WEEK 7. LEARN YOU SOME ERLANG

Presented by nolleh

CHAP.15 - CLIENT AND SERVERS

Callback to the future

Callback to the future

Our First OTP behavior

- We will see gen-server
- You already have seen a few functions from gen-server.

- Init
  - Returns
    - {ok, State}, {ok, State, Timeout}, {ok, State, hibernate}, {stop, Reason}, ignore
    - Timeout will result in a special message (timeout) was sent to the server. (handled with handle_info/2)
    - Process takes long time for getting reply and worried about memory, using hibernate (sacrifice processing power)

Callback to the future

- Handle_call
  - Used to work with synchronous messages
  - 3 arguments : Request / From / State
  - Similar with handle_call/3 in my_server
    - Biggest difference is how you reply to messages
  - 8 way to Returns
    - {reply,Reply,NewState}, {reply,Reply,NewState,Timeout}, {reply,Reply,NewState,hibernate}, {noreply,NewState}, {noreply,NewState,Timeout}, {noreply,NewState,hibernate}, {stop,Reason,Reply,NewState}, {stop,Reason,NewState}
    - No reply : You're taking care of sending reply back.. (gen_server:reply/2)

Callback to the future

- Handle_cast
  - A lot like one we had in my_server
  - Returns
    - {noreply,NewState}, {noreply,NewState,Timeout}, {noreply,NewState,hibernate}, {stop,Reason,NewState}
- Handle_info
  - Similar with handle_cast/2 and in fact returns same tuples.
  - Different part is callback is only there for message which is sent by
    - ! Operator
    - Init/1's timeout
    - Monitors notifications
    - 'EXIT' signal

Callback to the future

- terminate
  - called whenever one of the three handle_something functions returns {stop, Reason, NewState}, {stop, Reason, Reply, NewState}
  - Parameters
    - Reason / State
  - Also be called when its parent dies, if and only if the gen_server is trapping exits

  - Your janitor charge of locking the door after making sure everyone's gone. (helped by VM – delete all ETS Tables, closing ports..)

  - Return value is no matter (code stops)

Callback to the future

- code_change
  - Upgrade code
  - Code_change(PreviousVersion, State, Extra).
  - After convert, all you have to do is return the new state with {ok, NewState}
  - Extra variable is used in larger OTP deployment, where specific tools exist to upgrade entire releases on a VM

.BEAM me up, Scotty!

- Going to be the Kitty_gen_server

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

```
1> c(kitty_gen_server).
./kitty_gen_server.erl:2: Warning: undefined callback function code_change/3 (behaviour
'gen_server')
./kitty_gen_server.erl:2: Warning: undefined callback function handle_call/3 (behaviour
'gen_server')
./kitty_gen_server.erl:2: Warning: undefined callback function handle_cast/2 (behaviour
'gen_server')…
```

.BEAM me up, Scotty!

- Define your own behaviors
- behavior_info/1

```
-module(my_behaviour).
-export([behaviour_info/1]).

%% init/1, some_fun/0 and other/3 are now expected callbacks
behaviour_info(callbacks) -> [{init,1}, {some_fun, 0}, {other, 3}];
behaviour_info(_) -> undefined.
```

.BEAM me up, Scotty!

- replace start_link/0

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

- 1st parameter: callback module
- 2nd parameter: list of parameters to pass to init/1
- 3rd parameter: debugging options
- 4th parameter(optional): name to register the server with
- Return: {ok, Pid}

.BEAM me up, Scotty!

- All of these calls are a one-to-one change

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

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

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

.BEAM me up, Scotty!

- Code is now shorter, thanks to smarter abstractions!

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

handle_call({order, Name, Color, Description}, _From, Cats) ->
      if Cats =:= [] ->
            {reply, make_cat(Name, Color, Description), Cats};
         Cats =/= [] ->
            {reply, hd(Cats), tl(Cats)}
      end;
handle_call(terminate, _From, Cats) ->
      {stop, normal, ok, Cats}.

handle_cast({return, Cat = #cat{}}, Cats) ->
      {noreply, [Cat | Cats]}.
```

.BEAM me up, Scotty!

- New callbacks

```
handle_info(Msg, Cats) ->
    io:format("Unexpected message: ~p~n",[Msg]),
    {noreply, Cats}.
```

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

```
code_change(_OldVsn, State, _Extra) ->
    %% No change planned. The function is there for the behaviour,
    %% but will not be used. Only a version on the next
    {ok, State}.
```

.BEAM me up, Scotty!

- Using it

```
1> c(kitty_gen_server).
      {ok,kitty_gen_server}
2> rr(kitty_gen_server).
      [cat]
3> {ok, Pid} = kitty_gen_server:start_link().
      {ok,<0.253.0>}
4> Pid ! <<"Test handle_info">>.
      Unexpected message: <<"Test handle_info">>
      <<"Test handle_info">>
5> Cat = kitty_gen_server:order_cat(Pid, "Cat Stevens", white, "not actually a cat").
      #cat{name = "Cat Stevens",color = white,
            description = "not actually a cat"}
```

.BEAM me up, Scotty!

- Using it

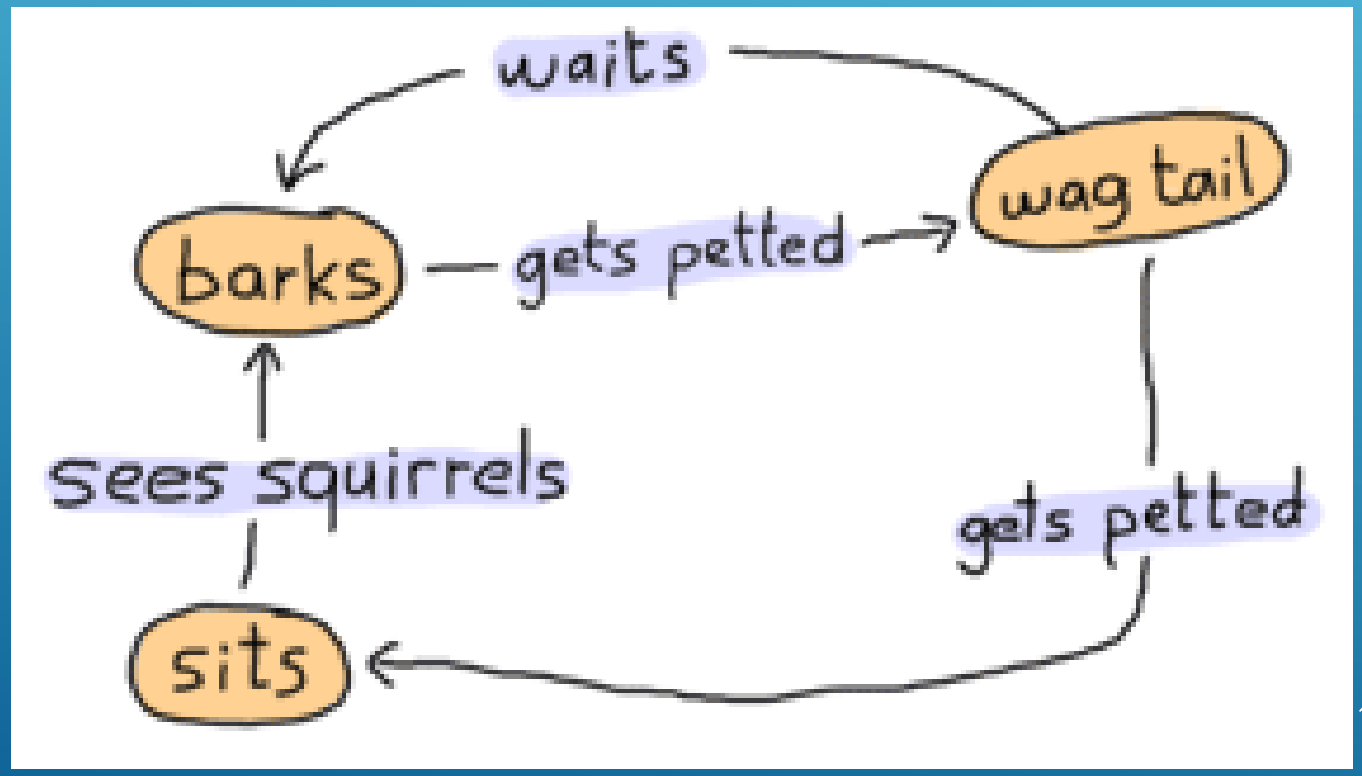
```
6> kitty_gen_server:return_cat(Pid, Cat).
      ok
7> kitty_gen_server:order_cat(Pid, "Kitten Mittens", black, "look at them little paws!").
      #cat{name = "Cat Stevens",color = white,
            description = "not actually a cat"}
8> kitty_gen_server:order_cat(Pid, "Kitten Mittens", black, "look at them little paws!").
      #cat{name = "Kitten Mittens",color = black,
            description = "look at them little paws!"}
9> kitty_gen_server:return_cat(Pid, Cat).
      ok
10> kitty_gen_server:close_shop(Pid).
      "Cat Stevens" was set free.
      ok
```

CHAP.16 – RAGE AGAINST THE FINITE-STATE-MACHINES

What are they?

What are they?

- Finite state machine
  - Have a finite number of states
  - Easier to understand with graphs and diagrams



What are they?

- Implements dog fsm

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

start() ->
      spawn(fun() -> bark() end).

squirrel(Pid) -> Pid ! squirrel.

pet(Pid) -> Pid ! pet.
```

What are they?

- Implements dog fsm

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

What are they?

- Implements dog fsm

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

What are they?

- dog fsm in use

```
6> c(dog_fsm).
      {ok,dog_fsm}
7> Pid = dog_fsm:start().
      Dog says: BARK! BARK!
      <0.46.0>
      Dog says: BARK! BARK!
      Dog says: BARK! BARK!
      Dog says: BARK! BARK!
8> dog_fsm:pet(Pid).
      pet
      Dog wags its tail
```

What are they?

- dog fsm in use

```
9> dog_fsm:pet(Pid).
      Dog is sitting. Goooooood boy!
      pet
10> dog_fsm:pet(Pid).
      Dog is confused
      pet
      Dog is sitting. Goooooood boy!
11> dog_fsm:squirrel(Pid).
      Dog says: BARK! BARK!
      squirrel
      Dog says: BARK! BARK!
```

What are they?

- dog fsm in use

```
12> dog_fsm:pet(Pid).
    Dog wags its tail
    pet
13> %% wait 30 seconds
    Dog says: BARK! BARK!
    Dog says: BARK! BARK!
    Dog says: BARK! BARK!
13> dog_fsm:pet(Pid).
    Dog wags its tail
    pet
14> dog_fsm:pet(Pid).
    Dog is sitting. Gooooood boy!
    pet
```

Generic finite-state Machines

- Init
  - Same init/1 as used for generic servers, except the return values
  - Returns
    - {ok, StateName, Data}, {ok, StateName, Data, Timeout}, {ok, StateName, Data, hibernate}, {stop, Reason}
    - StateName: atom, and represents the next callback functions
- StateName
  - StateName/2, StateName/3 is placeholder names and you are to decide what they will be
  - You defines sitting/2, sitting/3 (called for async/sync, repectively)

Generic finite-state Machines

- StateName
  - Sitting/2
    - Returns

      {next_state, NextStateName, NewStateData},
      {next_state, NextStateName, NewStateData, Timeout},
      {next_state, NextStateName, NewStateData, hibernate},
      {stop, Reason, NewStateData}

Generic finite-state Machines

- StateName
  - Sitting/3
    - Returns
      - {reply, Reply, NextStateName, NewStateData},
        {reply, Reply, NextStateName, NewStateData, Timeout},
        {reply, Reply, NextStateName, NewStateData, hibernate},
        {next_state, NextStateName, NewStateData},
        {next_state, NextStateName, NewStateData, Timeout},
        {next_state, NextStateName, NewStateData, hibernate},
        {stop, Reason, Reply, NewStateData},
        {stop, Reason, NewStateData}

Generic finite-state Machines

- handle_event
  - For Global event
  - The function takes arguments similar to StateName/2 with exception that it accepts a StateName (prev's state)
- handle_sync_event
  - Handle_sync_event/4 callback is to StateName/3
  - How we know whether an event is global or if it's meant to be sent to a specific state
    - Send_event/2, sync_send_event/2-3, send_all_state_event/2, sync_send_all_state_event/2-3

Generic finite-state Machines

- Code_change
  - Same as it did for gen_server, except that it takes an extra state parameter.
    - Code_change(OldVersion, StateName, Data, Extra)
  - Returns
    - {ok, NextStateName, NewStateData}
- Terminate
  - This should, again, act a bit like what we have for generic servers..

A Trading System Specification

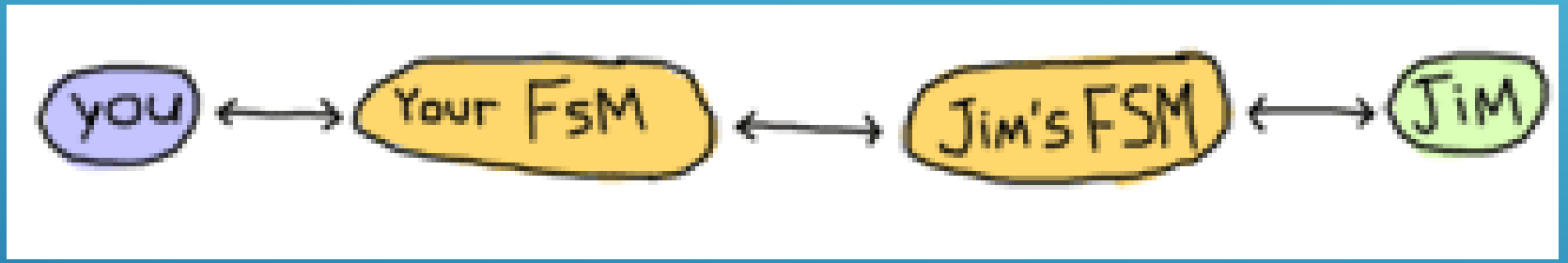
- Time to put all of this in practice
  - We will design and implement an item trading system for some fictional and non-existing video game
  - Rather than using a broker through which players route items and confirmations, we're going to implement a server where both players speak to each other directly (advantage of being distributable)
  - define actions that can be done by our players when trading
    - first is asking for a trade to be set up
    - The other user should aslo be able to accept the trade
    - Once the trade is set up, our users should be able to negotiate with each other
    - At any point in time, it should also make sense for any of players to cancel the whole trade

A Trading System Specification

- Time to put all of this in practice
  - Following actions should be possible
    - Ask for a trade
    - Accept a trade
    - Offer items
    - Retract an offer
    - Declare self as ready
    - Brutally cancel the trade

A Trading System Specification

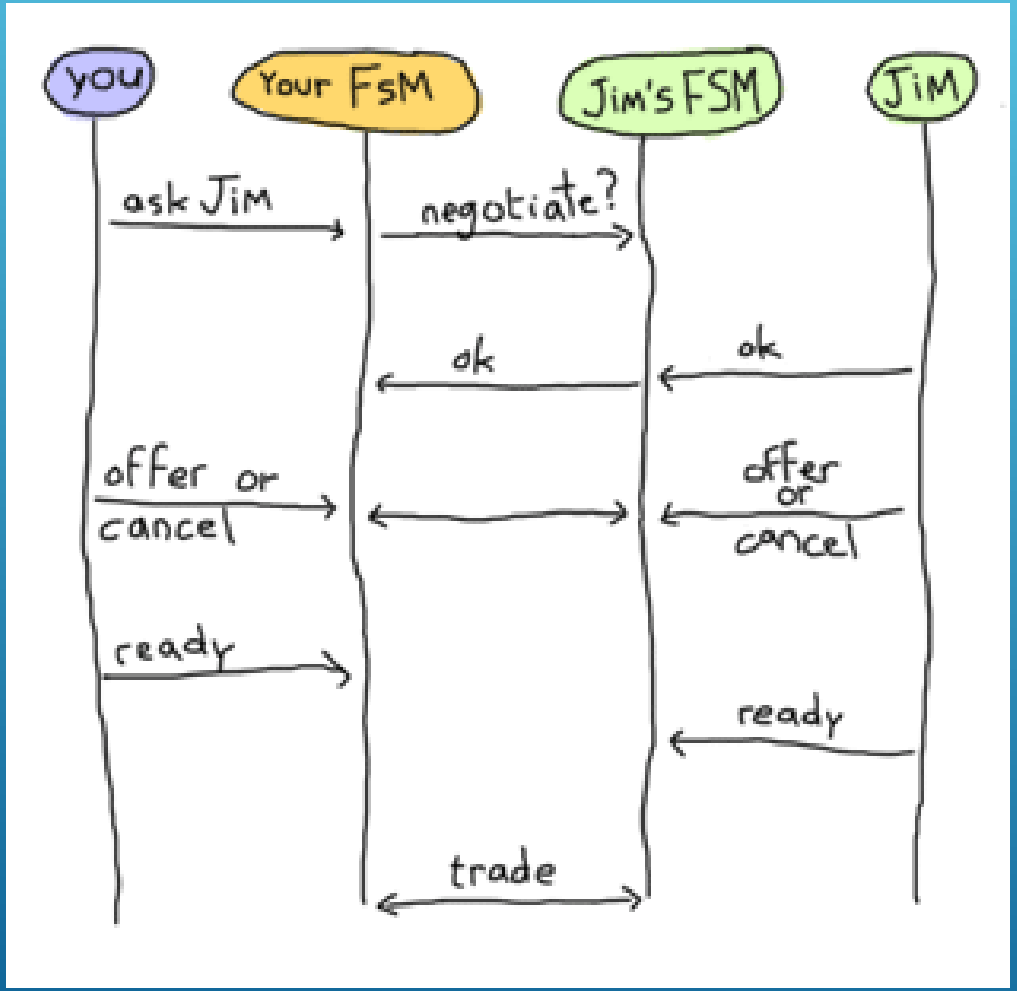
- Time to put all of this in practice
  - If ones actions is taken, the other player's FSM should be made aware of it



  - Do not use sync calls to avoid deadlock.
  - Jim might still make a synchronous call to his own FSM
    - There's no risk here because the FSM won't need to call Jim and so no deadlock can occur between them

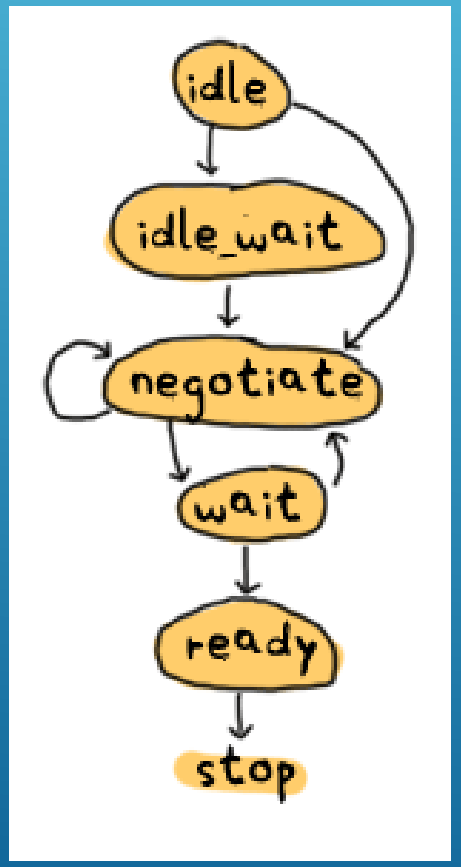
A Trading System Specification

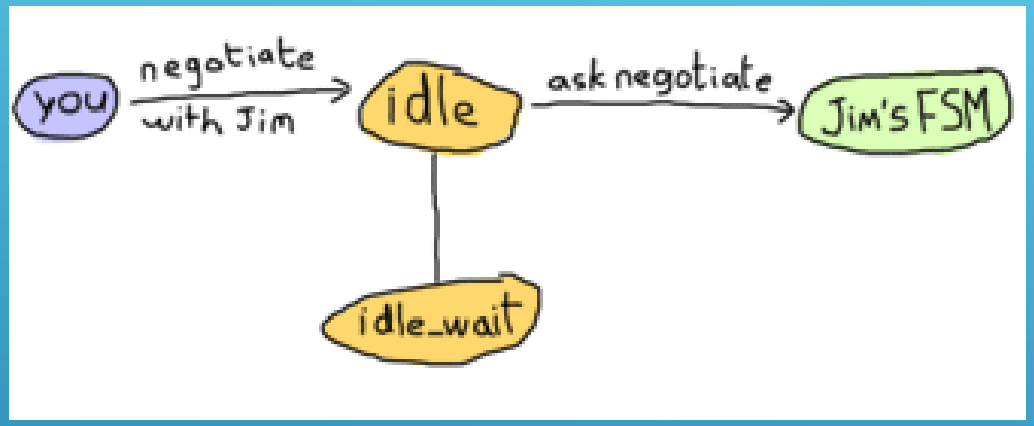
- Time to put all of this in practice
- ones

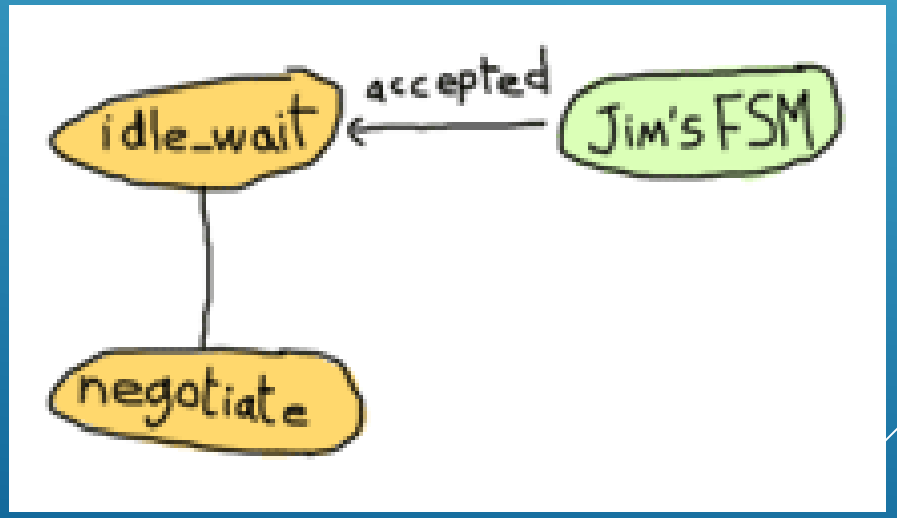


A Trading System Specification

- Time to put all of this in practice
  - Idle

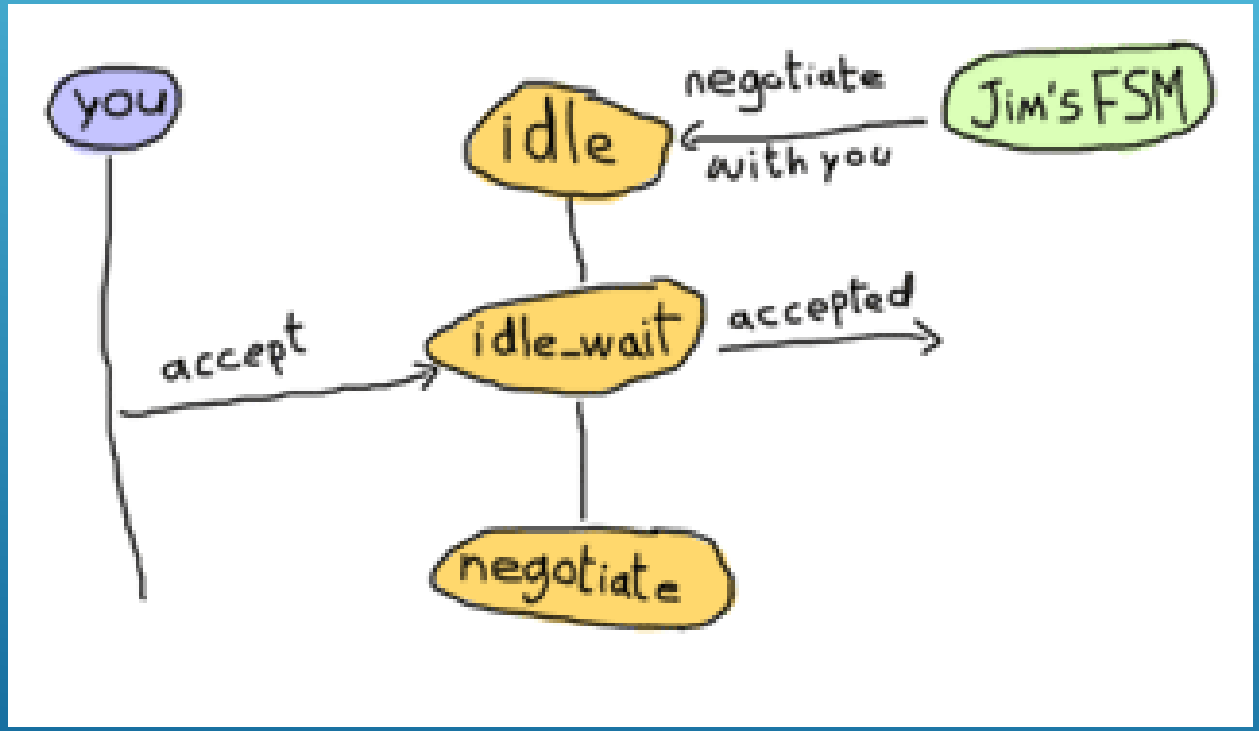






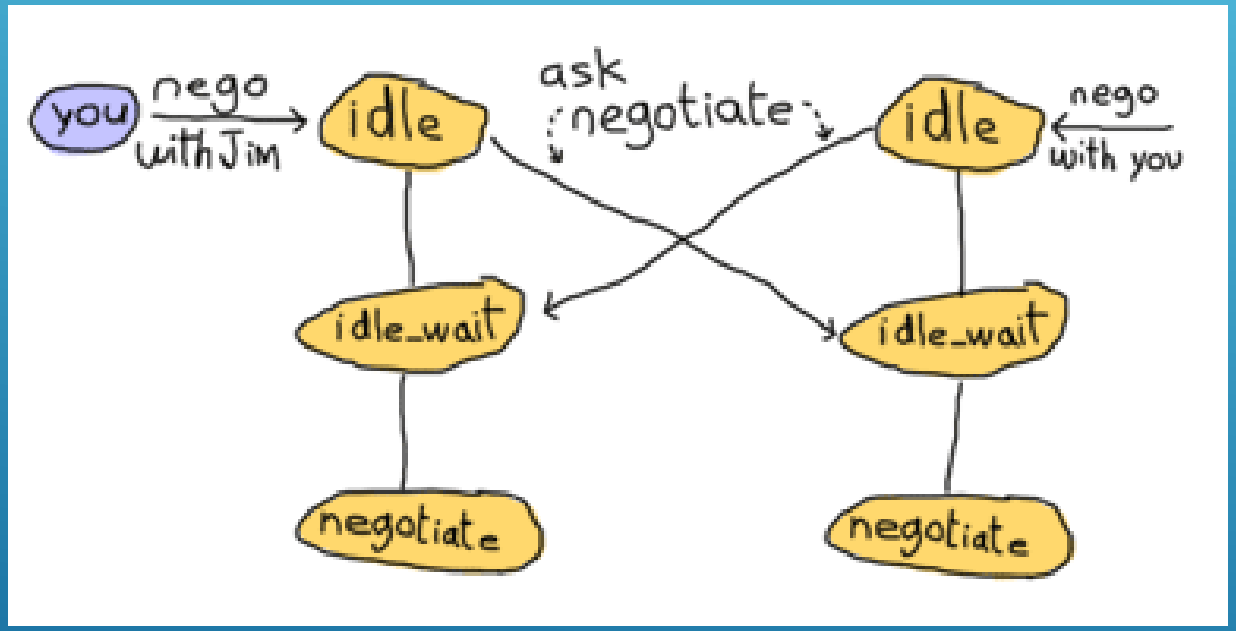
A Trading System Specification

- Time to put all of this in practice
  - Idle



A Trading System Specification

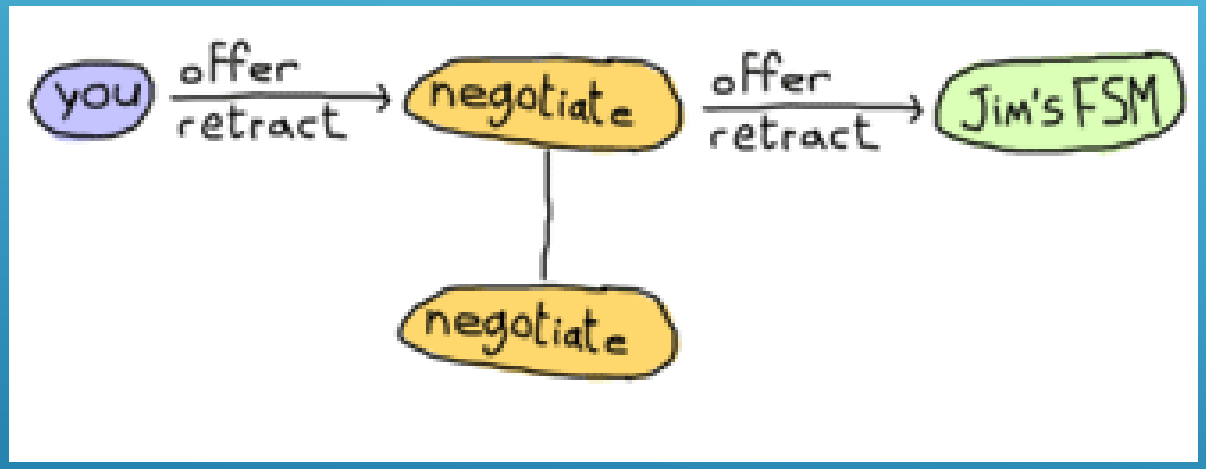
- Time to put all of this in practice
  - we ask other player to trade with us at the same time he askes us to trade ?



  - Recv ask nego, while in idle_wait state. (this is race, and can assume both user want to talk)

A Trading System Specification

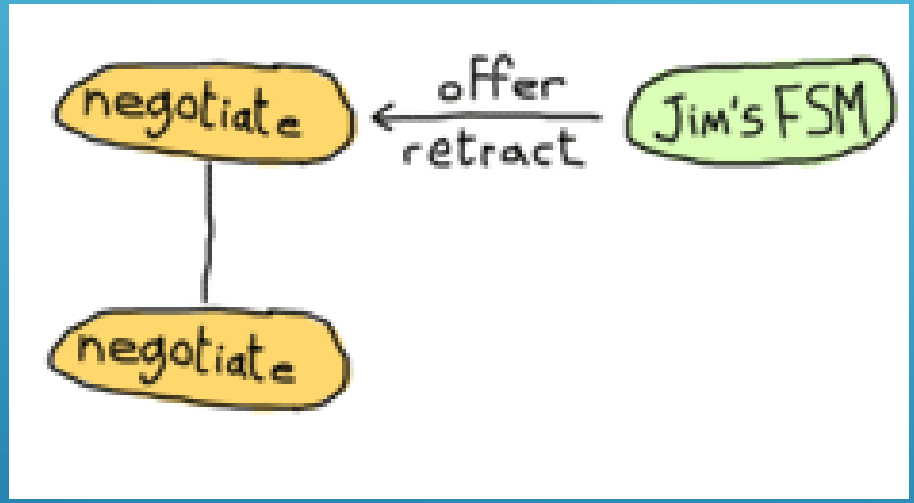
- Time to put all of this in practice
  - Users offering items and then retracting the offer



- All this does is forward our client's message to the other FSM
- Both FSM will need to hold a list of items offered by either player, so they can update that list when receiving such messages.

A Trading System Specification

- Time to put all of this in practice
  - Maybe the other player wants to offer items too



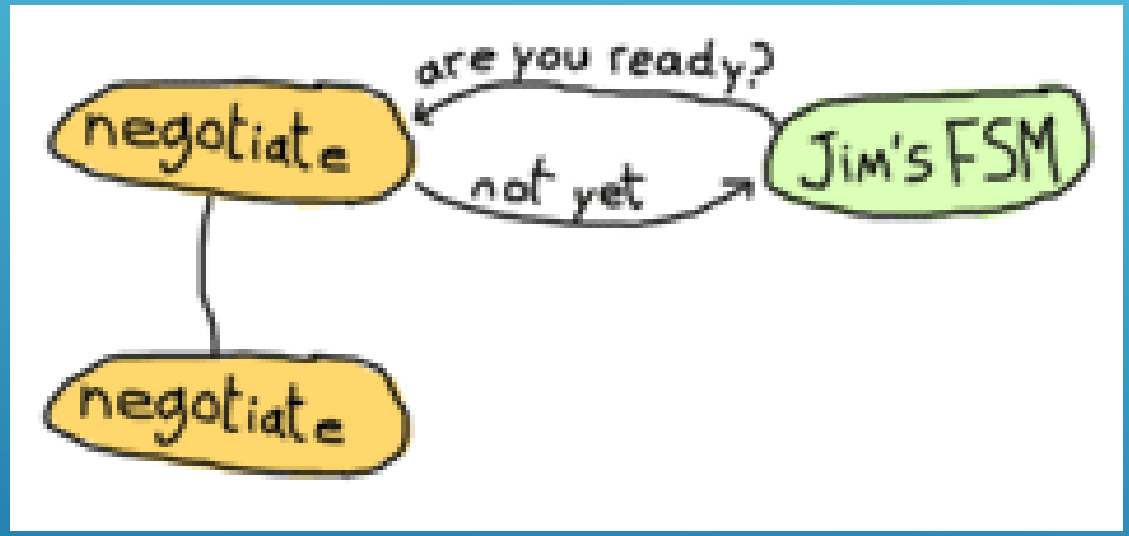
A Trading System Specification

- Time to put all of this in practice
  - Time to officialise the trade

  - We have to synchronise both players, we'll have to use an intermediary state, as we did for idle and idle_wait

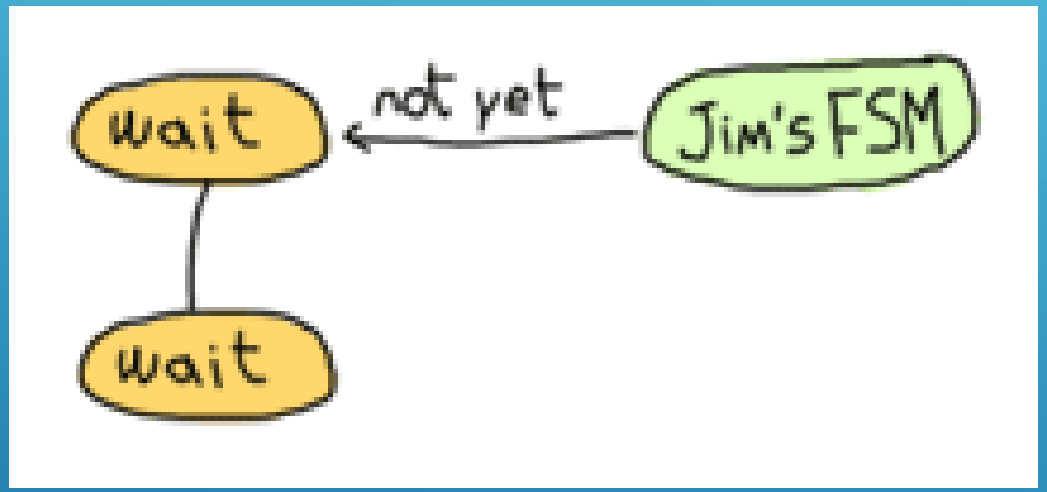
A Trading System Specification

- Time to put all of this in practice
  - Reply is depend on jim's FSM state.



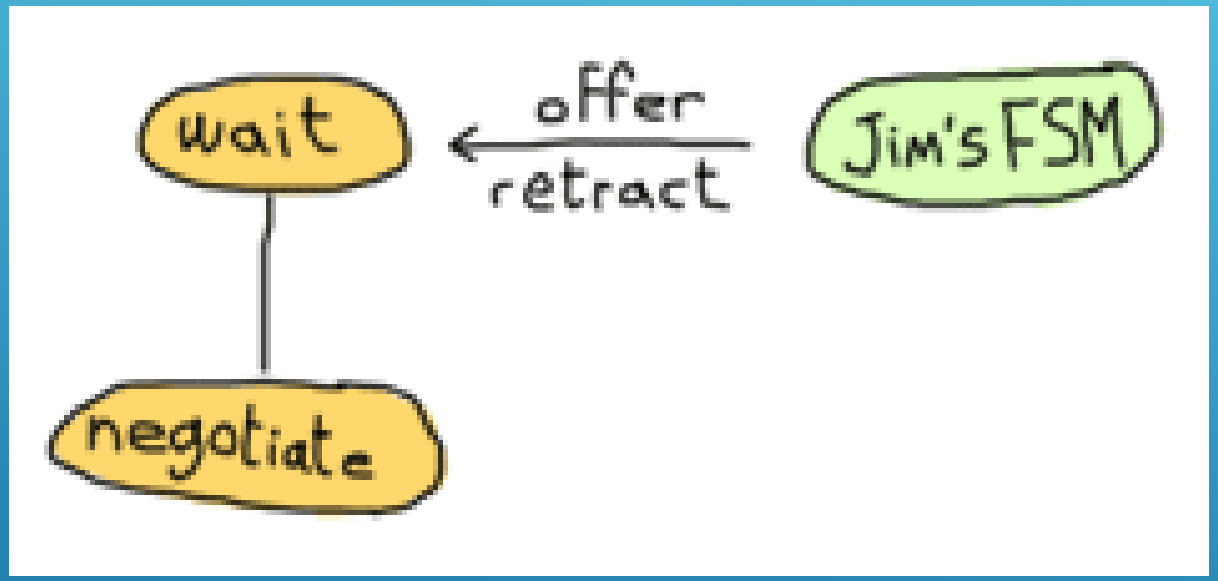
A Trading System Specification

- Time to put all of this in practice
  - Fsm's reply



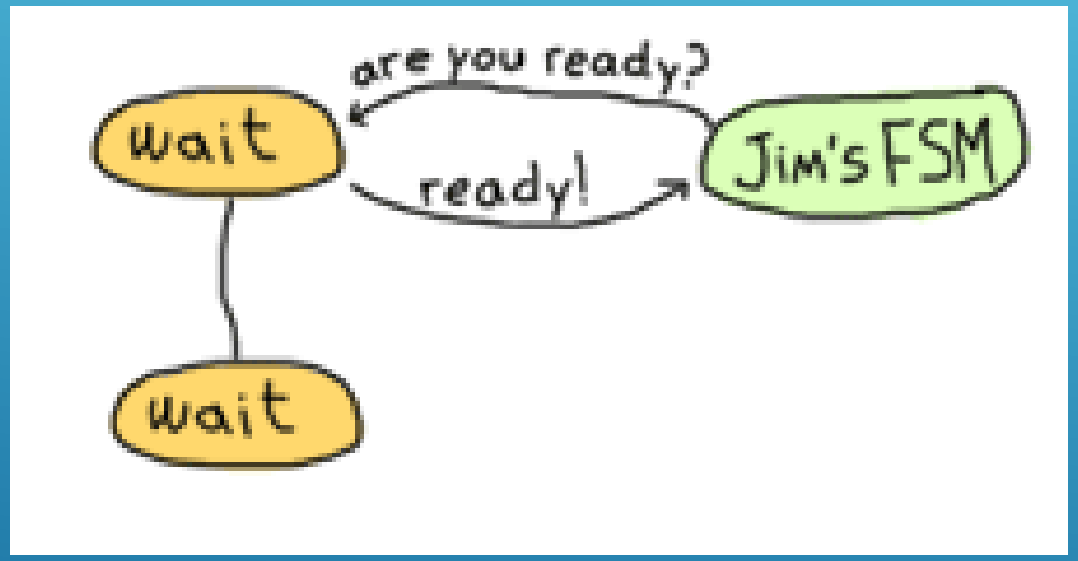
A Trading System Specification

- Time to put all of this in practice
  - Jim wants more negotiate



A Trading System Specification

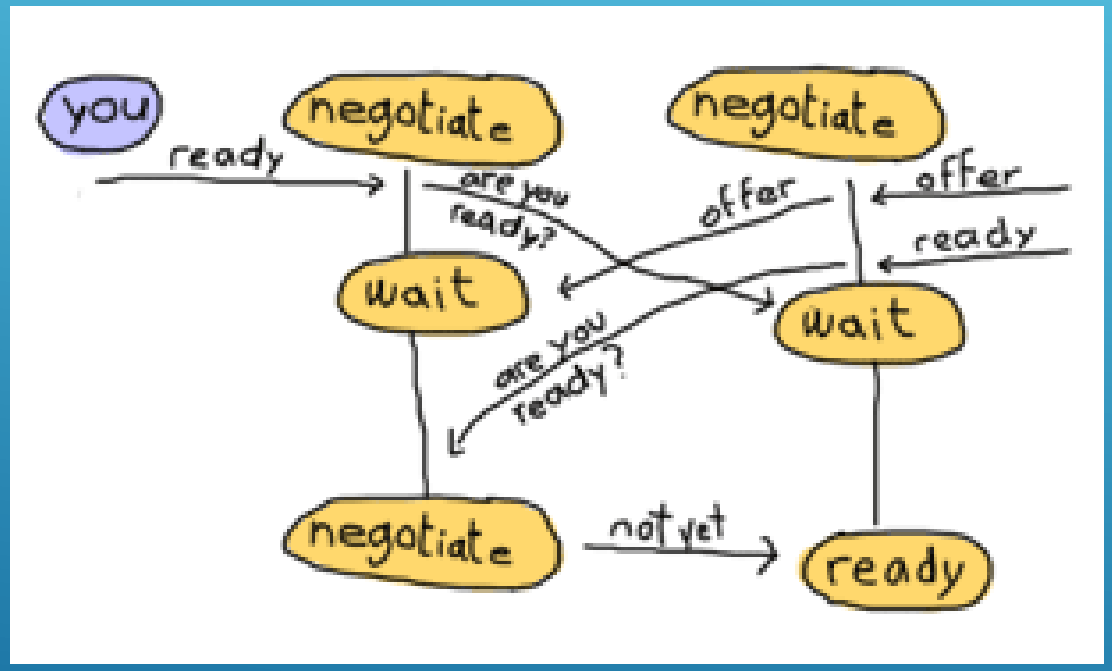
- Time to put all of this in practice
  - Ready both players



  - We stay in the waiting state and refuse to move to the ready state though.

A Trading System Specification

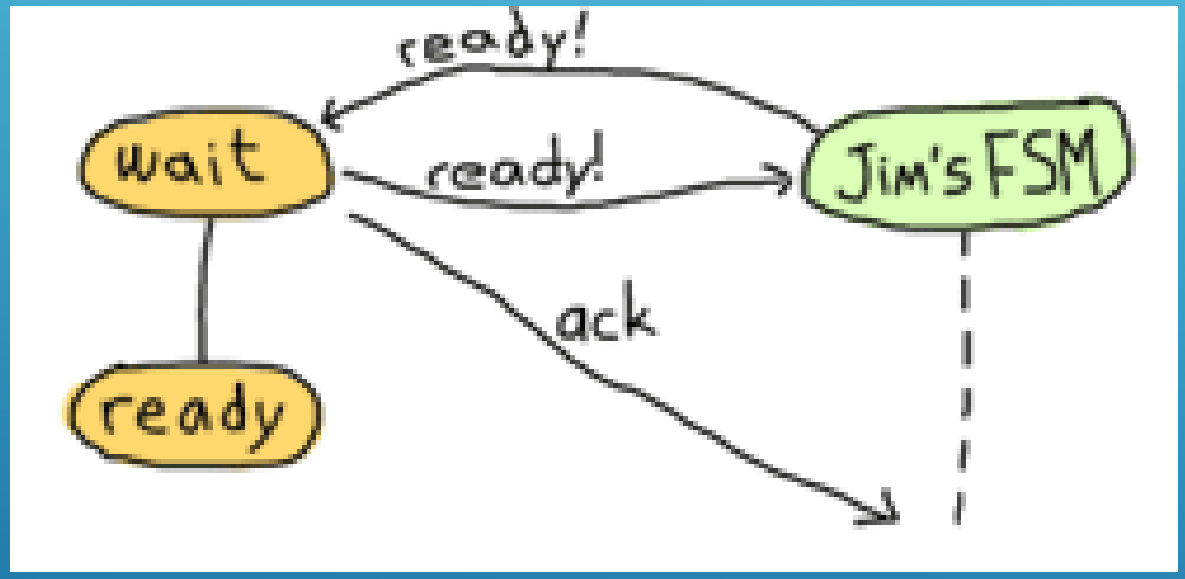
- Time to put all of this in practice
  - Race condition



- As soon as we read offer message, we switch back to negotiate state
- Jim told us he is ready
- Jim waiting indefinitely..

A Trading System Specification

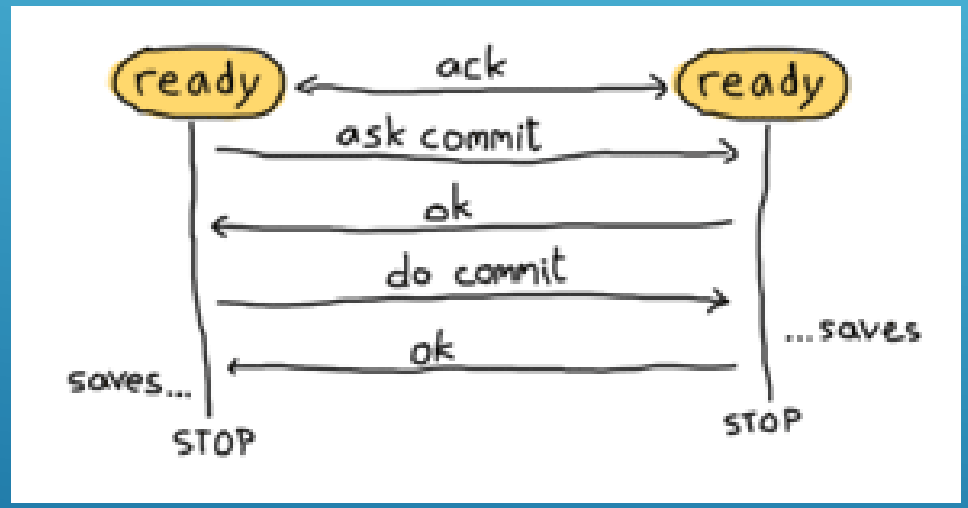
- Time to put all of this in practice
  - By adding one layer of indirection



- This will create a superfluous ‘ready!’ message in one of the two FSMs, but we’ll ignore in this case
- Ack message before moving to ready state

A Trading System Specification

- Time to put all of this in practice
  - Both players are ready and have basically given the finite-state machines all the control they need



- Simplisticv ersion of two phase commit
- Somehow, we need Global cancel message

A Trading System Specification

- Time to put all of this in practice
  - It is okay, you aren't fully understands.
  - Encourage you to make your own protocol.
  - Aks yourself
    - what happens if two people do the same actions very fast?
    - What if they chain two other events quickly?
    - What do I do with messages I don't handle when changing states?

Game trading between two players

- Use gen_fsm to create the interface
  - We will only need to export the player function and gen_fsm functions
    - the other FSM will also run within the trade_fsm module, and can access them from the inside

```
-module(trade_fsm).
-behaviour(gen_fsm).

%% public API
-export([start/1, start_link/1, trade/2, accept_trade/1,
         make_offer/2, retract_offer/2, ready/1, cancel/1]).
%% gen_fsm callbacks
-export([init/1, handle_event/3, handle_sync_event/4, handle_info/3,
         terminate/3, code_change/4,
         % custom state names
         idle/2, idle/3, idle_wait/2, idle_wait/3, negotiate/2,
         negotiate/3, wait/2, ready/2, ready/3]).
```

Game trading between two players

- Use gen_fsm to create the interface
  - Public api according to the protocol defined as before

```
%%% PUBLIC API
start(Name) ->
    gen_fsm:start(?MODULE, [Name], []).

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

%% ask for a begin session. Returns when/if the other accepts
trade(OwnPid, OtherPid) ->
    gen_fsm:sync_send_event(OwnPid, {negotiate, OtherPid}, 30000).

%% Accept someone's trade offer.
accept_trade(OwnPid) ->
    gen_fsm:sync_send_event(OwnPid, accept_negotiate).
```

Game trading between two players

- Use gen_fsm to create the interface
  - Public api according to the protocol defined as before

```
%% Send an item on the table to be traded
make_offer(OwnPid, Item) ->
      gen_fsm:send_event(OwnPid, {make_offer, Item}).

%% Cancel trade offer
retract_offer(OwnPid, Item) ->
      gen_fsm:send_event(OwnPid, {retract_offer, Item}).

%% Mention that you're ready for a trade. When the other
%% player also declares being ready, the trade is done
ready(OwnPid) ->
      gen_fsm:sync_send_event(OwnPid, ready, infinity).

%% Cancel the transaction.
cancel(OwnPid) ->
      gen_fsm:sync_send_all_state_event(OwnPid, cancel).
```

Game trading between two players

- Use gen_fsm to create the interface
  - Fsm to Fsm Functions

```
%% Ask the other FSM's Pid for a trade session
ask_negotiate(OtherPid, OwnPid) ->
      gen_fsm:send_event(OtherPid, {ask_negotiate, OwnPid}).

%% Forward the client message accepting the transaction
accept_negotiate(OtherPid, OwnPid) ->
      gen_fsm:send_event(OtherPid, {accept_negotiate, OwnPid}).
```

Game trading between two players

- Use gen_fsm to create the interface
  - Fsm to Fsm Functions

```
%% forward a client's offer
do_offer(OtherPid, Item) ->
      gen_fsm:send_event(OtherPid, {do_offer, Item}).

%% forward a client's offer cancellation
undo_offer(OtherPid, Item) ->
      gen_fsm:send_event(OtherPid, {undo_offer, Item}).
```

Game trading between two players

- Use gen_fsm to create the interface
  - Fsm to Fsm Functions

```
%% Ask the other side if he's ready to trade.
are_you_ready(OtherPid) ->
      gen_fsm:send_event(OtherPid, are_you_ready).

%% Reply that the side is not ready to trade
%% i.e. is not in 'wait' state.
not_yet(OtherPid) ->
      gen_fsm:send_event(OtherPid, not_yet).

%% Tells the other fsm that the user is currently waiting
%% for the ready state. State should transition to 'ready'
am_ready(OtherPid) ->
      gen_fsm:send_event(OtherPid, 'ready!').
```

Game trading between two players

- Use gen_fsm to create the interface
  - Fsm to Fsm Functions

```
%% Acknowledge that the fsm is in a ready state.
ack_trans(OtherPid) ->
      gen_fsm:send_event(OtherPid, ack).

%% ask if ready to commit
ask_commit(OtherPid) ->
      gen_fsm:sync_send_event(OtherPid, ask_commit).

%% begin the synchronous commit
do_commit(OtherPid) ->
      gen_fsm:sync_send_event(OtherPid, do_commit).
```

Game trading between two players

- Use gen_fsm to create the interface
  - Fsm to Fsm Functions

```
notify_cancel(OtherPid) ->
    gen_fsm:send_all_state_event(OtherPid, cancel).
```

Game trading between two players

- Use gen_fsm to create the interface
  - Gen_fsm callbacks

```
-record(state, {name="",
                other,
                ownitems=[],
                otheritems=[],
                monitor,
                from}).
```

  - We want hold
    - My name (for representation)
    - Others pid
    - Items we offer and the items the other offers
    - Reference of a monitor
    - From field, used to do delayed replies

Game trading between two players

- Use gen_fsm to create the interface
  - Gen_fsm callbacks

```
init(Name) ->
      {ok, idle, #state{name=Name}}.
```

```
%% Send players a notice. This could be messages to their clients
%% but for our purposes, outputting to the shell is enough.
notice(#state{name=N}, Str, Args) ->
      io:format("~s: "++Str++"~n", [N | Args]).

%% Unexpected allows to log unexpected messages
unexpected(Msg, State) ->
      io:format("~p received unknown event ~p while in state ~p~n",
      [self(), Msg, State]).
```

Game trading between two players

- Use gen_fsm to create the interface
  - Gen_fsm callbacks

```
idle({ask_negotiate, OtherPid}, S=#state{}) ->
        Ref = monitor(process, OtherPid),
        notice(S, "~p asked for a trade negotiation", [OtherPid]),
        {next_state, idle_wait, S#state{other=OtherPid, monitor=Ref}};
idle(Event, Data) ->
        unexpected(Event, idle),
        {next_state, idle, Data}.
```

Game trading between two players

- Use gen_fsm to create the interface
  - Gen_fsm callbacks

```
idle({negotiate, OtherPid}, From, S=#state{}) ->
      ask_negotiate(OtherPid, self()),
      notice(S, "asking user ~p for a trade", [OtherPid]),
      Ref = monitor(process, OtherPid),
      {next_state, idle_wait, S#state{other=OtherPid, monitor=Ref, from=From}};
idle(Event, _From, Data) ->
      unexpected(Event, idle),
      {next_state, idle, Data}.
```

Game trading between two players

- Use gen_fsm to create the interface
  - Gen_fsm callbacks

```
idle_wait({ask_negotiate, OtherPid}, S=#state{other=OtherPid}) ->
      gen_fsm:reply(S#state.from, ok),
      notice(S, "starting negotiation", []),
      {next_state, negotiate, S};
%% The other side has accepted our offer. Move to negotiate state
idle_wait({accept_negotiate, OtherPid}, S=#state{other=OtherPid}) ->
      gen_fsm:reply(S#state.from, ok),
      notice(S, "starting negotiation", []),
      {next_state, negotiate, S};
idle_wait(Event, Data) ->
      unexpected(Event, idle_wait),
      {next_state, idle_wait, Data}.
```

Game trading between two players

- Use gen_fsm to create the interface
  - Gen_fsm callbacks
    - The case of our FSM's client accepting the trade suggested by the other party

```
idle_wait(accept_negotiate, _From, S=#state{other=OtherPid}) ->
      accept_negotiate(OtherPid, self()),
      notice(S, "accepting negotiation", []),
      {reply, ok, negotiate, S};
idle_wait(Event, _From, Data) ->
      unexpected(Event, idle_wait),
      {next_state, idle_wait, Data}.
      Again, this one moves on to the negotiate state.
```

Game trading between two players

- Use gen_fsm to create the interface
  - Gen_fsm callbacks
    - Wrap item operations in their own functions

```
%% adds an item to an item list
add(Item, Items) ->
    [Item | Items].

%% remove an item from an item list
remove(Item, Items) ->
    Items -- [Item].
```

Game trading between two players

- Use gen_fsm to create the interface
  - Gen_fsm callbacks

```
negotiate({make_offer, Item}, S=#state{ownitems=OwnItems}) ->
      do_offer(S#state.other, Item),
      notice(S, "offering ~p", [Item]),
      {next_state, negotiate, S#state{ownitems=add(Item, OwnItems)}};

%% Own side retracting an item offer
negotiate({retract_offer, Item}, S=#state{ownitems=OwnItems}) ->
      undo_offer(S#state.other, Item),
      notice(S, "cancelling offer on ~p", [Item]),
      {next_state, negotiate, S#state{ownitems=remove(Item, OwnItems)}};
```

Game trading between two players

- Use gen_fsm to create the interface
  - Gen_fsm callbacks

```
%% other side offering an item
negotiate({do_offer, Item}, S=#state{otheritems=OtherItems}) ->
      notice(S, "other player offering ~p", [Item]),
      {next_state, negotiate, S#state{otheritems=add(Item, OtherItems)}};

%% other side retracting an item offer
negotiate({undo_offer, Item}, S=#state{otheritems=OtherItems}) ->
      notice(S, "Other player cancelling offer on ~p", [Item]),
      {next_state, negotiate, S#state{otheritems=remove(Item, OtherItems)}};
```

Game trading between two players

- Use gen_fsm to create the interface
  - Gen_fsm callbacks

```
negotiate(are_you_ready, S=#state{other=OtherPid}) ->
      io:format("Other user ready to trade.~n"),
      notice(S,
            "Other user ready to transfer goods:~n"
            "You get ~p, The other side gets ~p",
            [S#state.otheritems, S#state.ownitems]),
      not_yet(OtherPid),
      {next_state, negotiate, S};

negotiate(Event, Data) ->
unexpected(Event, negotiate),
{next_state, negotiate, Data}.
```

Game trading between two players

- Use gen_fsm to create the interface
  - Gen_fsm callbacks

```
negotiate(ready, From, S = #state{other=OtherPid}) ->
      are_you_ready(OtherPid),
      notice(S, "asking if ready, waiting", []),
      {next_state, wait, S#state{from=From}};
negotiate(Event, _From, S) ->
      unexpected(Event, negotiate),
      {next_state, negotiate, S}.
```

Game trading between two players

- Use gen_fsm to create the interface
  - Gen_fsm callbacks

```
wait({do_offer, Item}, S=#state{otheritems=OtherItems}) ->
      gen_fsm:reply(S#state.from, offer_changed),
      notice(S, "other side offering ~p", [Item]),
      {next_state, negotiate, S#state{otheritems=add(Item, OtherItems)}};
wait({undo_offer, Item}, S=#state{otheritems=OtherItems}) ->
      gen_fsm:reply(S#state.from, offer_changed),
      notice(S, "Other side cancelling offer of ~p", [Item]),
      {next_state, negotiate, S#state{otheritems=remove(Item, OtherItems)}};
```

Game trading between two players

- Use gen_fsm to create the interface
  - Gen_fsm callbacks

```
wait(are_you_ready, S=#state{}) ->
    am_ready(S#state.other),
    notice(S, "asked if ready, and I am. Waiting for same reply", []),
    {next_state, wait, S};
```

```
wait(not_yet, S = #state{}) ->
    notice(S, "Other not ready yet", []),
    {next_state, wait, S};
```

Game trading between two players

- Use gen_fsm to create the interface
  - Gen_fsm callbacks

```
wait('ready!', S=#state{}) ->
      am_ready(S#state.other),
      ack_trans(S#state.other),
      gen_fsm:reply(S#state.from, ok),
      notice(S, "other side is ready. Moving to ready state", []),
      {next_state, ready, S};

%% DOn't care about these!
wait(Event, Data) ->
      unexpected(Event, wait),
      {next_state, wait, Data}.
```

Game trading between two players

- Use gen_fsm to create the interface
  - Ack_trans/1, triggering without either player acting, begin commit.
  - Two phase commit require synchornouse communitaions, but can't have both FSMs starting the transaction at once, to avoid deadlock
    - Way to decide that one finite state machine should initiate the commit, while the other will sit and wait for orders from the first one
  - To do this
    - The pids of any process can be compatered and sorted.

```
priority(OwnPid, OtherPid) when OwnPid > OtherPid -> true;
priority(OwnPid, OtherPid) when OwnPid < OtherPid -> false.
```

Game trading between two players

- Use gen_fsm to create the interface

```
ready(ack, S=#state{}) ->
    case priority(self(), S#state.other) of
        true ->
            try
                notice(S, "asking for commit", []),
                ready_commit = ask_commit(S#state.other),
                notice(S, "ordering commit", []),
                ok = do_commit(S#state.other),
                notice(S, "committing...", []),
                commit(S),
                {stop, normal, S}
            catch Class:Reason ->
            %% abort! Either ready_commit or do_commit failed
            notice(S, "commit failed", []),
            {stop, {Class, Reason}, S}
            end;
```

Game trading between two players

- Use gen_fsm to create the interface

```
            false ->
                {next_state, ready, S}
                end;
ready(Event, Data) ->
      unexpected(Event, ready),
      {next_state, ready, Data}.
```

Game trading between two players

- Use gen_fsm to create the interface

```
ready(ask_commit, _From, S) ->
      notice(S, "replying to ask_commit", []),
      {reply, ready_commit, ready, S};
```

```
ready(do_commit, _From, S) ->
      notice(S, "committing...", []),
      commit(S),
      {stop, normal, ok, S};
ready(Event, _From, Data) ->
      unexpected(Event, ready),
      {next_state, ready, Data}.
```

  - The try-catch is for FSM dies / or player cancels the transactions, the sync calls will crash after a time out…

Game trading between two players

- Use gen_fsm to create the interface

```
commit(S = #state{}) ->
      io:format("Transaction completed for ~s. "
      "Items sent are:~n~p,~n received are:~n~p.~n"
      "This operation should have some atomic save "
      "in a database.~n",
      [S#state.name, S#state.ownitems, S#state.otheritems]).
```

Game trading between two players

- Use gen_fsm to create the interface

```
%% The other player has sent this cancel event
%% stop whatever we're doing and shut down!
handle_event(cancel, _StateName, S=#state{}) ->
      notice(S, "received cancel event", []),
      {stop, other_cancelled, S};
handle_event(Event, StateName, Data) ->
      unexpected(Event, StateName),
      {next_state, StateName, Data}.
```

Game trading between two players

- Use gen_fsm to create the interface

```
%% This cancel event comes from the client. We must warn the other
%% player that we have a quitter!
handle_sync_event(cancel, _From, _StateName, S = #state{}) ->
        notify_cancel(S#state.other),
        notice(S, "cancelling trade, sending cancel event", []),
        {stop, cancelled, ok, S};
%% Note: DO NOT reply to unexpected calls. Let the call-maker crash!
handle_sync_event(Event, _From, StateName, Data) ->
        unexpected(Event, StateName),
        {next_state, StateName, Data}.
```

Game trading between two players

- Use gen_fsm to create the interface

```
handle_info({'DOWN', Ref, process, Pid, Reason}, _, S=#state{other=Pid, monitor=Ref}) ->
        notice(S, "Other side dead", []),
        {stop, {other_down, Reason}, S};
handle_info(Info, StateName, Data) ->
        unexpected(Info, StateName),
        {next_state, StateName, Data}.
```

Game trading between two players

- Use gen_fsm to create the interface

```
code_change(_OldVsn, StateName, Data, _Extra) ->
    {ok, StateName, Data}.

%% Transaction completed.
terminate(normal, ready, S=#state{}) ->
    notice(S, "FSM leaving.", []);
    terminate(_Reason, _StateName, _StateData) ->
    ok.
```

That was quite something

- Once again, it's okay, despite of you aren't fully understand!
  - ask yourself
    - Can you understand how different events are handled depending on the state your process is in?
    - Do you know when to use send_event/2 and sync_send_event/2-3 as opposed to send_all_state_event/2 and sync_send_all_state_event/3?

Fit for the real world?

- There is a lot more stuff going on that could make trading even more complex
  - Items could be worn by characters and damaged by enemies while they're being traded
  - Maybe items could be moved in and out of inventory while being exchanged.
  - Are the players on the same server? ..
- Before trying to fit it in a game, make sure everything goes right