Erlang OTP

Programowanie w języku Erlangu

Plan

- Monitorowanie działania procesów
- Wzorce OTP
- gen_server
- supervisor
- gen_fsm, gen_event, application

Monitorowanie działania procesów

Wysoka dostępność

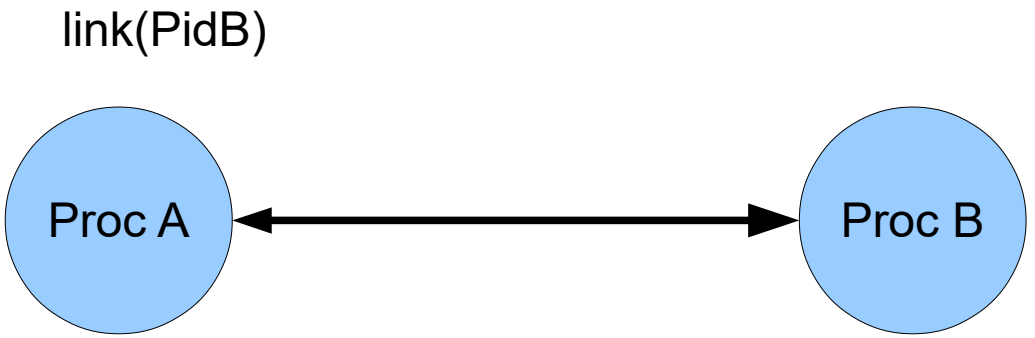
- Happy case – piszemy tylko to, co funkcja ma robić
- **"Let it crash"** approach
  - System umie sam „wstać”
  - Wykorzystajmy to do obsługi błędów
- Nie oznacza to, że nie obsługujemy żadnych błędów!

Linkowanie procesów

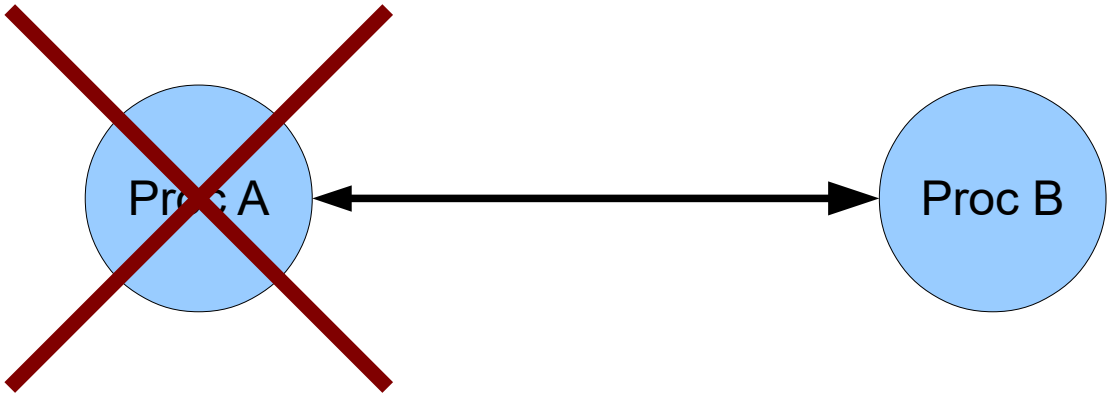
- Linkowanie – wbudowany w język mechanizm obsługi błędów
- BIF *link/1* tworzy dwukierunkowe połączenie między procesami
- Jeśli jeden z połączonych procesów zakończy działanie w sposób nienormalny, do połączonego procesu zostanie wysłany sygnał zakończenia

```
link(Pid).
```

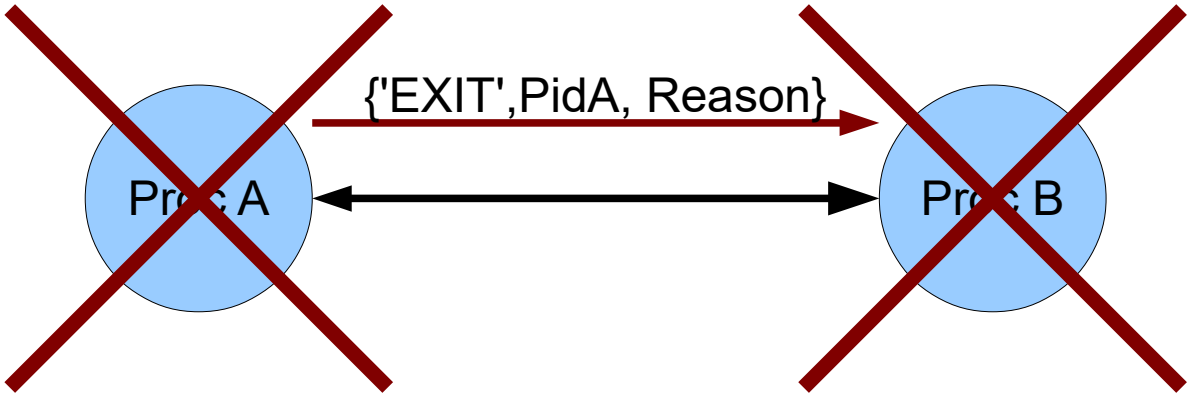
Linkowanie



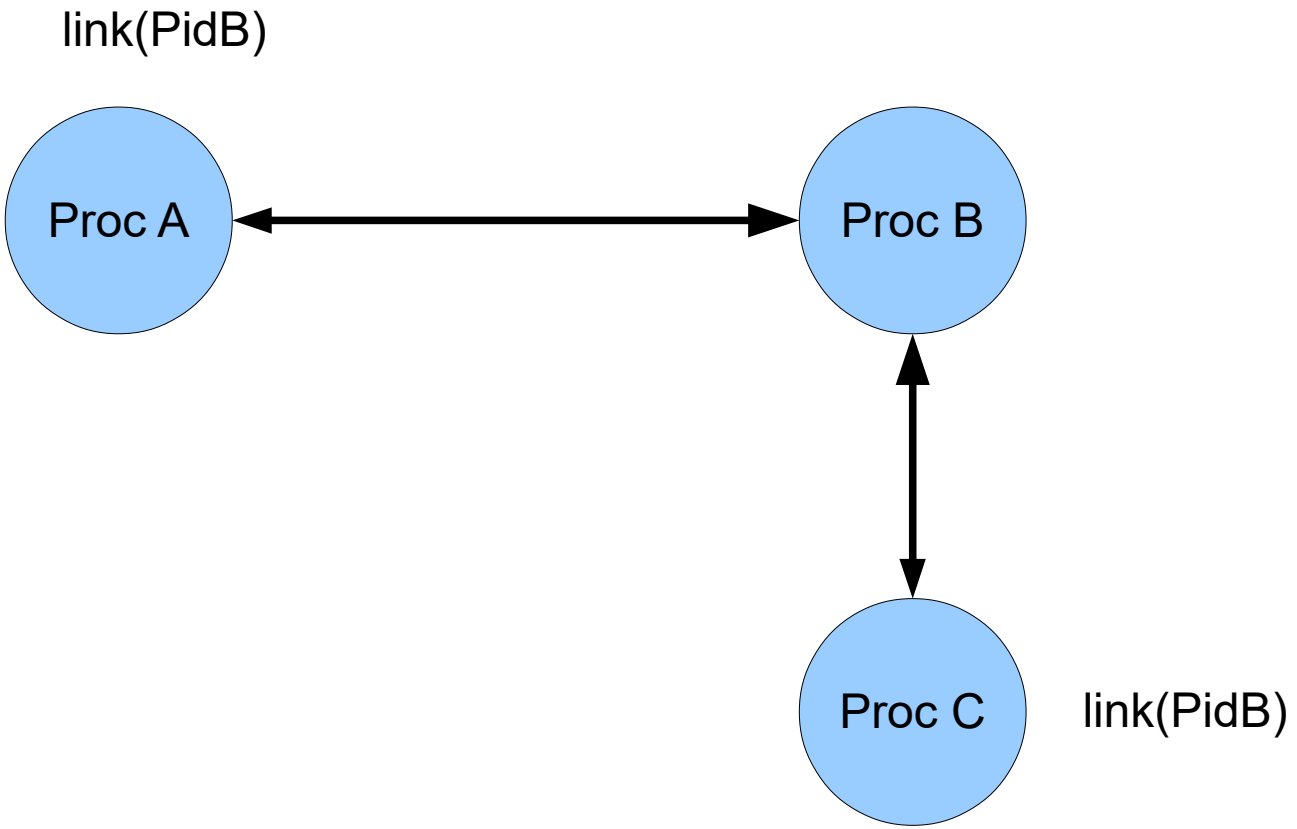
Linkowanie



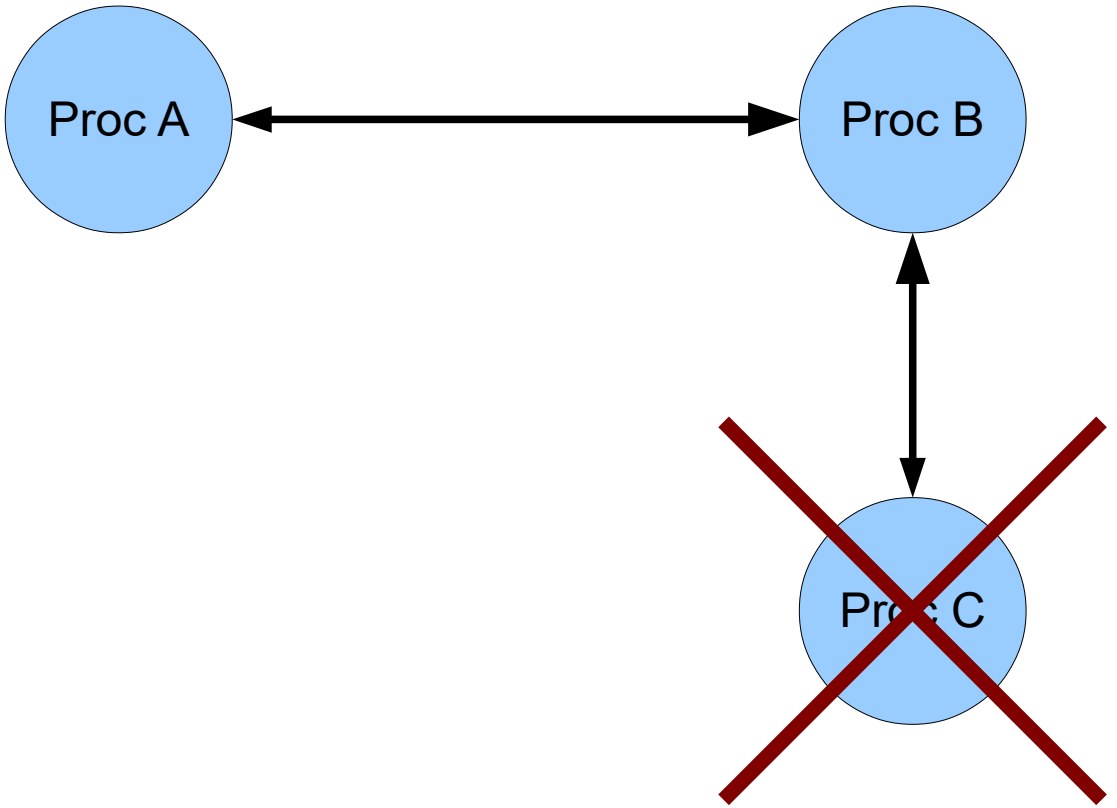
Linkowanie



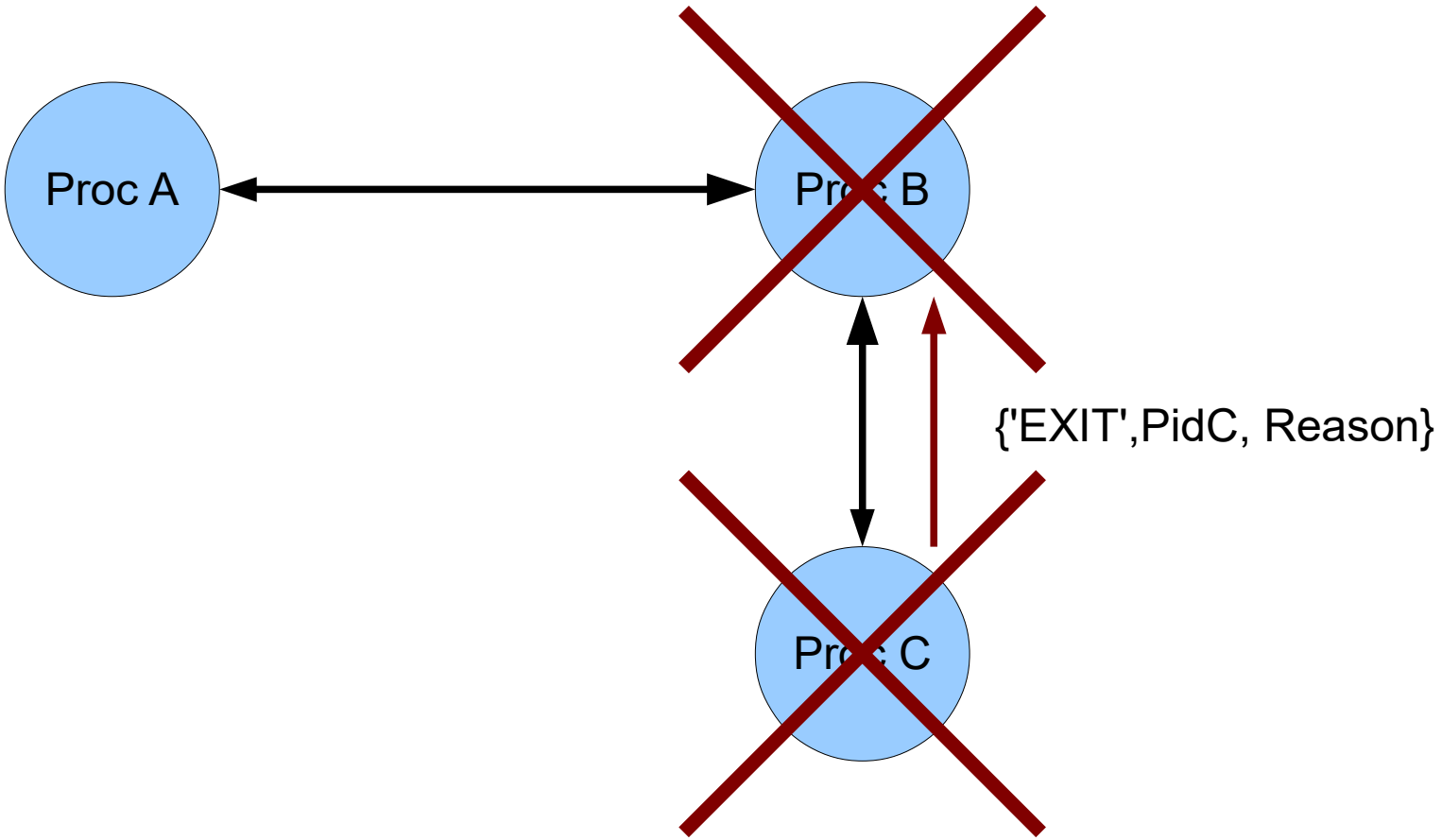
Linkowanie



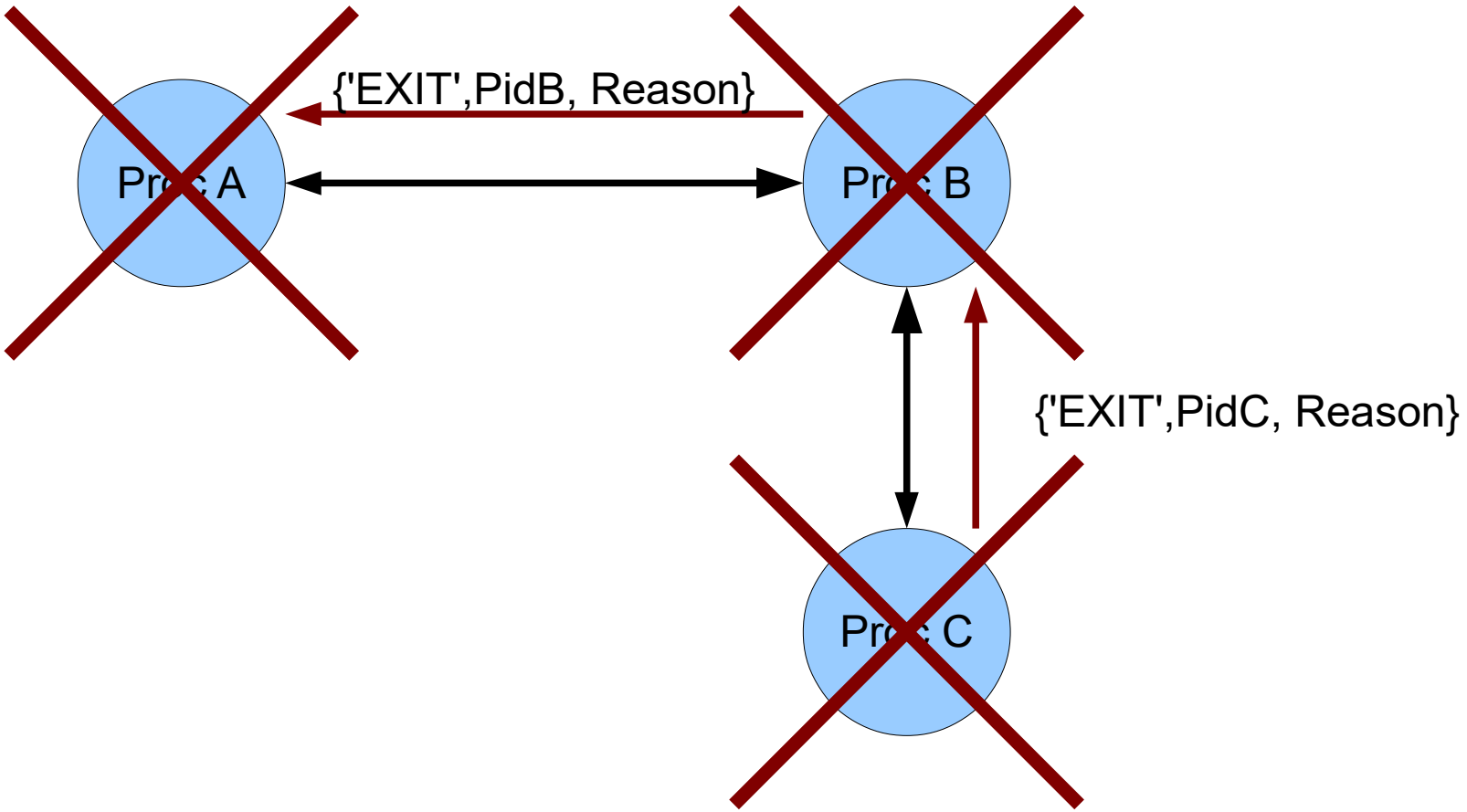
Linkowanie



Linkowanie



Linkowanie



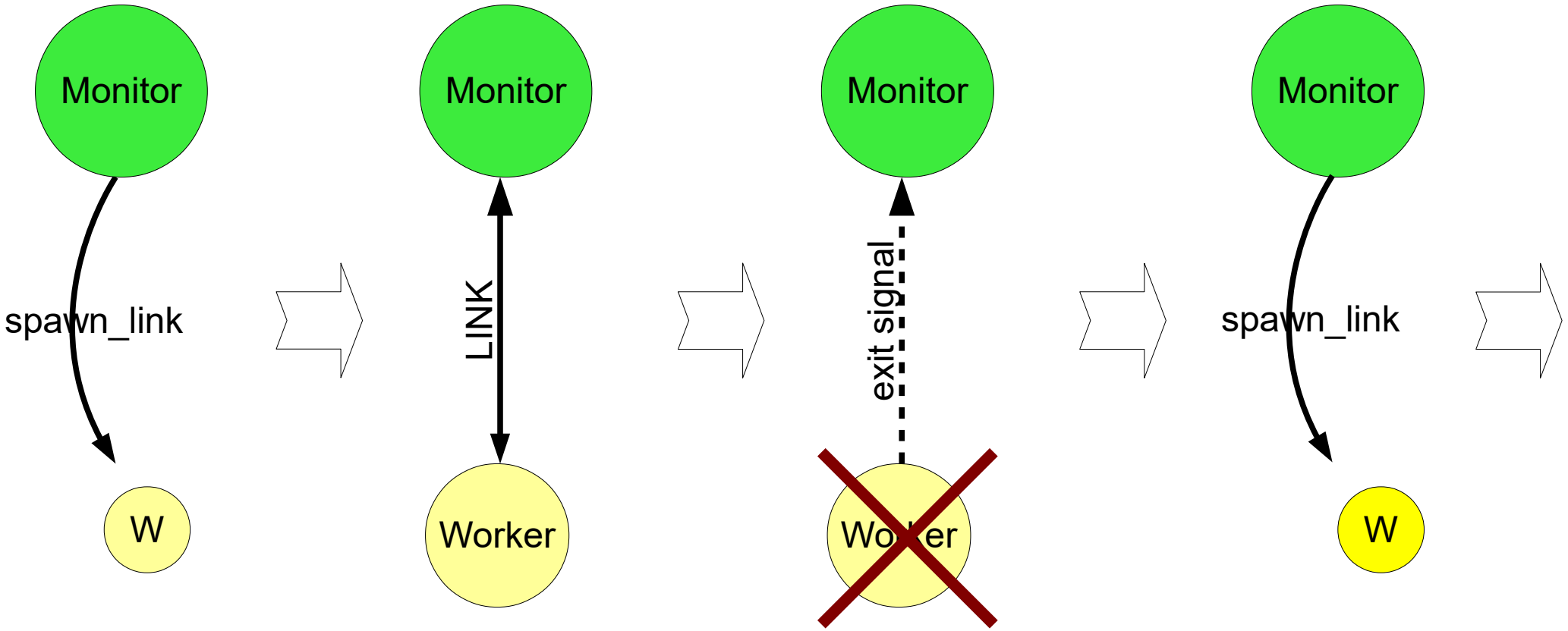
Przechwytywanie sygnałów zakończenia

- Proces może zadeklarować, że chce obsługiwać sygnały zakończenia
- Do procesu przechwytującego zostanie przekazana wiadomość *{'EXIT', Pid, Reason}*
- Przechwycony sygnał nie jest dalej propagowany

```
process_flag(trap_exit, true).
```

Monitorowanie procesów

- Monitor process, worker process



Problem natychmiastowego zakończenia

- Linkowanie służy do kontrolowania funkcjonowania utworzonego procesu
- Proces tworzony może jednak zakończyć działanie zanim proces go tworzący zdąży się z nim zlinkować
- Rozwiązanie: *spawn_link(module, function, args)*

```
start_and_link() ->
  Pid = spawn(mod, crash, []),
  do_sth(),
  link(Pid).


crash() →
  1 / 0.
```

Żądanie zakończenia procesu

- Zakończenie bieżącego procesu: *exit(Reason)*
- Zakończenie innego procesu: *exit(Pid, Reason)*

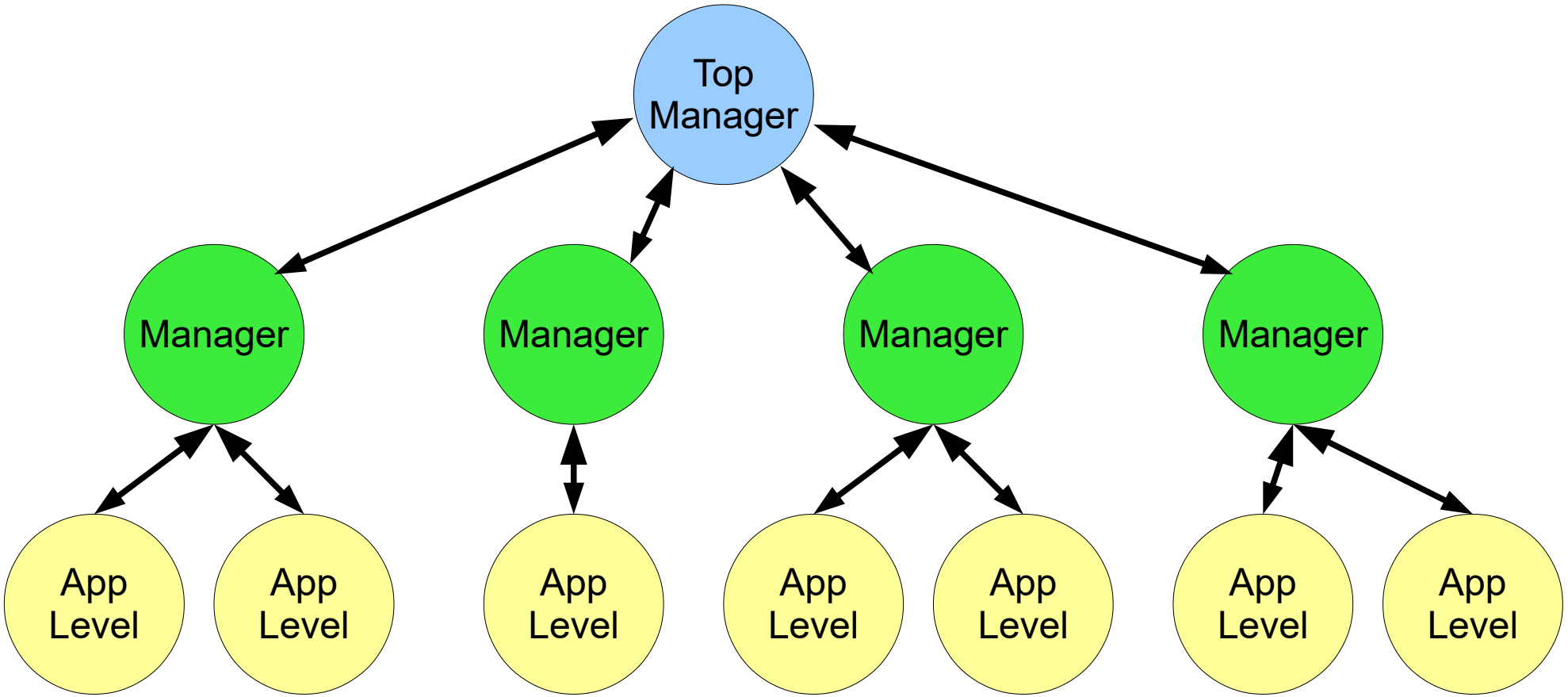
Propagacja i przechwytywanie

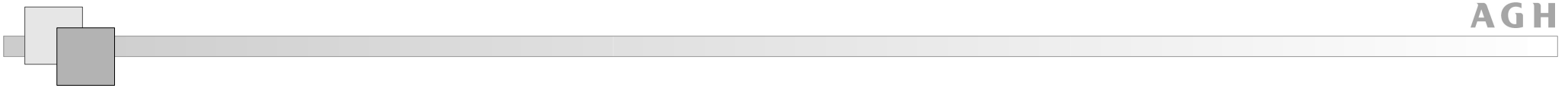
trap_exit / Przyczyna	**tak**	**nie**
Normal	Otrzymuje wiadomość: *{'EXIT', Pid, Przyczyna}*	Nic się nie dzieje
Other	Otrzymuje wiadomość: *{'EXIT', Pid, Przyczyna}*	Kończy działanie z przyczyną: *Other*
Kill	Kończy działanie z przyczyną: *killed*	Kończy działanie z przyczyną: *killed*

- Jeśli przyczyną jest atom *kill*, zakończenia nie da się przechwycić

Programowanie systemów wysokiej dostępności

- Hierarchia nadzorców nad warstwą aplikacji
- Podejście „happy case”





Wzorce OTP

Open Telecom Platform

- **Open**: pozwala na integrację z innymi technologiami:
  - Jinterface, erlinterface, TCP, UDP, Corba, ...
- **Telecom**: dedykowana do tworzenia rozwiązań dla zastosowań telekomunikacyjnych:
  - Masowo współbieżne
  - Odporne na awarie
  - Działające w czasie rzeczywistym
- **Platform**: dostarcza:
  - Biblioteki
  - Aplikacje
  - Narzędzia
  - Wzorce projektowe

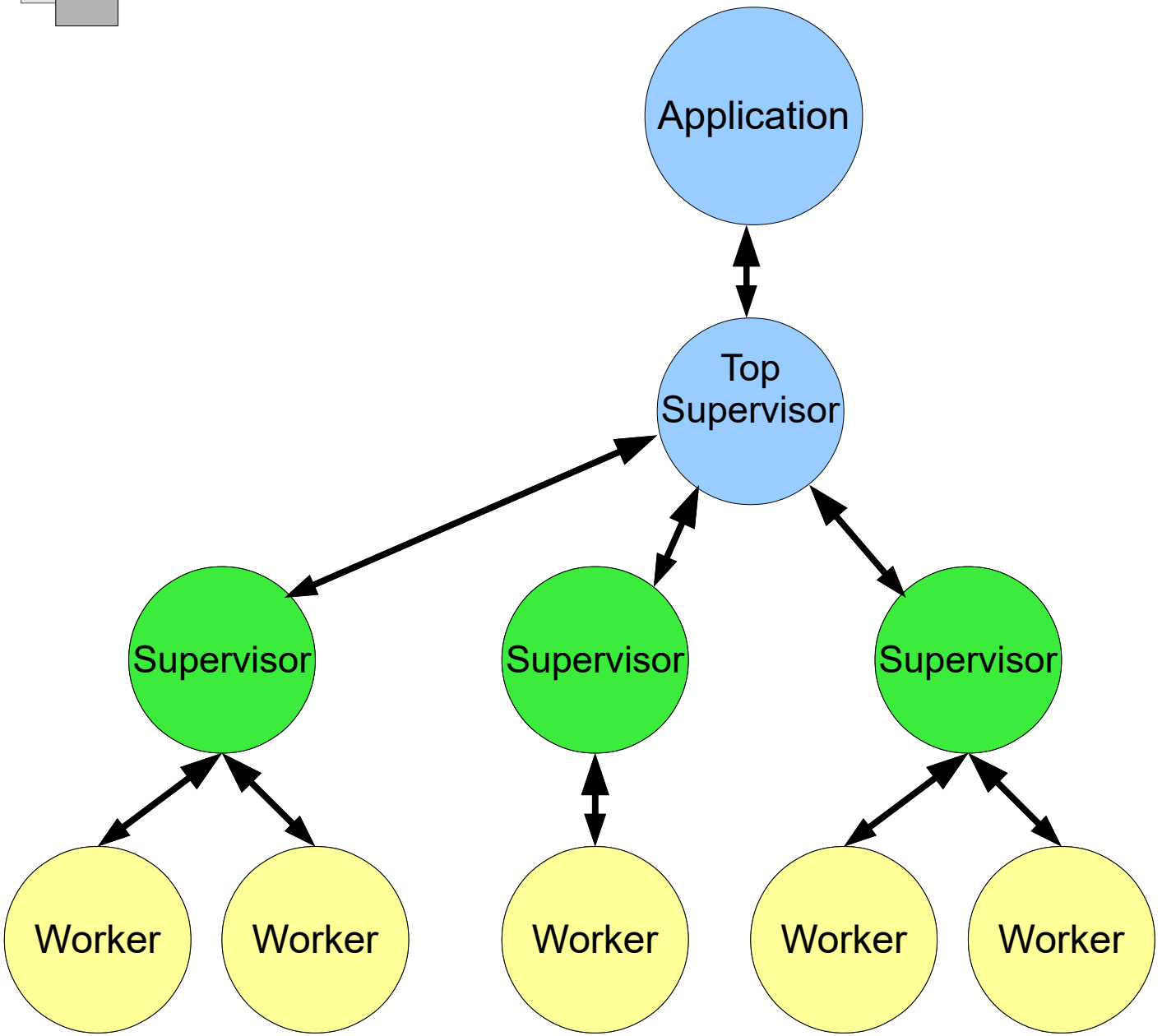
Wzorce OTP

- Zasady tworzenia kodu
- **Szkielety programów, zawierające ogólny kod**

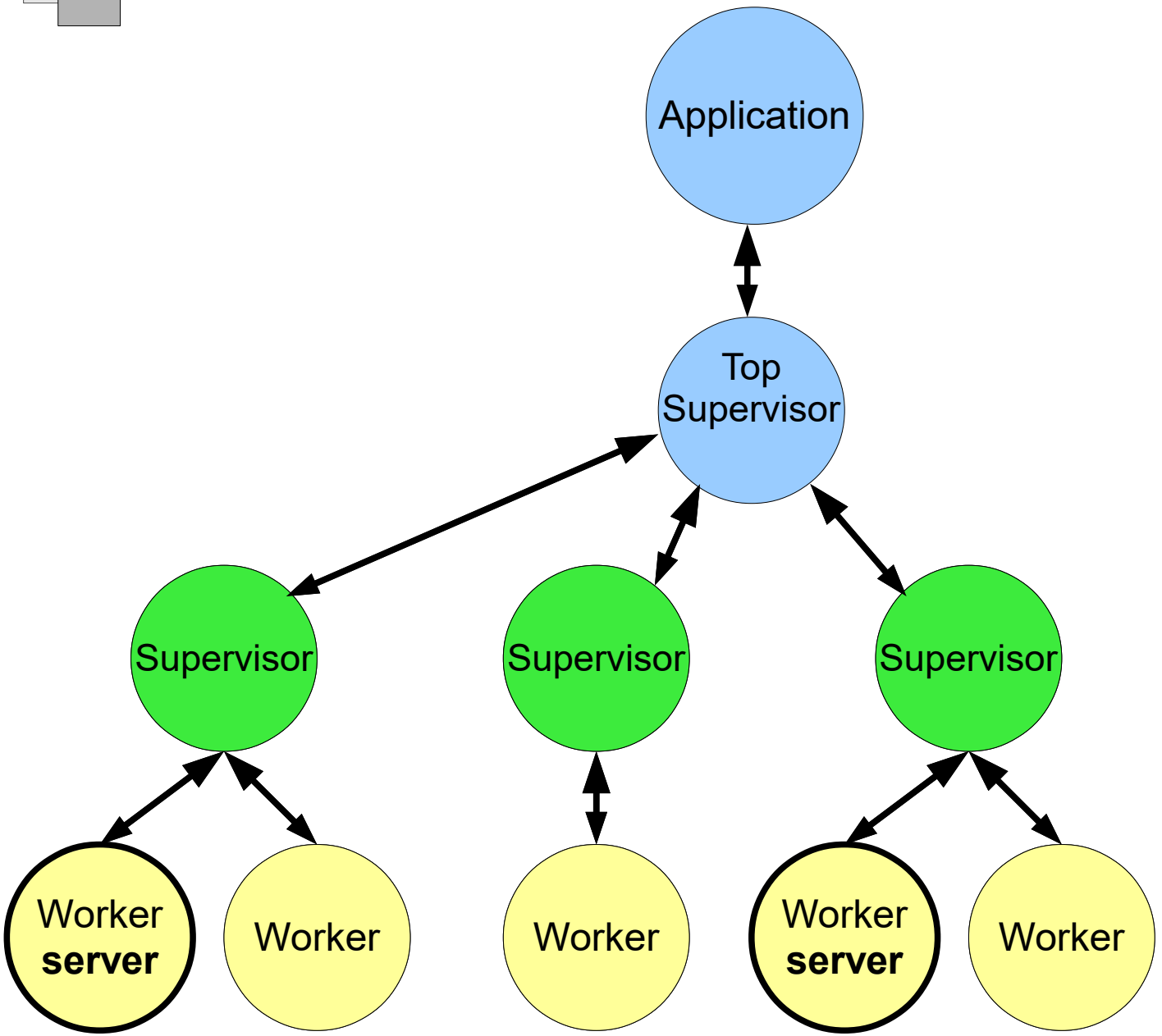
Szkielety – beahviours

- Rozwiązanie typowego problemu
- Zawiera wbudowane mechanizmy obsługi błędów
- Zapewnia logowanie, aktualizowanie, ...
- Tworzy zrozumiałą strukturę aplikacji

Struktura aplikacji

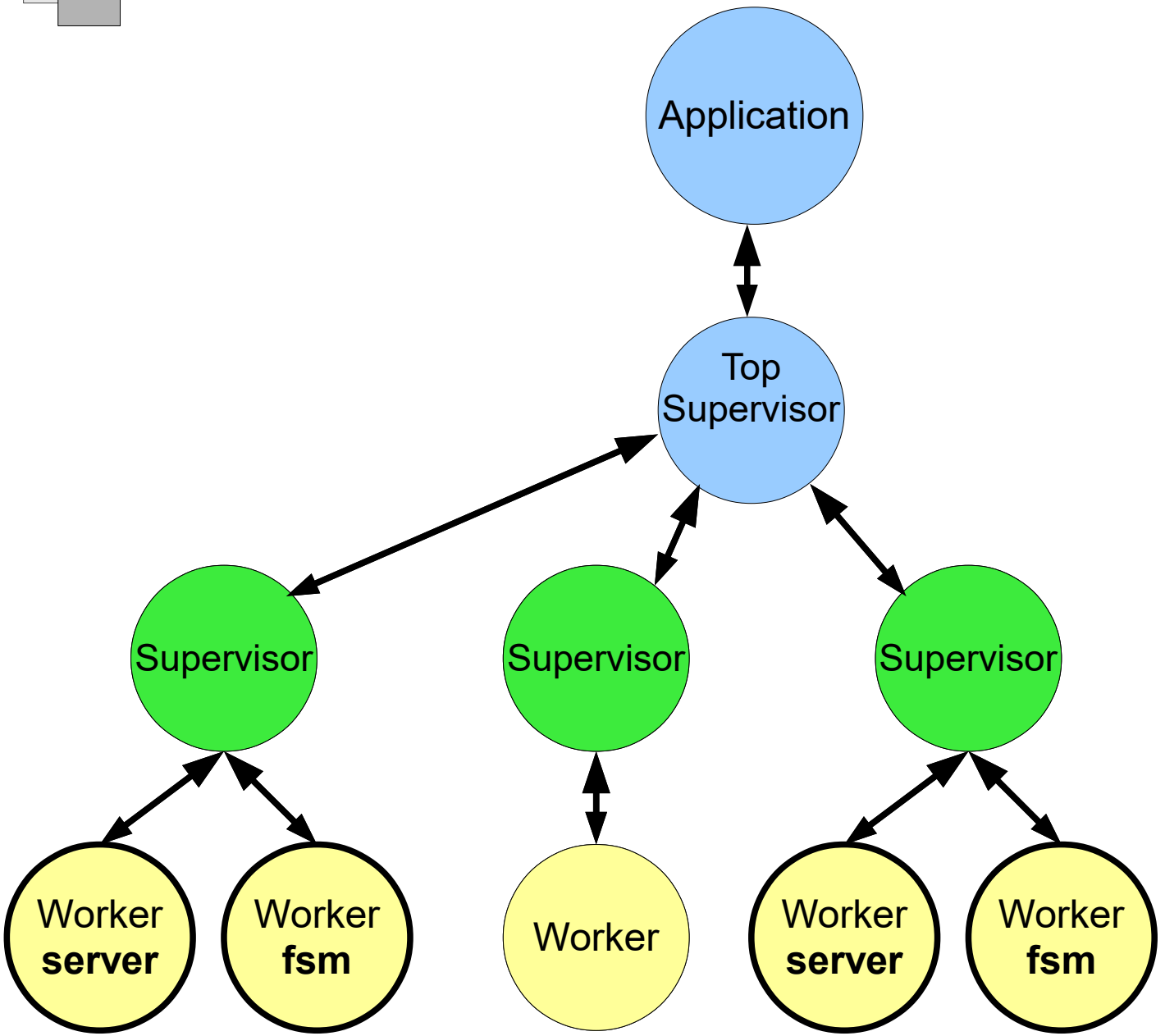


Struktura aplikacji



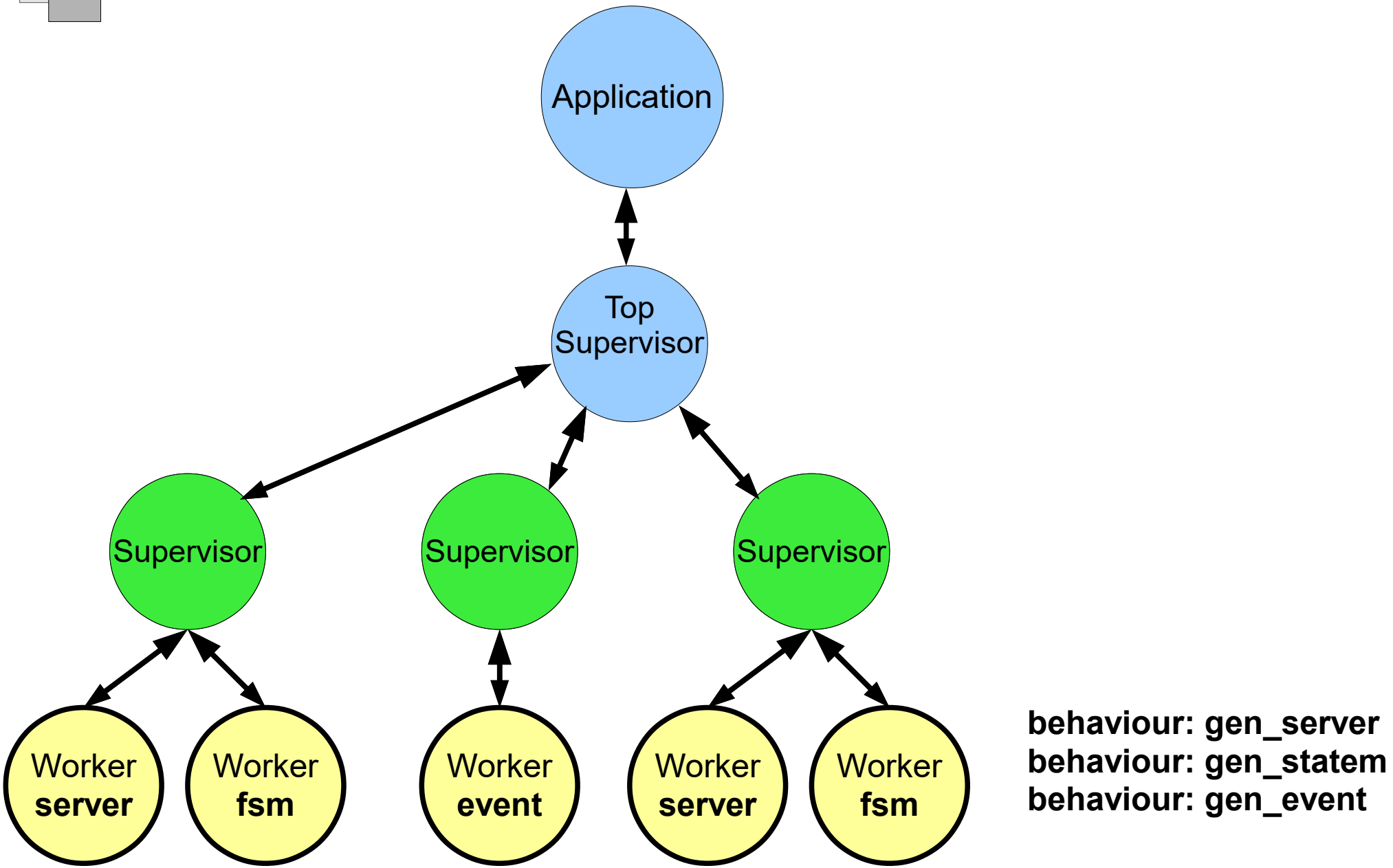
behaviour: gen_server

Struktura aplikacji



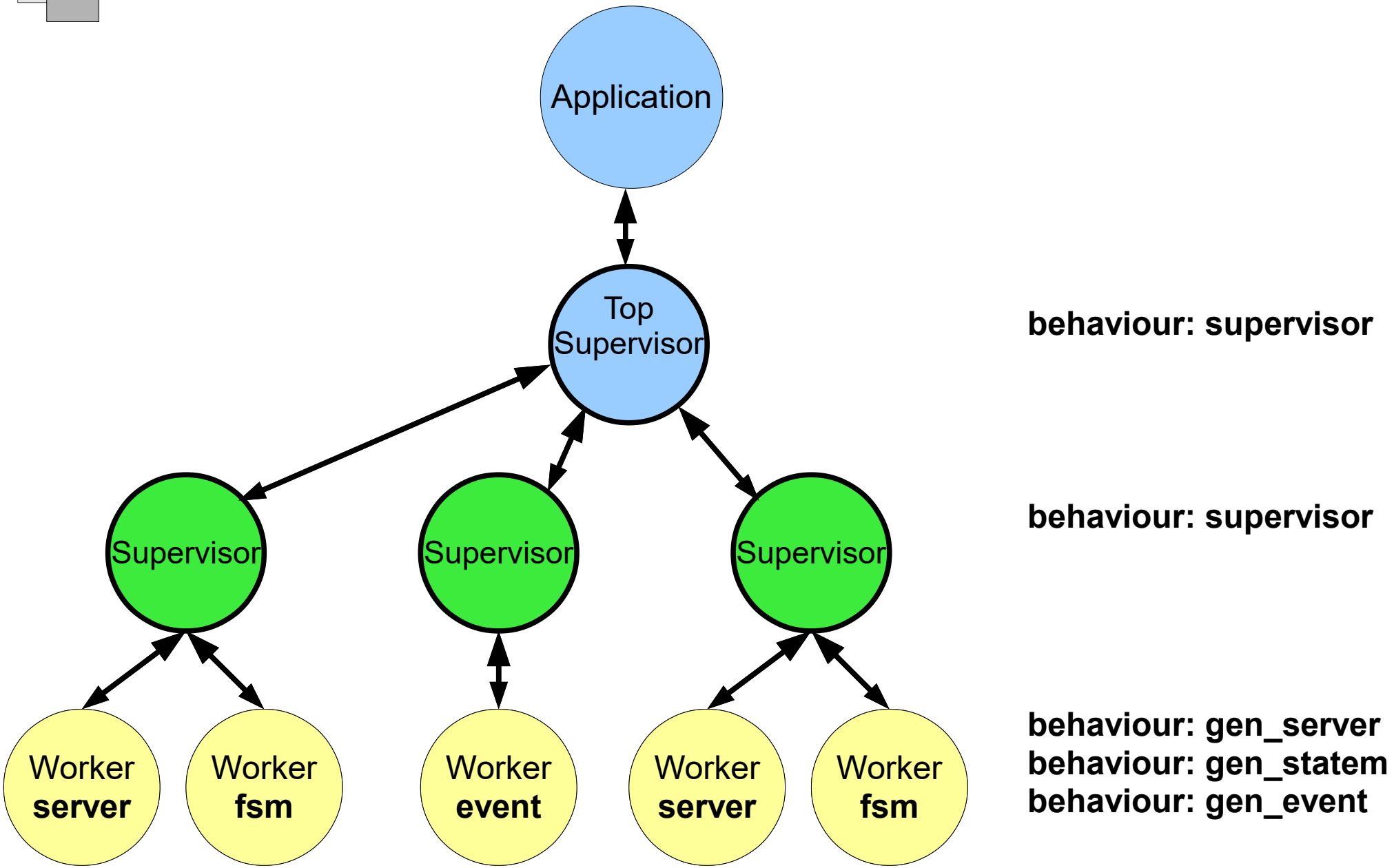
behaviour: gen_server
behaviour: gen_statem

Struktura aplikacji

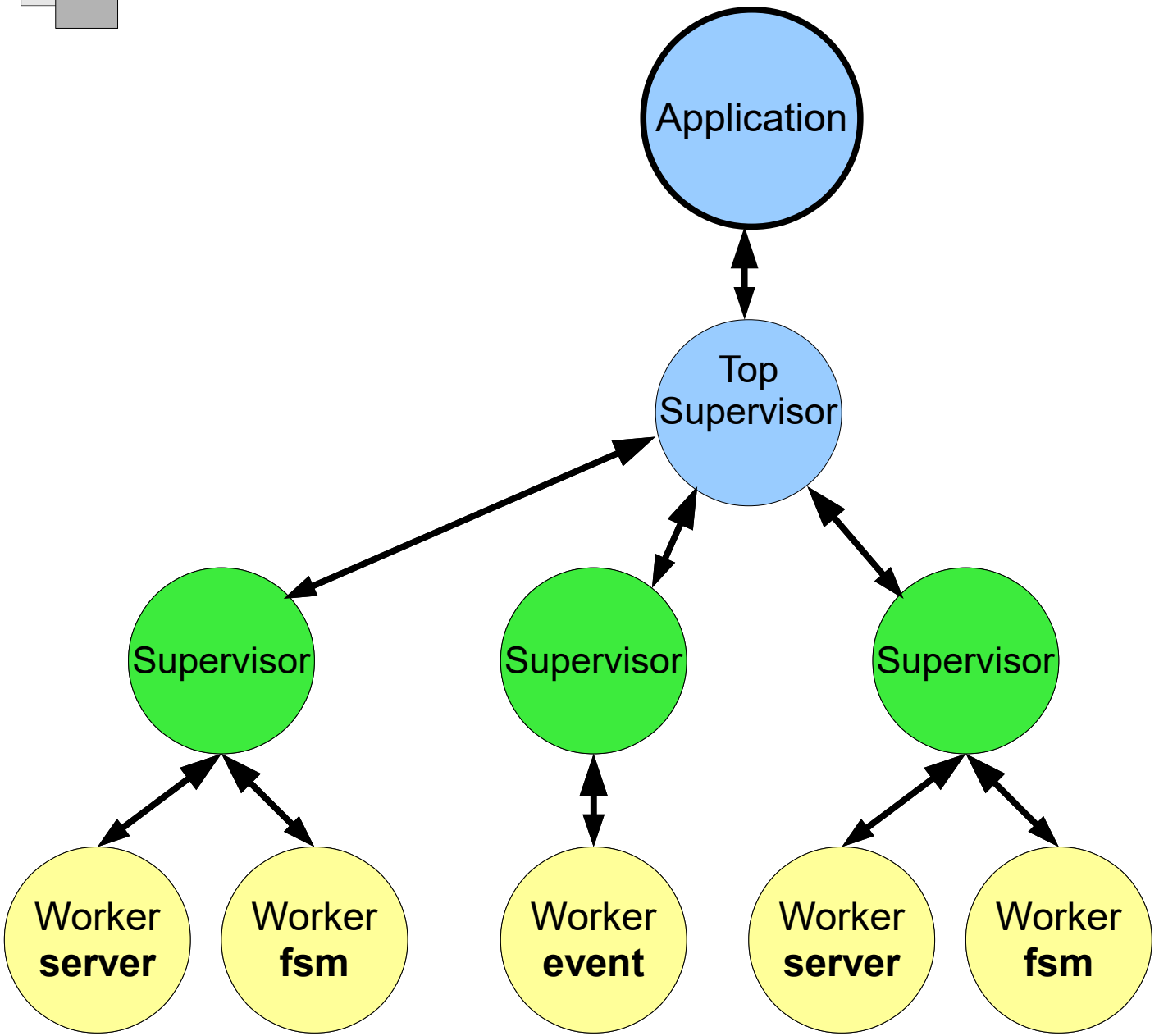


behaviour: gen_server
behaviour: gen_statem
behaviour: gen_event

Struktura aplikacji



Struktura aplikacji

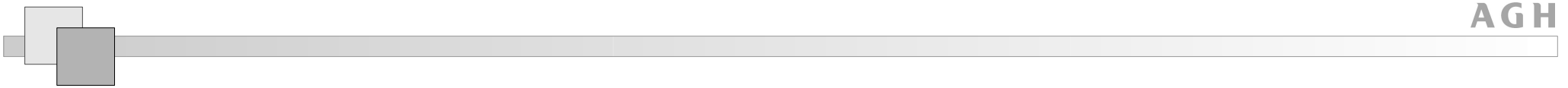


behaviour: application

behaviour: supervisor

behaviour: supervisor

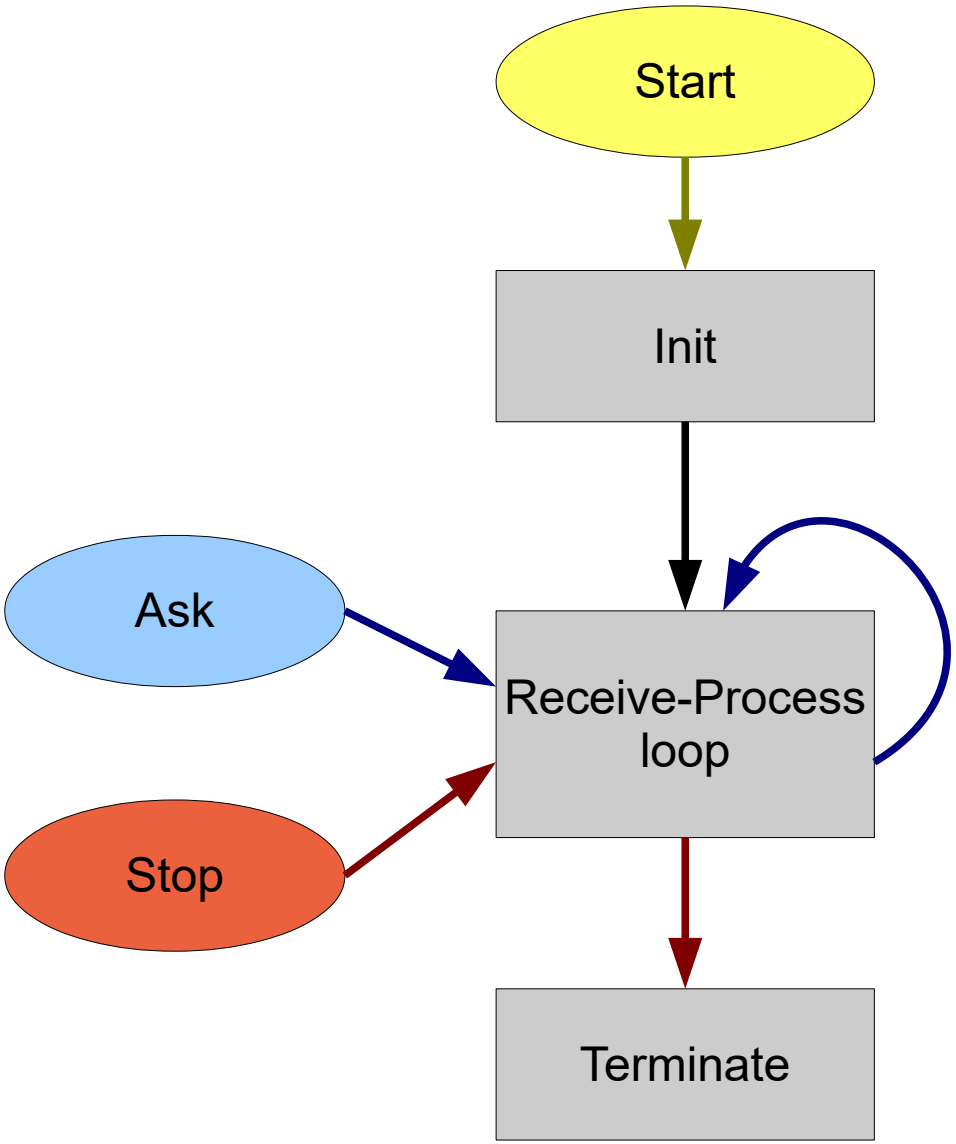
behaviour: gen_server
behaviour: gen_statem
behaviour: gen_event



gen_server

Serwer – podstawowy wzorzec

```
start(Args) ->
    spawn(server, init, Args).


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


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


terminate() ->
    ok.
```



gen_server

- Implementuje zachowanie typu klient-serwer
- Kod ogólny jest wykorzystywany przez moduł implementujący kod szczególny - tzw callback module

```
-module(var_server).
-behaviour(gen_server).

-export([start_link/0, .....

start_link() -> ........
```

Uruchamianie serwera

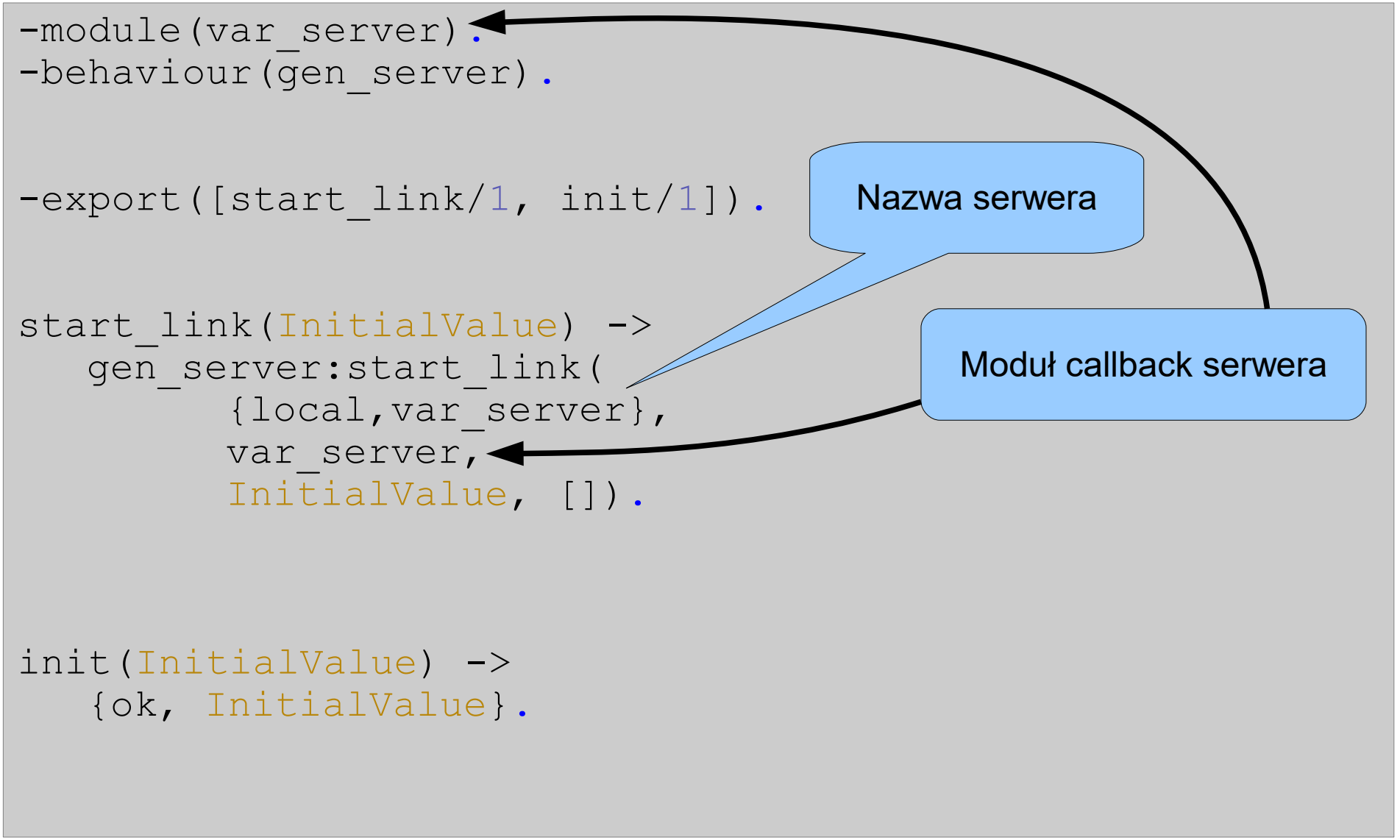
- Serwer uruchamia funkcja:

```
gen_server:start_link(
          {local,Name},
          Module,
          Arguments,
          Options).
```

- Name – nazwa, pod którą będzie zarejestrowany
- Module – moduł implementując funkcje callback
- Arguments – przekazywane do funkcji init/1
- Options – dodatki

Uruchamianie serwera zmiennej

```
-module(var_server).
-behaviour(gen_server).


-export([start_link/1, init/1]).


start_link(InitialValue) ->
   gen_server:start_link(
         {local,var_server},
         var_server,
         InitialValue, []).



init(InitialValue) ->
   {ok, InitialValue}.
```



Uruchamianie serwera zmiennej

```
-module(var_server).
-behaviour(gen_server).


-export([start_link/1, init/1]).


start_link(InitialValue) ->
   gen_server:start_link(
         {local,var_server},
         var_server,
         InitialValue, []).



init(InitialValue) ->
   {ok, InitialValue}.
```

Uruchamianie serwera zmiennej

```
-module(var_server).
-behaviour(gen_server).


-export([start_link/1, init/1]).


start_link(InitialValue) ->
   gen_server:start_link(
         {local,var_server},
         var_server,
         InitialValue, []).



init(InitialValue) ->
   {ok, InitialValue}.
```

Dane w pętli serwera

Zapytania synchroniczne

- Wysłanie zapytania synchronicznego:

```
gen_server:call(Name, Message) -> Value.
```

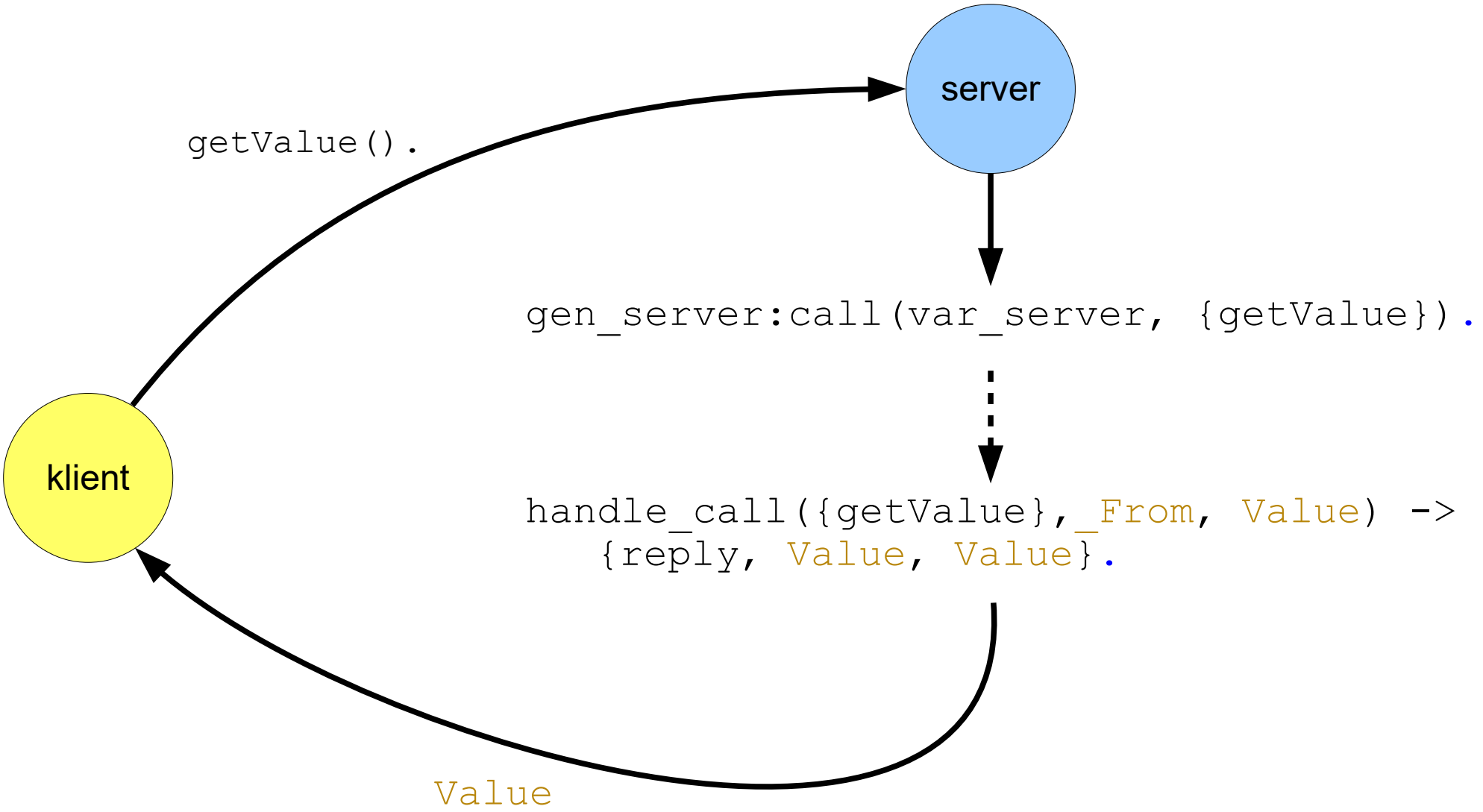
- Powoduje wywołanie funkcji callback:

```
handle_call(Message, From, LoopData) ->
   {reply, Reply, NewLoopData}.
```

Zapytania synchroniczne o wartość zmiennej

```
-module(var_server).
-behaviour(gen_server).
-version('1.0').

-export([start_link/1, init/1, handle_call/3]).

-export([getValue/0]).


%% user interface
getValue() ->
   gen_server:call(var_server, {getValue}).


%% callbacks
handle_call({getValue},_From, Value) ->
   {reply, Value, Value}.
```

Zapytania synchroniczne o wartość zmiennej



Zapytania asynchroniczne

- Wysłanie zapytania asynchronicznego:

```
gen_server:cast(Name, Message) -> ok.
```

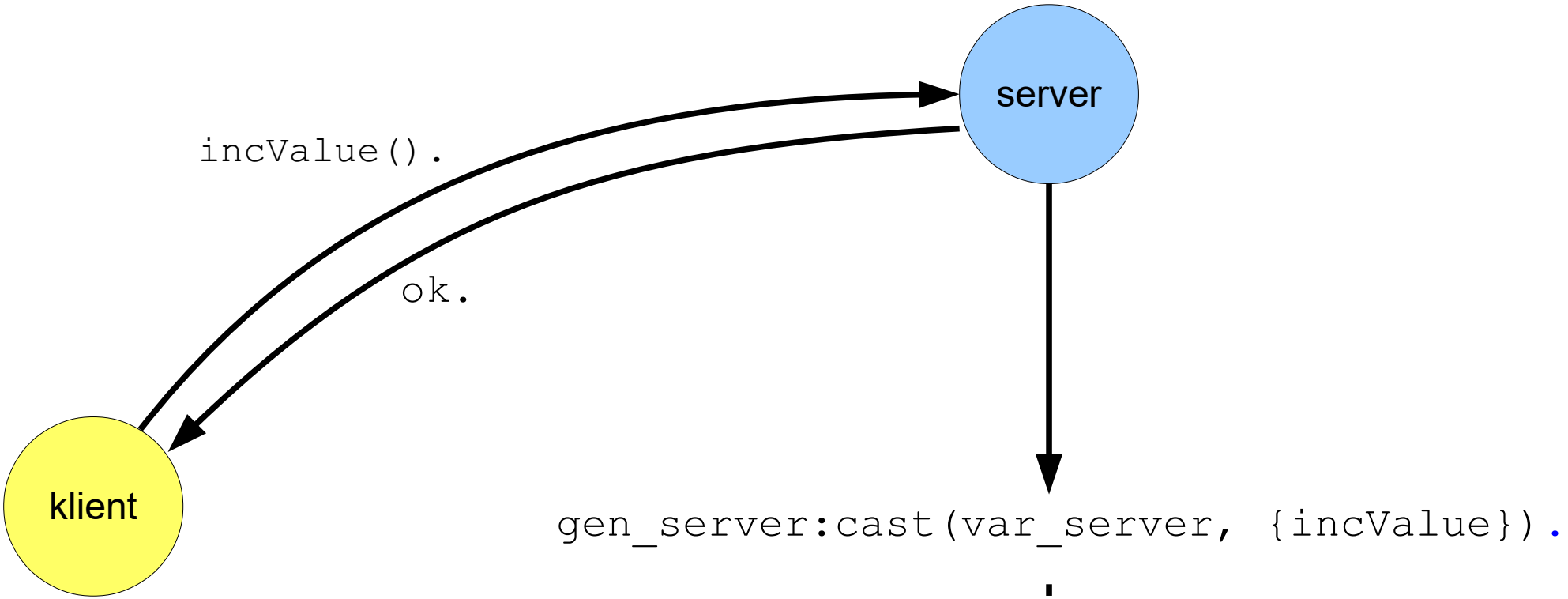
- Powoduje wywołanie funkcji callback:

```
handle_cast(Message, LoopData) ->
   {noreply, NewLoopData}.
```

Zapytania asynchroniczne – inkrementacja wartości

```
-module(var_server).
-behaviour(gen_server).
-version('1.0').

-export([start_link/1, init/1, handle_cast/2]).

-export([incValue/0]).


%% user interface
incValue() ->
   gen_server:cast(var_server, {incValue}).


%% callbacks
handle_cast({incValue}, Value) ->
   {noreply, Value+1}.
```

Zapytania asynchroniczne – inkrementacja wartości



Kończenie działania serwera

- Zwrócenie określonej wartości z funkcji callback:

```
init/1 ->          {stop, Reason, LoopData}
handle_call/3 ->   {stop, Reason, Reply, NewLoopData}
handle_cast/2 ->   {stop, Reason, NewLoopData}
```

- Wywoływana jest funkcja callback:

```
terminate(Reason, LoopData) ->
   ok.
```

Kończenie działania serwera zmiennej

```
-module(var_server).
-behaviour(gen_server).
-version('1.0').

-export([start_link/1, handle_cast/2, terminate/2]).

-export([stop/0]).

%% user interface
stop() ->
   gen_server: cast(var_server, stop).

%% callbacks
handle_cast(stop, Value) ->
   {stop, normal, Value}.

terminate(Reason, Value) ->
   io:format("Server: exit with value ~p~n", [Value]),
   Reason.
```

Kończenie w wyniku błędu

- terminate/2 będzie zawołane także w wyniku błędu
- Przyczyna zostanie przekazana do terminate/2:

```
terminate(Reason, LoopData) ->
```

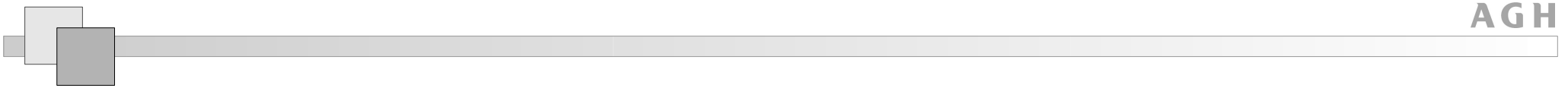
Pozostałe funkcje callback

- Wiadomości niezgodne z protokołem serwera zostaną przekazane do funkcji

```
handle_info(Message, LoopData) -> ...
```

- Zmiana wersji aplikacji może zostać obsłużona w funkcji:

```
code_change(OldVsn, State, Extra) -> ...
   {ok, NewState}.
```



supervisor

supervisor

- Korzysta z mechanizmów linkowania procesów
- Pozwala na startowanie procesów obserwowanych
- Nadzoruje ich działanie
- W razie potrzeby przywraca działanie

```
-module(atm_sups).
-version('1.0').
-behaviour(supervisor).

-export([start_link/1, init/1, stop/1]).

start_link() -> ........
```

Uruchamianie supervisora

- Supervisor jest uruchamiany przez funkcję:

```
supervisor:start_link(
         {local,Name},
         Module,
         Arguments).
```

- Name – nazwa, pod którą będzie zarejestrowany
- Module – moduł implementując funkcje callback
- Arguments – przekazywane do funkcji init/1

Supervisor dla serwera

```
-module(var_supervisor).
-behaviour(supervisor).


-export([start_link/1, init/1]).

start_link(InitValue) ->
   supervisor:start_link({local, varSupervisor},
                         ?MODULE, InitValue).


init(InitValue) ->
  {ok, {
      {one_for_all, 2, 3},
      [  {var_server,
         {var_server, start_link, [InitValue]},
         permanent, brutal_kill, worker, [var_server]}
      ]}
   }.
```

Funkcja init/1

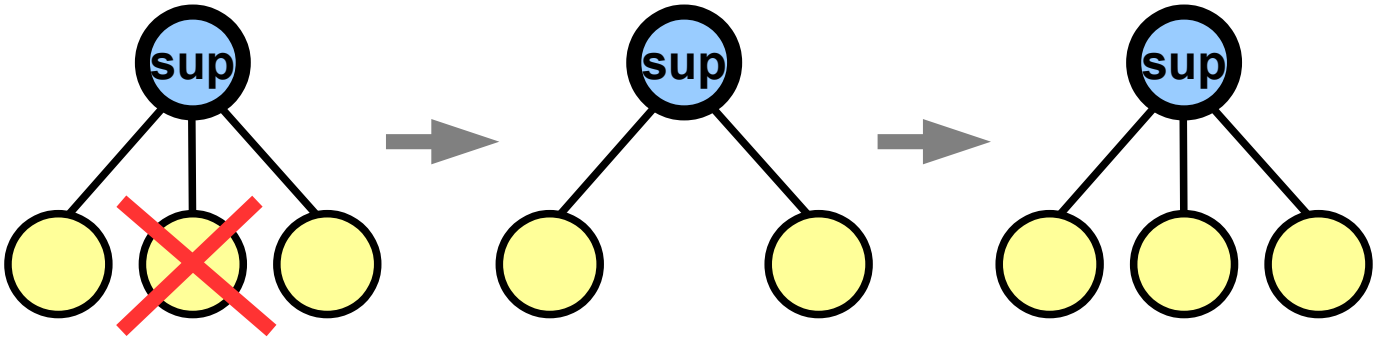
- Wołana przy uruchamianiu supervisora
- Musi dostarczyć informacje niezbędne do utworzenia zadzorcy

```
init(InitValue) ->
  {ok, SupervisorSpec}
```

- SupervisorSpec = {**RestartTuple**, ChildSpecList}

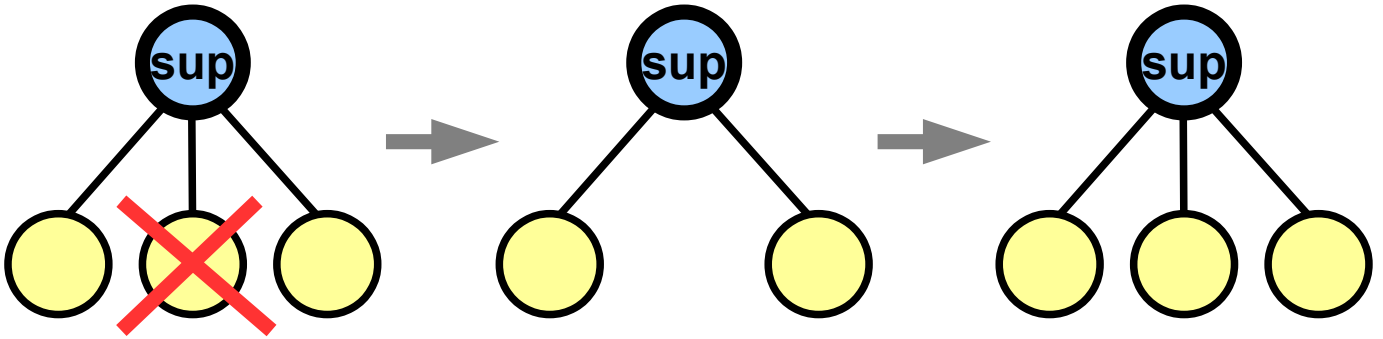
RestartTuple = {RestartType, MaxRestart, MaxTime}

one_for_one

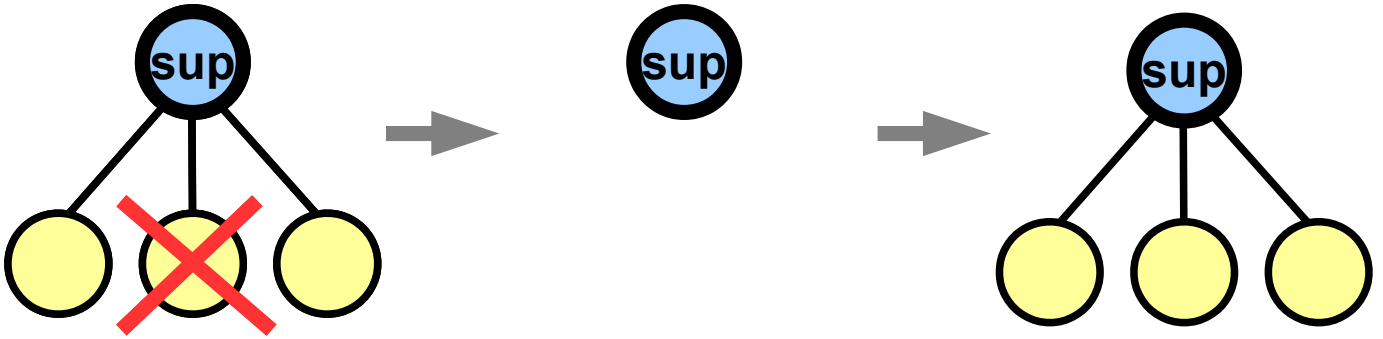


RestartTuple = {RestartType, MaxRestart, MaxTime}

one_for_one

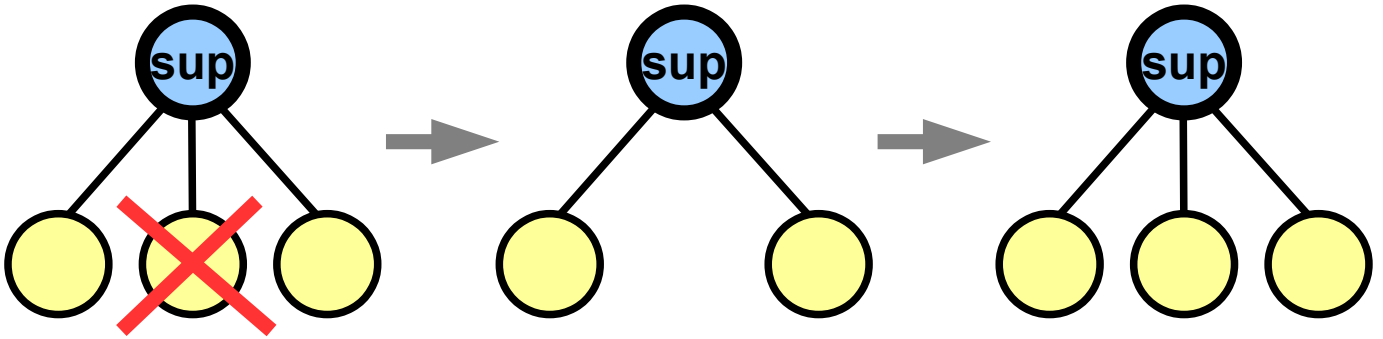


one_for_all

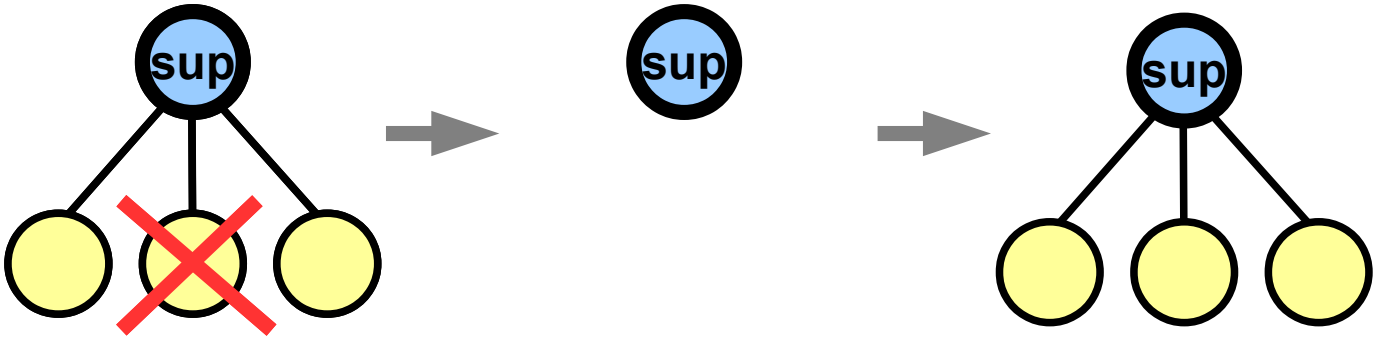


RestartTuple = {RestartType, MaxRestart, MaxTime}

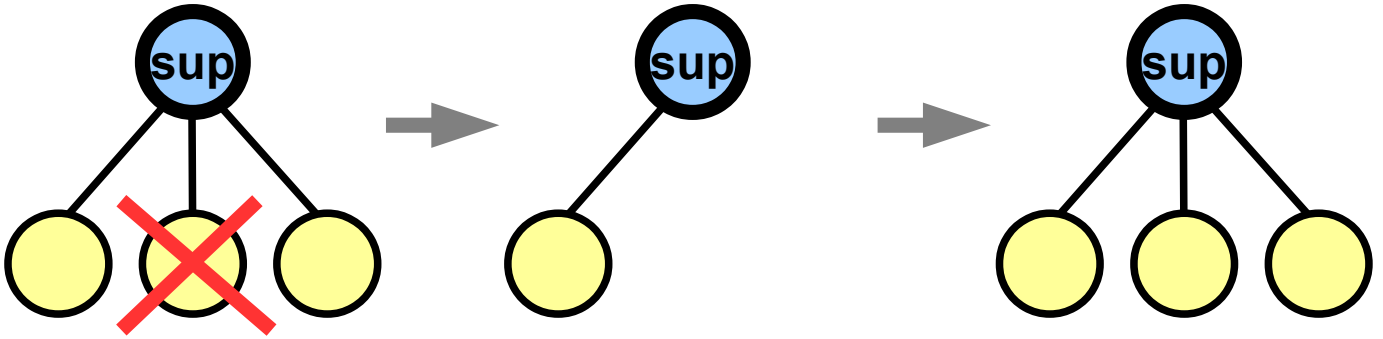
one_for_one



one_for_all



rest_for_one



RestartTuple = {RestartType, MaxRestart, MaxTime}

- MaxRestart – maksymalna liczba restartów, jaka może zajść w MaxTime

- MaxTime – jeśli MaxRestart zostanie przekroczony w ciągu MaxTime, supervisor kończy działanie

Funkcja init/1

- Wołana przy uruchamianiu supervisora
- Musi dostarczyć informacje niezbędne do utworzenia zadzorcy

```
init(InitValue) ->
  {ok, SupervisorSpec}
```

- SupervisorSpec = {RestartTuple, **ChildSpecList**}

ChildSpecList

- Lista elementów specyfikujących nadzorowane procesy
- ChildSpec = {Id,StartFunc,Restart,Shutdown,Type,Modules}

  - Id – nazwa do identyfikacji przez supervisora
  - StartFunc = {M, F, A}
  - Restart = permanent | transient | temporary
    - permanent - zawsze
    - transient – tylko po nienormalnym zakończeniu
    - temporary – nie jest restartowany

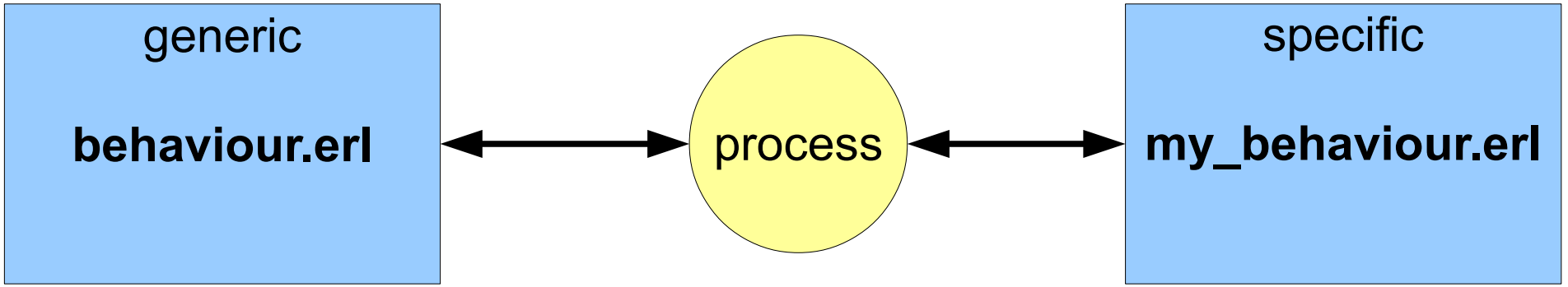
ChildSpecList

- Lista elementów specyfikujących nadzorowane procesy
- ChildSpec = {Id,StartFunc,Restart,Shutdown,Type,Modules}
  - Shutdown = int()>=0 | infinity | brutal_kill
    - Czas, jaki proces może poświęcić na zakończenie
    - infinity – nieskończoność
    - brutal_kill – nie będą wołane funkcje zakończenia
  - Type = worker | supervisor
  - Modules – lista callback modułów używanych przez proces – potrzebne przy aktualizacjach.

Supervisor dla serwera

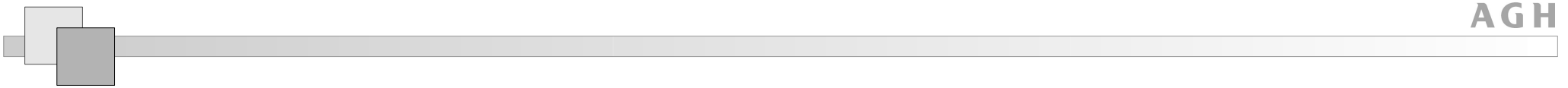
```
-module(var_supervisor).
-behaviour(supervisor).


-export([start_link/1, init/1]).

start_link(InitValue) ->
   supervisor:start_link({local, varSupervisor},
                         ?MODULE, InitValue).


init(InitValue) ->
  {ok, {
      {one_for_all, 2, 3},
      [  {var_server,
         {var_server, start_link, [InitValue]},
         permanent, brutal_kill, worker, [var_server]}
      ]}
   }.
```

Ogólna struktura wzorców OTP



- Startowanie
- Monitorowanie
- Wywoływanie funkcji obsługi zdarzeń
- Zatrzymywanie
- Sprzątanie

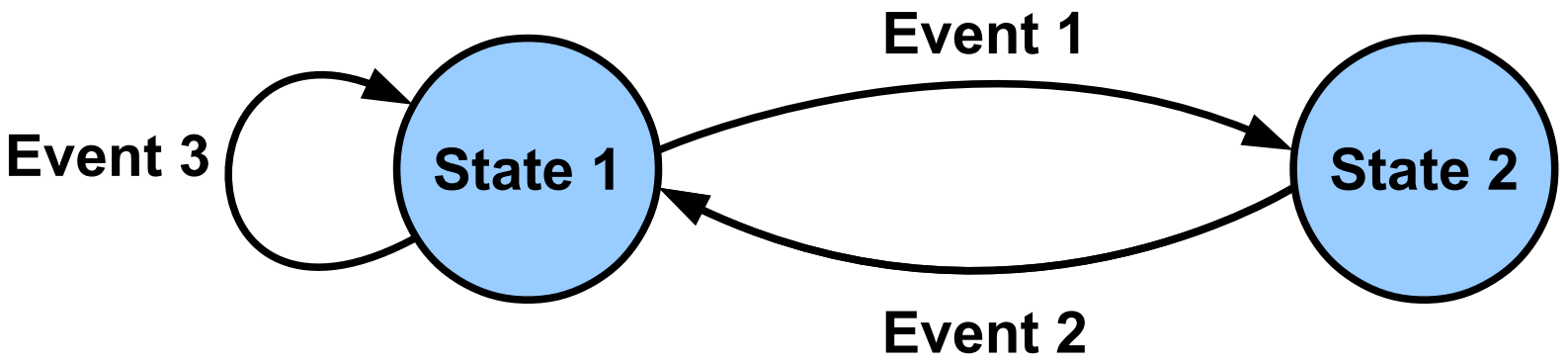
- Inicjalizcja
- Obsługa zdarzeń
- Zakańczanie



gen_statem, gen_event, application

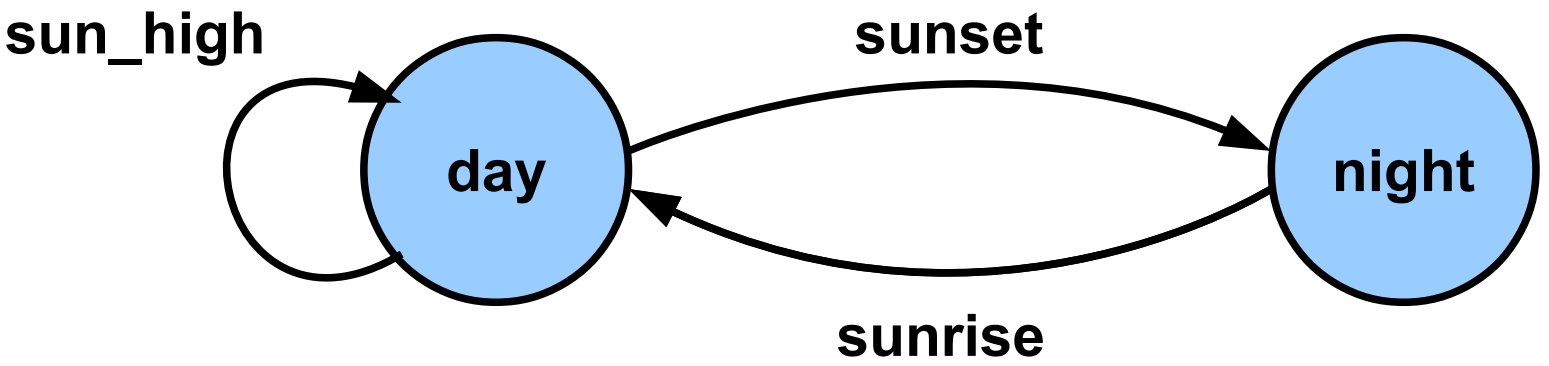
Automat skończony - Finite State Machine

- Skończony zbiór stanów
- Skończony zbiór możliwych przejść między stanami



Automat skończony w Erlangu: gen_statem

- Każdy stan to wywołanie określonej funkcji
- Każde przejście to przychodząca wiadomość




```
day() ->
   receive
      sunset -> night();
      sun_high -> day()
   end.
```

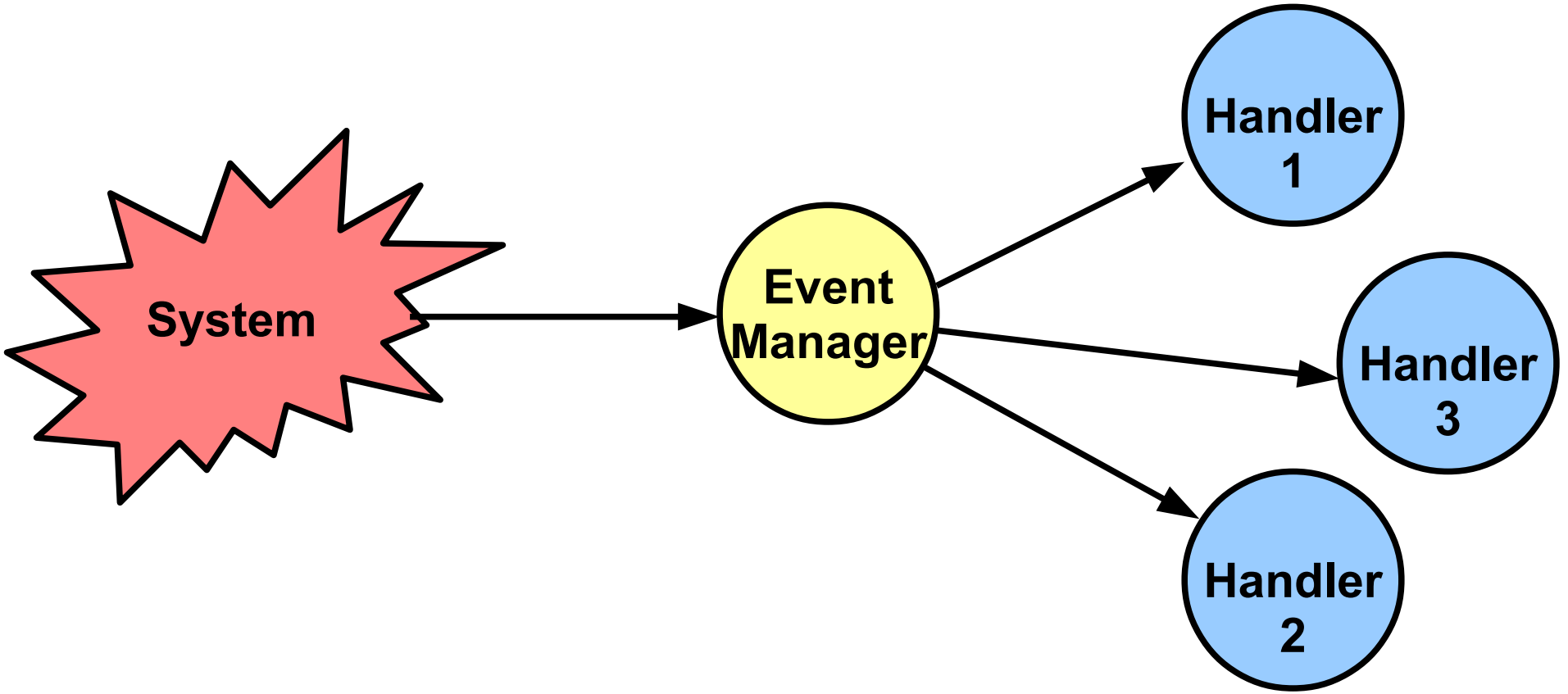
```
night() ->
   receive
      sunrise -> day()
   end.
```

Działanie statem

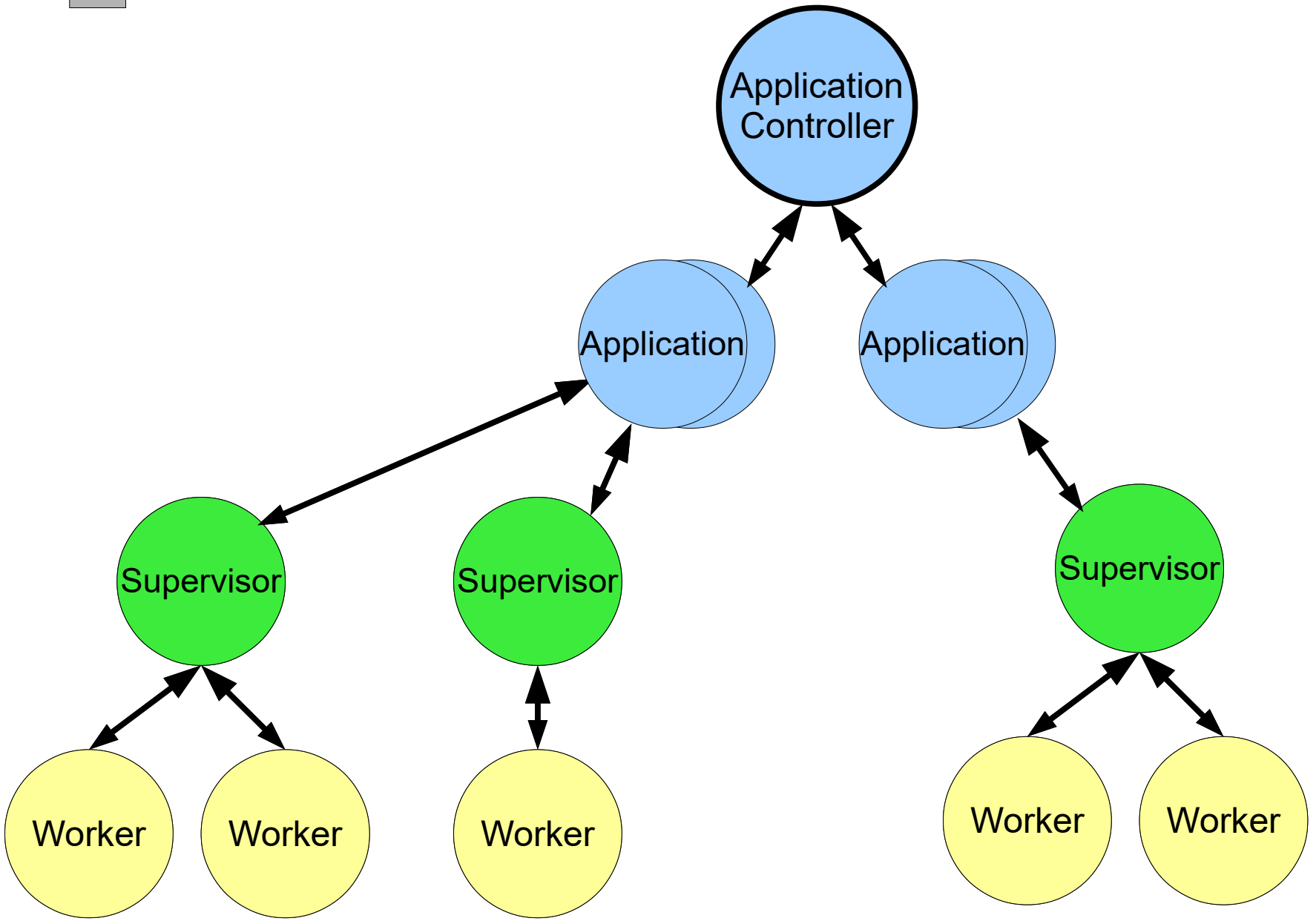
```
%% PUBLIC API
reportSunHigh() -> gen_statem:cast(sun_statem, sun_high).
reportSunset() -> gen_statem:cast(sun_statem, sunset).
reportSunrise() -> gen_statem:cast(sun_statem, surise).

start_link() ->
   gen_statem:start_link({local, sun_statem}, ?MODULE, [], []).

init([]) -> {ok, day, []}.
callback_mode()->state_functions.

%% HANDLERS
day(_Event, sun_high, []) -> {next_state, day, []};
day(_Event, sunset, []) -> {next_state, night, []}.
night(_Event, surise, []) -> {next_state, day, []}.

stopSun() -> gen_statem:stop(sun_statem).

terminate(Reason, StateName, StateData) -> ok.
```

Obsługa zdarzeń: gen_event

- Manager będzie otrzymywał zdarzenia określonego typu
- Handler rejestruje się u managera



Aplikacje w EVM



Aplikacja

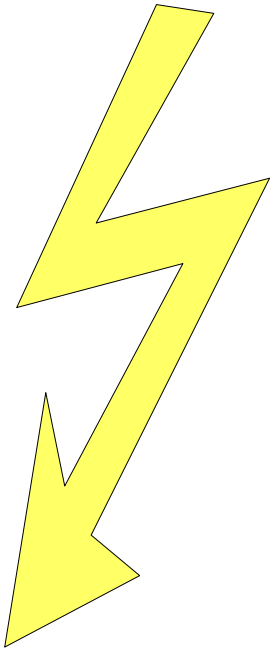
- Struktura organizująca kod, pozwalająca na zarządzanie stanem systemu
- Stan aplikacji: Loaded, Started, Stopped, Unloaded
- Posiada określoną strukturę plików i katalogów
- Narzędzie do tworzenia i aktualizowana: rebar, rebar3

```
ebin/
 - appl.app
 - *.beam
include/
priv/
src/
 - appl.erl
 - appl_sup.erl
 - appl_serv.erl
test/
 - appl_tests.erl
```

```
{application, appl,
        [{vsn, "1.0.0"},
                {modules, [appl.erl, appl_sup, appl_serv]},
                {registered, [appl]},
                {mod, {appl, []}}
]}.
```

lab6

- Prezentacje mikro-projektów (3 punkty)
  - Brak kartkówki i zadania
- Mikro-projekty dwuosobowe – Lightning talk
  - 12 minut
  - Prezentacja + pokaz na żywo
    - Podłączanie projektora...
  - Oceniamy wkład pracy i jakość prezentacji



- Zapisy na tematy – po następnym wykładzie

- **Własne tematy: ✉ do mnie przed następnym wykładem**