We've seen examples (servers, actors) where the essential *behavior* is written once and then *instantiated* with different functions.

The Erlang OTP (Open Telecommunication Platform) pursues this idea aggressively – almost all concurrent behavior is captured in library modules and then instantiated with *modules* containing purely sequential code.
Why is this good?

What is the advantage of using modules instead of functions for instantiation?
We want to generalize behaviors that extend beyond a single function to a set of related functions. Example – a server requires code for initialization as well as steady-state operation
Aside 1: process control systems folklore – system initialization often requires the intensive assistance of the system designers because PCSs are often designed without sufficient attention to the startup process.
Aside 2: does the OTP approach alleviate this by calling explicit attention to the startup phase?
Aside 3: what does this idea correspond to in Java?

```
%% basic server behavior
-module(server1).
-export([start/2, rpc/2]).

start(Name, Mod) ->
    register(Name, spawn(fun() -> loop(Name, Mod, Mod:init()) end)).

rpc(Name, Request) ->
    Name ! {self(), Request},
    receive
        {Name, Response} -> Response
    end.

loop(Name, Mod, State) ->
    receive
{From, Request} ->
            {Response, NextState} = Mod:handle(Request, State),
            From ! {Name, Response},
            loop(Name, Mod, NextState)
    end.
```

```
%% A name server callback module using basic server behavior

-module(name_server).
-export([init/0, add/2, whereis/1, handle/2]).
-import(server1, [rpc/2]).

%% client routines – aka client stubs
add(Name, Place) -> rpc(name_server, {add, Name, Place}).
whereis(Name)    -> rpc(name_server, {whereis, Name}).

%% callback routines
init() -> dict:new().

handle({add, Name, Place}, Dict) -> {ok, dict:store(Name, Place, Dict)};
handle({whereis, Name}, Dict)    -> {dict:find(Name, Dict), Dict}.
-------------------------------------------------------------------------------
%% then outside of name_server instantiate one using
1> server1:start(name_server, name_server).
2> name_server:add(joe, “at home”).
3> name_server:whereis(joe).
{ok, “at home”}
```

```
-module(server2).
-export([start/2, rpc/2]).
%% server behavior w/ transaction semantics

start(Name, Mod) ->
    register(Name, spawn(fun() -> loop(Name,Mod,Mod:init()) end)).

rpc(Name, Request) ->
    Name ! {self(), Request},
    receive
        {Name, crash} -> exit(rpc);
        {Name, ok, Response} -> Response
    end.

log_the_error(Name, Request, Why) ->
    io:format("Server ~p request ~p ~n"
      "caused exception ~p~n",
      [Name, Request, Why]).
```

```
%% loop for transactional semantics
loop(Name, Mod, OldState) ->
    receive
{From, Request} ->
    try Mod:handle(Request, OldState) of
            {Response, NewState} ->
                From ! {Name, ok, Response},
                loop(Name, Mod, NewState)#
    catch
            _:Why ->
                log_the_error(Name, Request, Why),
                %% send a message to cause the client to crash
                From ! {Name, crash},
                %% loop with the *original* state
                loop(Name, Mod, OldState)#
    end
    end.

%% the callback module doesn't change!
```

```
%% server behavior with “hot” code replacement
-module(server3).
-export([start/2, rpc/2, swap_code/2]).

start(Name, Mod) ->
    register(Name,
     spawn(fun() -> loop(Name,Mod,Mod:init()) end)).

rpc(Name, Request) ->
    Name ! {self(), Request},
    receive
        {Name, Response} -> Response
    end.
```

```
%% The server implements the swap_code operation
%% and passes other ops off to the callback module

swap_code(Name, Mod) -> rpc(Name, {swap_code, Mod}).

loop(Name, Mod, OldState) ->
    receive
{From, {swap_code, NewCallBackMod}} ->
    From ! {Name, ack},
    loop(Name, NewCallBackMod, OldState);
{From, Request} ->
    {Response, NewState} = Mod:handle(Request, OldState),
    From ! {Name, Response},
    loop(Name, Mod, NewState)#
    end.
```

Supervison Trees

Worker processes – gen_server, gen_fsm

The examples above are “toy” generic server code to illustrate the main ideas.
Real OTP gen_server details are different but same idea

Supervisor processes/Supervision trees:

Long-lived systems need to protect against failures of software (and hardware)
supervisor:start(CallBackModule, Arguments)
CBM contains init/1 function
init/1 returns {ok, SupervisorSpec, ChildSpecList}
Each ChildSpec is

{Id, {M, F, A}, …, Type, …}

SupervisorSpec has form {RestartStrategy, AllowedRestarts, InSeconds}
Restart strategy is one of: one_for_one, one_for_all, rest_for_one
If restarting too frequently, the *supervisor* terminates!

The gen_server of OTP implements these behaviors and more, including support for *supervision trees.*

These code-swapping servers are a key component of high-availability systems – you can keep a system up for years while updating the software as it runs.

Joe Armstrong describes this in his PhD thesis “Making Reliable Distributed Systems in the Presence of Software Errors” -- it's worth a read.