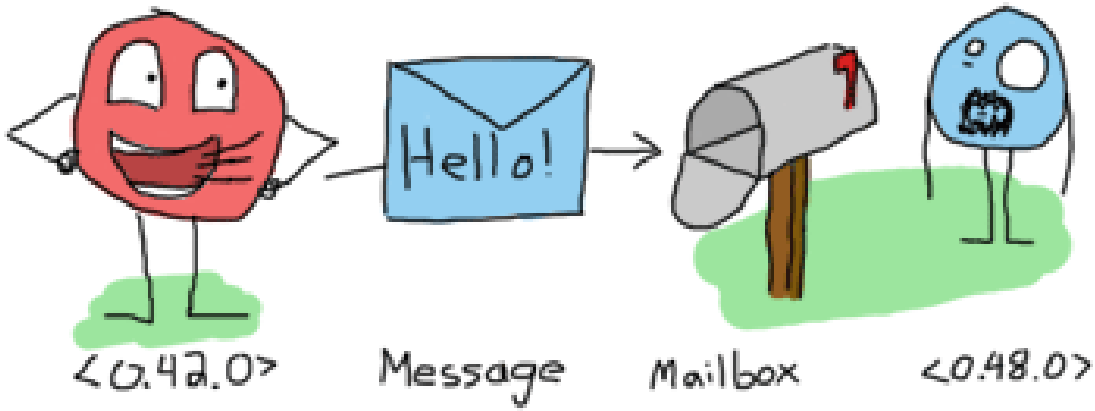
IMPLEMENTAREA CONCURENTEI IN LIMBAJE DE PROGRAMARE

Ioana Leustean



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

➢Bibliografie

Joe Armstrong, Programming Erlang, Second Edition 2013

Fred Hébert, Learn You Some Erlang For Great Good, 2013
Varianta online

Jim Larson, Erlang for Concurrent Programming, ACM Queue, 2008

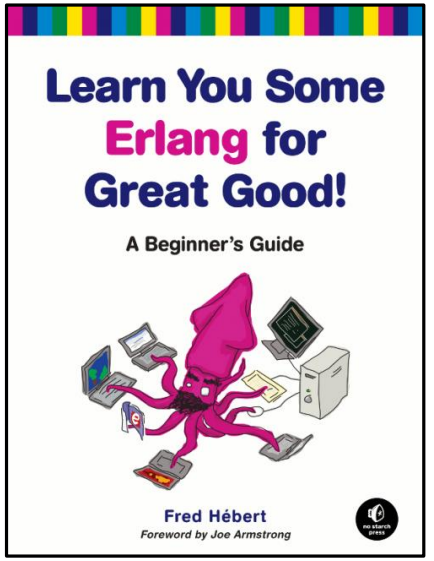
ACTOR MODEL

"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



Varianta online

➢ Concurenta in Erlang este implementata folosind urmatoarele primitive:

```
Pid = spawn (fun)

Pid = spawn (module, fct, args)

Pid ! Message

receive … end
```

➢ receive … end

```
receive
Pattern1 when Guard1 -> Expr1;
Pattern2 when Guard2 -> Expr2;
Pattern3 -> Expr3
end
```

Cand ajunge la o instructiune receive un proces scoate un mesaj din mailbox si incearca sa ii gaseasca un pattern.

Daca nu gaseste un mesaj in mailbox procesul se blocheaza si asteapta un mesaj care se potriveste cu un pattern.

receive este singura instructiune care blocheaza procesul!

➢Trimiterea mesajelor: **Pid ! msg**

Mesajul **msg** este trimis procesului cu id-ul **Pid.** Mesajul este un termen Erlang.

```
myrec() ->
  receive
   {do_A, X} ->  prelA(X);
   {do_B, X} ->  prelB(X);
              _ -> io:format("Nothing to do ~n")
  end.
```

```
2> c(myconc).
{ok,myconc}
3> Rec=spawn(myconc, myrec,[]).
<0.40.0>
4> Rec! {do_A,2}.
A
{do_A,2}
A
End A
5> Rec! {do_B,2}.
{do_B,2}
6> f(Rec).
ok
7> Rec=spawn(myconc, myrec,[]).
<0.45.0>
8> Rec! {do_B,2}.
B
{do_B,2}
B
End B
```

```
9> f(Rec).
ok
10> Rec=spawn(myconc, myrec,[]).
<0.49.0>
11> Rec! fjrhjh.
Nothing to do
fjrhjh
```

f(X)
elibereaza X
http://erlang.org/doc/man/shell.html

➢ ?MODULE

macro care intoarce numele modulului curent

```
start() -> spawn(?MODULE, myrec, []).

myrec() ->
  receive
   {do_A, X} ->  prelA(X);
   {do_B, X} ->  prelB(X);
              _ -> io:format("Nothing to do ~n")
  end.
```

```
3> Pid = myconc1:start().
<0.112.0>
4> Pid ! {do_A,2}.
A
{do_A,2}
A
End A
```

➢ Cilent-Server (Exemplu simplu: doubling service)

```
-module(myserv).
-export([start_server/0, server_loop/0,client/2]).


start_server() ->  spawn(myserv, server_loop, []).


server_loop() ->
  receive
     {From, {double, Number}} -> From ! {self(),(Number*2)},
                                 server_loop() ;

     {From,_} -> From ! {self(),error},
                 server_loop()
  end.
```

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

```
3> c(myserv).
{ok,myserv}
4> Server = myserv:start_server().
<0.43.0>
5> myserv:client(Server,{double,15675}).
31350
6> myserv:client(Server,nothing).
error
7> myserv:client(Server, {double, 887}).
1774
```

➢ 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.
```

```
-module(servtemplate2).
-compile(export_all).

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

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

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

        end.
```

register(atom, Pid)
whereis(atom) -> Pid| undefined

➢ Message passing with data storage

- Procesul (serverul) este un frigider care accepta doua tipuri de comenzi:

  - depoziteaza alimente,
  - scoate alimente

- Acelasi aliment poate fi depozitat de mai multe ori si poate fi scos de cate ori a fost depozitat.

- La fiecare moment trebuie sa stim ce alimente se gasesc in frigider (starea procesului).

- Starea procesului se transmite prin parametrii functiilor.

kitchen.erl
http://learnyousomeerlang.com/

➢ Message passing with data storage

```
fridgef(FoodList) ->
        receive
        % comanda store
        % comanda take
        ....
        end.
```

```
store(Pid, Food) ->
        Pid ! {self(), {store, Food}},
        receive
                {Pid, Msg} -> Msg
        end.
```

```
take(Pid, Food) ->
        Pid ! {self(), {take, Food}},
        receive
                {Pid, Msg} -> Msg
        end.
```

kitchen.erl

http://learnyousomeerlang.com/

➢ Message passing with data storage

kitchen.erl

http://learnyousomeerlang.com/

```
fridgef(FoodList) ->
        receive
            {From, {store, Food}}  ->   From ! {self(), ok},
                                        fridgef([Food|FoodList]);

            {From, {take, Food}} ->  case lists:member(Food, FoodList) of
                                        true ->  From ! {self(), {ok, Food}},
                                                 fridgef(lists:delete(Food, FoodList));
                                        false ->   From ! {self(), not_found},
                                                   fridgef(FoodList)
                                     end;
             terminate   ->      ok
        end.
```

```
6> c(kitchen).
{ok,kitchen}
7> Fridge = kitchen:start([milk, cheese, ham]).
<0.99.0>
8> kitchen:store(Fridge, juice).
ok
9> kitchen:take(Fridge, milk).
{ok,milk}
10> kitchen:take(Fridge, juice).
{ok,juice}
11> kitchen:take(Fridge, juice).
not_found
```

➢ Varianta: registered process, comenzile **show** (pentru a vizualiza starea) si **terminate**

```
start(FoodList) -> register(fridge, spawn(fun()-> fridgef(FoodList) end)).
```

```
store(Food) ->
   fridge! {self(), {store, Food}},
   receive
      {fridge, Msg} -> Msg
   end.

take( Food) ->
   fridge ! {self(), {take, Food}},
   receive
      {fridge, Msg} -> Msg
   end.
```

```
show() ->
   fridge ! {self(), show},
   receive
      {fridge, Msg} -> Msg
   end.

terminate() ->
      fridge ! {self(), terminate},
      receive
        {fridge, Msg} -> Msg
      end.
```

- Varianta: registered process, comenzile **show** (pentru a vizualiza starea) si **terminate**

```
fridgef(FoodList) ->
   receive
      {From, {store, Food}} -> From ! {fridge, ok},
                               fridgef([Food|FoodList]);
      {From, {take, Food}} ->
                  case lists:member(Food, FoodList) of
                     true ->   From ! {fridge, {ok, Food}},
                               fridgef(lists:delete(Food, FoodList));
                     false ->  From ! {fridge, not_found},
                               fridgef(FoodList)
                   end;
      {From,show} ->  From ! {fridge, FoodList},
                      fridgef(FoodList);
      {From,terminate} -> From ! {fridge, done}
    end.
```

mykitchen.erl

```
2> c(mykitchen).
{ok,mykitchen}
3> mykitchen:start([milk, apple]).
true
4> mykitchen:take(milk).
{ok,milk}
5> mykitchen:store(orange).
ok
6> mykitchen:show().
[orange,apple]
7> mykitchen:terminate().
done
```

➢ receive … after … end

```
receive
Pattern1 when Guard1 -> Expr1;
Pattern2  when Guard2 -> Expr2;
Pattern3 -> Expr3
…
after T ->
      ExpressionT
end
```

- procesul asteapta pana cand primeste un mesaj care se potriveste cu un pattern sau pana cand expira timpul .
- timpul T este exprimat in milisecunde
- procesul va astepta maxim T milisecunde sa primeasca un mesaj
- daca nici un mesaj care se potriveste cu un patern nu este primit in timpul T, procesul executa ExpressionT

➢receive … after … end

```
sleep(T) ->
      receive
        after T ->
              ok
      end.
```

nu exista mesaje de prelucrat, procesul va fi blocat T milisecunde

```
flush() ->
  receive
          _      -> flush()
    after 0      ->
          ok
end.
```

orice mesaj se potriveste cu patternul, deci apelul recursiv va goli coada de mesaje, dupa care procesul va continua

```
after 0   -> …
```

verifica coada de mesaje si apoi continua

daca aceasta clauza lipseste, procesul se va bloca cand coada de mesaje se goleste

➢receive … after … end

Procesul este un ceas care apeleaza o functie repetat dupa un interval de timp fixat.

```
-module(clock).
-export([start/2, stop/0]).
start(Time, Fun) ->
      register(clock, spawn(fun() -> tick(Time, Fun) end)).

stop() -> clock ! stop.

tick(Time, Fun) ->
  receive
       stop ->  void
 after Time ->
                  Fun(),
                  tick(Time, Fun)   end.


currentTime() ->
      io:format("TICK~p~n",[erlang:monotonic_time()]).
```

```
2> c(clock).
{ok,clock}
3> clock:start(3000, fun clock:currentTime/0).
true
TICK119568384
TICK122640384
TICK125727744
TICK128799744
TICK131871744
TICK134943744
TICK138015744
TICK141087744
TICK144159744
4> clock:stop().
stop
```

https://pragprog.com/titles/jaerlang2/source_code

mykitchen3.erl

```
fridgef(FoodList) ->
 receive
      {From, {store, Food}} -> From ! {fridge, ok},
                               fridgef([Food|FoodList]);
      {From, {take, Food}} ->
                  case lists:member(Food, FoodList) of
                    true ->   From ! {fridge, {ok, Food}},
                              fridgef(lists:delete(Food, FoodList));
                    false ->  From ! {fridge, not_found},
                              fridgef(FoodList)
                  end;
      {From,show} ->  From ! {fridge, FoodList},
                      fridgef(FoodList)
  end,
  io:format("al doilea receive~n"),
  receive
      {From,terminate} -> From ! {fridge, done}
   end.
```

```
1> c(mykitchen3).
{ok,mykitchen3}
2> mykitchen3:start([]).
true
3> mykitchen3:store(apple).
ok
4> mykitchen3:terminate().
```

procesul este blocat!

Ctrl-G deblocheaza shell-ul

```
User switch command
 --> i
 --> s
 --> c
Eshell V8.3  (abort with ^G)
1> f().
ok
```

f() dezleaga variabilele

http://erlang.org/doc/man/shell.html

➢ receive … after … end

```
fridgef(FoodList) ->
 receive
      {From, {store, Food}} -> From ! {fridge, ok},
                               fridgef([Food|FoodList]);
      {From, {take, Food}} ->
                  case lists:member(Food, FoodList) of
                    true ->   From ! {fridge, {ok, Food}},
                              fridgef(lists:delete(Food, FoodList));
                    false ->  From ! {fridge, not_found},
                              fridgef(FoodList)
                   end;
      {From,show} ->  From ! {fridge, FoodList},
                      fridgef(FoodList)

  after 30000 ->   timeout
 end,
  io:format("al doilea receive~n"),
  receive
      {From,terminate} -> From ! {fridge, done}
   end.
```

```
1> c(mykitchen3).
{ok,mykitchen3}
2> mykitchen3:start([]).
true
3> mykitchen3:terminate().
al doilea receive
done
```

- receive … after … end

```
fridgef(FoodList) ->
 receive
      {From, {store, Food}} -> From ! {fridge, ok},
                               fridgef([Food|FoodList]);
      {From, {take, Food}} ->
                case lists:member(Food, FoodList) of
                  true ->   From ! {fridge, {ok, Food}},
                            fridgef(lists:delete(Food, FoodList));
                  false ->  From ! {fridge, not_found},
                            fridgef(FoodList)
                end;
       {From,show} ->  From ! {fridge, FoodList},
                       fridgef(FoodList);
      {From,terminate} -> From ! {fridge, done}
  after 30000 ->   timeout
 end,
receive
    gata -> io:format("Sunt gata~n")
end.
```

```
1> c(mykitchen4).
{ok,mykitchen4}
2> mykitchen4:start([apple]).
true
3> mykitchen4:gata().
gata
4> mykitchen4:show().
[apple]
5> mykitchen4:store(water).
ok
6> mykitchen4:show().
[water,apple]
Sunt gata
```

mesajul **gata** este prelucrat dupa ce se iese din primul **receive**, adica dupa ce au trecut 30 sec de cand s-a intrat in primul **receive**.

➢ receive … after … end

```
receive
Pattern1 when Guard1 -> Expr1;
Pattern2  when Guard2 -> Expr2;
Pattern3 -> Expr3
…
after T ->
      ExpressionT
end
```

- La intrarea in **receive**, daca exista un **after**, se porneste un timer.
- Mesajele din coada sunt investigate in ordinea sosirii; daca un mesaj se potriveste cu un pattern atunci expresia corespunzatoare este prelucrata.
- Mesajele care nu se potrivesc cu nici un pattern sunt puse intr-o coada separate (*save queue*).
- Daca nu mai sunt mesaje in coada procesul se suspenda si asteapta venirea unui nou mesaj; la venirea acestuia, numai el este prelucrat, nu si mesajele din *save queue.*
- Cand un mesaj se potriveste cu un pattern, mesajele din *save queue* sunt puse la loc in coada si timerul se sterge.
- Daca timpul T s-a scurs fara ca un mesaj sa se potriveasca unui pattern, atunci ExpressionT se executa, iar mesajele din *save queue* sunt puse inapoi in coada.

- Selective receives

```
5> self()! hi, self() ! low.
low
6> flush().
Shell got hi
Shell got low
ok
```

```
flush() ->
  receive
        _       -> flush()
    after 0     ->
        ok
end.
```

```
important() ->
    receive
      {Priority, X} when Priority > 10 ->   [X|important()]
      after 0 ->
            normal()
    end.

normal() ->
     receive
         {_,X} ->
               [X|normal()]
      after 0 -> []
end.
```

Varianta a functiei flush() care ordoneaza mesajele dupa prioritati

```
2> c(sel).
{ok,sel}
3> self()! {5, low1}, self() ! {9,low2}, self() ! {15, high1}, self()!{11,high2}
.
{11,high2}
4> sel:important().
[high1,high2,low1,low2]
```

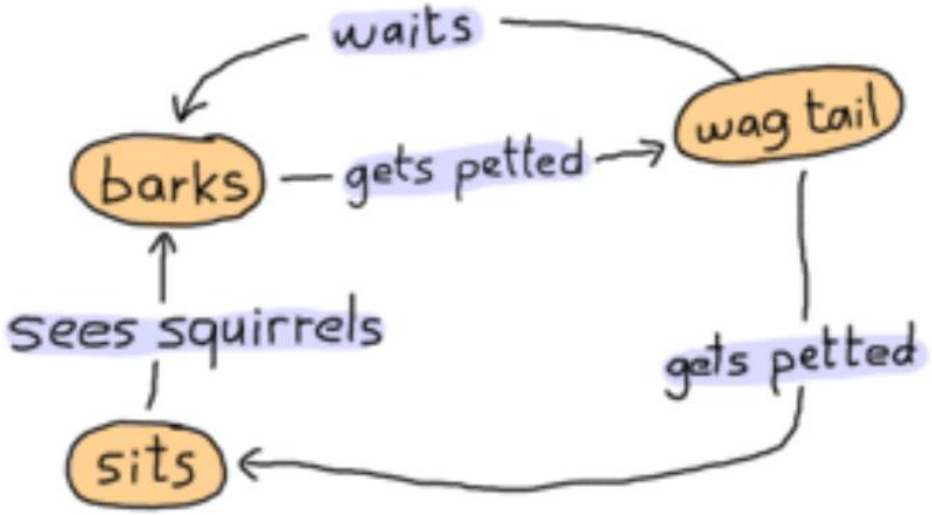
➢ Cilent-Server (simple) template

```
-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



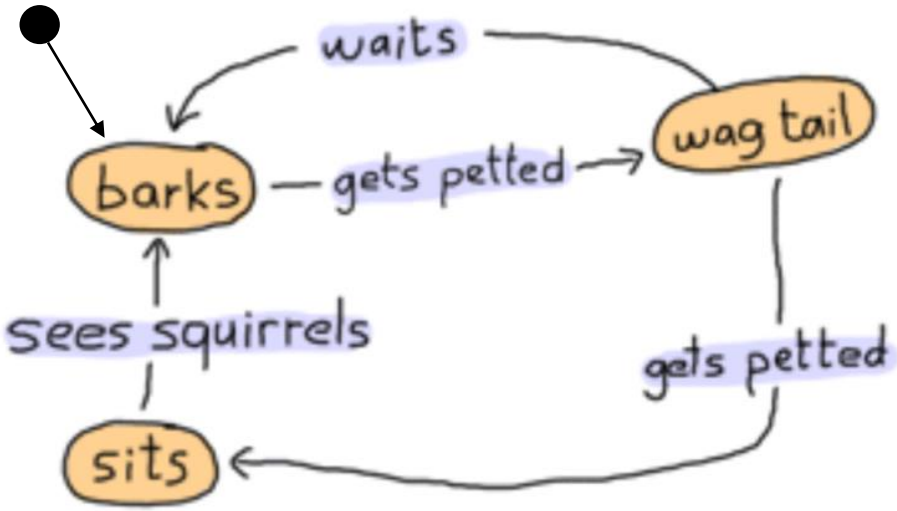
Starile ={barks, sits, wag_tail}
Actiunile ={gets_petted, see_squirrels, waits}

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

➢Finite-State Machine

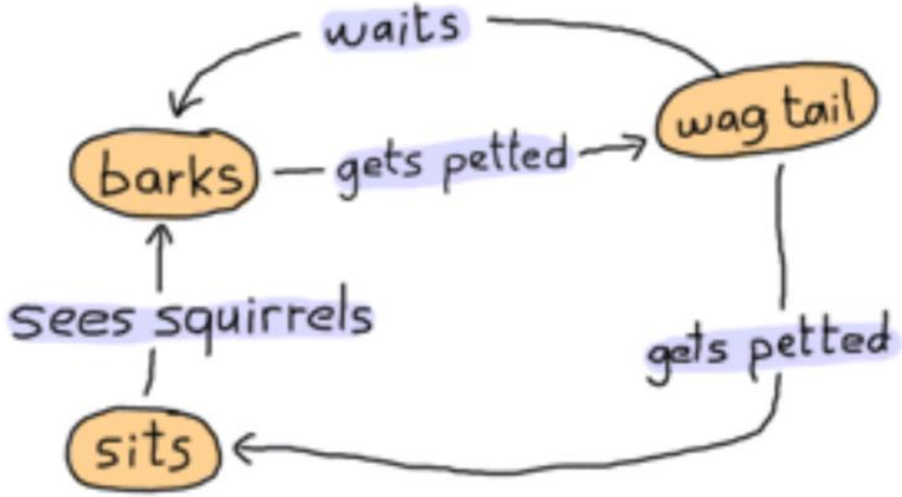
actiunile sunt implementate prin mesaje si sunt vizibile in exterior

```
-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.
```



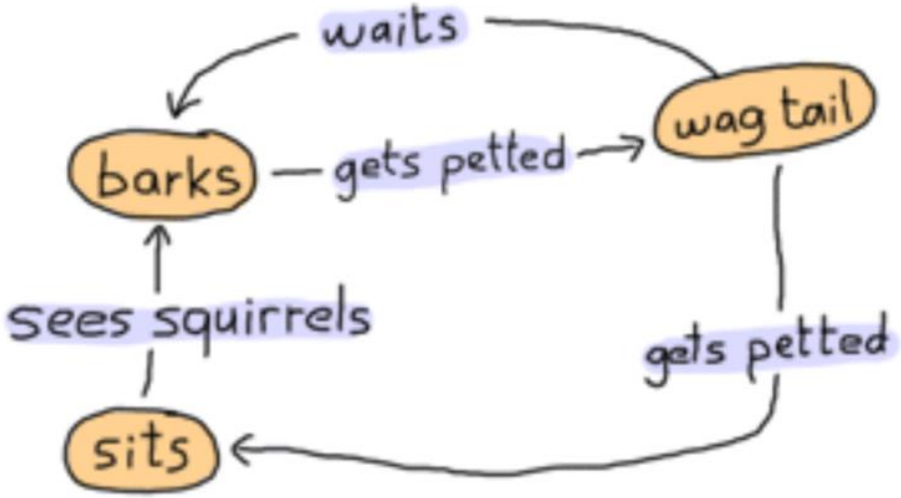
➢Finite-StateMachine: implementarea starilor

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



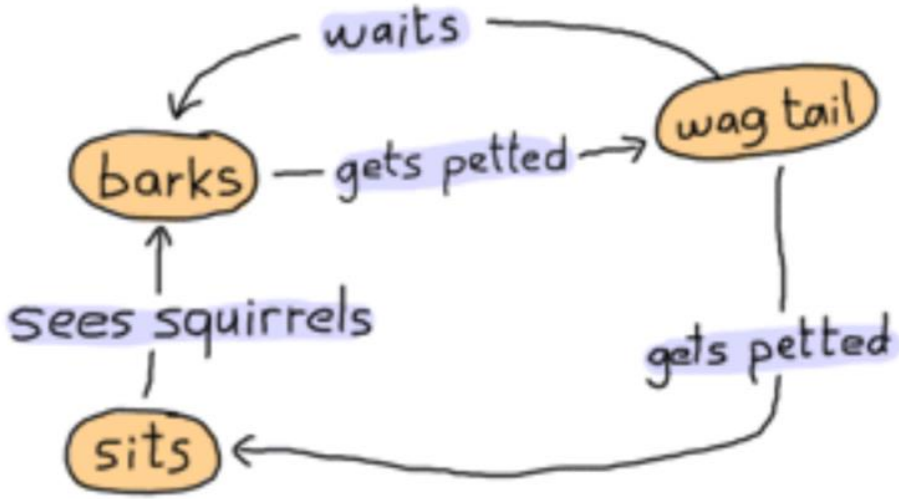
➢Finite-StateMachine: implementarea starilor

```
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.
```



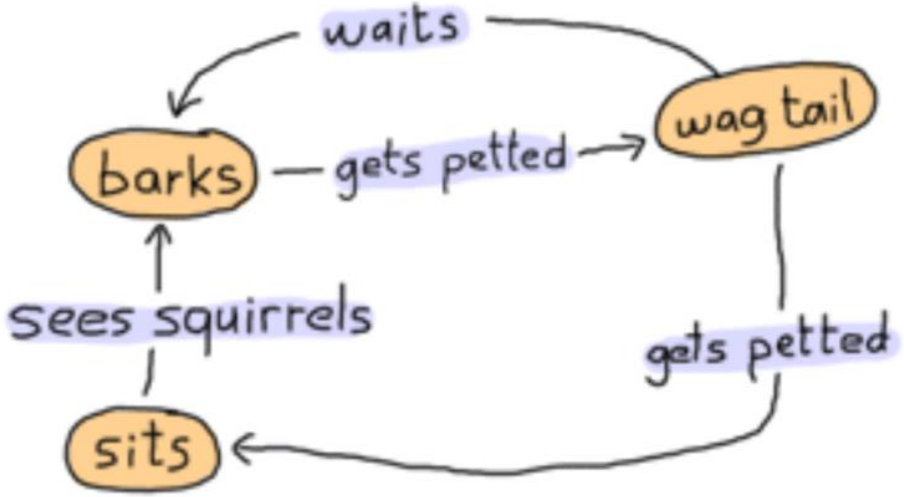
➢ Finite-StateMachine: implementarea starilor

```
sit() ->
    io:format("Dog is sitting. Goooooood 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