

Coupling Data and Behaviour

(in Elixir)

2018-04-17

Guilherme de Maio



@nirev ?



Based in São Paulo @ Brasil

Elixir @ xerpa.com.br (2015)

SP Elixir Meetups regular ;)

lot's of Java before



Full-time @ Telnyx (Remote)

Based in São Paulo @ Brasil

Elixir @ xerpa.com.br (2015)

SP Elixir Meetups regular ;)

lot's of Java before



Full-time @ Telnyx (Remote)



Based in São Paulo @ Brasil

Elixir @ xerpa.com.br (2015)

SP Elixir Meetups regular ;)

lot's of Java before





Data and Behaviour

How so?

“

One of the fundamental principles of object-oriented design is to **combine data and behavior**, so that the basic elements of our system (objects) combine both together.

Michael Fowler

“

Finally, another **downside** to object-oriented programming is the **tight coupling between function and data**. In fact, the Java programming language forces you to build programs entirely from class hierarchies, restricting all functionality to containing methods in a highly restrictive “Kingdom of Nouns” (Yegge 2006).

Fogus/Houser | The Joy of Clojure

Call Control

Binary strings via socket

Call Control: needs

- **Several commands sent via socket as binary strings**
- **Each command can take different parameters**
- **Adding new commands should be easy**
- **Sending commands should be the same for all**
- **Crossconcerns: logging, authorization, etc...**



Call Control using Objects

Beware: pseudocode

CallControl using Objects

```
interface Command {  
    String render();  
}
```



CallControl using Objects

```
class Commander {  
    def send_command(Command command) {  
        command_str = command.render()  
        Socket.send(command_str)  
    }  
}
```



CallControl using Objects

```
class Play impl Command {  
    string uuid;  
    string audio;  
    string channels;  
  
    def render() {  
        "playback #{uuid} " +  
        " #{audio} #{channels}"  
    }  
}
```

CallControl using Objects

```
class Play impl Command {  
    string uuid;  
    string audio;  
    string channels;  
  
    def render() {  
        "playback #{uuid} " +  
        " #{audio} #{channels}"  
    }  
}
```

```
class Record impl Command {  
    string uuid;  
    string output;  
    string format;  
    string channels  
  
    def render() {  
        "record #{uuid} " +  
        " #{output} " +  
        "format=#{format}" +  
        "channels=#{channels}"  
    }  
}
```

CallControl using Objects

- Adding new commands should be easy ✅

```
class Whatever impl Command {  
    string uuid;  
    string audio;  
    string channels;  
  
    def render() {...}  
}
```


CallControl using Objects

```
class Commander {  
    def send_command(command) {  
        command_str = command.render()  
        Socket.send(command_str)  
    }  
}
```



Properties:

- Extensible
- Open/Closed

CallControl using Objects

```
class Whatever impl Command {  
    string uuid;  
    string audio;  
    string channels;  
  
    def render() {...}  
}
```

What if we need more methods??

CallControl using Objects

```
class Whatever impl Command {  
    string uuid;  
    string audio;  
    string channels;  
  
    def render() {...}  
    def log() {...}  
}
```

What if we need more methods??

```
interface Command {  
    String render();  
    void log();  
}
```

CallControl using Objects

```
class Whatever impl Command, Loggable {
```

```
    string uuid;  
    string audio;  
    string channels;
```

```
    def render() {...}
```

```
    def log() {...}
```

```
}
```

What if we need more methods??

```
interface Command {
```

```
    String render();
```

```
}
```

```
interface Loggable {
```

```
    void log();
```

```
}
```



CallControl using Objects

- Several commands sent via socket as binary strings ✓
- Each command can take different parameters ✓
- Adding new commands should be easy ✓
- Sending commands should be the same for all ✓
- Crossconcerns: logging, authorization, etc... ✓



Functional Commands ;)

Functional version

```
defmodule Commander do  
  def send(command) do  
    command ▷ render() ▷ socket_send()  
  end  
end
```



Functional version

```
defmodule Play do
  defstruct [
    :uuid,
    :audio,
    :channels]
end
```

```
defmodule Record do
  defstruct [
    :uuid,
    :output,
    :format,
    :channels]
end
```



Functional version

```
defmodule Play do
  defstruct [
    :uuid,
    :audio,
    :channels]
end
```

```
defmodule Record do
  defstruct [
    :uuid,
    :output,
    :format,
    :channels]
end
```

```
defmodule Commander do
  def send(command) do
    command ▷ render() ▷ socket_send()
  end

  def render(%Play{} = p) do
    "playback #{p.uuid} #{p.audio} #{p.channels}"
  end

  def render(%Record{} = r) do
    "record #{r.uuid} " ◇ "#{r.output} " ◇
    "format=#{r.format}" ◇ "channels=#{r.channels}"
  end
end
```

Functional version

So far:

How to add a new command?

How to add a new function?



Functional version

So far:

How to add a new command?

How to add a new function?



Functional version

This is the way of doing this with Erlang.

With pattern-matching it's possible to dispatch per struct but it's not possible to do it following “open/closed” principle.

This is the sort of thing that makes the "raison d'être" of Elixir.



“

The problem I have with **Erlang** is that the language is somehow **too simple**, making it very hard to eliminate boilerplate and structural duplication. Conversely, the resulting code gets a bit messy, being harder to write, analyze, and modify. After coding in Erlang for some time, I thought that **functional programming is inferior to OO**, when it comes to efficient code organization.

Sasa Juric | Why Elixir

Protocols :)

Protocols

```
defprotocol Command do  
  def render(cmd)  
end
```



Protocols

```
defmodule Play do
  defstruct [
    :uuid,
    :audio,
    :channels]
end

defmodule Record do
  defstruct [
    :uuid,
    :output,
    :format,
    :channels]
end

defimpl Command, for: Play do
  def render(p) do
    "playback #{p.uuid} #{p.audio} #{p.channels}"
  end
end

defimpl Command, for: Record do
  def render(r) do
    "record #{r.uuid} #{r.output} " <
    "format=#{r.format} channels=#{r.channels}"
  end
end
```

Protocols

```
defmodule Commander do
  def send(command) do
    command
    ▷ Command.render()
    ▷ socket_send()
  end
end
```



Protocols

- Several commands sent via socket as binary strings ✓
- Each command can take different parameters ✓
- Adding new commands should be easy ✓
- Sending commands should be the same for all ✓
- Crossconcerns: logging, authorization, etc... ✓

Protocols

- Several commands sent via socket as binary strings ✓
- Each command can take different parameters ✓
- Adding new commands should be easy ✓
- Sending commands should be the same for all ✓
- Crossconcerns: logging, authorization, etc... ✓



Protocols: what, why, how

WHAT

Protocols: inspired by Clojure



- Provide a high-performance, dynamic polymorphism construct as an alternative to interfaces
- Support best part of interfaces, while avoiding the drawbacks

Protocols: inspired by Clojure

```
(defprotocol P  
  (foo [x])  
  (bar [x y]))
```

```
(deftype Foo [a b c]  
  P  
  (foo [x] a)  
  (bar [x y] (+ c y)))
```

```
(foo (Foo. 1 2 3))  
⇒ 1
```

```
(bar (Foo. 1 2 3) 42)  
⇒ 45
```


Protocols: inspired by Clojure

```
(defprotocol P  
  (foo [x])  
  (bar [x y]))
```

```
(deftype Foo [a b c]  
  P  
  (foo [x] a)  
  (bar [x y] (+ c y)))
```

```
(foo (Foo. 1 2 3))
```

```
⇒ 1
```

```
(bar (Foo. 1 2 3) 42)
```

```
⇒ 45
```

```
defprotocol P do  
  def foo(x)  
  def bar(x, y)  
end
```

```
defmodule Foo do  
  defstruct [:a, :b, :c]
```

```
  defimpl P do  
    def foo(f), do: f.a  
    def bar(f, y) do: f.c + y  
  end
```

```
end
```

Protocols: what can you do

```
defprotocol Size do
  def size(data)
end
```


Protocols: what can you do

→ Implement it for native types

```
defprotocol Size do
  def size(data)
end
```

```
defimpl Size, for: BitString do
  def size(string), do: byte_size(string)
end
```

```
defimpl Size, for: Map do
  def size(map), do: map_size(map)
end
```



Protocols: what can you do

→ Implement it for native types

```
defprotocol Size do
  def size(data)
end
```

```
defimpl Size, for: BitString do
  def size(string), do: byte_size(string)
end
```

```
defimpl Size, for: Map do
  def size(map), do: map_size(map)
end
```

```
iex> Size.size "abacate"
7
```

```
iex> Size.size %{a: 1, b: 2}
2
```



Protocols: what can you do

→ Implement it for structs

```
defprotocol Size do
  def size(data)
end
```

```
defmodule Bag do
  defstruct [:items]
```

```
iex> Size.size %Bag{items: [1,2,3]}
3
```

```
  defimpl Size do
    def size(bag), do: Enum.count(bag.items)
  end
end
```

Protocols: what can you do

→ Implement it for structs, decoupled from module definition

```
defprotocol Size do
  def size(data)
end
```

```
defmodule Bag do
  defstruct [:items]
end
```

```
defimpl Size, for: Bag do
  def size(bag), do: Enum.count(bag.items)
end
```



Protocols: what can you do

→ Have defaults

```
defprotocol Size do
  def size(data)
end
```

```
defimpl Size, for: Any do
  def size(_), do: 0
end
```

```
defmodule Sizeless do
  @derive [Size]
  defstruct [:wat]
end
```

```
iex> Size.size %Sizeless{wat: :wat}
0
```

Protocols: what can you do

→ Have defaults

```
defprotocol Size do
  def size(data)
end
```

```
defimpl Size, for: Any do
  def size(%{size: size}), do: size
  def size(_), do: 0
end
```

```
defmodule Sizeable do
  @derive [Size]
  defstruct [:size]
end
```

```
iex> Size.size %Sizeable{size: 42}
42
```


Protocols: what can you do



HOW

Protocols: how

```
defmodule Data do  
  defstruct [:x]  
end
```

```
%Data{x: 1} = %{__struct__: Data, x: 1}
```

Protocols: how

```
defmodule Protocolz do
  def dispatch(function_name, data) do
    struct_module = data.__struct__
    :erlang.apply(struct_module, function_name, [data])
  end
end
```



Protocols: how

```
defmodule Play do
  defstruct [:id, :audio]
  def render(p) do
    "playback #{p.id} #{p.audio}"
  end
end
```

```
defmodule Record do
  defstruct [:id, :output]
  def render(r) do
    "record #{r.id} #{r.output}"
  end
end
```

```
play = %Play{id: "19831", audio: "audio.mp3"}
Protocolz.dispatch(:render, play)
```

```
record = %Record{id: "908301931", output: "record.mp3"}
Protocolz.dispatch(:render, record)
```

Protocols: how

```
defmodule Protocolz do
  defmacro defimplz(protocol, [for: struct_module], [do: block]) do
    quote do
      defmodule Module.concat([unquote(protocol), unquote(struct_module)]) do
        unquote(block)
      end
    end
  end
end

def dispatch(protocol, function_name, data) do
  struct_module = data.__struct__
  impl_module = Module.concat(protocol, struct_module)
  :erlang.apply(impl_module, function_name, [data])
end
end
```

Protocols: how

```
defimplz Command, for: Play do
  def render(p) do
    "playback #{p.id} #{p.audio}"
  end
end
```

```
defimplz Command, for: Record do
  def render(r) do
    "record #{r.id} #{r.output}"
  end
end
```

```
play = %Play{id: "19831", audio: "audio.mp3"}
Protocolz.dispatch(Command, :render, play) # Command.Play.render()

record = %Record{id: "908301931", output: "record.mp3"}
Protocolz.dispatch(Command, :render, record) # Command.Record.render()
```

Protocols: how

```
defmacro defprotocol(name, do: block) do
  quote do
    defmodule unquote(name) do
      import Protocol, only: [def: 1]
      # Invoke the user given block
      _ = unquote(block)
    end
  end
end
```


Protocols: how

```
defmacro def({name, _, args}) do
```

```
  quote do
```

```
    name = unquote(name)
```

```
    arity = unquote(arity)
```

```
  Kernel.def unquote(name)(unquote_splicing(args)) do
```

```
    impl_for!(term).unquote(name)(unquote_splicing(args))
```

```
  end
```

```
end
```

```
end
```

Protocols: how

```
defprotocol Command do
  def render(x)
end
```

```
defmodule Command do
  def render(x) do
    impl_for!(x).render(x)
  end

  defp impl_for!(struct) do
    target = Module.concat(__MODULE__, struct)

    case Code.ensure_compiled?(target) do
      true → target.__impl__(:target)
      false → nil
    end
  end
end
```

Protocols: how



Protocols: how

```
defmodule Command do
  def render(x) do
    impl_for!(x).render(x)
  end

  defp impl_for!(struct) do
    target = Module.concat(__MODULE__, struct)

    case Code.ensure_compiled?(target) do
      true → target.__impl__(:target)
      false → nil
    end
  end
end
```

Protocols: how

```
defmodule Command do
```

```
  def render(x) do
```

```
    impl_for!(x).render(x)
```

```
  end
```

```
defp impl_for!(struct)
```

```
  target = Module.__MODULE__, struct)
```

```
  case Code._compiled?(target) do
```

```
    true  -> get.__impl__(:target)
```

```
    false -> nil
```

```
  end
```

```
end
```

```
end
```

CONSOLIDATION

WHY

String.chars

```
defmodule Product do
  defstruct title: "", price: 0

  defimpl String.Chars do
    def to_string(%Product{title: title, price: price}) do
      "#{title}, ${price}"
    end
  end
end
```

Poison

```
defmodule Person do
  @derive [Poison.Encoder]
  defstruct [:name, :age]
end
```

```
defimpl Poison.Encoder, for: Person do
  def encode(%{name: name, age: age}, opts) do
    Poison.Encoder.BitString.encode("#{name} (#{age})", opts)
  end
end
```


infinitered/elasticsearch-elixir

```
defimpl Elasticsearch.Document, for: MyApp.Post do
  def id(post), do: post.id
  def type(_post), do: "post"
  def parent(_post), do: false
  def encode(post) do
    %{
      title: post.title,
      author: post.author
    }
  end
end
```

What about Behaviour?

Behaviour vs Protocols

```
defprotocol Command do
  @spec render(t) :: String.t
  def render(cmd)
end
```

```
defmodule Commander do
  @type r :: :ok | :error
  @callback render(Command.t) :: r
end
```



Behaviour vs Protocols

```
defprotocol Command do
  @spec render(t) :: String.t
  def render(cmd)
end
```

```
defmodule Commander do
  @type r :: :ok | :error
  @callback send(Command.t) :: r
end
```

```
defmodule FakeCommander do
  @behaviour Commander
  def send(_command), do: :ok
end
```

```
defmodule SocketCommander do
  @behaviour Commander
  def send(command) do
    ...
  end
end
```

| Protocols | Behaviours |
|--------------------------------------|--|
| Dispatch on type | Dispatch on module |
| Contract for Data | Contract for modules |
| Elixir-only | Erlang too |
| Ex: Encoding/ Serializing structs | Ex: sending email via SMTP/IMAP, API or mocks |

Bottomline?

Coupling data and behaviour can be **GOOD**,
and Elixir protocols are f!%*@*#%*@* awesome!



Check us out!
www.telnyx.com

Thank You!

❤ ElixirConf.EU 2018

Guilherme de Maio

guilherme@taming-chaos.com

 @nirev