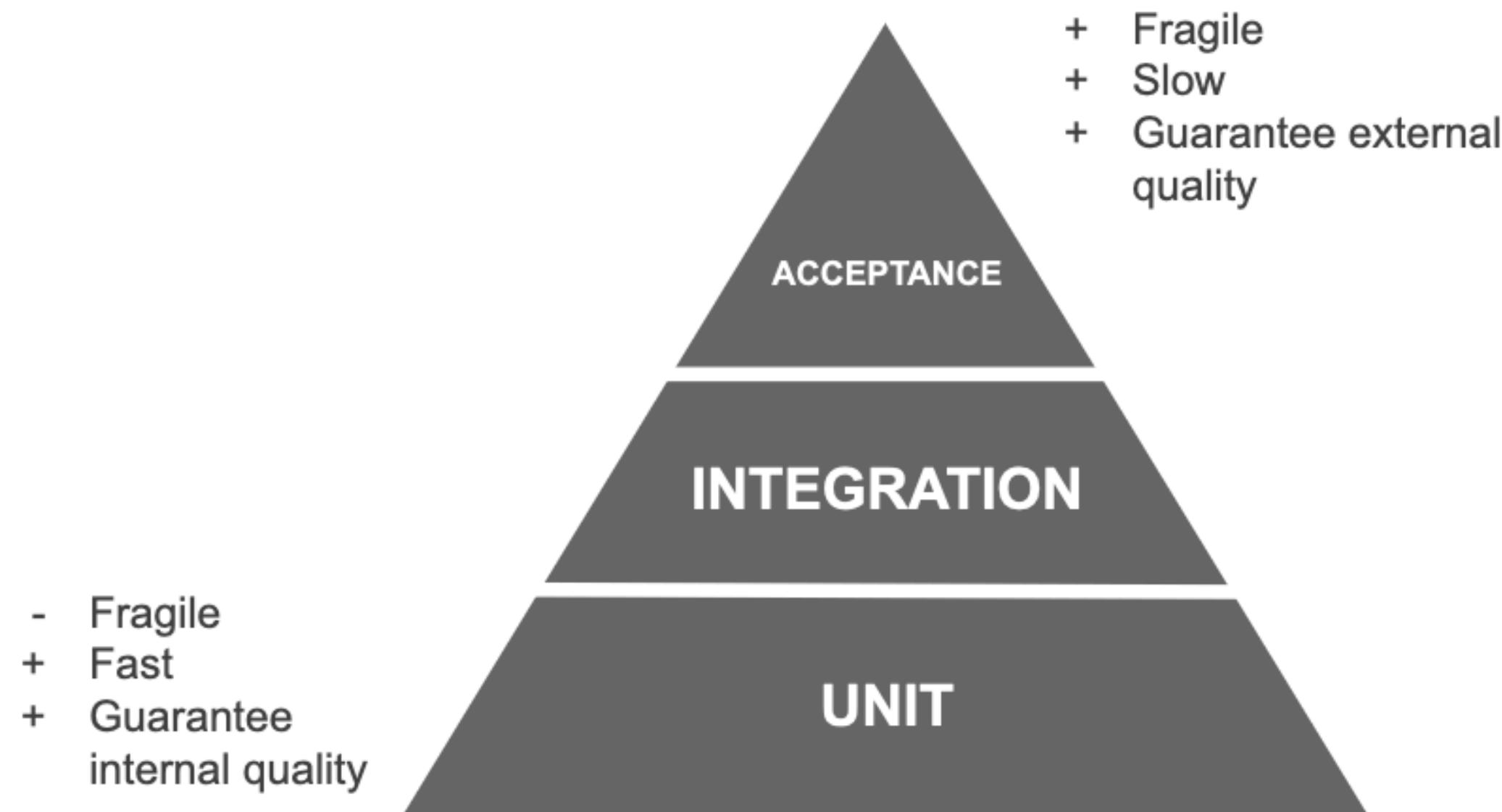
# Why testing?

- [x] Being self-confident
- [x] Organizing thoughts
- [x] Keeping the costs low
- [x] Bringing quality to the code



What are the types of  
tests?

# Test pyramid



# Acceptance:

- Express a usage scenario.
- End to end
- Close to the UI
- Slow
- Guarantee External Quality



# Integration:



- Test between acceptance and unit
- Test the behavior of 2 or more entities

# Unit:

- Tests the behavior of one entity
- Earlier detect mistakes
- Run Faster
- Guarantee internal quality
- Easier to fix mistakes



Imagine that we have an app called  
Greenbox

# Greenbox

Online store that sells organic beauty products, where users can choose a diffent variety of products and build its own box.

- We have a stock that changes its prices every *10 minutes*, due to our crazy promotions.

# Let's practice?

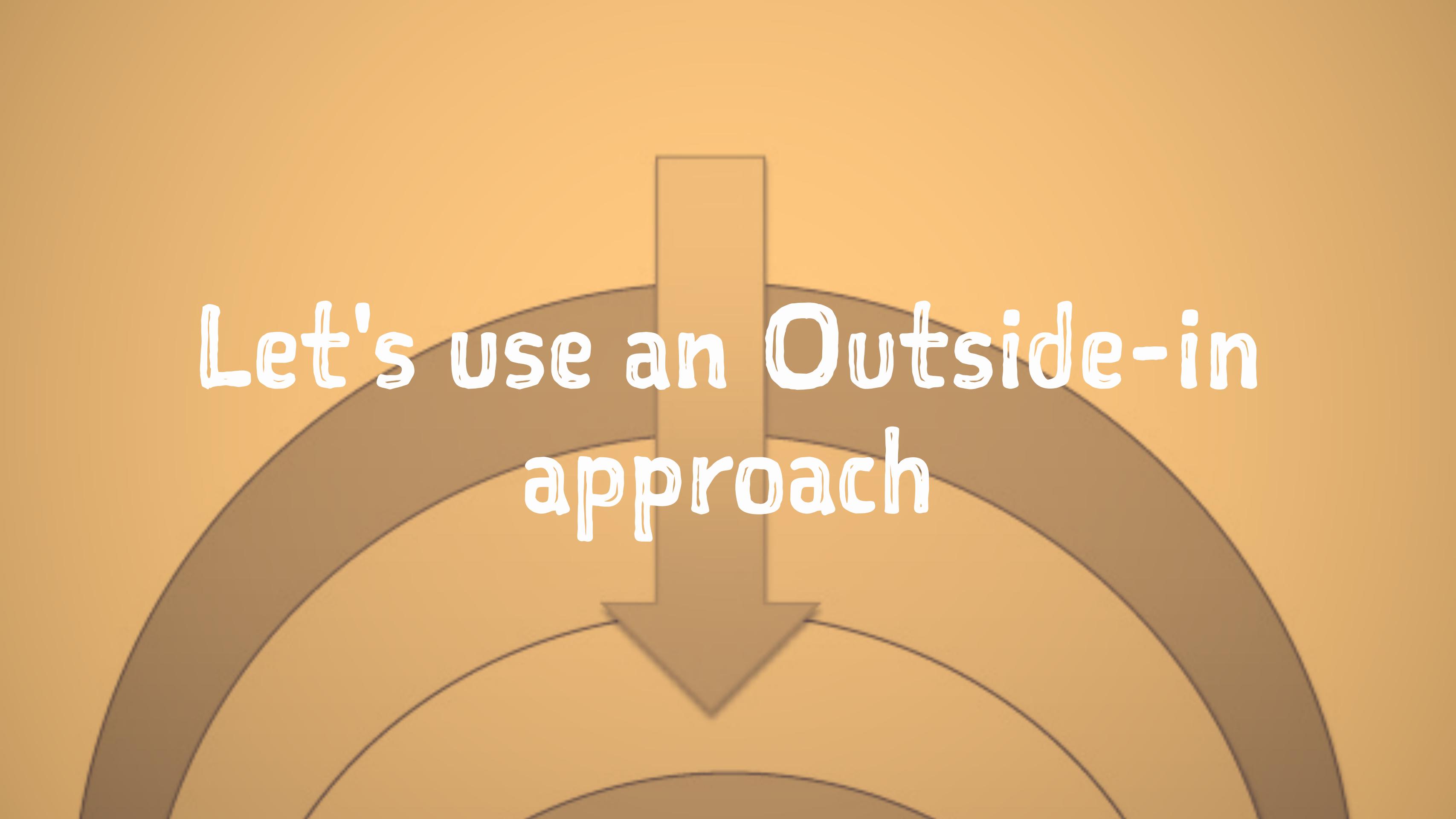
As a User, I want to fetch products from abcdpricing.com so that we can store the current name and price of a given product.

# Acceptance Criteria:

- All *id*, *products name* and *price* should be fetched time to time.
- The product name should be *capitalized*
- The price should be in a dollar format, like: *\$12.50*

# Basically what we have to do:

- 1) Fetch Products from the API
- 2) Build a structure with id, capitalized name, and price
- 3) Consume the data



Let's use an Outside-in  
approach

# What is the primary outside layer of our tasks?

- [] Fetch Products from the API
- [] Build a structure with id, capitalized name, and price
- [] Consume the data

# What is the primary outside layer of our tasks?

- [] Fetch Products from the API
- [] Build a structure with id, capitalized name, and price
- [!] Consume the data**

To fetch the products time to time, we  
are going to use a GenServer

# What is a GenServer?

"A GenServer is a process like any other process in Elixir, and it can be used to keep state, execute code asynchronously and so on."

-- Elixir Documentation

```
# [1] Consume the data
```

```
defmodule GreenBox.PriceUpdater do
  use GenServer

  def start_link do
    GenServer.start_link(__MODULE__, [])
  end

  def init(state) do
    {:ok, state}
  end

  def list_products(pid) do
    GenServer.call(pid, :list_products)
  end
```

# [1] Consume the data

```
def list_products(pid) do
  GenServer.call(pid, :list_products)
end
```

```
def handle_call(:list_products, _, state) do
  {:reply, state, state}
end
```

# What is the primary outside layer of our tasks?

[2] Fetch Products from the API

- [] Build a structure with id, capitalized name, and price
- [!] Consume the data

## # [2] Fetch Products from the API

```
defmodule GreenBox.PriceUpdater do
  use GenServer

  def start_link do
    GenServer.start_link(__MODULE__, [])
  end

  def init(_) do
    state = fetch_products()
    schedule_work()
    {:ok, state}
  end
```

```
# [2] Fetch Products from the API

@doc """
Run the job and reschedule it to run again after some time.
"""

def handle_info(:get_products, _state) do
  products = fetch_products()
  schedule_work()

  {:noreply, products}
end

defp fetch_products do
  response = HTTPoison.get!("http://abcdpricing.com/products")
  Poison.decode!(response.body)
end

@time_to_consume 10000 * 60 # 10 minutes
defp schedule_work do
  Process.send_after(self(), :get_products, @time_to_consume)
end
```

What is the primary outside layer of our tasks?

[2] Fetch Products from the API

[3] Build a structure with *id*, *capitalized name* and *price*

[!] Consume the data

# [3] Build a structure with id, capitalized name and price

```
def init(_) do
  state = build_products()
  schedule_work()
  {:ok, state}
end
```

```
defp build_products do
  fetch_products()
  |> process_products()
end
```

```
# [3] Build a structure with id, capitalized name and price
```

```
defp fetch_products do
  response = HTTPoison.get!("http://abcdpricing.com/products")
  Poison.decode!(response.body)
end

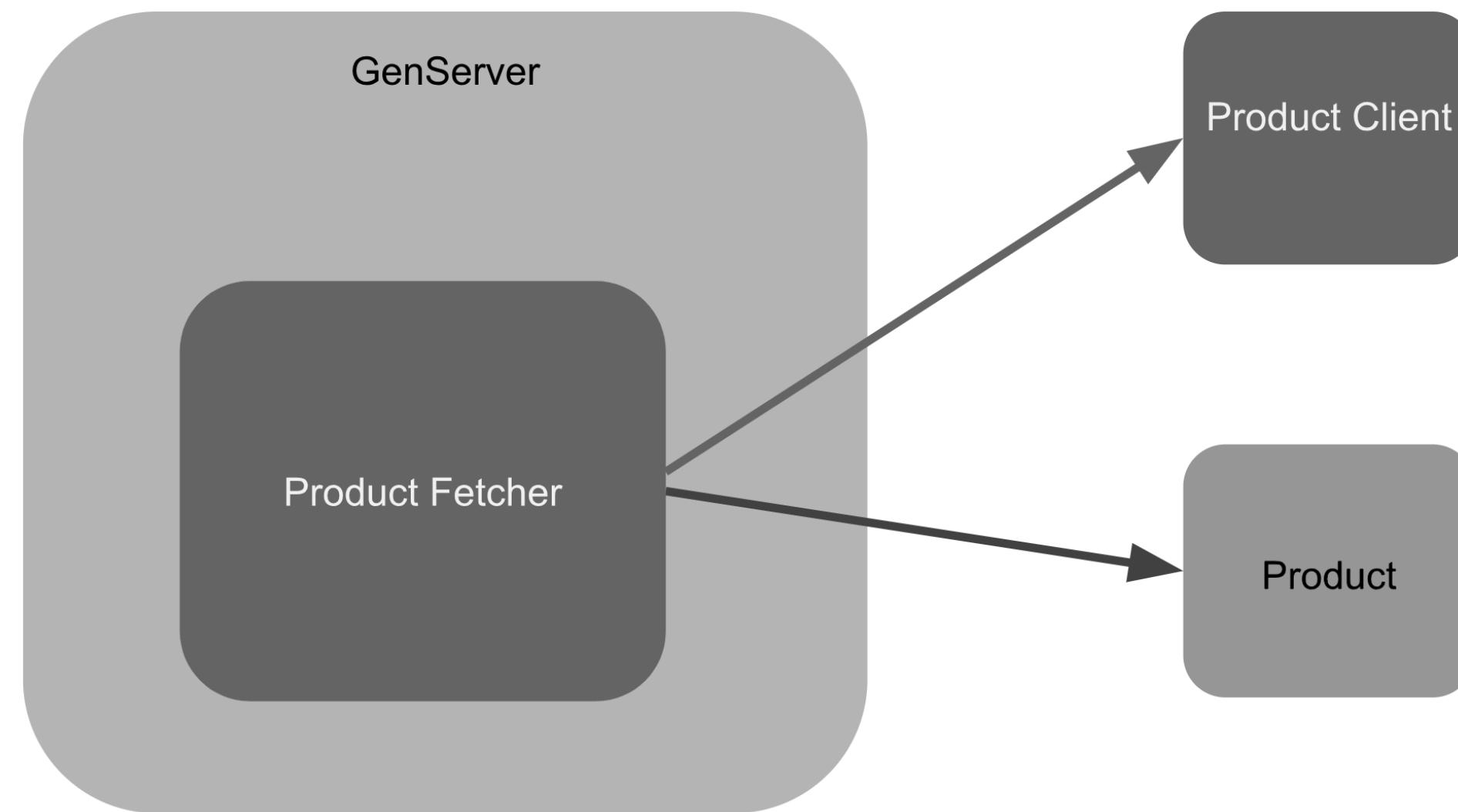
defp process_products(products) do
  Enum.map(products, fn %{id: id, name: name, price: price} =>
    new_name = String.capitalize(name)
    new_price = "$#{price/100}"
    %{
      id: id,
      name: new_name,
      price: new_price
    }
  end)
end
```

How can I test a  
GenServer?

Be careful to not test your servers  
through the callbacks  
otherwise you are going to test the  
GenServer implementation.

Change your Design!

# Fetch Products Architecture



Let's build an integration test to guide  
the development

```
# Let's build an INTEGRATION TEST

defmodule Greenbox.ProductFetcherTest do
  use ExUnit.Case, async: true
  alias Greenbox.ProductFetcher

  # Specifications into code
  describe "Given a request to fetch a list of products" do
    test "builds a list of products with id, capitalized name and price in dollar" do
      products = ProductFetcher.build()

      assert [
        %{id: "1234", name: "Blue ocean cream", price: _},
        %{id: "1235", name: "Sea soap", price: _}
      ] = products
    end
  end
end
```

```
# Let tests guide the development
```

```
test "builds a product with the price with a dollar sign" do
  product =
    ProductFetcher.build()
    |> List.first()

  # Expected format "$12.45"
  assert Regex.match?(~r(\$\d+\.\d+), product.price)
end
```

# Product Fetcher - A new entity

```
defmodule Greenbox.ProductFetcher do
  def build do
    fetch_products() 
    |> process_products()
  end

  defp fetch_products do
    response = HTTPoison.get!("http://abcdpricing.com/products")
    Poison.decode!(response.body)
  end

  defp process_products(products) do
    Enum.map(products, fn %{id: id, name: name, price: price} ->
      %{
        id: id,
        name: capitalize_name(name),
        price: price_to_money(price)
      }
    end)
  end
end
```

# Listen to your code...

```
defp process_products(products) do
  Enum.map(products, fn %{id: id, name: name, price: price} ->
    %{
      id: id,
      name: capitalize_name(name),
      price: price_to_money(price)
    }
  end)
end
```

# Listen to your code

```
defp price_to_money(price) do
  "$#{price / 100}"
end
```

```
defp capitalize_name(name) do
  String.capitalize(name)
end
```

# Build the *unit tests* to handle product structure

```
defmodule Greenbox.ProductTest do
  use ExUnit.Case, async: true
  alias Greenbox.Product

  describe "Given a product" do
    test "transforms its name by capitalizing it" do
      # Setup
      product_name = "BLUE SOAP"

      # Exercise
      capitalized_name = Product.capitalize_name(product_name)

      # Verify
      assert capitalized_name == "Blue soap"
    end
  end
end
```

```
# Build the unit tests to handle product structure

test "transforms the price in cents to dollar" do
  # Setup
  product_price_in_cents = 1253

  # Exercise
  product_price = Product.price_to_money(product_price_in_cents)

  # Verify
  assert product_price == "$12.53"



```

# Product Entity

```
defmodule Greenbox.Product do
  defstruct [:id, :name, :price]

  def price_to_money(price) do
    "$#{price / 100}"
  end

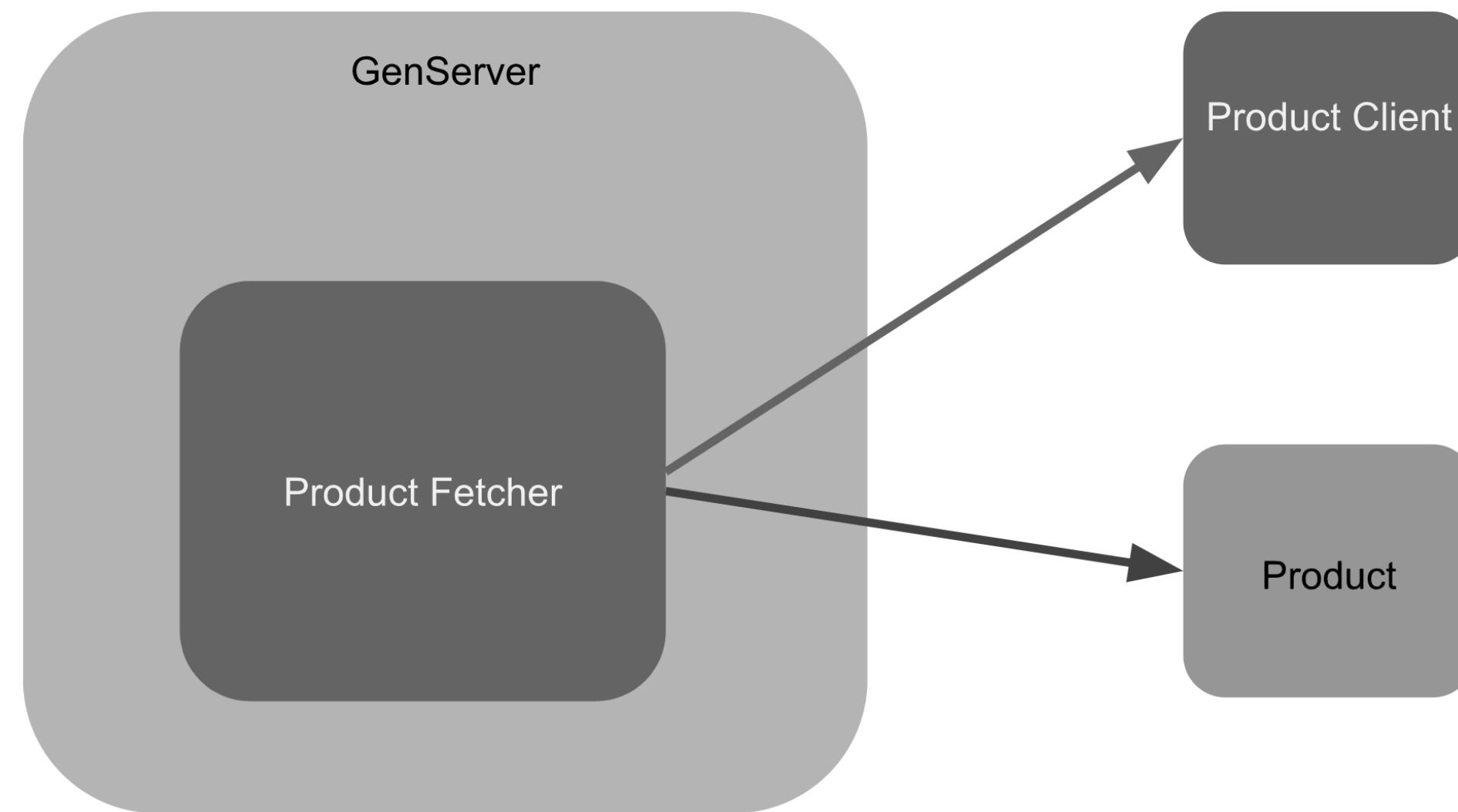
  def capitalize_name(name) do
    String.capitalize(name)
  end
end
```

And finally, build a client to call the external API

```
defmodule Greenbox.ProductClient do
  def fetch_products do
    response = url() |> HTTPoison.get!()
    Poison.decode!(response.body)
  end

  defp url do
    Application.get_env(:greenbox, :abc_products_url)
  end
end
```

# Fetch Products Architecture



Did you notice that we are hitting the  
API every time we run our tests?

## Call to the external API

```
defmodule Greenbox.ProductClient do
  def fetch_products do
    response = url() |> HTTPoison.get!()
    Poison.decode!(response.body)
  end

  defp url do
    Application.get_env(:greenbox, :abc_products_url)
  end
end
```

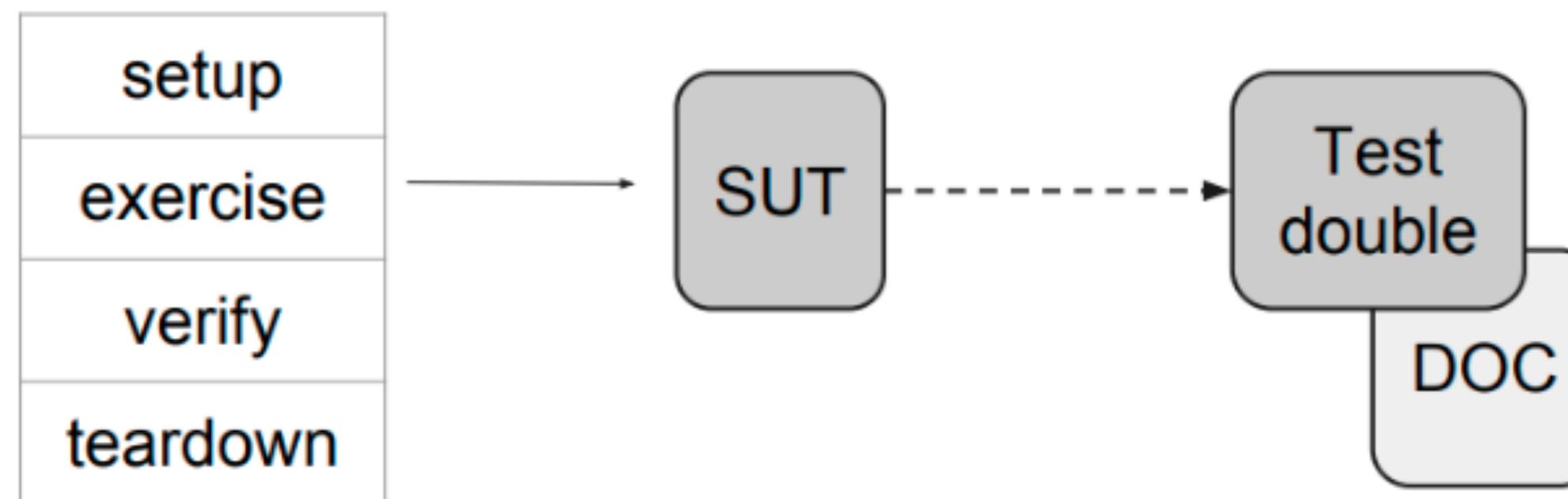
Test Double, how to stub in  
Elixir?

# Test Double

SUT: System Under Test

DOC: Collaborator

Double: Is the object that substitutes the real DOC



Let's start creating our  
Double

# Fake client

```
# test/support/fake_client.ex
defmodule Greenbox.FakeClient do

  def fetch_products do
    [
      %{id: "1234", name: "BLUE OCEAN CREAM", price: Enum.random(8000..10000)},
      %{id: "1235", name: "SEA SOAP", price: Enum.random(5000..60000)}
    ]
  end
end
```

# Configure the Fake Client

```
defmodule Greenbox.MixProject do
  use Mix.Project

  def project do
    [
      app: :greenbox,
      version: "0.1.0",
      elixir: "~> 1.7",
      elixirc_paths: elixirc_paths(Mix.env()),
      start_permanent: Mix.env() == :prod,
      deps: deps()
    ]
  end

  # Specifies which paths to compile per environment.
  defp elixirc_paths(:test), do: ["lib", "test/support"]
  defp elixirc_paths(_), do: ["lib"]
end
```

```
# config/config.exs
config :greenbox,
  abc_products_client: Greenbox.ProductClient
```

```
# config/test.exs
```

```
config :greenbox,  
  abc_products_client: Greenbox.FakeClient
```

# Other ways to stub requests in Elixir

- Bypass (<https://github.com/PSPDFKit-labs/bypass>)
- Mox (<https://github.com/plataformatec/mox>)

# What about Doctests?

---

Are they supposed to substitute tests?

```
# Doctest
```

```
defmodule Greenbox.Product do
  defstruct [:id, :name, :price]

  @doc """
  Converts price in cents to a string money format.
```

```
## Example:
```

```
iex> Greenbox.Product.price_to_money(1245)
"$12.45"
```

```
"""
```

```
def price_to_money(price) do
  "$#{price / 100}"
end
```

How tests can reflect specifications and  
help us to build confident code?

- Write clear *test descriptions*
- Follow the *specifications*
- Think *outside-in*
- Think in the Test *Pyramid*
- Use *stubs* or build fake clients
- *Don't test callbacks*
- Abstract your code into *modules*

# Thank you!

<https://github.com/rafaelrochasilva/greenbox>

[https://github.com/rafaelrochasilva/  
testinginelixir\\_talk](https://github.com/rafaelrochasilva/testinginelixir_talk)

[http://blog.plataformatec.com.br/2018/11/starting-  
with-elixir-the-study-guide/](http://blog.plataformatec.com.br/2018/11/starting-with-elixir-the-study-guide/)

## References:

<https://github.com/plataformatec/mox>

<https://github.com/PSPDFKit-labs/bypass>

<https://github.com/keathley/wallaby>