WEEK6 - PRESENTED BY NOLLEH

# LEARN U ERLANG

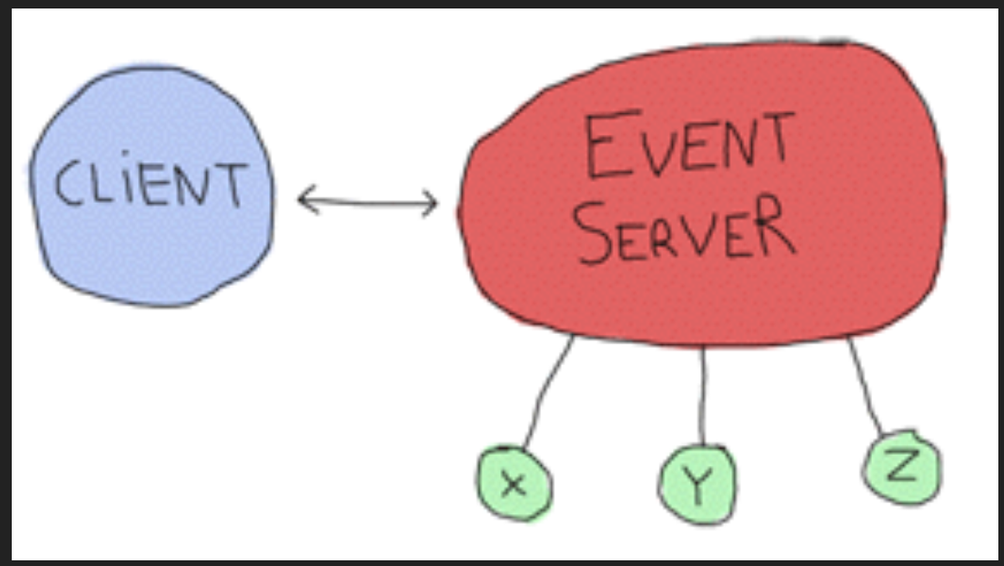
CHAPTER.13

# DESIGNING A CONCURRENT PROGRAM

# UNDERSTANDING THE PROBLEM

- roles
  - add event ( event contains deadline / name / desc )
  - show an warning when time has comes up
  - cancel event
  - no persistent disk storage
  - update while it is running
  - interaction via command line

# UNDERSTANDING THE PROBLEM




- event server
  - Accepts subscriptions from clients
  - Forwards notifications from event processes to each of the subscribers
  - Accepts messages to add events (and start the x, y, z processes needed)
  - Can accept messages to cancel an event and subsequently kill the event processes
  - Can be terminated by a client
  - Can have its code reloaded via the shell.

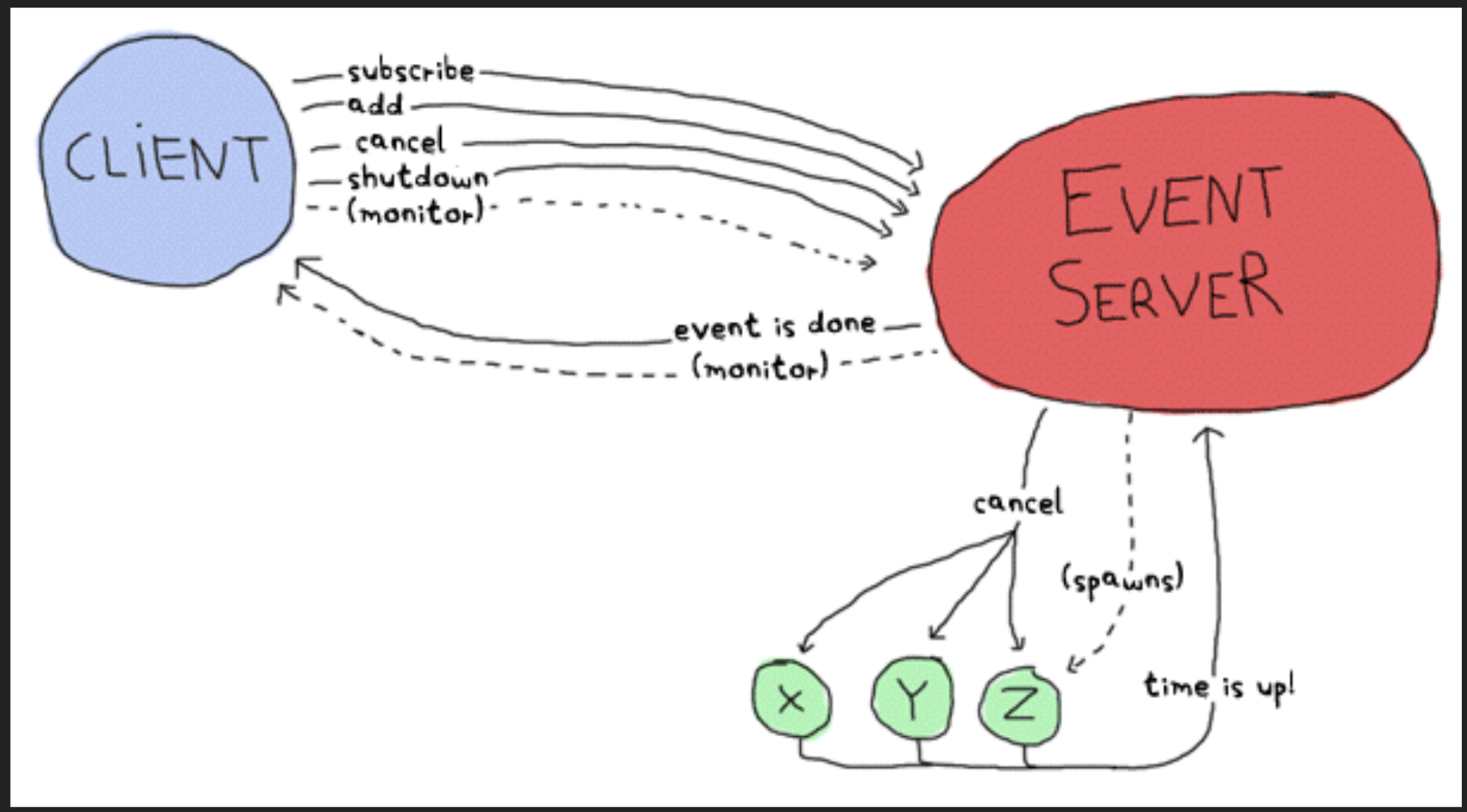
# UNDERSTANDING THE PROBLEM

- client
  - Subscribes to the event server and receive notifications as messages. Each of these could potentially be a gateway to the different interaction points mentioned above (GUI, web page, instant messaging software, email, etc.)
  - Asks the server to add an event with all its details
  - Asks the server to cancel an event
  - Monitors the server (to know if it goes down)
  - Shuts down the event server if needed

# UNDERSTANDING THE PROBLEM

- x, y and z:
  - Represent a notification waiting to fire (they're basically just timers linked to the event server)
  - Send a message to the event server when the time is up
  - Receive a cancellation message and die

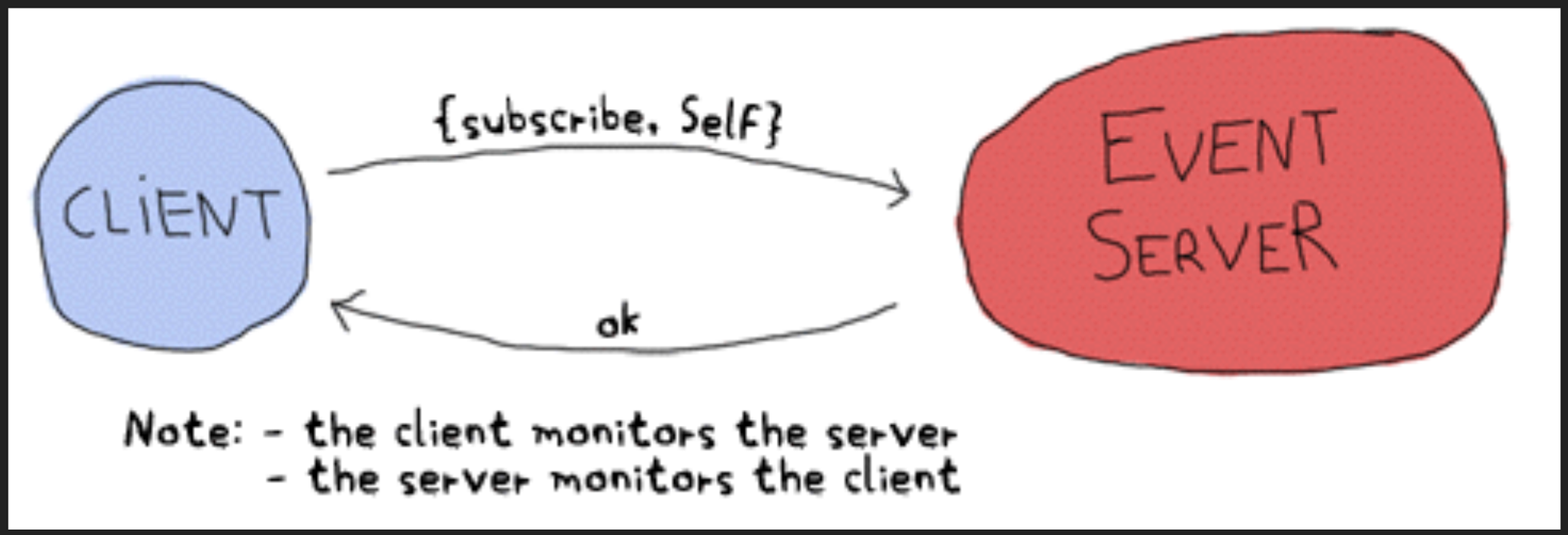
# UNDERSTANDING THE PROBLEM




- for real program,
  - timer:send_after/2-3 to avoid spawning too many processes.

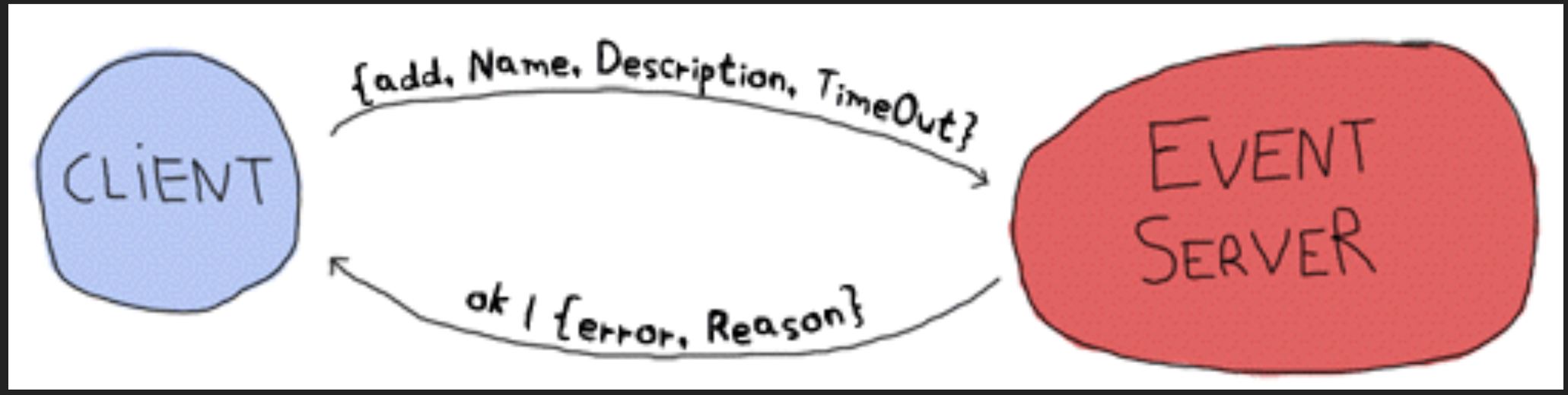
# DEFINING THE PROTOCOL

- subscribe



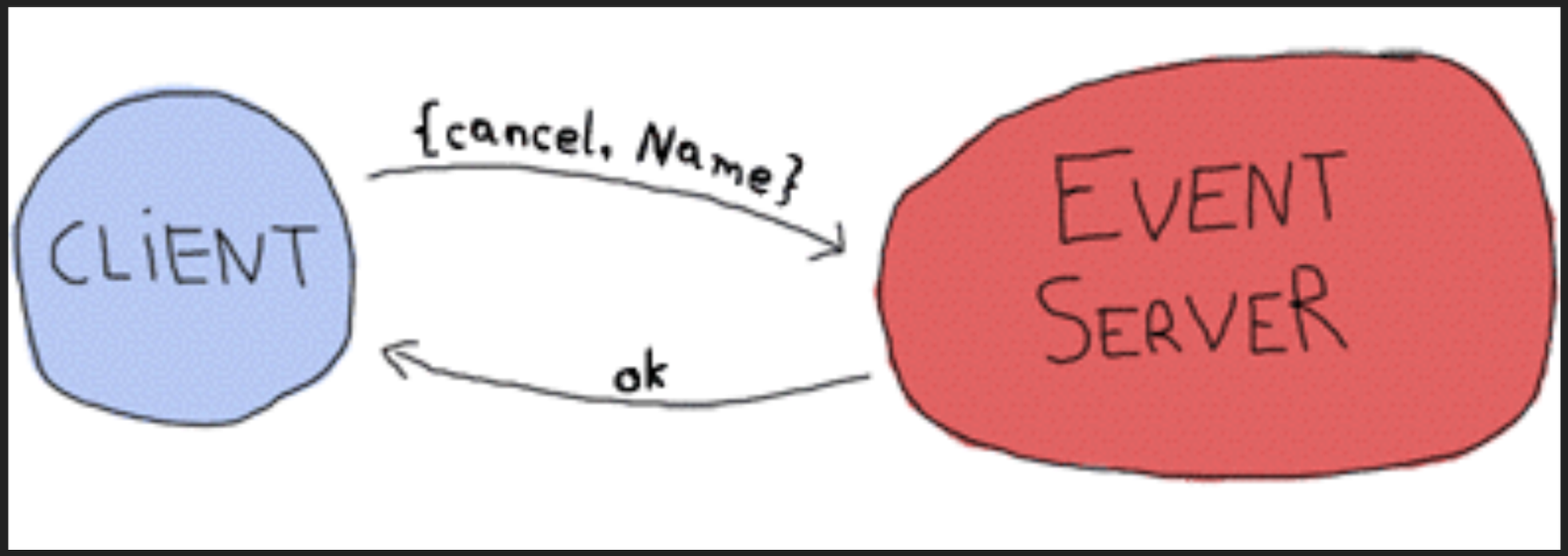
# DEFINING THE PROTOCOL

- add



# DEFINING THE PROTOCOL

- cancel

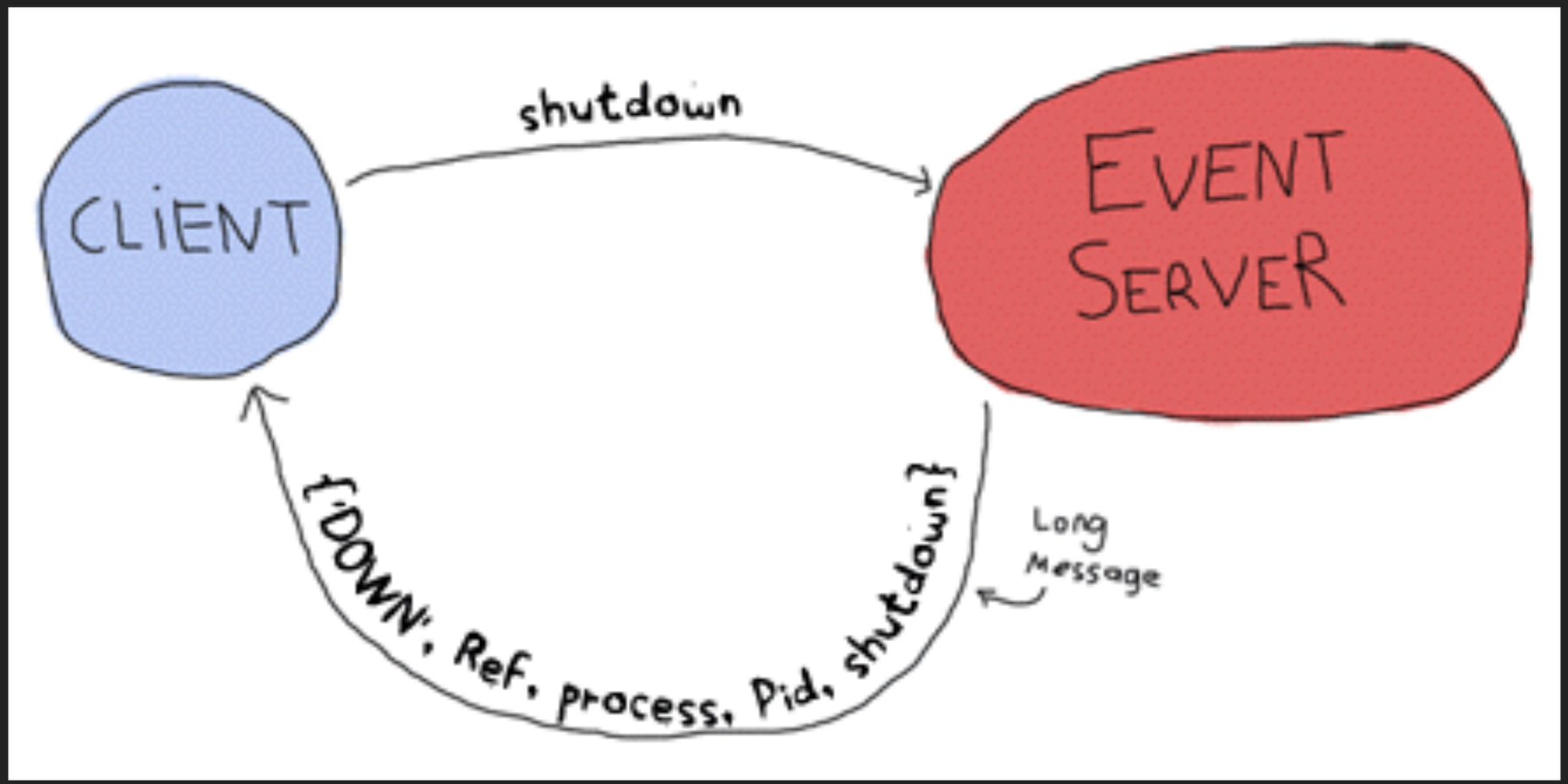


# DEFINING THE PROTOCOL

- notification

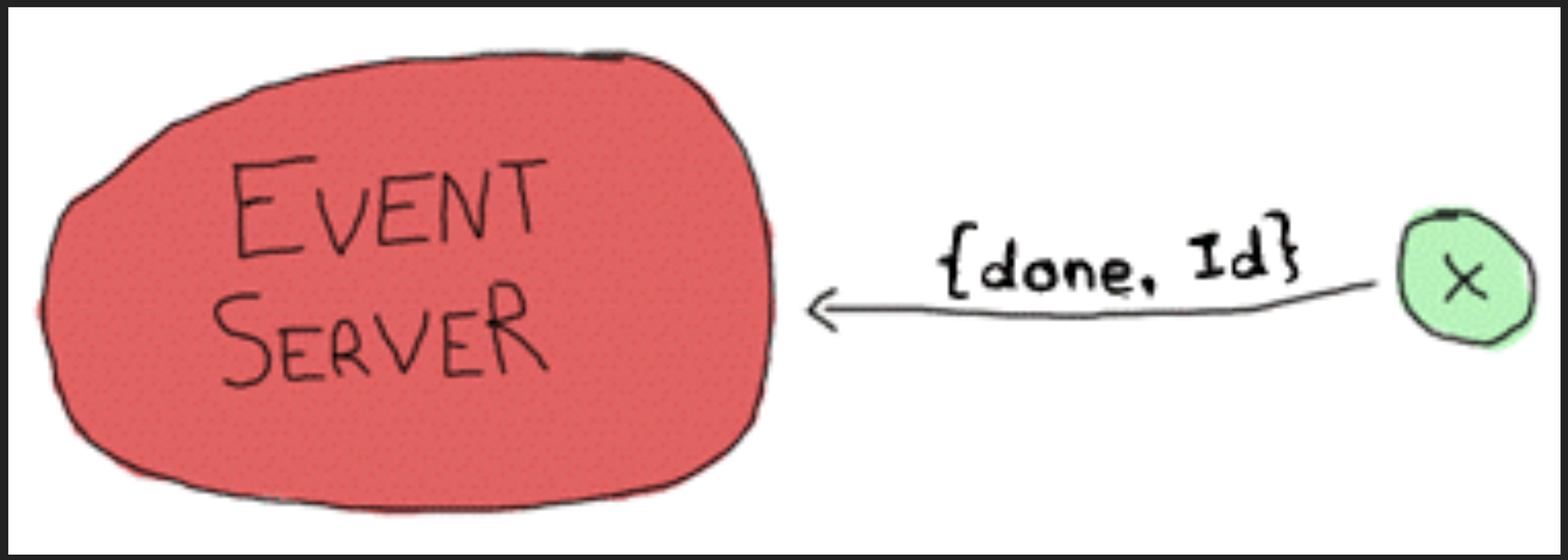
# DEFINING THE PROTOCOL

- shutdown



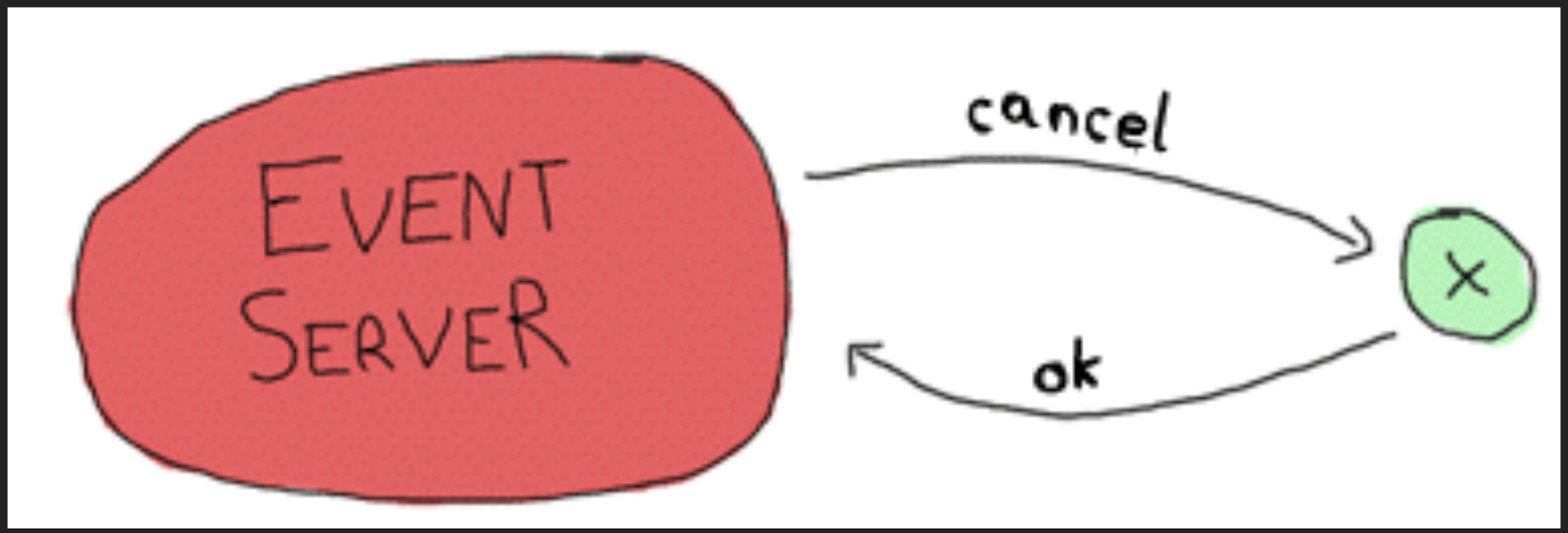
# DEFINING THE PROTOCOL

- notification



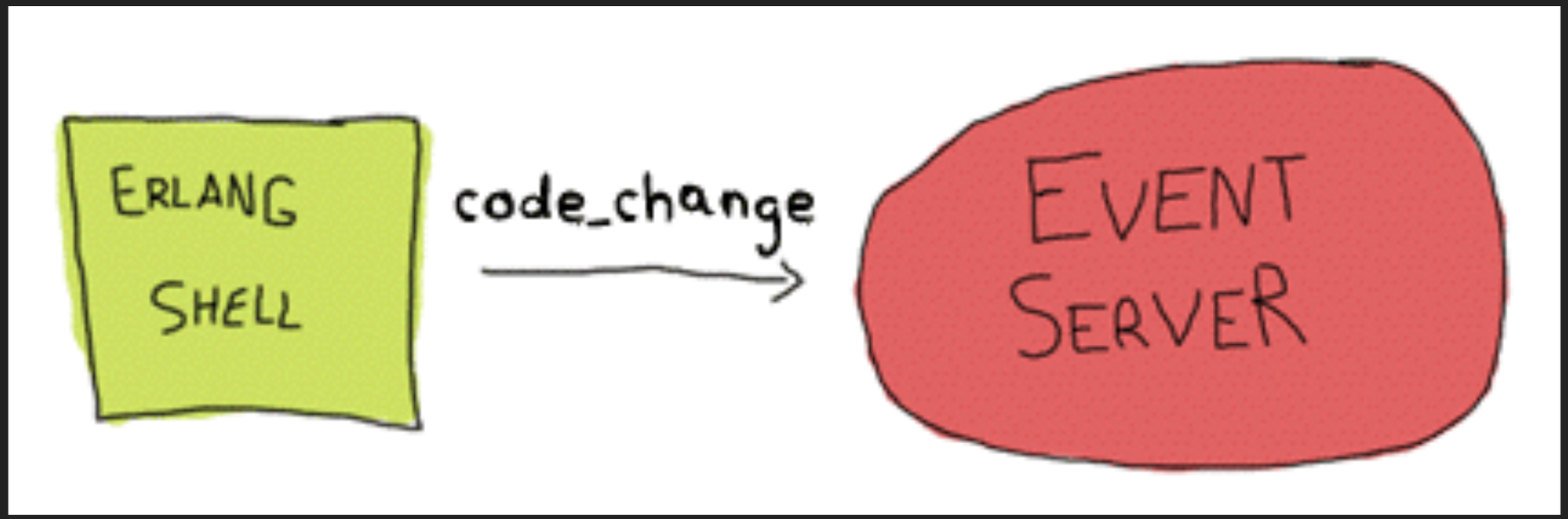
# DEFINING THE PROTOCOL

- cancel



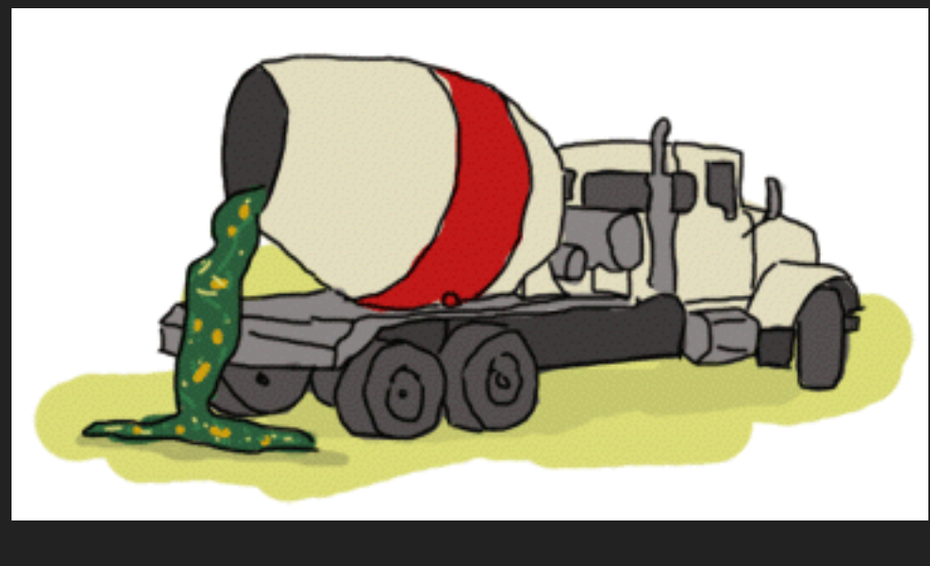
# DEFINING THE PROTOCOL

- code change



# LAY THEM FOUNDATION

- project
  - ebin/ (compiled)
  - include/ (header)



  - priv/ (executables that might have to interact with Erlang, such as specific drivers and whatnot)
  - src/ (private hrl, erl)

# AN EVENT MODULE

- project
  - {Pid, Ref, Message}, where Pid is the sender and Ref is a unique message identifier to help know what reply came from who.

```
loop(State) ->
  receive
    {Server, Ref, cancel} ->
      ...
    after Delay ->
      ...
  end.
```

# AN EVENT MODULE

- state
  - contains
  - timeout / name of event / event server pid (for noti)

```
-module(event).
-compile(export_all).
-record(state, {server,
     name="",
     to_go=0}).
```

# AN EVENT MODULE

- state
  - contains
    - timeout / name of event / event server pid (for noti)

```
loop(S = #state{server=Server}) ->
  receive
  {Server, Ref, cancel} ->
    Server ! {Ref, ok}
  after S#state.to_go*1000 ->
    Server ! {done, S#state.name}
  end.
```

# AN EVENT MODULE

- test event

```
6> c(event).
  {ok,event}
7> rr(event, state).
  [state]
8> spawn(event, loop, [#state{server=self(), name="test", to_go=5}]).
  <0.60.0>
9> flush().
  ok
10> flush().
  Shell got {done,"test"}
  ok
```

# AN EVENT MODULE

- test event

```
11> Pid = spawn(event, loop, [#state{server=self(), name="test",
 to_go=500}]).
  <0.64.0>
12> ReplyRef = make_ref().
  #Ref<0.0.0.210>
13> Pid ! {self(), ReplyRef, cancel}.
  {<0.50.0>,#Ref<0.0.0.210>,cancel}
14> flush().
  Shell got {#Ref<0.0.0.210>,ok}
  ok
```

- we don't expect it to come from anywhere specific (any place will do, we won't match on the receive) nor should we want to reply to it

# AN EVENT MODULE

- limit of timer

```
15> spawn(event, loop, [#state{server=self(), name="test",
to_go=365*24*60*60}]).
  <0.69.0>
16>
=ERROR REPORT==== DD-MM-YYYY::HH:mm:SS ===
Error in process <0.69.0> with exit value: {timeout_value,[{event,loop,
1}]}
```

# AN EVENT MODULE

- split it

```
%% Because Erlang is limited to about 49 days (49*24*60*60*1000) in
%% milliseconds, the following function is used
normalize(N) ->
  Limit = 49*24*60*60,
  [N rem Limit | lists:duplicate(N div Limit, Limit)].
```

- ex.  98*24*60*60+4 -> [4,4233600,4233600]

# AN EVENT MODULE

- split it

```
loop(S = #state{server=Server, to_go=[T|Next]}) ->
  receive
    {Server, Ref, cancel} ->
      Server ! {Ref, ok}
  after T*1000 ->
    if Next =:= [] ->
        Server ! {done, S#state.name};
      Next =/= [] ->
        loop(S#state{to_go=Next})
    end
end.
```

# AN EVENT MODULE

- normalize helper

```
start(EventName, Delay) ->
  spawn(?MODULE, init, [self(), EventName, Delay]).

start_link(EventName, Delay) ->
  spawn_link(?MODULE, init, [self(), EventName, Delay]).

%%% Event's innards
init(Server, EventName, Delay) ->
  loop(#state{server=Server,
    name=EventName,
    to_go=normalize(Delay)}).
```

# AN EVENT MODULE

- cancel helper

```
cancel(Pid) ->
  %% Monitor in case the process is already dead
  Ref = erlang:monitor(process, Pid),
  Pid ! {self(), Ref, cancel},
  receive
    {Ref, ok} ->
      erlang:demonitor(Ref, [flush]),
      ok;
    {'DOWN', Ref, process, Pid, _Reason} ->
      ok
  end.
```

# AN EVENT MODULE

- test

```
19> event:start("Event", 0).
  <0.103.0>
20> flush().
  Shell got {done,"Event"}
  ok
21> Pid = event:start("Event", 500).
  <0.106.0>
22> event:cancel(Pid).
  ok
```

# AN EVENT MODULE

- I don’t want noti-time as second

```
time_to_go(TimeOut={{_,_,_}, {_,_,_}}) ->
  Now = calendar:local_time(),
ToGo = calendar:datetime_to_gregorian_seconds(TimeOut) -
   calendar:datetime_to_gregorian_seconds(Now),
Secs = if ToGo > 0  -> ToGo;
             ToGo =< 0 -> 0
          end,
normalize(Secs).
```

# AN EVENT MODULE

- init function using time_to_go

```
init(Server, EventName, DateTime) ->
  loop(#state{server=Server,
              name=EventName,
              to_go=time_to_go(DateTime)}).
```

# THE EVENT SERVER

- skeleton

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

loop(State) ->
  receive
    {Pid, MsgRef, {subscribe, Client}} ->
    ...
    {Pid, MsgRef, {add, Name, Description, TimeOut}} ->
    ...
    {Pid, MsgRef, {cancel, Name}} ->
    ...
```

# THE EVENT SERVER

- skeleton

```
    {done, Name} ->
    ...
    shutdown ->
    ...
    {'DOWN', Ref, process, _Pid, _Reason} ->
    ...
    code_change ->
    ...
    Unknown ->
      io:format("Unknown message: ~p~n",[Unknown]),
      loop(State)
  end.
```

# THE EVENT SERVER

- declare state

```
-record(state, {events,    %% list of #event{} records
                clients}). %% list of Pids

-record(event, {name="",
                description="",
                pid,
                timeout={{1970,1,1},{0,0,0}}}).
```

# THE EVENT SERVER

- declare state

```
loop(S = #state{}) ->
receive
...
end.
```

```
init() ->
  %% Loading events from a static file could be done here.
  %% You would need to pass an argument to init telling where the
  %% resource to find the events is. Then load it from here.
  %% Another option is to just pass the events straight to the server
  %% through this function.
  loop(#state{events=orddict:new(),
              clients=orddict:new()}).
```

# THE EVENT SERVER

- implement subscribe

```
{Pid, MsgRef, {subscribe, Client}} ->
  Ref = erlang:monitor(process, Client),
  NewClients = orddict:store(Ref, Client, S#state.clients),
  Pid ! {MsgRef, ok},
  loop(S#state{clients=NewClients});
```

# THE EVENT SERVER

- implement add event

```
valid_datetime({Date,Time}) ->
  try
    calendar:valid_date(Date) andalso valid_time(Time)
  catch
    error:function_clause -> %% not in {{Y,M,D},{H,Min,S}} format
      false
  end;
valid_datetime(_) ->
  false.
```

# THE EVENT SERVER

- implement add event

```
valid_time({H,M,S}) -> valid_time(H,M,S).
valid_time(H,M,S) when H >= 0, H < 24,
                       M >= 0, M < 60,
                        S >= 0, S < 60 -> true;
valid_time(_,_,_) -> false.
```

# THE EVENT SERVER

- implement add event

```
{Pid, MsgRef, {add, Name, Description, TimeOut}} ->
  case valid_datetime(TimeOut) of
    true ->
      EventPid = event:start_link(Name, TimeOut),
      NewEvents = orddict:store(Name,
                  #event{name=Name,
                         description=Description,
                         pid=EventPid,
                         timeout=TimeOut},
                         S#state.events),
      Pid ! {MsgRef, ok},
      loop(S#state{events=NewEvents});
```

# THE EVENT SERVER

- implement add event

```
    false ->
        Pid ! {MsgRef, {error, bad_timeout}},
        loop(S)
end;
```

# THE EVENT SERVER

- implement cancel event

```
{Pid, MsgRef, {cancel, Name}} ->
  Events = case orddict:find(Name, S#state.events) of
               {ok, E} ->
                 event:cancel(E#event.pid),
                 orddict:erase(Name, S#state.events);
               error ->
                 S#state.events
             end,
Pid ! {MsgRef, ok},
loop(S#state{events=Events});
```

# THE EVENT SERVER

- implement handle done

```
{done, Name} ->
  case orddict:find(Name, S#state.events) of
    {ok, E} ->
      send_to_clients({done, E#event.name, E#event.description},
                      S#state.clients),
      NewEvents = orddict:erase(Name, S#state.events),
      loop(S#state{events=NewEvents});
    error ->
      %% This may happen if we cancel an event and
      %% it fires at the same time
      loop(S)
end;
```

# THE EVENT SERVER

- implement send_to_client

```
send_to_clients(Msg, ClientDict) ->
  orddict:map(fun(_Ref, Pid) -> Pid ! Msg end, ClientDict).
```

- others

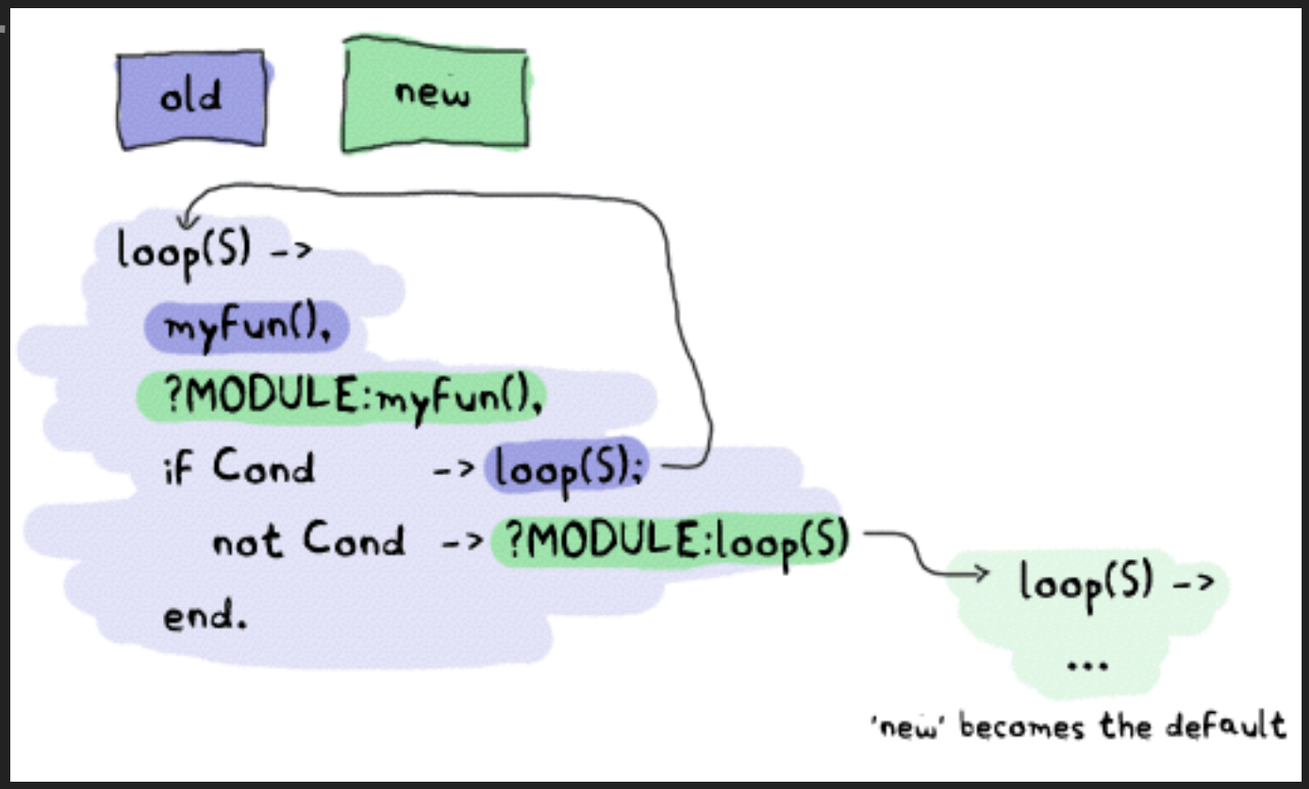
```
shutdown ->
  exit(shutdown);
{'DOWN', Ref, process, _Pid, _Reason} ->
  loop(S#state{clients=orddict:erase(Ref, S#state.clients)});
code_change ->
  ?MODULE:loop(S);
Unknown ->
  io:format("Unknown message: ~p~n",[Unknown]),
  loop(S)
```

# HOT CODE LOVING

- code server
  - basically a VM process in charge of an ETS table
    - ETS table : in-memory database table
      - A new version of a module is automatically loaded when compiling it with c(Module), loading with l(Module) or loading it with one of the many functions of the code module.
- local and external call
  - external calls are always done on the newest version of the code available in the code server

# HOT CODE LOVING

- code server



# HOT CODE LOVING

- more generic code update

```
-module(hotload).
-export([server/1, upgrade/1]).

server(State) ->
  receive
    update ->
      NewState = ?MODULE:upgrade(State),
      ?MODULE:server(NewState);
    SomeMessage ->
      %% do something here
      server(State)  %% stay in the same version no matter what.
  end.
```

# I SAID, HIDE YOUR MESSAGE

- hiding

```
start() ->
  register(?MODULE, Pid=spawn(?MODULE, init, [])),
  Pid.
start_link() ->
  register(?MODULE, Pid=spawn_link(?MODULE, init, [])),
  Pid.
terminate() ->
  ?MODULE ! shutdown.
```

- we should only have one running at a time

# I SAID, HIDE YOUR MESSAGE

- subscribe

```
subscribe(Pid) ->
  Ref = erlang:monitor(process, whereis(?MODULE)),
  ?MODULE ! {self(), Ref, {subscribe, Pid}},
receive
  {Ref, ok} ->
    {ok, Ref};
  {'DOWN', Ref, process, _Pid, Reason} ->
    {error, Reason}
after 5000 ->
  {error, timeout}
end.
```

# I SAID, HIDE YOUR MESSAGE

- add_event

```
add_event(Name, Description, TimeOut) ->
  Ref = make_ref(),
?MODULE ! {self(), Ref, {add, Name, Description, TimeOut}},
receive
  {Ref, Msg} -> Msg
after 5000 ->
  {error, timeout}
end.
```

# I SAID, HIDE YOUR MESSAGE

- add_event

```
add_event2(Name, Description, TimeOut) ->
  Ref = make_ref(),
?MODULE ! {self(), Ref, {add, Name, Description, TimeOut}},
receive
  {Ref, {error, Reason}} -> erlang:error(Reason);
  {Ref, Msg} -> Msg
after 5000 ->
  {error, timeout}
end.
```

# I SAID, HIDE YOUR MESSAGE

- cancel

```
cancel(Name) ->
  Ref = make_ref(),
  ?MODULE ! {self(), Ref, {cancel, Name}},
  receive
    {Ref, ok} -> ok
  after 5000 ->
    {error, timeout}
  end.
```

# I SAID, HIDE YOUR MESSAGE

- accumulate all messages during a given period of time

```
listen(Delay) ->
  receive
    M = {done, _Name, _Description} ->
    [M | listen(0)]
  after Delay*1000 ->
    []
  end.
```

# A TEST DRIVE

- vim Emakefile

```
{'src/*', [debug_info,
        {i, "src"},
        {i, "include"},
        {outdir, "ebin"}]}.
```

- erl -make
- erl -pa ebin/
  - add path for look in erlang module
- make:all([load])
  - find Emakefile -> recompile -> load

# A TEST DRIVE

```
1> evserv:start().
  <0.34.0>
2> evserv:subscribe(self()).
  {ok,#Ref<0.0.0.31>}
3> evserv:add_event("Hey there", "test", FutureDateTime).
  ok
4> evserv:listen(5).
  []
5> evserv:cancel("Hey there").
  ok
6> evserv:add_event("Hey there2", "test", NextMinuteDateTime).
ok
7> evserv:listen(2000).
  [{done,"Hey there2","test"}]
```

# ADDING SUPERVISION

- supervisor

```
-module(sup).
-export([start/2, start_link/2, init/1, loop/1]).


start(Mod,Args) ->
  spawn(?MODULE, init, [{Mod, Args}]).

start_link(Mod,Args) ->
  spawn_link(?MODULE, init, [{Mod, Args}]).

init({Mod,Args}) ->
  process_flag(trap_exit, true),
  loop({Mod,start_link,Args}).
```

# ADDING SUPERVISION

- supervisor

```
loop({M,F,A}) ->
  Pid = apply(M,F,A),
  receive
    {'EXIT', _From, shutdown} ->
      exit(shutdown); % will kill the child too
    {'EXIT', Pid, Reason} ->
      io:format("Process ~p exited for reason ~p~n",[Pid,Reason]),
      loop({M,F,A})
  end.
```

# ADDING SUPERVISION

- using supervisor

```
1> c(evserv), c(sup).
  {ok,sup}
2> SupPid = sup:start(evserv, []).
  <0.43.0>
3> whereis(evserv).
  <0.44.0>
4> exit(whereis(evserv), die).
  true
  Process <0.44.0> exited for reason die
5> exit(whereis(evserv), die).
  Process <0.48.0> exited for reason die
  true
```

# ADDING SUPERVISION

- using supervisor

```
6> exit(SupPid, shutdown).
  true
7> whereis(evserv).
  undefined
```

- The supervisor demonstrated here is only the most basic form that exists and is not exactly fit for production environments compared to the real thing.

# NAMESPACES (OR LACK THERE OF)

- using prefix
  - renamed to reminder_evserv, reminder_sup and reminder_event.
- Some programmers then decide to add a module, named after the application itself, which wraps common calls
- No need to synchronize them, no locks, no real main loop

CHAPTER.14

# WHAT IS OTP?

# IT'S THE OPEN TELECOM PLATFORM!

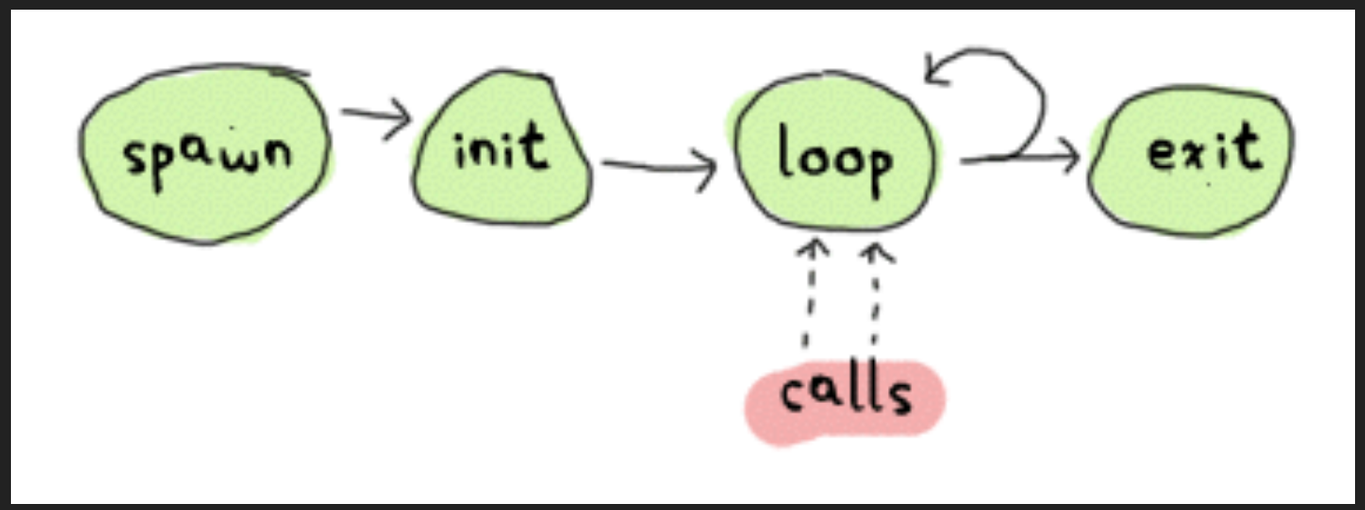
- meaning?
  - OTP stands for Open Telecom Platform, although it's not that much about telecom anymore (it's more about software that has the property of telecom applications)
- erlang's greatness comes from
  - concurrency + distribution + error handling capabilities, and otp

# IT'S THE OPEN TELECOM PLATFORM!

- There were a few 'gotchas' here and there
  - on how to avoid race conditions or to always remember that a process could die at any time. There was also hot code loading, naming processes and adding supervisors, to name a few.
- The OTP framework takes care of this by grouping these essential practices
  - Every Erlang programmer should use them

# THE COMMON PROCESS, ABSTRACTED

- In most processes, we had a function in charge of spawning the new process, a function in charge of giving it its initial values, a main loop, etc.
- these parts are usually present



# THE COMMON PROCESS, ABSTRACTED

- crafted code
  - with the advantage of being used for years in the field
  - also being built with far more caution than we were with our implementations.
- fault-tolerant manner

# THE BASIC SERVER

- kitty_server

```
%%%%% Naive version
-module(kitty_server).

-export([start_link/0, order_cat/4, return_cat/2, close_shop/1]).

-record(cat, {name, color=green, description}).

%%% Client API
start_link() -> spawn_link(fun init/0).
```

# THE BASIC SERVER

- kitty_server - api, order_cat

```
%% Synchronous call
order_cat(Pid, Name, Color, Description) ->
  Ref = erlang:monitor(process, Pid),
Pid ! {self(), Ref, {order, Name, Color, Description}},
receive
  {Ref, Cat} ->
    erlang:demonitor(Ref, [flush]),
    Cat;
  {'DOWN', Ref, process, Pid, Reason} ->
    erlang:error(Reason)
after 5000 ->
  erlang:error(timeout)
end
```

# THE BASIC SERVER

- kitty_server - api, return_cat

```
%% This call is asynchronous
return_cat(Pid, Cat = #cat{}) ->
  Pid ! {return, Cat},
  ok.
close_shop(Pid) ->
  Ref = erlang:monitor(process, Pid),
  Pid ! {self(), Ref, terminate},
  receive
    {Ref, ok} -> erlang:demonitor(Ref, [flush]), ok;
    {'DOWN', Ref, process, Pid, Reason} -> erlang:error(Reason)
  after 5000 -> erlang:error(timeout)
end.
```

# THE BASIC SERVER

- kitty_server - server_fun, init, loop

```
%%% Server functions
init() -> loop([]).

loop(Cats) ->
  receive
    {Pid, Ref, {order, Name, Color, Description}} ->
      if Cats =:= [] ->
        Pid ! {Ref, make_cat(Name, Color, Description)},
        loop(Cats);
      Cats =/= [] -> % got to empty the stock
        Pid ! {Ref, hd(Cats)},
        loop(tl(Cats))
      end;
```

# THE BASIC SERVER

- kitty_server - server_fun, loop2

```
{return, Cat = #cat{}} ->
  loop([Cat|Cats]);
{Pid, Ref, terminate} ->
  Pid ! {Ref, ok},
  terminate(Cats);
Unknown ->
  %% do some logging here too
  io:format("Unknown message: ~p~n", [Unknown]),
  loop(Cats)
end.
```

# THE BASIC SERVER

- kitty_server - private server_fun

```
%%% Private functions
make_cat(Name, Col, Desc) ->
  #cat{name=Name, color=Col, description=Desc}.

terminate(Cats) ->
  [io:format("~p was set free.~n",[C#cat.name]) || C <- Cats],
  ok.
```

# THE BASIC SERVER

- kitty_server - private server_fun

```
1> c(kitty_server).
  {ok,kitty_server}
2> rr(kitty_server).
  [cat]
3> Pid = kitty_server:start_link().
  <0.57.0>
4> Cat1 = kitty_server:order_cat(Pid, carl, brown, "loves to burn
                  bridges").
  #cat{name = carl,color = brown,
  description = "loves to burn bridges"}
5> kitty_server:return_cat(Pid, Cat1).
  ok
```

# THE BASIC SERVER

- kitty_server - private server_fun

```
6> kitty_server:order_cat(Pid, jimmy, orange, "cuddly").
  #cat{name = carl,color = brown,
  description = "loves to burn bridges"}
7> kitty_server:order_cat(Pid, jimmy, orange, "cuddly").
  #cat{name = jimmy,color = orange,description = "cuddly"}
8> kitty_server:return_cat(Pid, Cat1).
  ok
9> kitty_server:close_shop(Pid).
  carl was set free.
  ok
10> kitty_server:close_shop(Pid).
  ** exception error: no such process or port
  in function  kitty_server:close_shop/1
```

# THE BASIC SERVER

- we can see patterns we've previously applied
    - The sections where we set monitors up and down, apply timers, receive data, use a main loop, handle the init function, etc.

# THE BASIC SERVER

- let's generic~

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

call(Pid, Msg) ->
  Ref = erlang:monitor(process, Pid),
  Pid ! {self(), Ref, Msg},
  receive
    {Ref, Reply} -> erlang:demonitor(Ref, [flush]), Reply;
    {'DOWN', Ref, process, Pid, Reason} ->
      erlang:error(Reason)
  after 5000 ->
    erlang:error(timeout)
  end.
```

# THE BASIC SERVER

- let's generic~

```
%% Synchronous call
order_cat(Pid, Name, Color, Description) ->
  my_server:call(Pid, {order, Name, Color, Description}).

%% This call is asynchronous
return_cat(Pid, Cat = #cat{}) ->
  Pid ! {return, Cat},
  ok.

%% Synchronous call
close_shop(Pid) ->
  my_server:call(Pid, terminate).
```

# THE BASIC SERVER

- next generic chunk is not obvious..
  - Note that every process we've written so far has a loop where all the messages are pattern matched

```
loop(Module, State) ->
  receive
    Message -> Module:handle(Message, State)
  end.
```

```
handle(Message1, State) -> NewState1;
handle(Message2, State) -> NewState2;
...
handle(MessageN, State) -> NewStateN.
```

# THE BASIC SERVER

- async call / sync call
  - It would be pretty helpful if our generic server implementation could provide a clear way to know which kind of call is which.
- we will need to match different kinds of messages in my_server:loop/2
  - add atom sync

# THE BASIC SERVER

- async call / sync call

```
call(Pid, Msg) ->
  Ref = erlang:monitor(process, Pid),
  Pid ! {sync, self(), Ref, Msg},
  receive
    {Ref, Reply} ->
      erlang:demonitor(Ref, [flush]),
      Reply;
    {'DOWN', Ref, process, Pid, Reason} ->
      erlang:error(Reason)
    after 5000 ->
      erlang:error(timeout)
  end.
```

# THE BASIC SERVER

- async call / sync call

```
cast(Pid, Msg) ->
  Pid ! {async, Msg},
  ok.
```

```
loop(Module, State) ->
  receive
    {async, Msg} ->
      loop(Module, Module:handle_cast(Msg, State));
    {sync, Pid, Ref, Msg} ->
      loop(Module, Module:handle_call(Msg, Pid, Ref, State))
  end.
```

# THE BASIC SERVER

- disappointing thing
  - The programmers who will use my_server will still need to know about references when sending synchronous messages and replying to them.

```
loop(Module, State) ->
  receive
    {async, Msg} ->
      loop(Module, Module:handle_cast(Msg, State));
    {sync, Pid, Ref, Msg} ->
      loop(Module, Module:handle_call(Msg, {Pid, Ref}, State))
  end.
```

# THE BASIC SERVER

- and now,
  - they can be passed as a single argument to the other function as a variable with a name like From.
- we'll provide a function to send replies that should understand what From contains.

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

# THE BASIC SERVER

- and now our code is,

```
-module(my_server).
-export([start/2, start_link/2, call/2, cast/2, reply/2]).
start(Module, InitialState) ->
  spawn(fun() -> init(Module, InitialState) end).
start_link(Module, InitialState) ->
  spawn_link(fun() -> init(Module, InitialState) end).

...call.. cast... reply...
init(Module, InitialState) ->
  loop(Module, Module:init(InitialState)).

...loop...
```

# THE BASIC SERVER

- kitty server 2 as, callback module

```
-record(cat, {name, color=green, description}).

start_link() -> my_server:start_link(?MODULE, []).

order_cat(Pid, Name, Color, Description) ->
  my_server:call(Pid, {order, Name, Color, Description}).

return_cat(Pid, Cat = #cat{}) ->
  my_server:cast(Pid, {return, Cat}).

close_shop(Pid) ->
  my_server:call(Pid, terminate).
```

# THE BASIC SERVER

- kitty server 2 as, callback module

```
%%% Server functions
init([]) -> []. %% no treatment of info here!

handle_call({order, Name, Color, Description}, From, Cats) ->
  if Cats =:= [] ->
       my_server:reply(From, make_cat(Name, Color, Description)),
       Cats;
     Cats =/= [] ->
       my_server:reply(From, hd(Cats)),
       tl(Cats)
  end;
```

# THE BASIC SERVER

- kitty server 2 as, callback module

```
handle_call(terminate, From, Cats) ->
  my_server:reply(From, ok),
  terminate(Cats).
handle_cast({return, Cat = #cat{}}, Cats) ->
  [Cat|Cats].

%%% Private functions
make_cat(Name, Col, Desc) ->
  #cat{name=Name, color=Col, description=Desc}.
terminate(Cats) ->
  [io:format("~p was set free.~n",[C#cat.name]) || C <- Cats],
  exit(normal).
```

# SPECIFIC VS GENERIC

- now you understand OTP!
  - taking all generic component
  - extracting library...
- If you're going to have larger applications then it might be worth it to separate generic parts of your code from the specific sections.
- U have many server - client..
  - if all these servers share the same common my_server abstraction, you substantially reduce that complexity

# SPECIFIC VS GENERIC

- This means you reduce a lot of time tracking and solving bugs
- when separating the generic from the specific is that we instantly made it much easier to test our individual modules.
- first kitty server need to spawn,..give right state...
- on the otherhand, requires us to run the function calls over the 'handle_call/3' and 'handle_cast/2'

# SPECIFIC VS GENERIC

- when someone optimizes that single backend
  - every process using it become faster!
  - that's what happens with the OTP framework.
- we need to also consider to our module, (kitty)
  - named processes, configuring the timeouts, adding debug information, what to do with unexpected messages, how to tie in hot code loading, handling specific errors, abstracting away the need to write most replies, handling most ways to shut a server down, making sure the server plays nice with supervisors, etc.
- the Erlang/OTP team managed to handle all of that for you with the gen_server behaviour