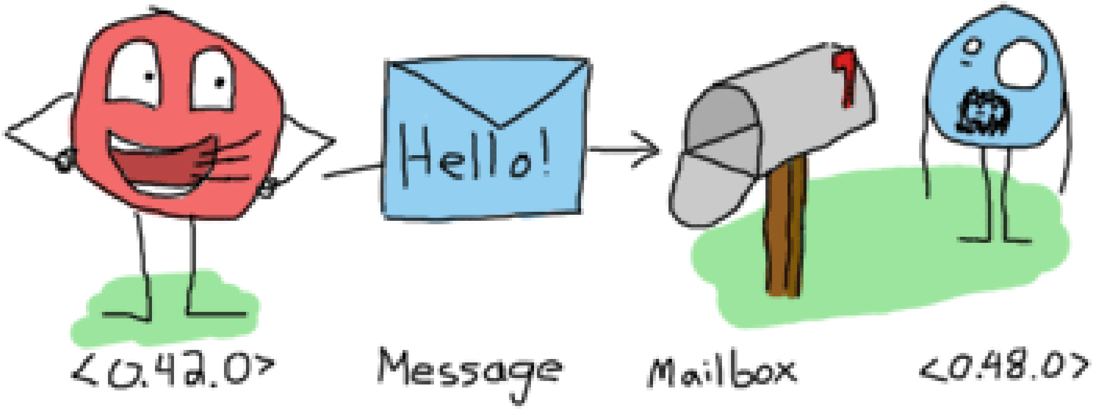
IMPLEMENTAREA CONCURENTEI IN LIMBAJE DE PROGRAMARE

Ioana Leustean



http://learnyousomeerlang.com/the-hitchhikers-guide-to-concurrency#dont-panic

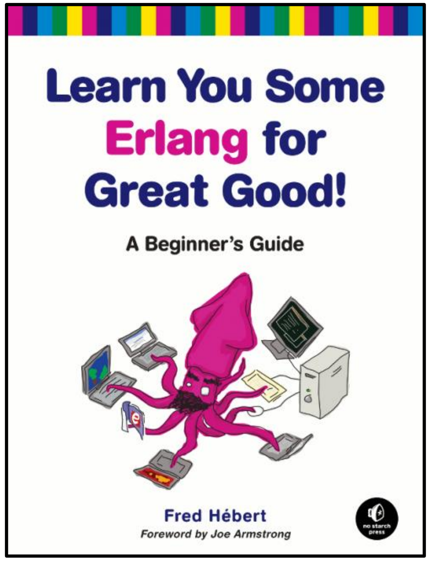
ACTOR MODEL

"Erlang's actor model can be imagined as a world where everyone is sitting alone in their own room and can perform a few distinct tasks. Everyone communicates strictly by writing letters and that's it. While it sounds like a boring life (and a new age for the postal service), it means you can ask many people to perform very specific tasks for you, and none of them will ever do something wrong or make mistakes which will have repercussions on the work of others; they may not even know the existence of people other than you (and that's great).

To escape this analogy, Erlang forces you to write actors (processes) that will share no information with other bits of code unless they pass messages to each other. Every communication is explicit, traceable and safe."

Fred Hébert, Learn You Some Erlang For Great Good

http://learnyousomeerlang.com/introduction#what-is-erlang



Learn You Some Erlang for Great Good!

A Beginner's Guide

Fred Hébert

Foreword by Joe Armstrong

Varianta online

- Concurenta in Erlang este implementata folosind urmatoarele primitive:

```
Pid = spawn (fun)

Pid = spawn (module, fct, args)

Pid ! Message

receive … end
receive … after … end
```

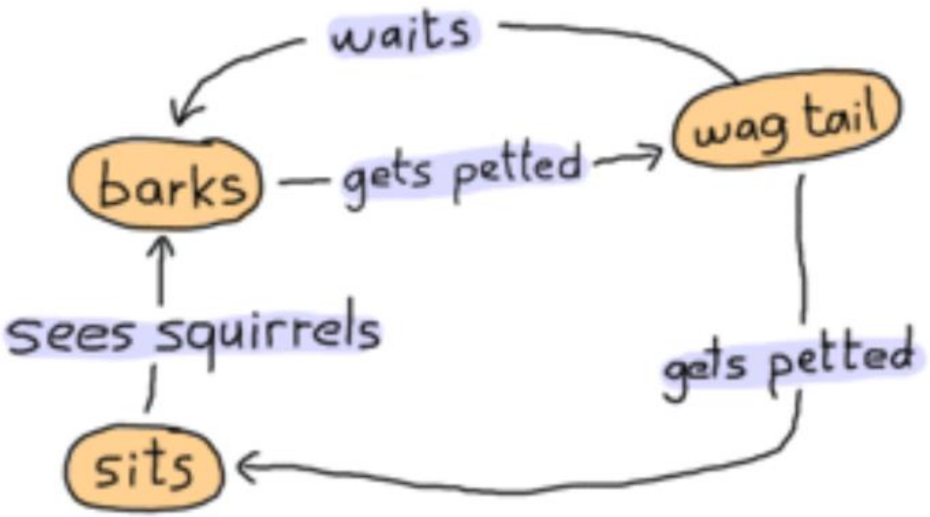
➢ Cilent-Server (simple) template

```
-module(servtemplate1).
-compile(export_all).

start_server() ->  spawn(?MODULE, server_loop, []).

 client(Pid, Request) ->
        Pid ! {self(), Request},
        receive
               {Pid, Response} -> Response
         end.

server_loop() ->
   receive
       .....
       {From, Request} -> From ! {self(),Response} ,
                                      server_loop()


       end.
```

- trimiterea mesajelor se face asincron
- **call(Pid, Request)** apel sincron: mesajul este trimis asincron dar procesul este blocat pana primeste raspunsul
- **cast(Pid, Request)** apel asincron

```
cast(Pid, Request) ->
        Pid ! {self(), Request},
        ok.
```

➢Finite-State Machine



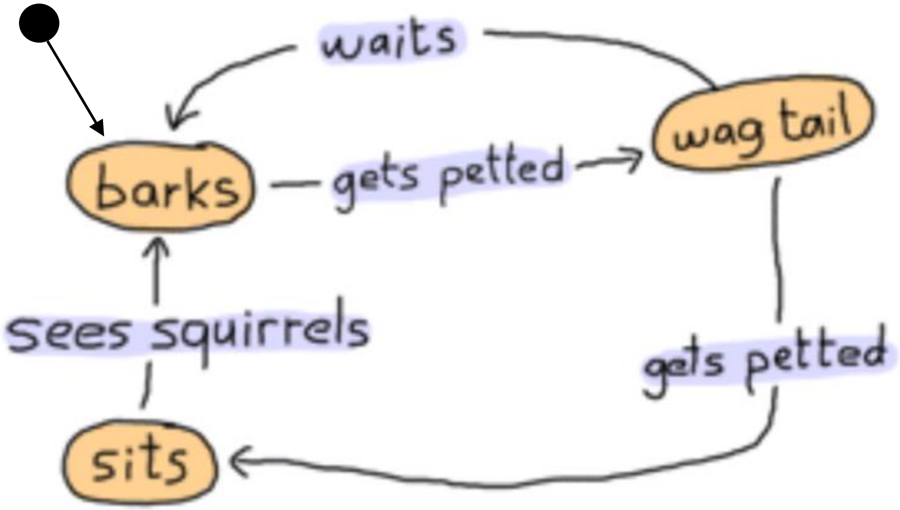
Starile ={barks, sits, wag_tail}
Actiunile ={gets_petted, see_squirrels, waits}

dog as a state-machine
http://learnyousomeerlang.com/finite-state-machines#what-are-they

➢Finite-State Machine

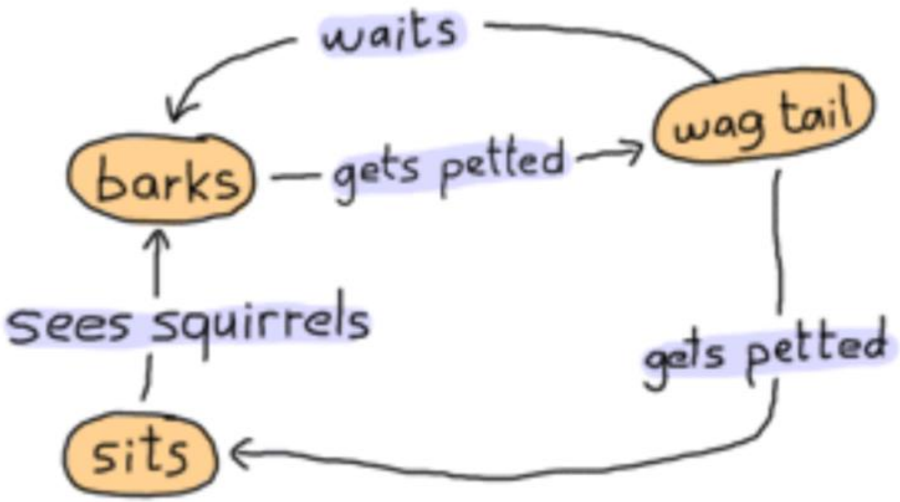
actiunile sunt implementate prin mesaje si sunt vizibile in exterior

```
-module(dog_fsm).
-export([start/0, squirrel/1, pet/1]).

start() ->
    spawn(fun() -> bark() end).     % starea initiala

%actiunea  see_squirrels
squirrel(Pid) -> Pid ! squirrel.

%actiunea  gets_petted
pet(Pid) -> Pid ! pet.
```



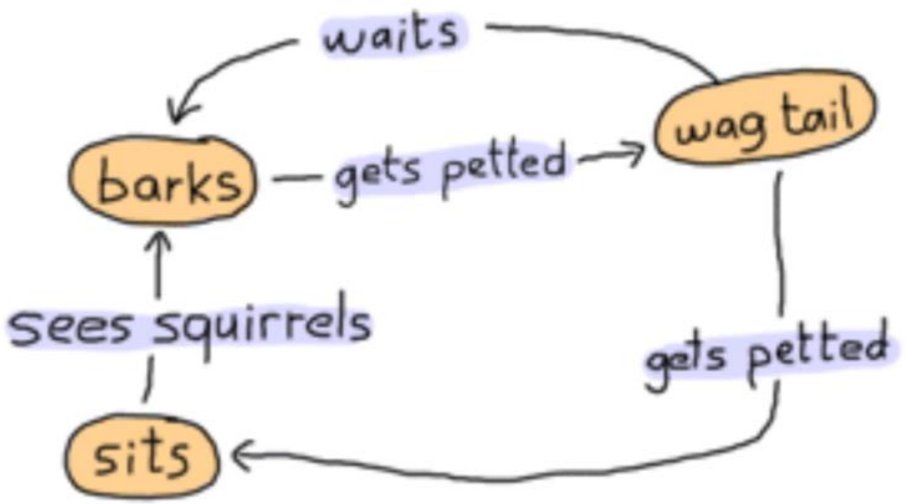
➢Finite-StateMachine: implementarea starilor

```
bark() ->
    io:format("Dog says: BARK! BARK!~n"),
   receive
     pet ->
             wag_tail();
       _ ->
            io:format("Dog is confused~n"),
            bark()
  after 2000 ->
              bark()
end.
```



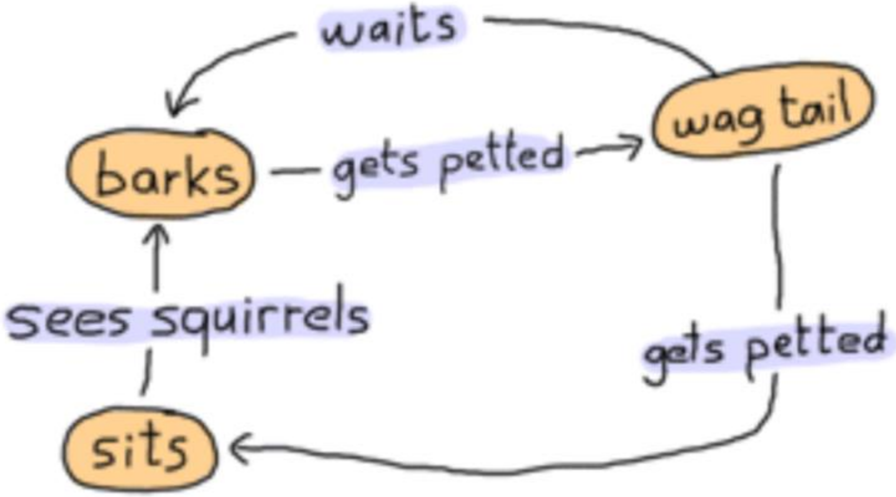
➢Finite-StateMachine: implementarea starilor

```
wag_tail() ->
        io:format("Dog wags its tail~n"),
        receive
              pet ->
                          sit();
               _ ->
                     io:format("Dog is confused~n"),
                     wag_tail()
        after 30000 ->
                          bark()      % actiunea waits

       end.
```

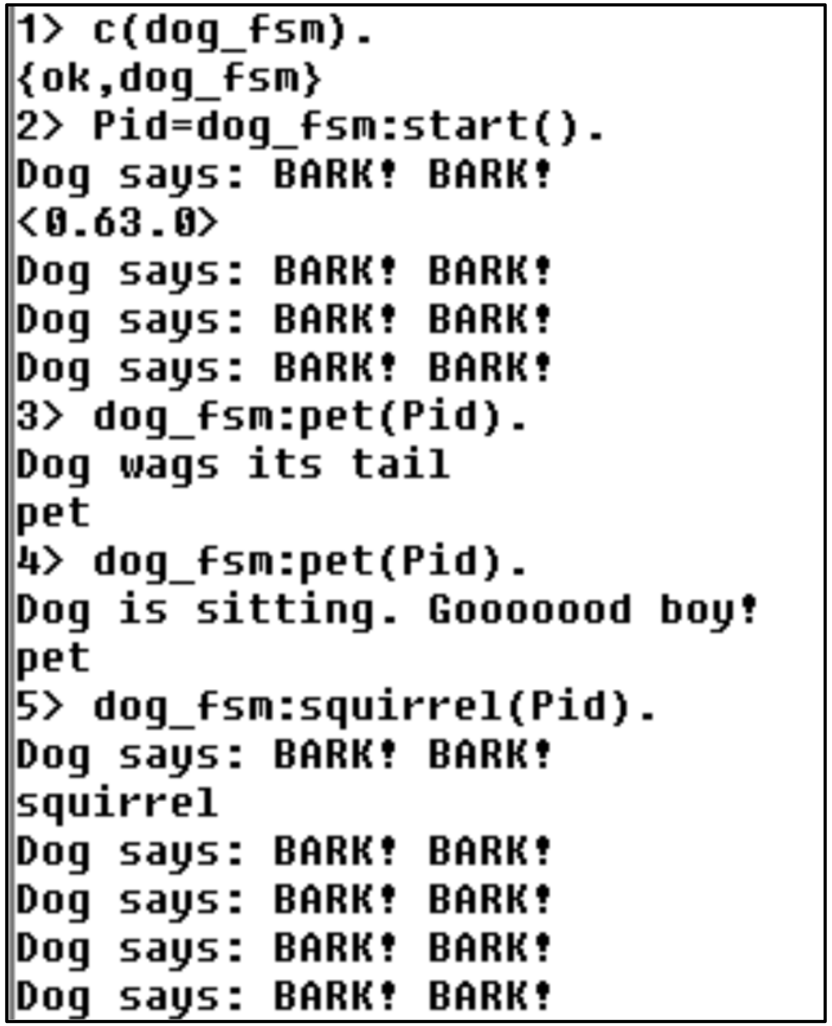


Finite-StateMachine: implementarea starilor

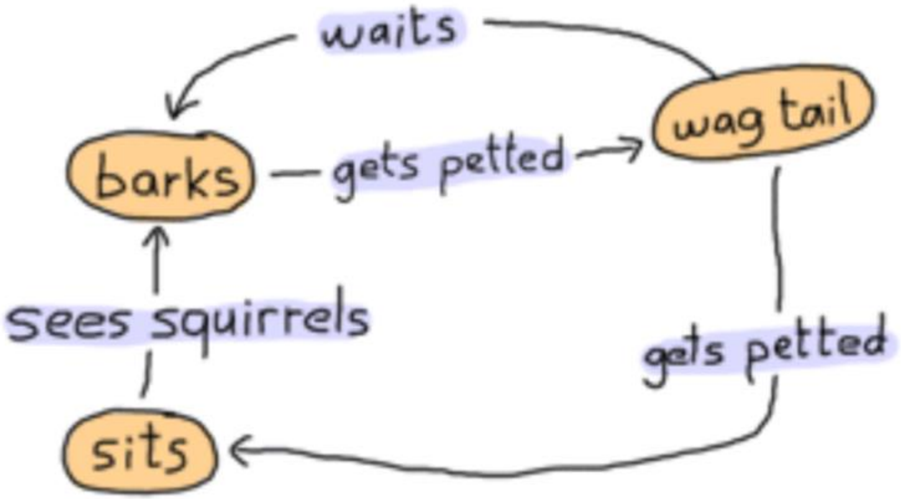
```
sit() ->
    io:format("Dog is sitting. Gooooood boy!~n"),
    receive
        squirrel ->
            bark();
        _ ->
            io:format("Dog is confused~n"),
            sit()
    end.
```



➢Finite-State Machines



```
1> c(dog_fsm).
{ok,dog_fsm}
2> Pid=dog_fsm:start().
Dog says: BARK! BARK!
<0.63.0>
Dog says: BARK! BARK!
Dog says: BARK! BARK!
Dog says: BARK! BARK!
3> dog_fsm:pet(Pid).
Dog wags its tail
pet
4> dog_fsm:pet(Pid).
Dog is sitting. Gooooood boy!
pet
5> dog_fsm:squirrel(Pid).
Dog says: BARK! BARK!
squirrel
Dog says: BARK! BARK!
Dog says: BARK! BARK!
Dog says: BARK! BARK!
Dog says: BARK! BARK!
```



http://learnyousomeerlang.com/finite-state-machines#what-are-they

OTP

> OTP stands for Open Telecom Platform, although it's not that much about telecom anymore (it's more about software that has the property of telecom applications, but yeah.) If half of Erlang's greatness comes from its concurrency and distribution and the other half comes from its error handling capabilities, then the OTP framework is the third half of it.
> http://learnyousomeerlang.com/what-is-otp#its-the-open-telecom-platform

OTP components:

- Supervision trees
- Behaviours
  - gen_server
  - gen_fsm
  - supervisor
- Applications
  - Mnesia(database)
  - Debugger

http://erlang.org/doc/design_principles/des_princ.html