BEHAVIOURS

Behaviours

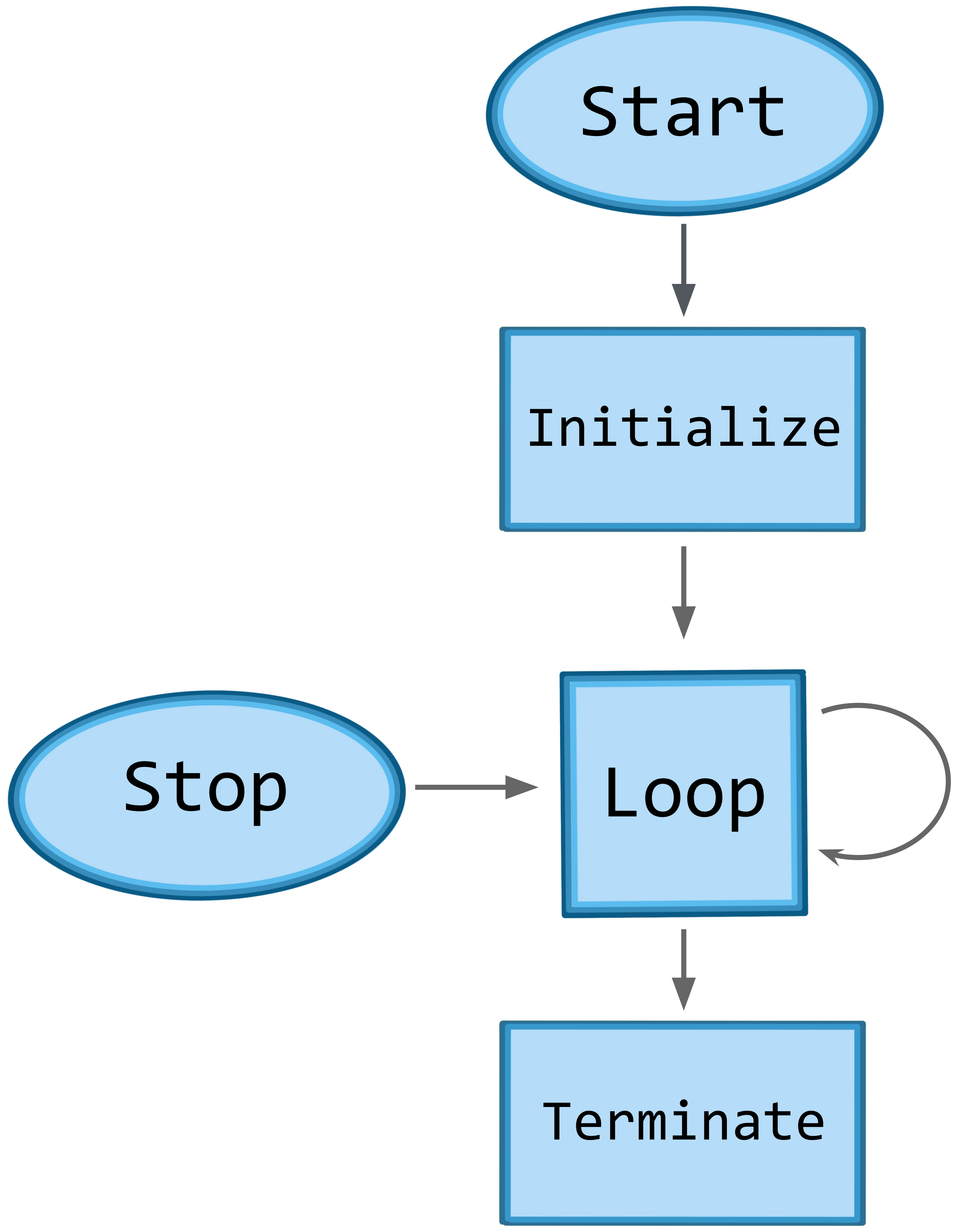
- Design Principles
- Behaviours
- A Server Example

Design Patterns

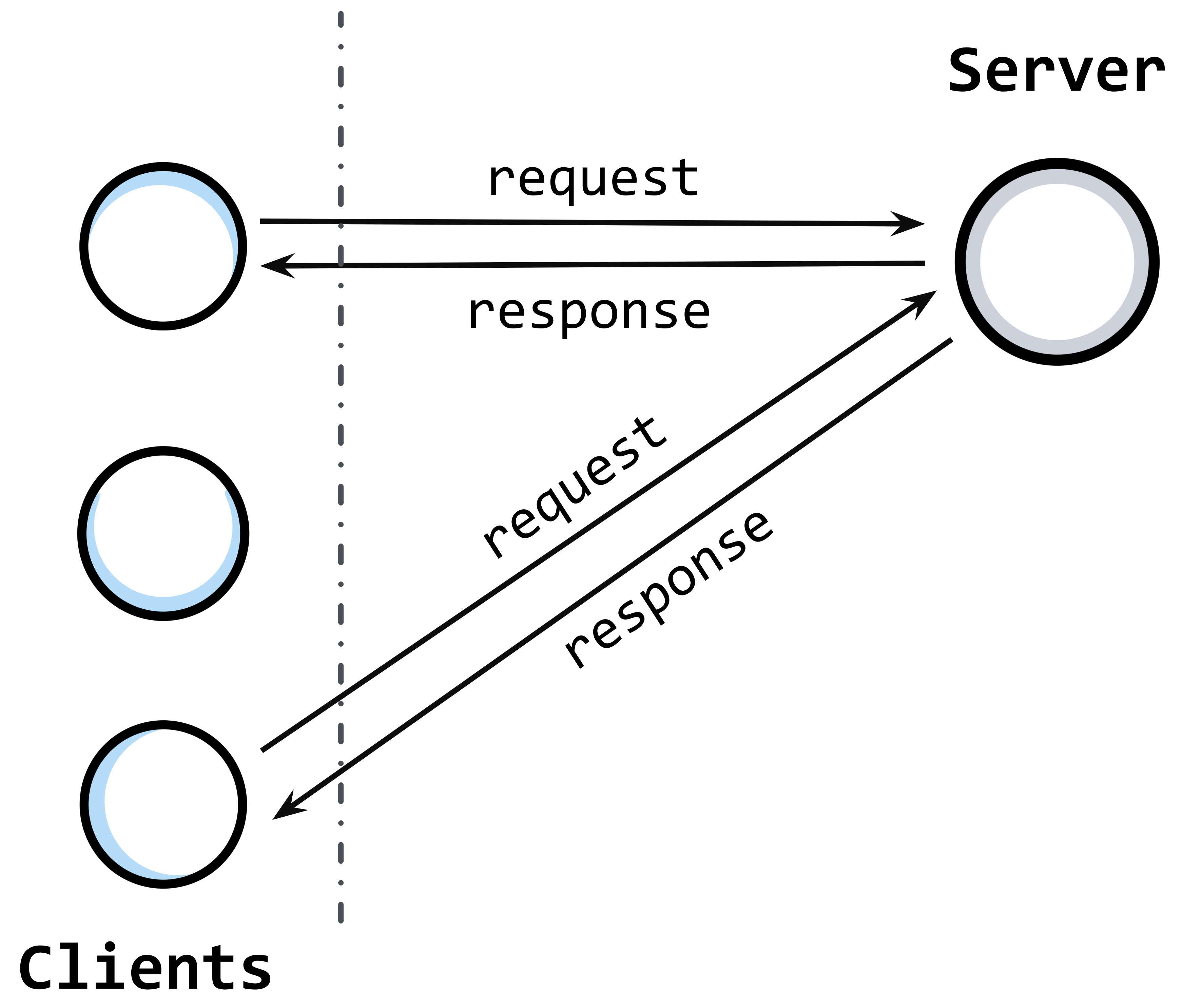
- ▶ OTP Behaviours are a formalisation of design patterns
- ▶ Processes share similar structures and life cycles
  - They are started
  - They receive messages and send replies
  - They are terminated (or crash)
- ▶ Even if they perform different tasks, they will perform them following a set of patterns
- ▶ Each design pattern solves a specific problem

More on Processes: process skeleton

```
start(Args) ->
   spawn(server, init, [Args])

init(Args) ->
  State = initialize_state(Args),
  loop(State).

loop(State) ->
   receive
      {handle, Msg} ->
         NewState = handle(Msg, State),
         loop(NewState);
      stop -> terminate(State)
   end.

terminate(State) -> clean_up(State).
```



Design Patterns



- ➢ Take a Client-Server architecture
- ➢ What behaviours will differ between systems?
- ➢ What similarities will there be between systems?

Design Patterns

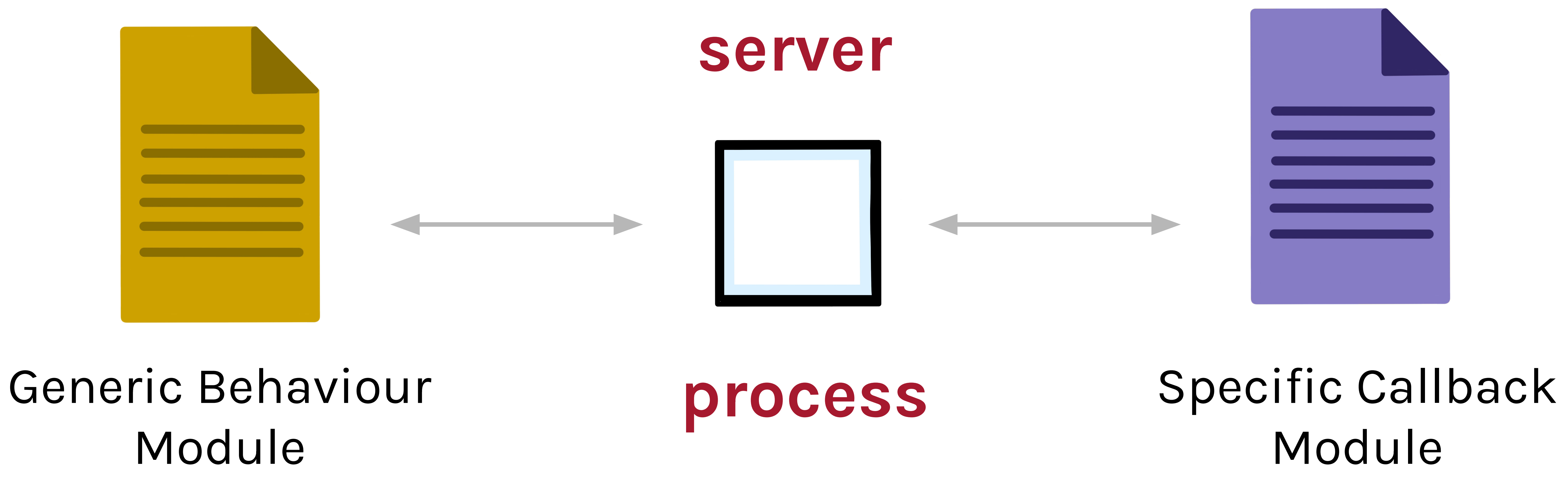
Generic

- Spawning the server
- Storing the loop data
- Sending requests to the server
- Sending replies to the client
- Receiving server replies
- Stopping the server

Specific

- Initialising the server state
- The loop data
- The client requests
- Handling client requests
- Contents of server reply
- Cleaning up

Behaviours



- The idea is to split the code in two parts
- The generic part is called the **generic behaviour**
  - They are provided by OTP as library modules
- The specific part is called the **callback module**
  - They are implemented by the programmer

Behaviours

Generic Servers

Used to model client-server behaviours

Generic State Machines

Used for state machine programming

Generic Event Handler/Manager

Used for writing event handlers

Supervisors

Used for fault-tolerant supervision trees

Application

Used to encapsulate resources and functionality

Behaviours

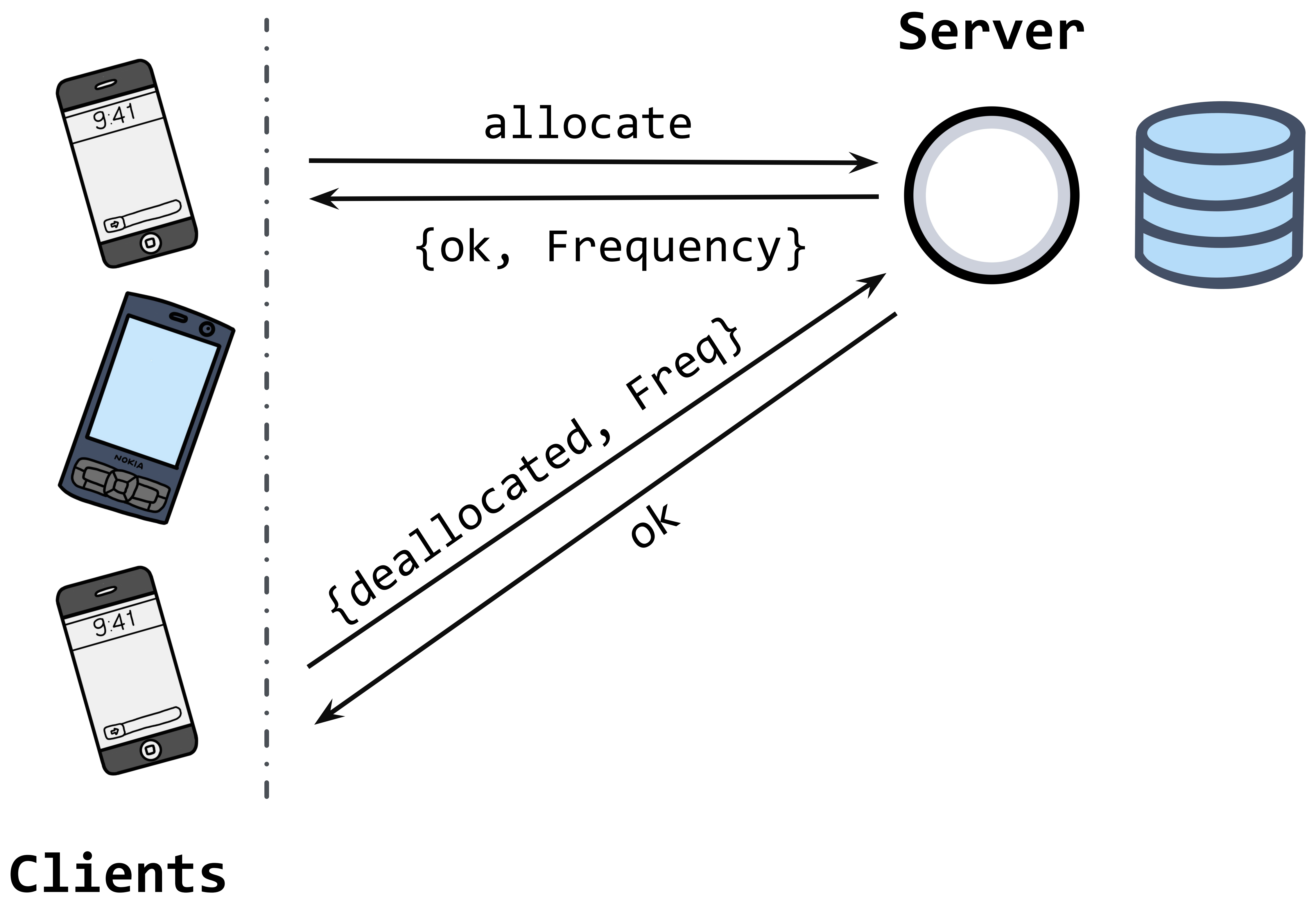
Pros

- ▶ Less code to develop
- ▶ Less bugs
- ▶ Solid well tested base
- ▶ Free built-in functionality
  - Log, Trace, Statistics, Extensible
- ▶ Common Programming Style
- ▶ Component-based terminology

Cons

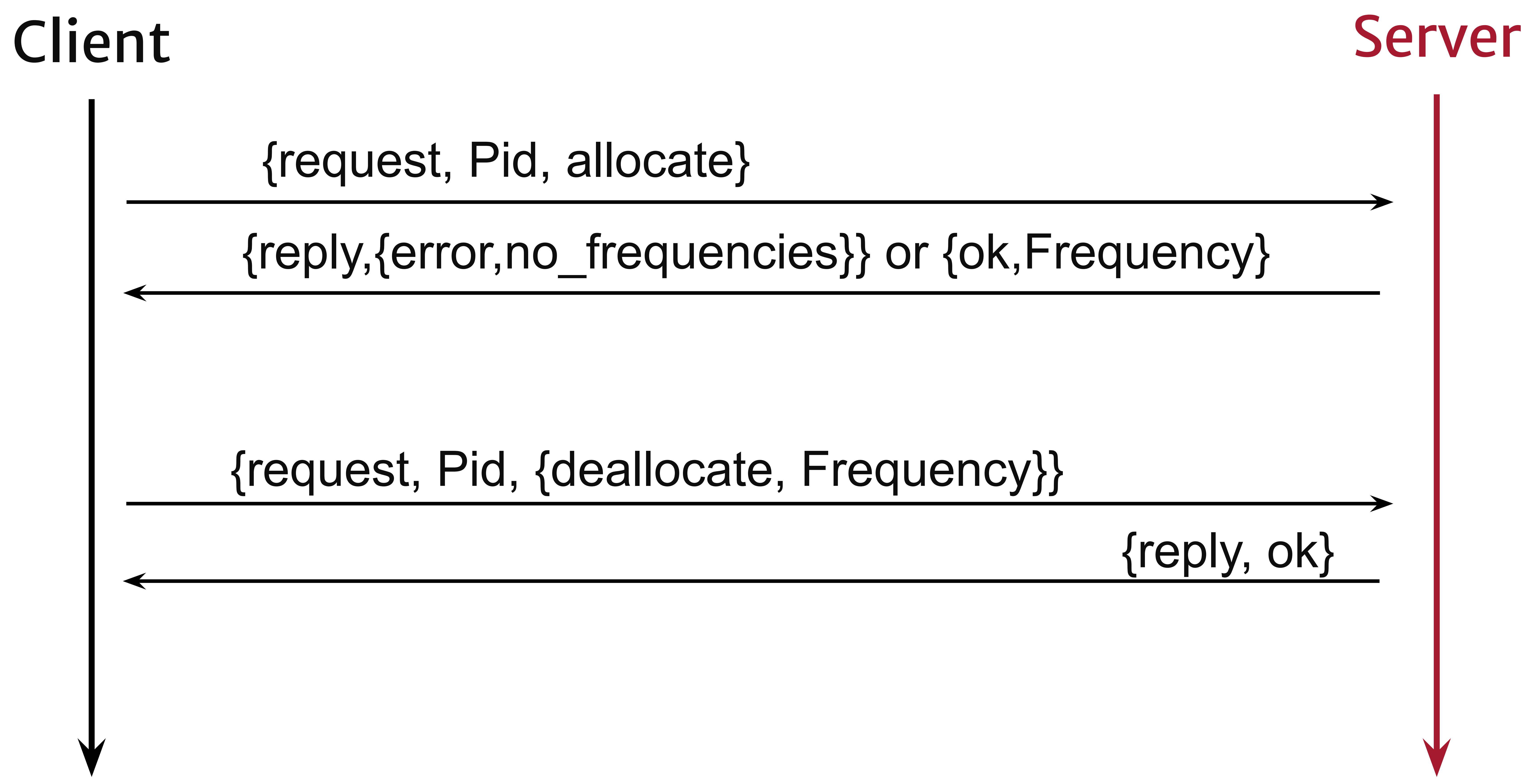
- ▶ Steep learning curve
- ▶ Affects performance

A Server Example



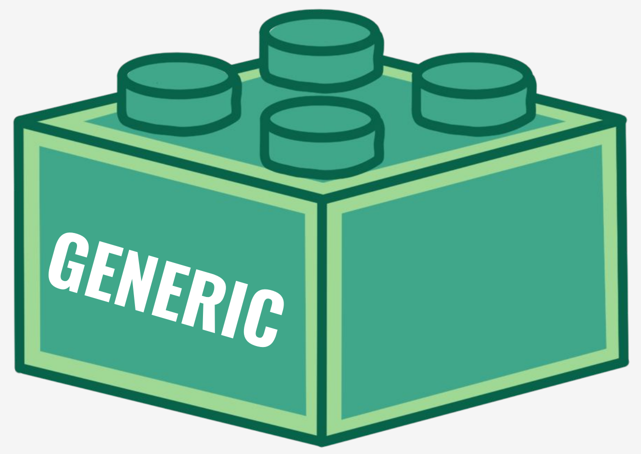
The following server is responsible for allocating and deallocating frequencies on behalf of mobile phones

A Server Example


Client
Server
{request, Pid, allocate}
{reply,{error,no_frequencies}} or {ok,Frequency}
{request, Pid, {deallocate, Frequency}}
{reply, ok}

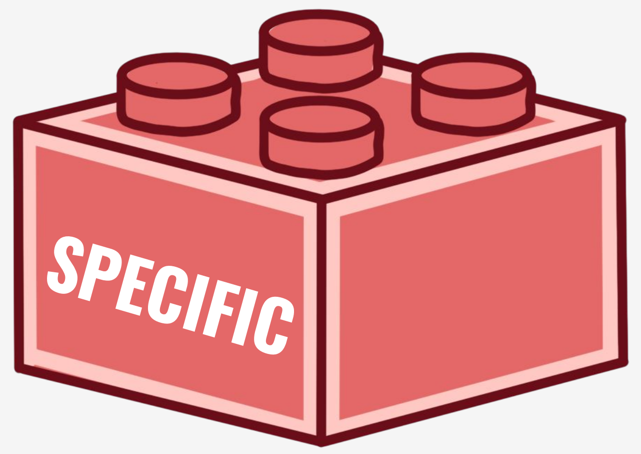
A Server Example

```
-module(frequency).
-export([start/0, stop/0, allocate/0, deallocate/1]).
-export([init/0]).

start() ->
    register(frequency, spawn(frequency, init, [])).

init() ->
    Frequencies = {get_frequencies(), []},
    loop(Frequencies).

get_frequencies() -> [10,11,12,13,14,15].
```



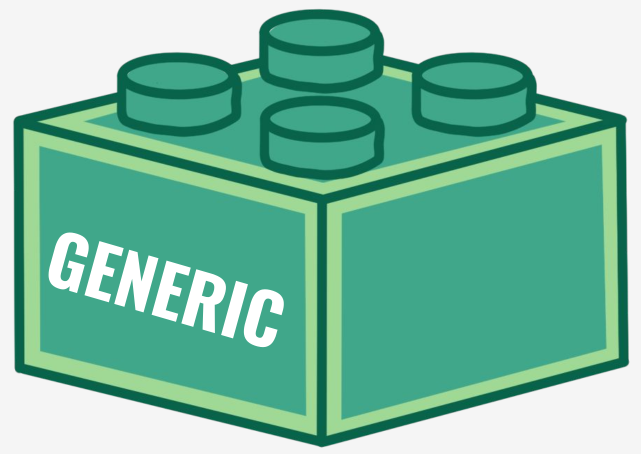
A Server Example

```
-module(frequency).
-export([start/0, stop/0, allocate/0, deallocate/1]).
-export([init/0]).

start() ->
    register(frequency, spawn(frequency, init, [])).

init() ->
    Frequencies = {get_frequencies(), []},
    loop(Frequencies).

get_frequencies() -> [10,11,12,13,14,15].
```



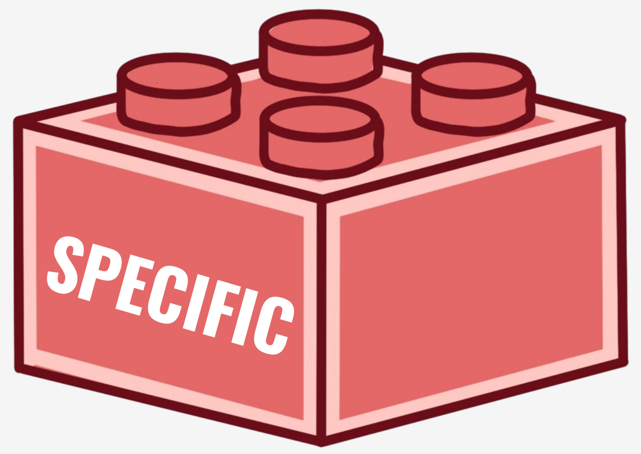
A Server Example

```
%%  The client Functions
stop()           -> call(stop).
allocate()       -> call(allocate).
deallocate(Freq) -> call({deallocate, Freq}).

%% We hide all message passing and the message protocol in
%% a functional interface.

call(Message) ->
    frequency ! {request, self(), Message},
    receive
       {reply, Reply} -> Reply
    end.
reply(Pid, Message) ->
  Pid ! {reply, Message}.
```



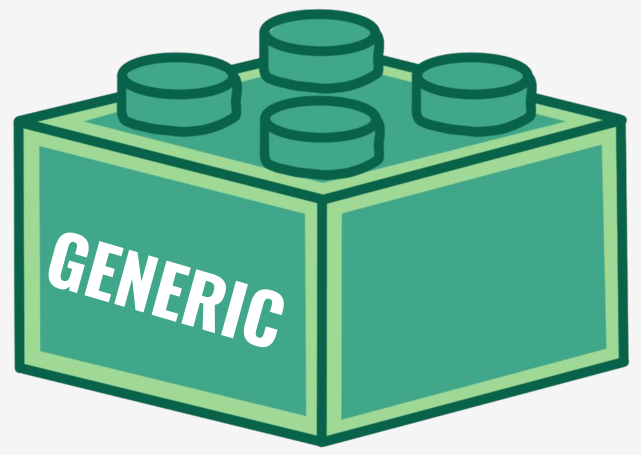
A Server Example

```
%%  The client Functions
stop()           -> call(stop).
allocate()       -> call(allocate).
deallocate(Freq) -> call({deallocate, Freq}).

%% We hide all message passing and the message protocol in
%% a functional interface.

call(Message) ->
    frequency ! {request, self(), Message},
    receive
       {reply, Reply} -> Reply
    end.
reply(Pid, Message) ->
  Pid ! {reply, Message}.
```



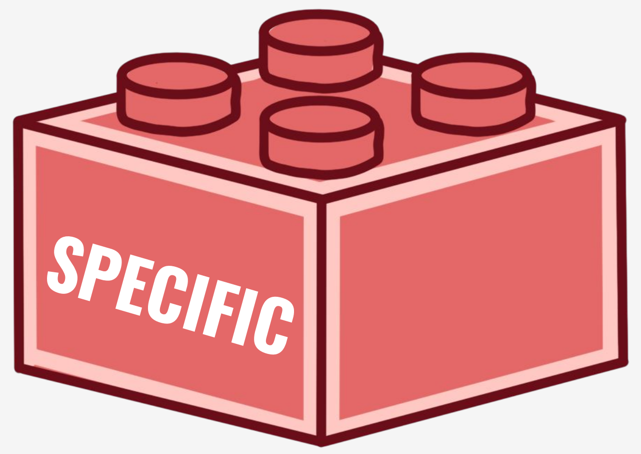
A Server Example

```
loop(Frequencies) ->
  receive
   {request, Pid, allocate} ->
     {NewFrequencies, Reply} = allocate(Frequencies, Pid),
     reply(Pid, Reply),
     loop(NewFrequencies);
   {request, Pid , {deallocate, Freq}} ->
      NewFrequencies = deallocate(Frequencies, Freq),
      reply(Pid, ok),
      loop(NewFrequencies);
   {request, Pid, stop} ->
      reply(Pid, ok)
  end.
```



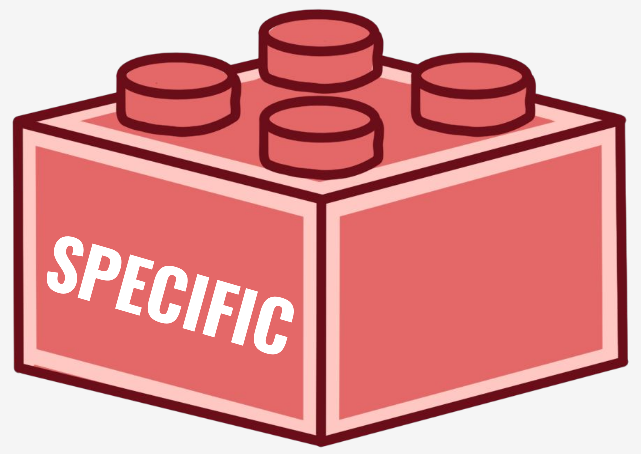
A Server Example

```
loop(Frequencies) ->
  receive
   {request, Pid, allocate} ->
     {NewFrequencies, Reply} = allocate(Frequencies, Pid),
     reply(Pid, Reply),
     loop(NewFrequencies);
   {request, Pid , {deallocate, Freq}} ->
      NewFrequencies = deallocate(Frequencies, Freq),
      reply(Pid, ok),
      loop(NewFrequencies);
   {request, Pid, stop} ->
      reply(Pid, ok)
  end.
```

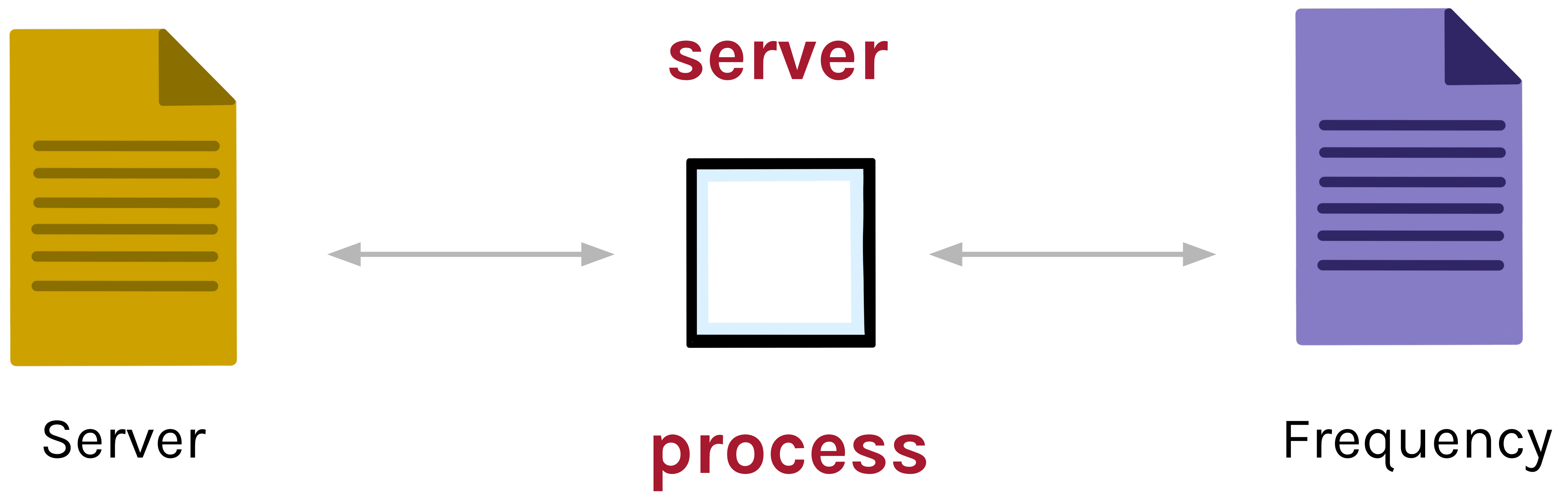


A Server Example

```erlang
%% The Internal Functions
%% Help functions used to allocate and deallocate frequencies.

allocate({[], Allocated}, Pid) ->
    {{[], Allocated}, {error, no_frequencies}};
allocate({[Freq|Free], Allocated}, Pid) ->
    {{Free, [{Freq, Pid}|Allocated]}, {ok, Freq}}.

deallocate({Free, Allocated}, Freq) ->
    NewAllocated = lists:keydelete(Freq, 1, Allocated),
    {[Freq|Free],  NewAllocated}.
```



Behaviours



- ➢ Place all the generic code in the **server** module
- ➢ Place all the specific code in the **frequency** module

A Server Example

Generic

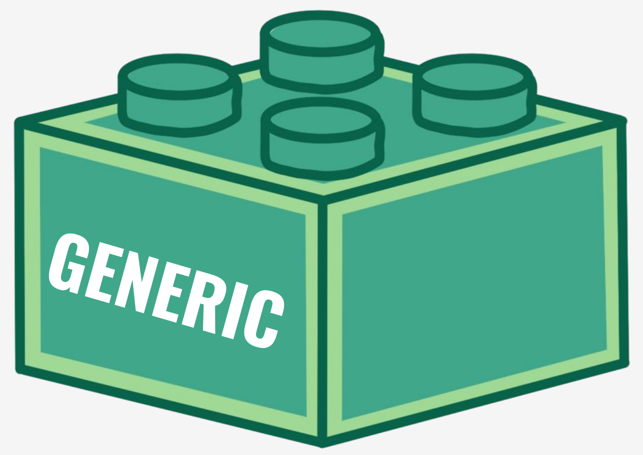
```
-module(server).
-export([start/2, stop/1,
         call/2, init/2]).

start(Mod, Args) ->
    Pid = spawn(server,init,
                [Mod,Args]),
    register(Mod, Pid).

stop(Mod) -> Mod ! stop.

init(Mod, Args) ->
    State = Mod:init(Args),
    loop(Mod, State).
```

Specific

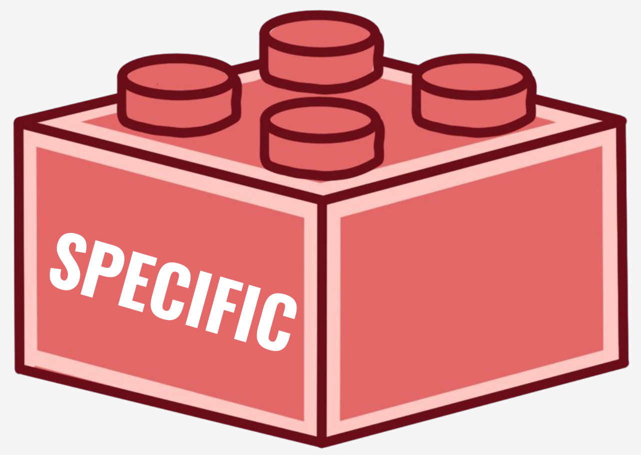
```
-module(frequency).
-export([start/0,stop/0,
         init/1,handle/2,
         allocate/0,deallocate/1]).

start() ->
    server:start(frequency,[]).
stop() ->
    server:stop(frequency).

init(_Args) ->
    {get_frequencies(), []}.
get_frequencies() ->
    [10,11,12,13,14,15].
```

A Server Example

```
call(Pid, Message) ->
    Pid ! {request, self(), Message},
    receive
        {reply, Reply} -> Reply
    end.

reply(Pid, Message) ->
    Pid ! {reply, Message}.
```

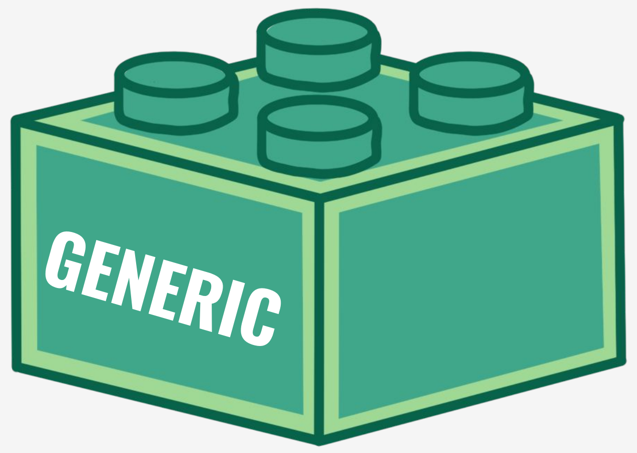


```
allocate()->
    server:call(frequency, {allocate, self()}).

deallocate(Freq) ->
    server:call(frequency, {deallocate, Freq}).
```

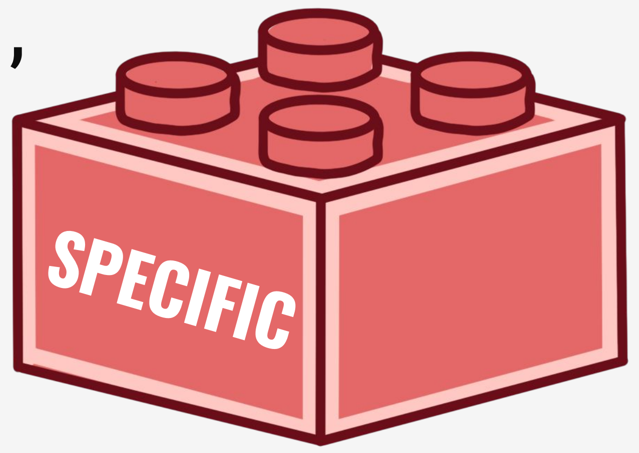


A Server Example

```
loop(Mod, State) ->
    receive
        {request, Pid, Msg} ->
            {NewState, Reply} = Mod:handle(Msg, State),
            reply(Pid, Reply),
            loop(Mod, NewState);
        stop -> ok
    end.
```



```
handle({allocate, Pid}, Frequencies) ->
    {NewFrequencies, Reply} = allocate(Frequencies, Pid),
    {NewFrequencies, Reply};
handle({deallocate, Freq}, Frequencies) ->
    Reply = ok,
    {deallocate(Frequencies, Freq), Reply}.
```



Behaviours

- Design Principles
- Behaviours
- A Server Example