No silicon heaven?
Preposterous!
Where would all the calculators go?

The road to GenServer

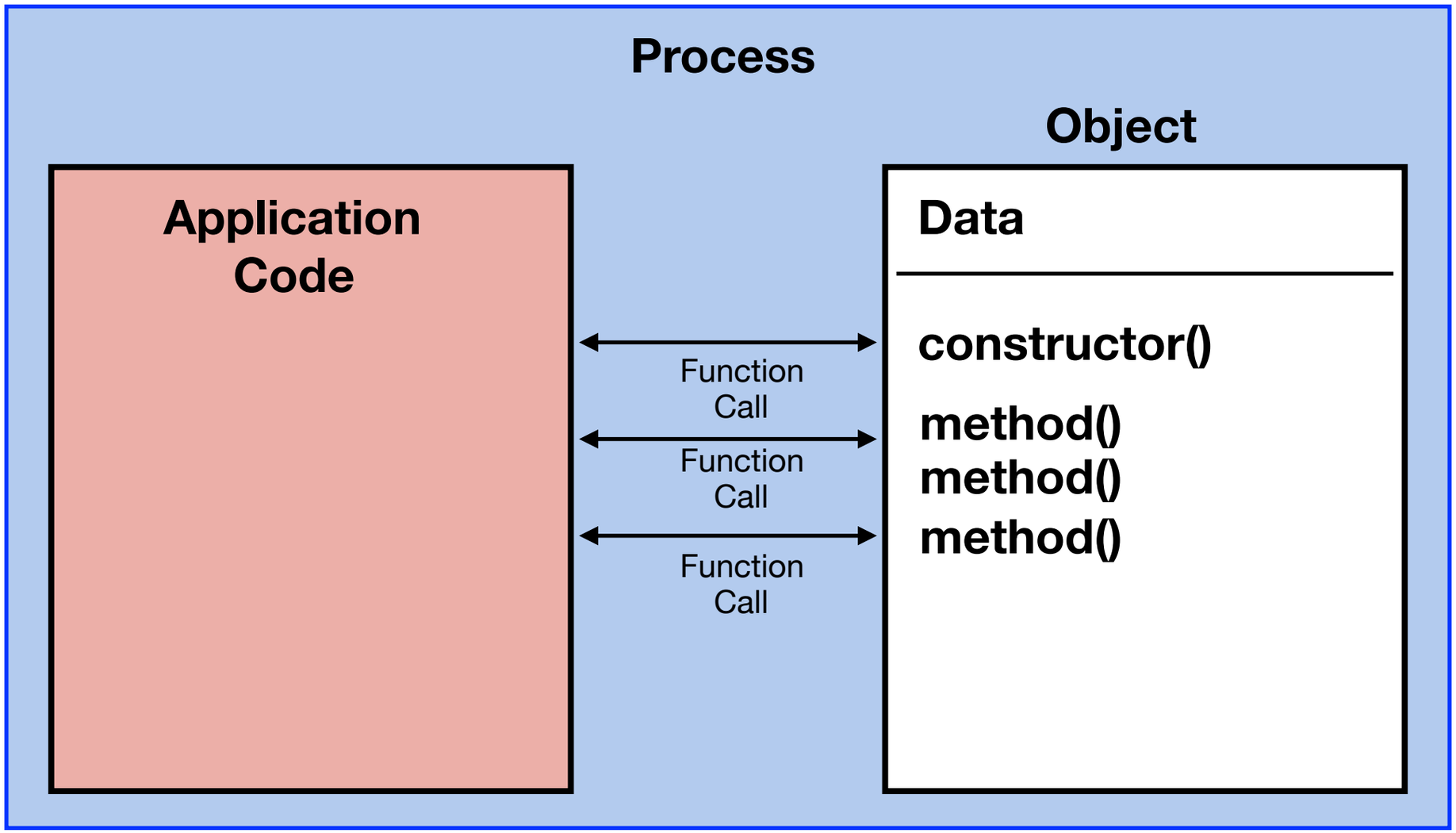
roger.wenham@gmx.net

About me

- Studied Electrical and Electronic Engineering.
- Worked for ICL as a digital electronics engineer.
- Worked for an engine test company on hardware/software for 10 years.
- Worked for myself as a software consultant for 15 years. (UK/DE)
- A company made me an offer I couldn’t refuse.
- Worked as an employee of a banking sector company (ATM Management).
- Now semi-retired, but I still enjoy programming and learning new languages.
- Assembler, C, Delphi/Lazarus, C++, Python, Erlang, learning Elixir

Modern Object Implementation

An object encapsulates some data and the code operating on that data.



Modern Object Problems

- If the object crashes, the whole program crashes!

- If the program or object, that the object is part of crashes, the encapsulated data is lost.

- The object can possibly share global data with the rest of the program. If the shared global data is changed from outside the object:

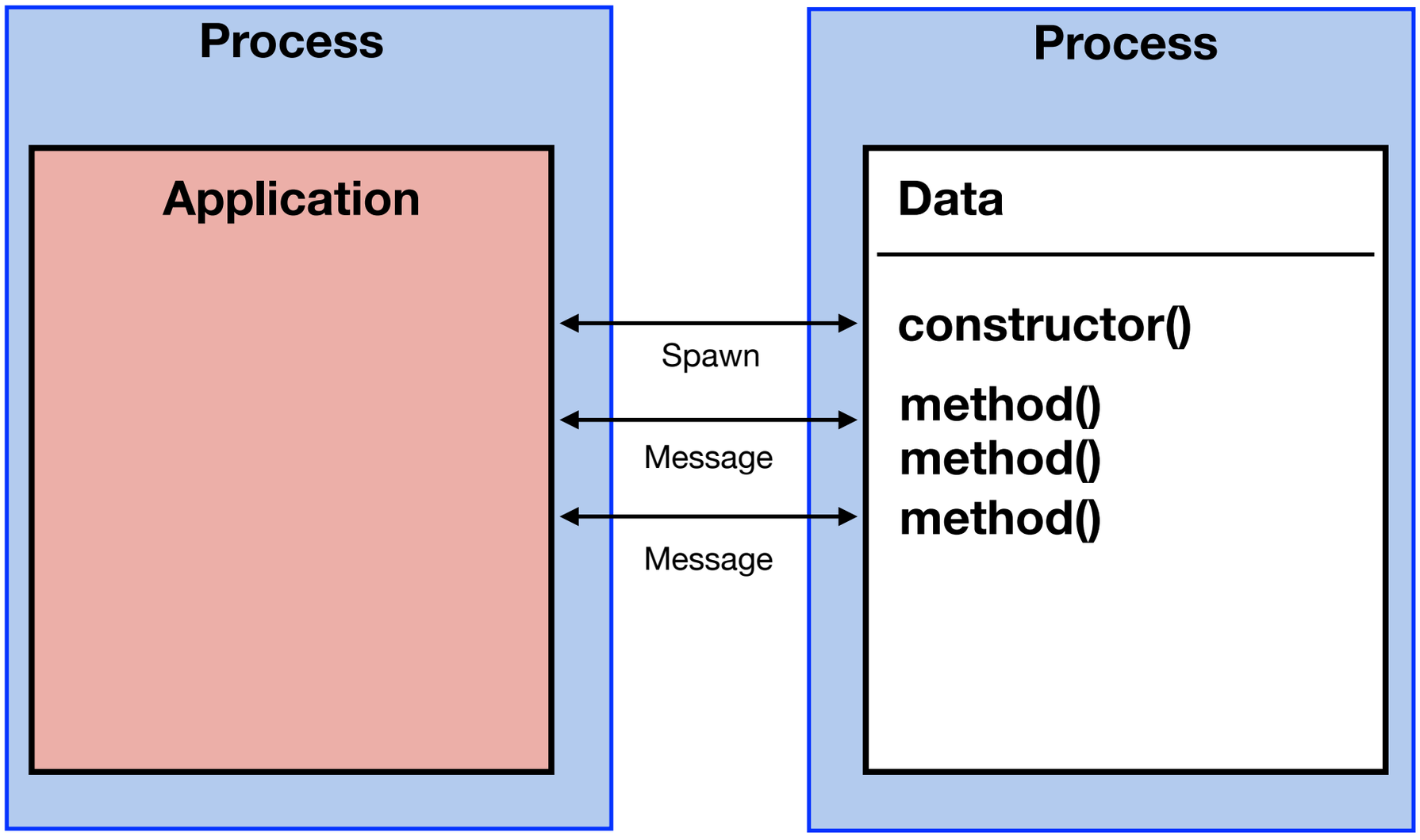
  - The results of method calls become indeterminate.

  - A method call may cause a crash, causing the whole program to crash.

- The program can possibly manipulate the object state without using the objects methods. Same effect as above!

Using Processes

The application and object code run in separate processes, possibly on different machines.



How to implement this in Elixir

- The process containing the object is started using the spawn/3 function.

- Method calls and responses are replaced by messages. send(PID or Name, data)

- The object maintains its internal data in a tail recursive loop.

Starting a process

- spawn(mod, fun, args) starts a new process.

- register(PID, process_name) registers the process with the system so that, in future, it can be referenced by name.

```elixir
def start(name) do
  Process.register(spawn(NameServer1, :loop, [name, %{}]), name)
  :ok
end
```

Messaging

- Message functions (send and receive) are first class members of the language:

```
def rpc(name, request) do
  send(name, {self(), request})

  receive do
    {_name, response} -> response
  end
end
```

```
receive do
  {from, request} ->
    response = handleRequest(request)
    send(from, {self(), response})
end
```

Tail Recursion

- If the last thing a function does is call itself, there's no need to make the call. Instead, the runtime simply jumps back to the start of the function. If the recursive call has arguments, then these replace the original parameters.

```
def loop(name, state) do
  receive do
    {from, request} ->
      {response, newState} = handleRequest(request, state)
      send(from, {name, response})
      loop(name, newState)
  end
end
```

Now for a server.
Putting the pieces together.

I mean I've tried to fit in I've really tried. I even tried learning what Functional Programming was.

```elixir
defmodule NameServer1 do
  @moduledoc """
  Very simple name server supporting two methods:
  add: Add a name and a place.
  find: Given a name, return the place or nil
  """

  def start(name) do
    Process.register(spawn(NameServer1, :loop, [name, %{}]), name)
    :ok
  end

  def rpc(name, request) do
    send(name, {self(), request})

    receive do
      {_name, response} -> response
    end
  end

  def loop(name, state) do
    receive do
      {from, request} ->
        {response, newState} = handleRequest(request, state)
        send(from, {name, response})
        loop(name, newState)
    end
  end

  defp handleRequest({:add, name, place}, state) do
    newState = Map.put(state, name, place)
    {:ok, newState}
  end

  defp handleRequest({:find, name}, state) do
    {state[name], state}
  end
end
```

Constructor.
Spawn a new process, running loop() and register its PID with a name.

Method interface.
Send a message to the server, requesting it to perform an action.

The server.
Preserves the server data (State), and calls the methods and returns the result.

Methods.
The functions that implement the server actions.

NameServer1 Test

- Create the server
- Add a name and place
- Retrieve the place by name

```elixir
defmodule GenServerSlidesTest do
  use ExUnit.Case
  doctest NameServer1

  test "1 - start the server" do
    assert NameServer1.start(:my_server) == :ok
    assert NameServer1.rpc(:my_server, {:add, :dwayne, "Red Dwarf"}) == :ok
    assert NameServer1.rpc(:my_server, {:find, :dwayne}) == "Red Dwarf"
  end
end
```

```elixir
defmodule NameServer1 do
  @moduledoc """
  Very simple name server supporting two methods:
  add: Add a name and a place.
  find: Given a name, return the place or nil
  """

  def start(name) do
    Process.register(spawn(NameServer1, :loop, [name, %{}]), name)
    :ok
  end

  def rpc(name, request) do
    send(name, {self(), request})

    receive do
      {_name, response} -> response
    end
  end

  def loop(name, state) do
    receive do
      {from, request} ->
        {response, newState} = handleRequest(request, state)
        send(from, {name, response})
        loop(name, newState)
    end
  end

  defp handleRequest({:add, name, place}, state) do
    newState = Map.put(state, name, place)
    {:ok, newState}
  end

  defp handleRequest({:find, name}, state) do
    {state[name], state}
  end
end

```

The client and server parts.

Interface

Functions called by the process using the server.

Server

The server.

Implementation

Functions called by the server to implement the server logic.

What happens if a method call crashes?

- The whole server will crash! Not good

Lister: “I’m going to use my brains for the first time in my life.”
Kryten: “Considering the circumstances, sir, do you really believe that’s wise?”

```elixir
defmodule NameServer2 do
  @moduledoc """
  Very simple name server supporting transactions:
  - In the event of a crash, the caller is sent a message.
  - All other processes using the server will not be affected.
  """
  def start(name) do…
  end

  def rpc(name, request) do…
  end

  def loop(name, state) do
    receive do
      {from, request} ->
        try do
          {response, newState} = handleRequest(request, state)
          send(from, {name, response})
          loop(name, newState)
        rescue
          thrown_value ->
            send(from, {:exception, "#{inspect(thrown_value)}"})
            loop(name, state)
        end
    end
  end

  defp handleRequest({:add, name, place}, state) do…
  end

  defp handleRequest({:find, name}, state) do…
  end

  defp handleRequest({:crash, name}, state) do
    1 / 0
    {state[name], state}
  end
end
```

Interface

Functions called by the process using the server.

Server

Runs in the server Process.

Implementation

Functions called by the server process to implement the server logic.

Nameserver2 Test

- Start the server.
- Add data.
- Crash the server.
- Check the data is still there.

```elixir
defmodule NameServer2Test do
  use ExUnit.Case
  doctest NameServer2

  test "2 - crashing server" do
    assert NameServer2.start(:my_server2) == :ok
    assert NameServer2.rpc(:my_server2, {:add, :dwayne, "Red Dwarf"}) == :ok

    assert NameServer2.rpc(:my_server2, {:crash, :dwayne}) ==
             "%ArithmeticError{message: \"bad argument in arithmetic expression\"}"

    assert NameServer2.rpc(:my_server2, {:find, :dwayne}) == "Red Dwarf"
  end
end

```

The method calls are a bit “clunky”

- NameServer2.rpc(:my_server2, {:add, :dwayne, "Red Dwarf”})

- How about:

- NameServer2.add(:my_server2, {:dwayne, "Red Dwarf"})

```elixir
defmodule NameServer3 do
  @moduledoc """
  Very simple name server supporting transactions:
  - In the event of a crash, the caller is sent a message.
  - All other processes using the server will not be affected.
  """
  def start(name) do…
  end

  def add(serverName, name, place) do
    rpc(serverName, {:add, name, place})
  end

  def find(serverName, name) do
    rpc(serverName, {:find, name})
  end

  defp rpc(name, request) do…
  end

  def loop(name, state) do…
  end

  defp handleRequest({:add, name, place}, state) do…
  end

  defp handleRequest({:find, name}, state) do…
  end
end
```

Interface

Server

Implementation

Nameserver3 Test

- Start the server.
- Add and retrieve data using named calls

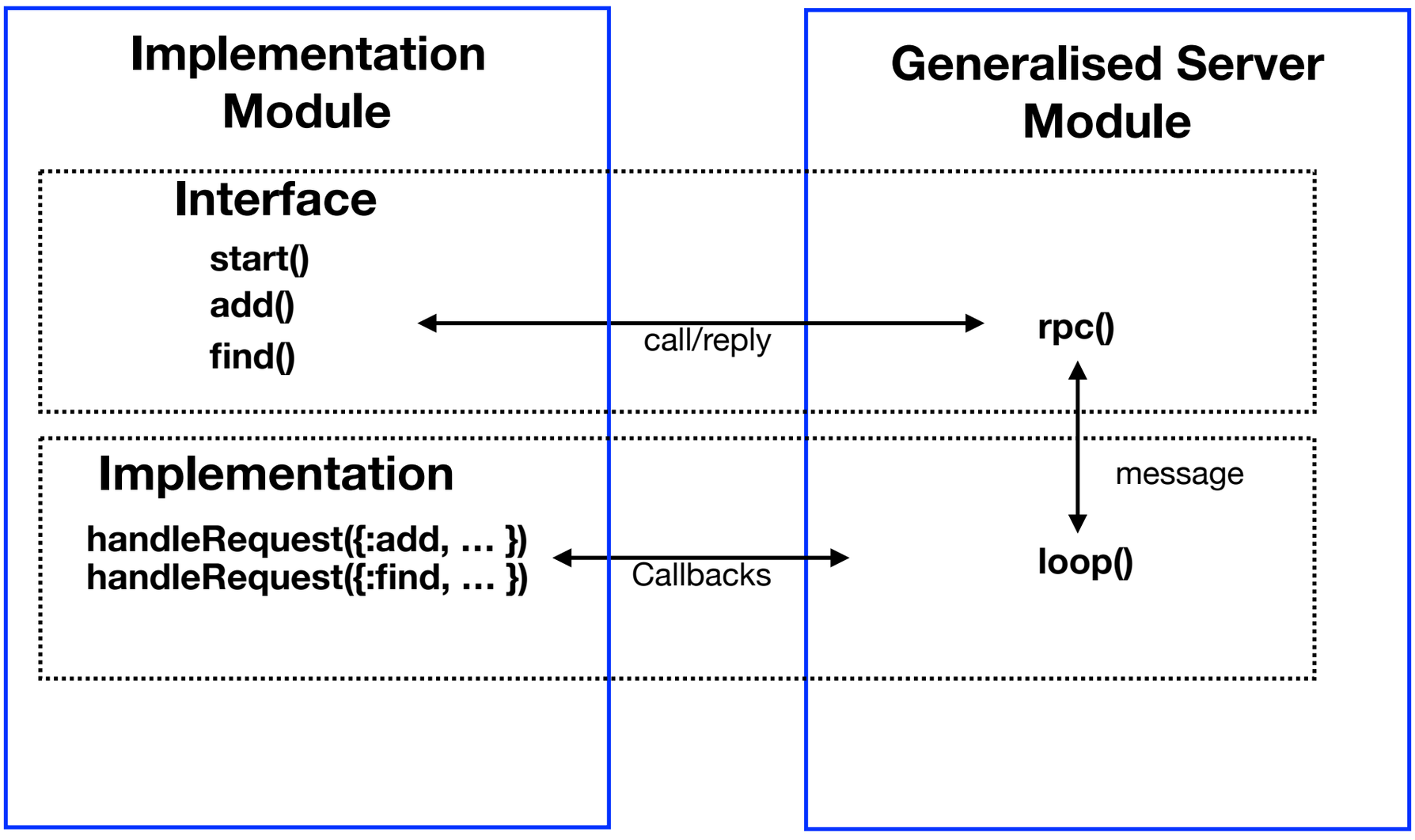
```elixir
defmodule NameServer3Test do
  use ExUnit.Case
  doctest NameServer3

  test "1 - start the server" do
    assert NameServer3.start(:my_server3) == :ok
    assert NameServer3.add(:my_server3, :dwayne, "Red Dwarf") == :ok
    assert NameServer3.find(:my_server3, :dwayne) == "Red Dwarf"
  end
end
```

We are getting close to GenServer!
Code Layout.

Splitting out the general parts of the server.

A module view of things….



The Final Step!
Port the code to use GenServer

```elixir
defmodule NameServer4 do
  @moduledoc """
  Very simple name server supporting transactions, using GenServer:
  """
  use GenServer

  ## Interface ----------------------------------------------------
  # def start(name) do
  #   Process.register(spawn(NameServer3, :loop, [name, %{}]), name)
  #   :ok
  # end
  def start(name) do
    {:ok, pid} = GenServer.start(__MODULE__, :ok, name: name)
    :ok
  end

  # def add(serverName, name, place) do
  #   rpc(serverName, {:add, name, place})
  # end
  def add(serverName, name, place) do
    GenServer.call(serverName, {:add, name, place})
  end

  # def find(serverName, name) do
  #   rpc(serverName, {:find, name})
  # end
  def find(serverName, name) do
    GenServer.call(serverName, {:find, name})
  end

  ## Implementation -----------------------------------------------
  @impl true
  def init(_) do
    {:ok, %{}}
  end

  # defp handleRequest({:add, name, place}, state) do
  #   newState = Map.put(state, name, place)
  #   {:ok, newState}
  # end
  @impl true
  def handle_call({:add, name, place}, _from, state) do
    newState = Map.put(state, name, place)
    {:reply, :ok, newState}
  end

  # defp handleRequest({:find, name}, state) do
  #   {state[name], state}
  # end
  @impl true
  def handle_call({:find, name}, _from, state) do
    {:reply, state[name], state}
  end
end
```

Interface

Implementation

```elixir
defmodule NameServer5 do
  @moduledoc """
  Very simple name server supporting transactions, using GenServer:
  """
  use GenServer

  ## Interface ----------------------------------------------------
  def start(name) do
    {:ok, pid} = GenServer.start(__MODULE__, :ok, name: name)
    :ok
  end

  def add(serverName, name, place) do
    GenServer.cast(serverName, {:add, name, place})
  end

  def find(serverName, name) do
    GenServer.call(serverName, {:find, name})
  end

  ## Implementation -----------------------------------------------
  @impl true
  def init(_) do
    {:ok, %{}}
  end

  @impl true
  def handle_cast({:add, name, place}, _from, state) do
    newState = Map.put(state, name, place)
    {:noreply, newState}
  end

  @impl true
  def handle_call({:find, name}, _from, state) do
    {:reply, state[name], state}
  end
end
```

Interface

Registered name of the server, cleared on exit.

Term passed to init() callback

Callback module (This module)

Implementation

Cast: does not send a reply. The calling process does not have to wait.

Call: does send a reply.

Nameserver5 Test

- Nothing Changed using GenServer

```elixir
defmodule NameServer3Test do
  use ExUnit.Case
  doctest NameServer3

  test "1 - start the server" do
    assert NameServer3.start(:my_server3) == :ok
    assert NameServer3.add(:my_server3, :dwayne, "Red Dwarf") == :ok
    assert NameServer3.find(:my_server3, :dwayne) == "Red Dwarf"
  end
end
```

```elixir
defmodule NameServer5Test do
  use ExUnit.Case
  doctest NameServer5

  test "1 - start the server" do
    assert NameServer5.start(:my_server5) == :ok
    assert NameServer5.add(:my_server5, :dwayne, "Red Dwarf") == :ok
    assert NameServer5.find(:my_server5, :dwayne) == "Red Dwarf"
  end
end
```

But how do I stop the server?

- The server will be stopped by:
    - Calling GenServer.stop() in the interface section.
    - A callback returning {:stop, how, state} (call or cast function).
    - A callback raising an exception.
    - A callback returning an invalid value.
- The function terminate() will be called (if present), allowing any shutdown actions to be performed.
- The server name will be de-registered.

```elixir
defmodule NameServer6 do
  @moduledoc """
  Two ways to stop the server
  """
  use GenServer

  def start(name) do…
  end

  def add(serverName, name, place) do…
  end

  def find(serverName, name) do…
  end

  def shutdown(serverName) do
    GenServer.cast(serverName, :shutdown)
  end

  def shutdown1(serverName) do
    GenServer.stop(serverName, :normal)
  end

  @impl true
  def init(_) do…
  end

  @impl true
  def terminate(_reason, _state) do
    # Cleanup code
  end

  @impl true
  def handle_cast({:add, name, place}, state) do…
  end

  def handle_cast(:shutdown, state) do
    # do something
    {:stop, :normal, state}
  end

  @impl true
  def handle_call({:find, name}, _from, state) do…
  end
end
```

Interface

Implementation

Called when the server is terminating because of:

- Callback returns {:stop, reason, … }
- Callback raises an exception.
- Callback returns an invalid value.
- GenServer.stop(…) is called.
- …

GenServer Callbacks

- **init(args) (interface - start(), start_link())**
  - {:ok, state *[,timeout]*}
  - {:stop, reason}
- **handle_call(request, from, state) (interface - call())**
  - {:reply, reply, new_state *[, timeout]*}
  - {:stop, reason *[,reply]*, new_state}
- **handle_cast(request, state) (interface - cast())**
  - {:noreply, new_state *[, timeout]*}
  - {:stop, reason, new_state}
- **handle_info(msg, state) (async - message or timeout)**
  - msg = term or :timeout
  - {:noreply, new_state *[,timeout]*}
  - {:stop, reason, new_state}
- **terminate(reason, state) (Callback return = {:stop …..} or interface stop call)**

Oh No! Not Dwayne Dibley

Further Information

- Examples: https://github.com/DwayneDibley
- Docs: https://hexdocs.pm/elixir/GenServer.html
- Docs: http://erlang.org/doc/man/gen_server.html
- Programming Erlang (Joe Armstrong)
- https://www.youtube.com/watch?v=QBIWMgPZo2Q