Elixir 2

A fun talk on “let it crash”

http://ferd.ca/the-zen-of-erlang.html

Erlang: “Let it crash!”

“The view is that you don't need to program defensively. If there are any errors, the process is automatically terminated, and this is reported to any processes that were monitoring the crashed process. In fact, defensive programming in Erlang is frowned upon.”

Erlang: “Let it crash!”

“Only program the happy case, what the specification says the task is supposed to do.”

“When writing code from a specification, the specification says what the code is supposed to do, it does not tell you what you’re supposed to do if the real world situation deviates from the specification”

“So what do the programmers do? ... they take ad hoc decisions”

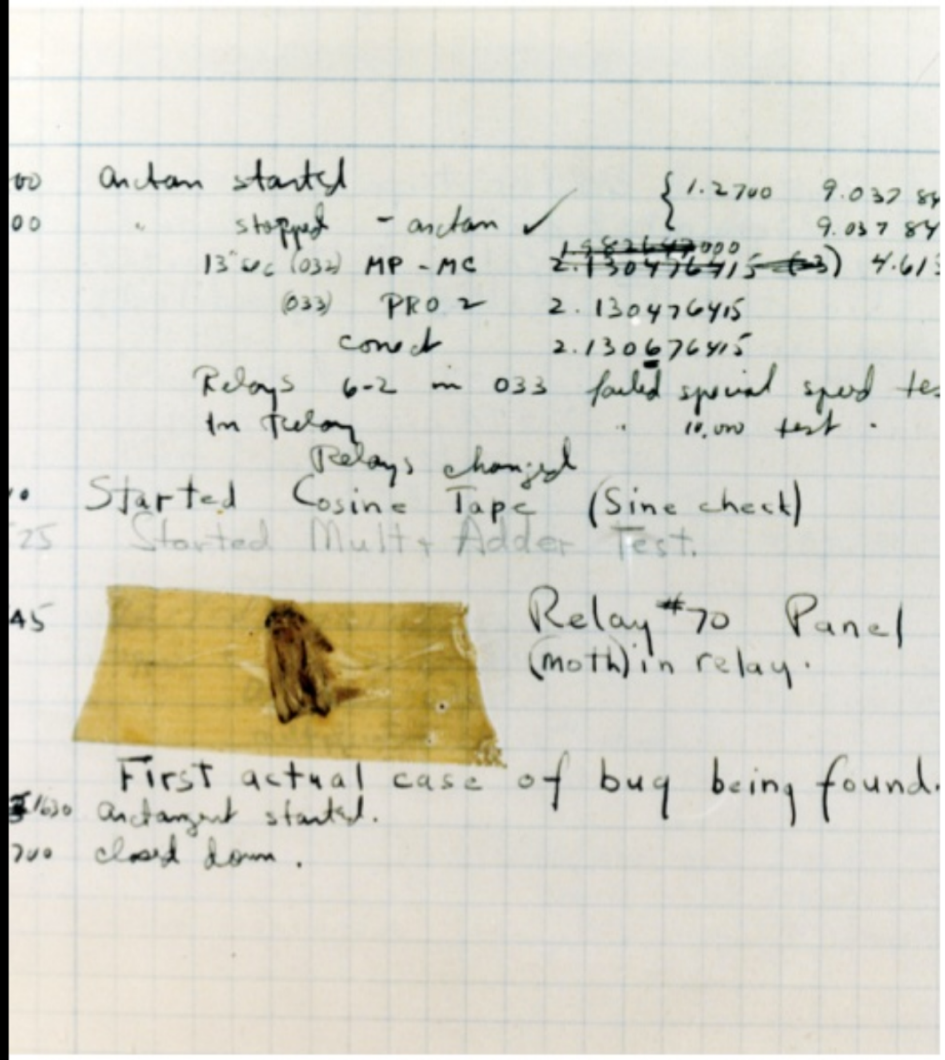
Erlang: “Let it crash!”

“If a hardware failure requires any immediate administrative action, the service simply won’t scale cost-effectively and reliably. The entire service must be capable of surviving failure without human administrative interaction. Failure recovery must be a very simple path and that path must be tested frequently.

Armando Fox of Stanford has argued that the best way to test the failure path is never to shut the service down normally. Just hard-fail it. This sounds counter-intuitive, but if the failure paths aren’t frequently used, they won’t work when needed.”

Erlang: “Let it crash!”

- Do I know how I’m supposed to handle an error here?
- If not, then should I handle it? (Thus going back to specification)
- If something goes wrong in this function call, **can** I continue?
  - E.g. Line 2 depends on a file descriptor from line 1
- When I restart, can I **recover** from the crash here?
  - If not then it is a good indication that you need a specification for how to handle the error. Can I reset my state? Do I need to clean up? etc are also good questions to ask yourself
- Am I crashing in a valid place?
  - E.g. you should never crash (intentionally) inside a gen_server unless it is some kind of a worker process. A “main” process shouldn’t really be allowed to crash in the same sense as a worker process can. A worker process, say for an HTTP request, shouldn’t really be very defensive (perhaps a top-level try-catch to return some useful error)



WHY RESTARTING WORKS

HEISENBUGS & FRIENDS

EASE OF FINDING BUGS IN DEVELOPMENT

	REPEATABLE	TRANSIENT
CORE FEATURE	EASY	HARD
SECONDARY FEATURE	EASY, OFTEN OVERLOOKED	HARD

BUGS THAT HAPPEN IN PRODUCTION

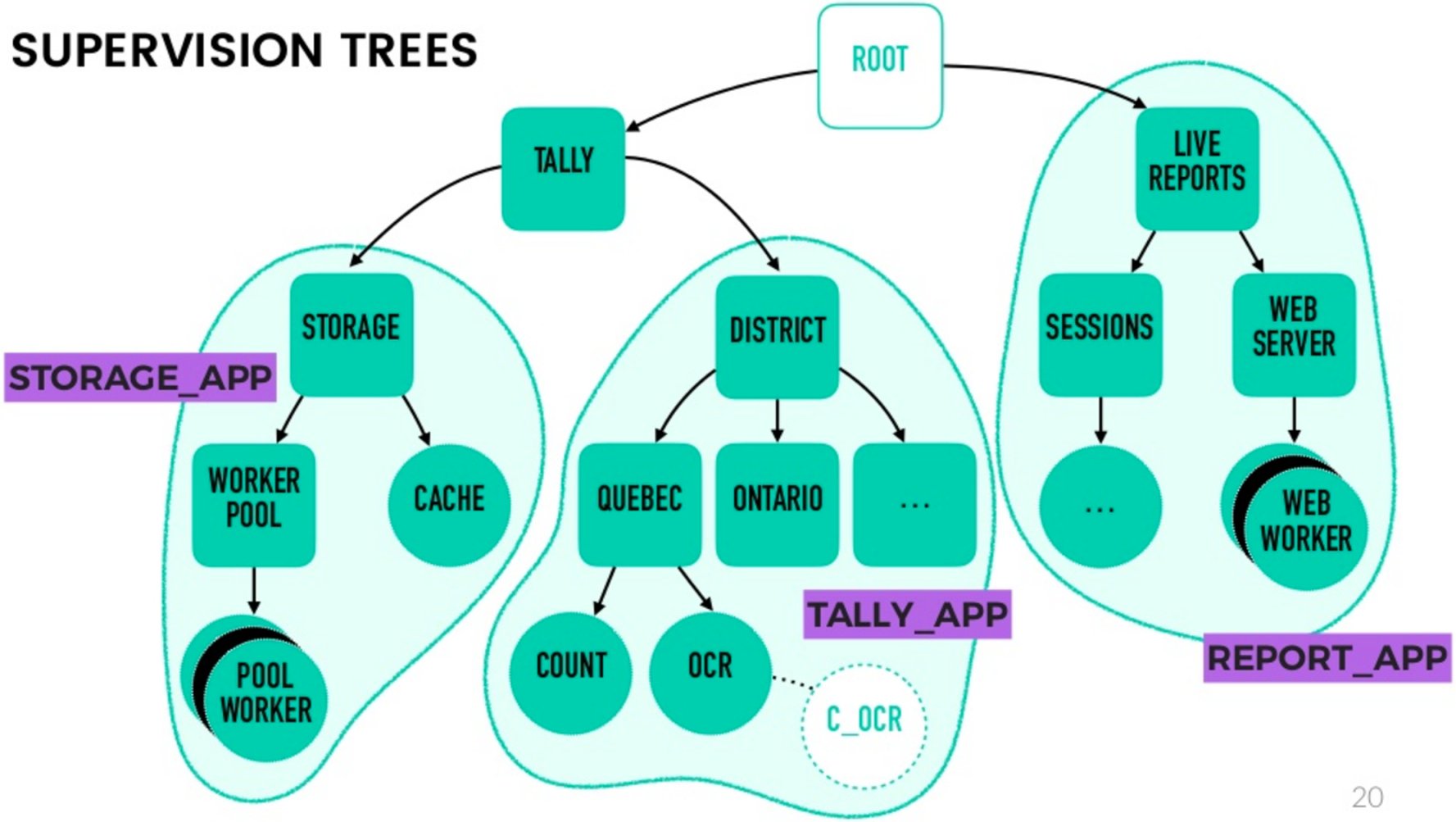
	REPEATABLE	TRANSIENT
CORE FEATURE	SHOULD NEVER	ALL THE TIME
SECONDARY FEATURE	PRETTY OFTEN	ALL THE TIME

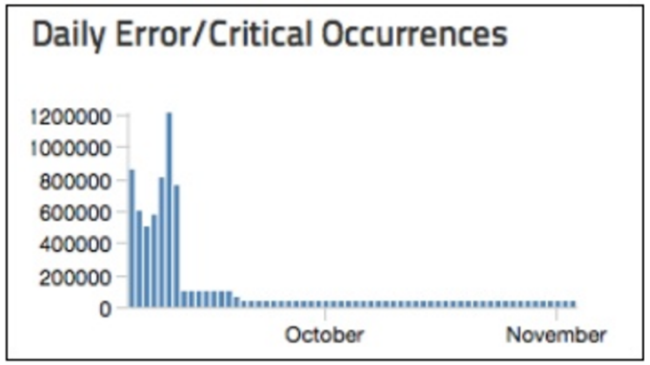
BUGS HANDLED BY RESTARTS

	REPEATABLE	TRANSIENT
CORE FEATURE	NO	YES
SECONDARY FEATURE	DEPENDS	YES

Clip slide

SUPERVISION TREES





SLEEP AT NIGHT

HOPEFULLY

but probably not...

Great system properties define

- what is critical or not to the survival of the system
- what is allowed to fail or not, and at which frequency it can do so before it is no longer sustainable
- how software should boot according to which guarantees, and in what order
- how software should fail, meaning it defines the legal states of partial failures you find yourself in, and how to roll back to a known stable state when this happens
- how software is upgraded (because it can be upgraded live, based on the supervision structure)
- how components interdepend on each other

“This is all extremely valuable. What's more valuable is forcing every developer to think in such terms from early on.”

Erlang and Elxir programs can achieve extreme reliability, *not* by never crashing, but by recovering after crashes.

Implementing a “Let It Crash” Elixir program…

? Can parts of the system crash and reboot without causing the program to fail?

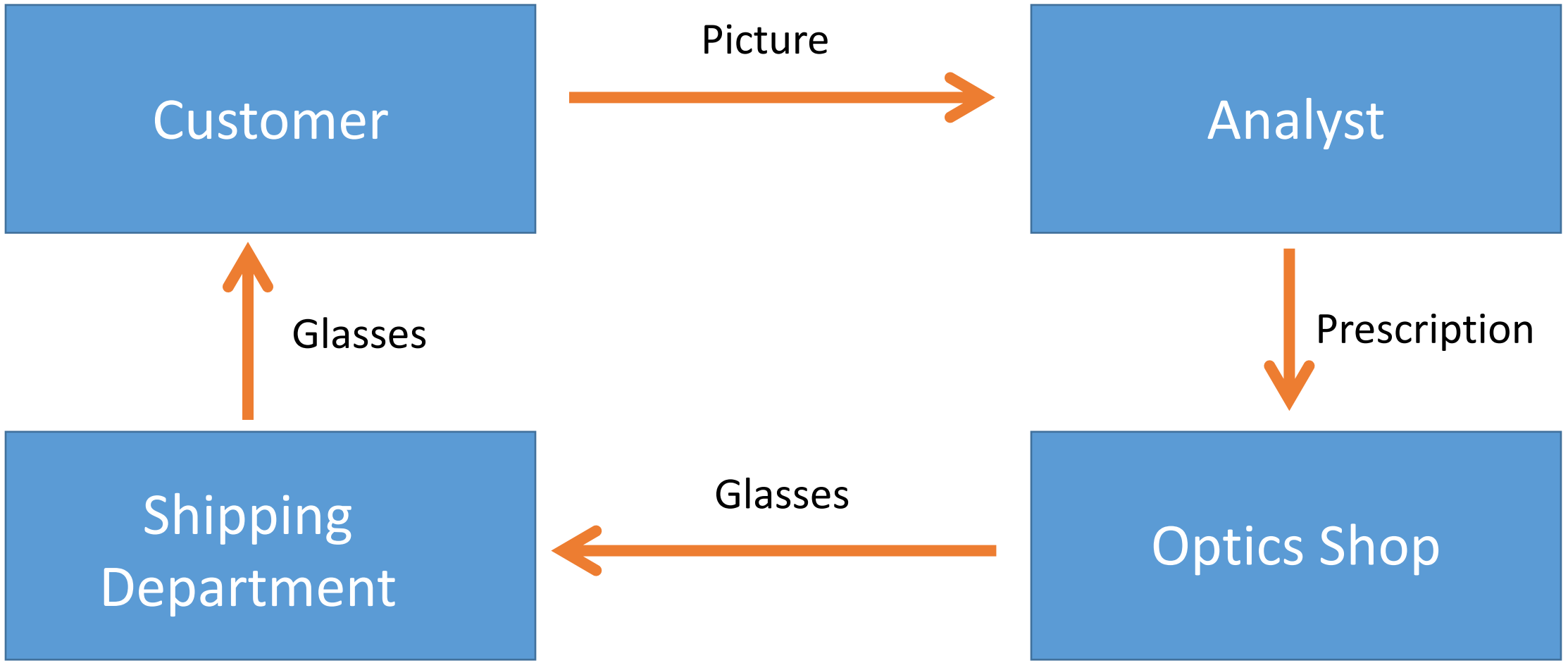
? Can the system scale to larger and larger systems effectively?

Please consider the following…

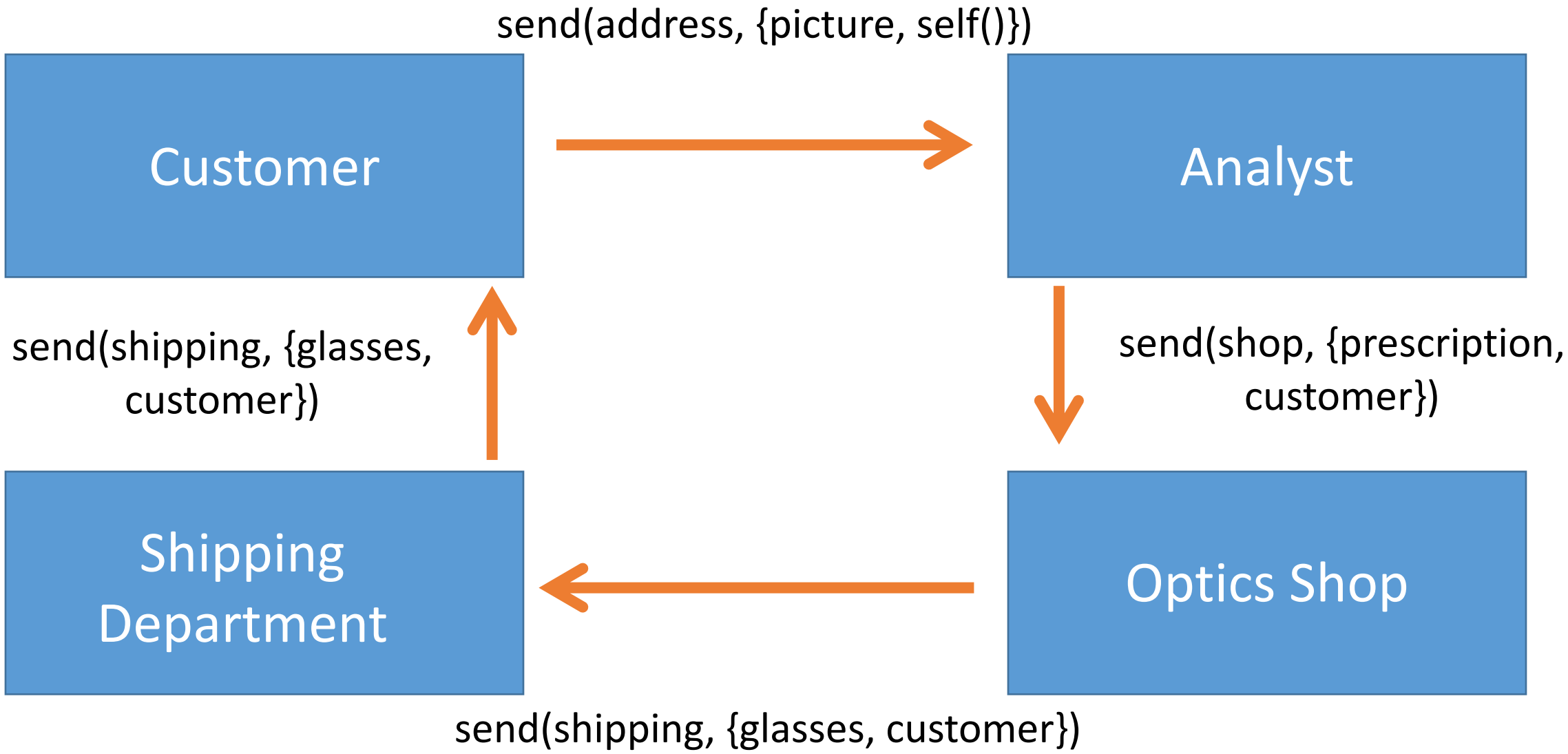
Glasses O-Matic
The Eye Company of the Future

Waste no more time coming to the eye doctor. Simply send us a picture of your eyes. Our computer will analyze your eyes and determine the glasses you need. We take care of your prescription in our specialized optics shop, and we mail you your glasses right away.

Glasses O-Matic



Possible Implementation 1: PID Passing



Faults of PID Passing

- Requires a lot of manual tracking of pids.
- Can become unclear what is a variable and what is a pid.
- Requires a lot of manual monitoring and notifications.
  - “All processes, please be aware that the customer moved to ….”
  - “All processes, the optics shop burned down. The new one is at…”
- May require a process to pass many pids for future processes that may need them.
- If a process crashes, it may also lose the pids of the processes that were going to receive its message.

PID Passing

Can parts of the system crash and reboot without causing the program to fail?

Can the system scale to larger and larger systems effectively?

Possible Implementation 2: Registering Processes (The Phonebook Method).

- Make a list of known processes. Register them in a global database that all processes can access.
  - Process.register(***pid, :Name***) -- gives the ***pid*** a globally accessible name.
  - Process.unregister(***pid***) -- removes the registration; if a registered process dies, it is automatically unregistered
  - Process.whereis(***:Name***) -- gets the pid of a registered process, or nil if no such process

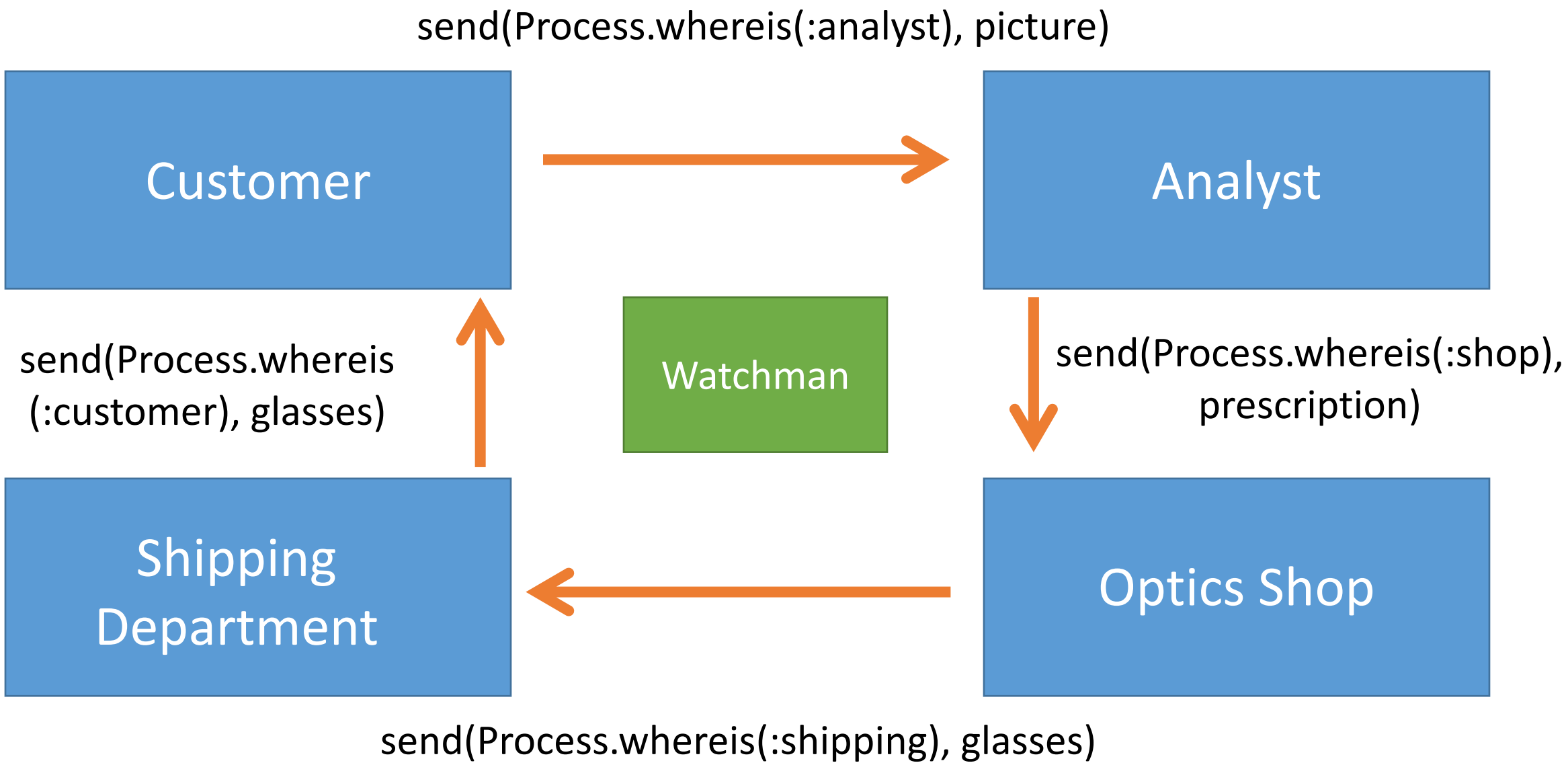
Linking and Monitoring processes

- You can link two processes--this means, if one process dies, the other receives an exit signal
  - Linking is symmetric; if A is linked to B, B is linked to A
  - If a “normal” process receives an exit signal, it too will exit

- Process.link(***pid***) - Links the calling process to the process at the ***pid.***
- You can also have a process monitor another--this means, if one process dies, the other receives a message about the exit.
  - If process A monitors B, and B dies, A does not and can reboot B.

- Process.monitor(***pid***) - Allows the calling process to monitor the process at the ***pid.***

Possible Implementation 2: Registering



Registering Processes

- Pros
  - Eliminates the need to send pid information with messages.
  - Allows for a single process to monitor multiple processes simultaneously.
  - Can quickly reboot a process without having to notify other processes.
    - “No one will ever know that the optics shop burned down…”
  - Processes don’t need to know the chain of command. Simply communicate directly with the process that needs the information.
- Cons
  - Requires strict adherence to naming conventions or clear documentation on used names.

Registering Processes

Can parts of the system crash and reboot without causing the program to fail?

Can the system scale to larger and larger systems effectively?

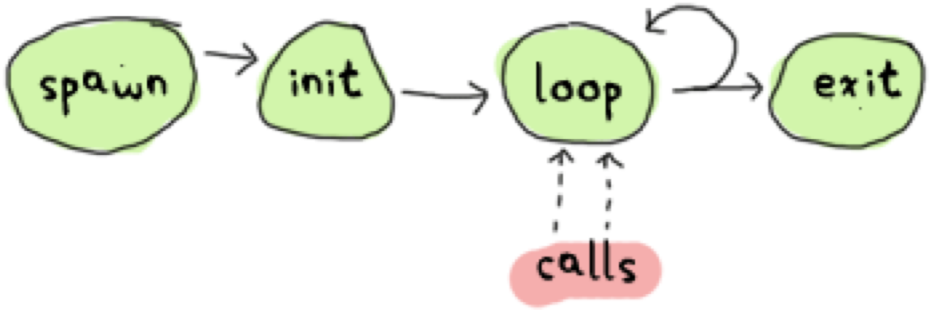
Possible Implementation 3: Behaviours

- Because so many concurrent applications have similar setups, the language has built-in hardened libraries that abstract common patterns.
- These libraries are known as **Behaviours** and do the hard work of knowing when to send and receive, keeping consistent protocols, linking appropriate processes, etc.
- These libraries are part of the Erlang Open Telecom Platform (**OTP**).
- These libraries are also available in Elixir.

Some Useful Behaviours

- **GenServer**
  - For implementing the server of a client-server relation
- **Supervisor**
  - For implementing a supervisor in a supervision tree
- **FSM**
  - For implementing finite-state machines
- **GenEvent**
  - For implementing event handling functionality

GenServer



- Can generate a standard interface for a client-server setup.
- Allows the developer to focus on functionality and callbacks.
- Has the following built in functions that the client can call:
  - GenServer.start_link/2 – Start the GenServer and link it to the current process.
  - GenServer.call/3 – Make a synchronous call to the server and wait for its reply.
  - GenServer.cast/2 – Make an asynchronous request to the server.
  - GenServer.stop/2 – Stop the server with a given reason.
  - GenServer.whereis/1 – Get the pid of a process known to the server.

Implementing a GenServer - Callbacks

- def init(state) do -

  Used to initialize the server's state and do all of the one-time tasks that it will depend on.

- def handle_call(request, from, state) do–

  Used to handle incoming **synchronous** messages (a response is needed).

- def handle_cast(request, state) do–

  Used to handle incoming **asynchronous** messages (no response is needed).

GenServer Implementation

```
defmodule Analyst do
        use GenServer

        def init(_) do
                IO.puts "Eye Analyst is ready to go!"
                Process.register(self(), :analyst)
                {:ok, self()}
        end

        def handle_cast(picture, _) do
                prescription = do_deep_eye_analysis(picture)
                GenServer.cast(Process.whereis(:shop), {prescription, pid})
        end

        def handle_call(_,_,_) do
                {:reply, "I don't respond directly. Please cast your message."}
        end
end
```

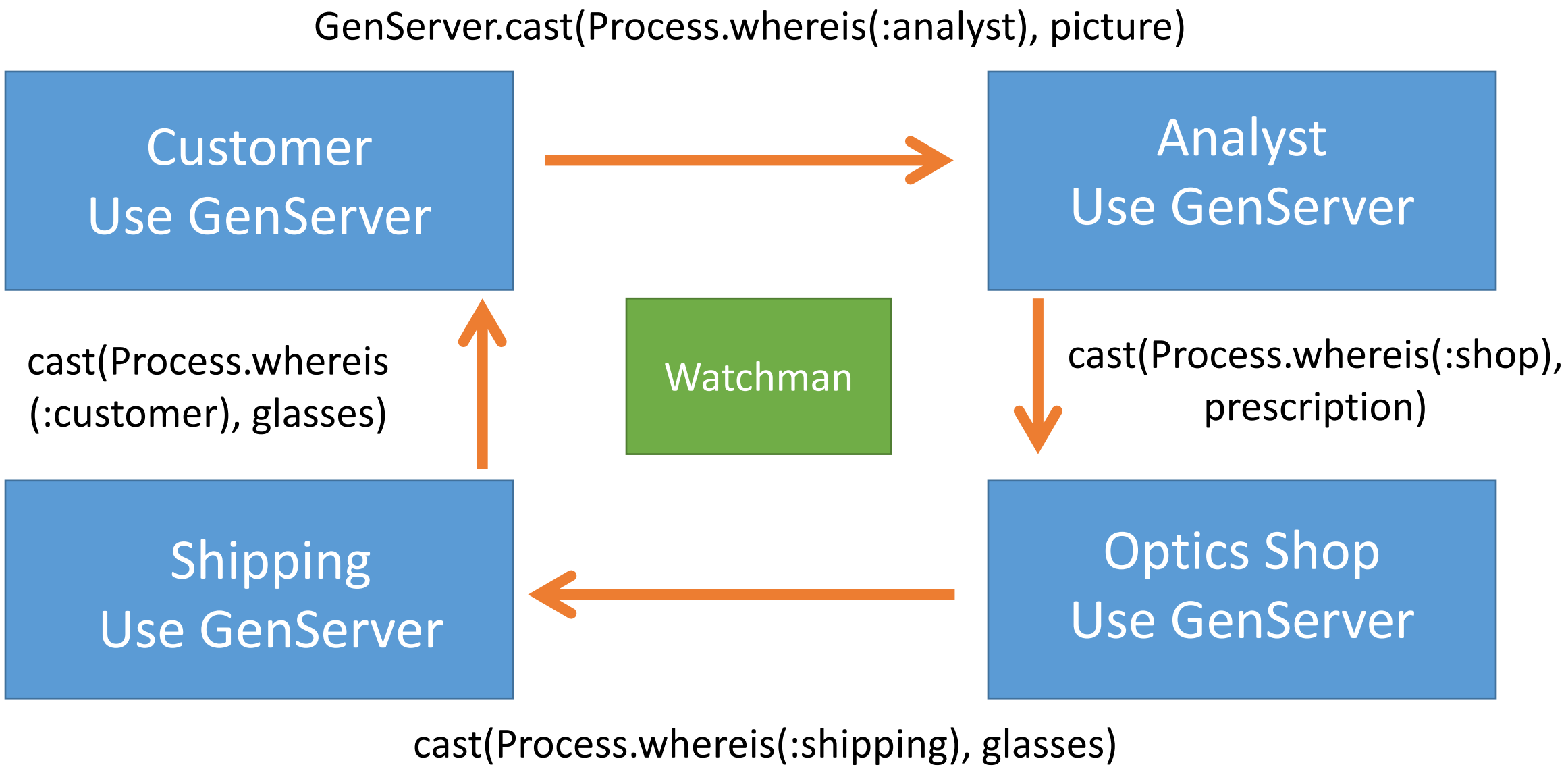
Console Output for Client

iex(51)> {:ok, pid} = GenServer.start_link(Analyst, [])
"Eye Analyst is ready to go!"
{:ok, #PID<0.167.0>}

iex(52)> GenServer.call(pid, picture)
"I don’t respond directly. Please cast your message."

iex(53)> GenServer.cast(pid, picture)
:ok

Possible Implementation 3: GenServers



GenServers

- Pros
  - All the same pros as before, but now it is a lot easier to handle incoming information.
  - Protocols stay consistent.
- Cons
  - Still doesn’t solve our process registering scaling problem.
  - For larger systems, it still doesn’t work to have a single watchmen to revive dead processes.

GenServers

Can parts of the system crash and reboot without causing the program to fail?

Can the system scale to larger and larger systems effectively?

Thinking About Interdependencies

- If we ever want to solve our scaling problem, we need an efficient way of dealing with the interdependence of processes.
- We want processes that depend on each other to know about each other and restart when there dependencies restart, but not restart the whole system.
- We want processes to only know about the processes that they need for them to function properly.
- This is done through **Local Name Servers** and **Supervisors**

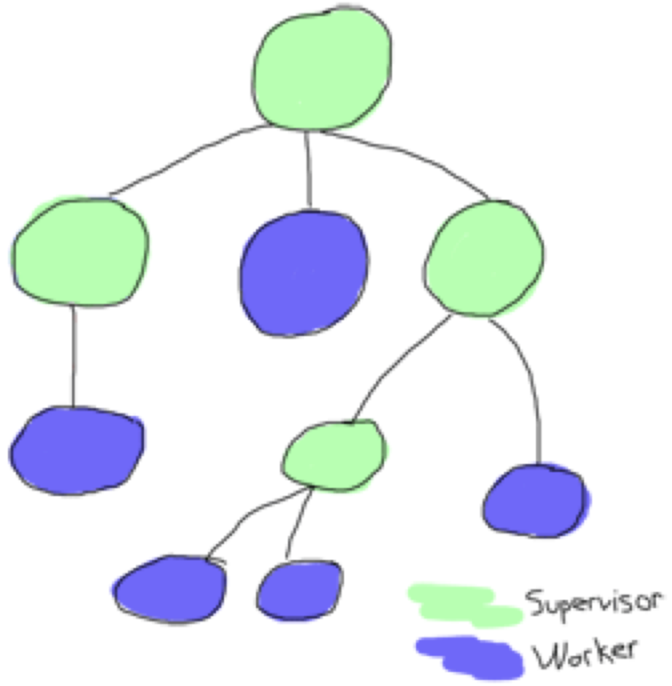
Local Name Servers

- Similar to process registration, but only involves keeping a small list for interdependent processes.
- Allows us to group processes together and keep track of naming within the grouped processes.
- Usually implemented with a GenServer (which you will do in your lab).
- Allows for the building of API facades that don’t require the outside developer to know intermediate names of processes.

Supervisor

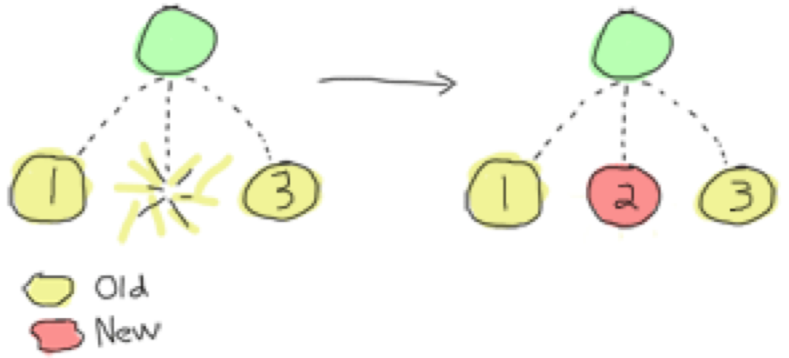
- Can monitor sub processes
  - which can be either workers or other supervisors
- Workers are usually not trusted

- Can provide orderly shutdown mechanisms
- Can also brutally kill off subtrees, if necessary

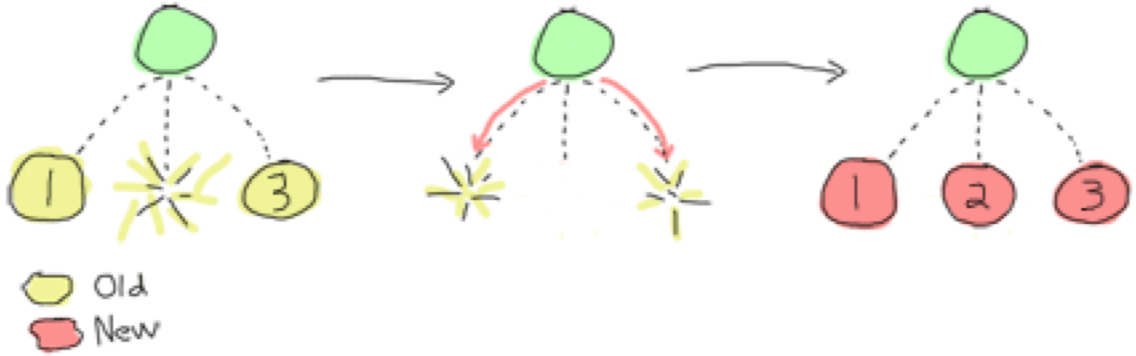


Restart strategies

- :one_for_one
  - if a supervisor supervises many subprocesses, and only one fails, only restart that one process

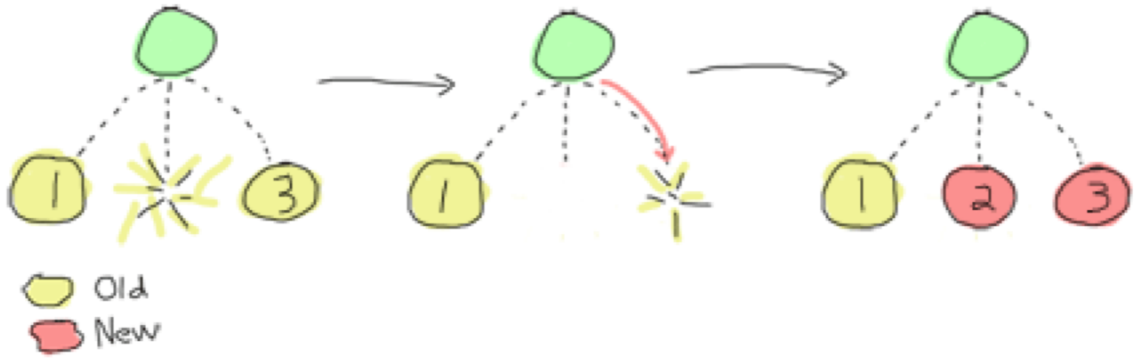


- :one_for_all
  - if any subprocess fails, all subprocesses are restarted



Restart strategies

- :rest_for_one
  - if you have a chain dependency, only processes that (transitively) depend on the failed process are restarted



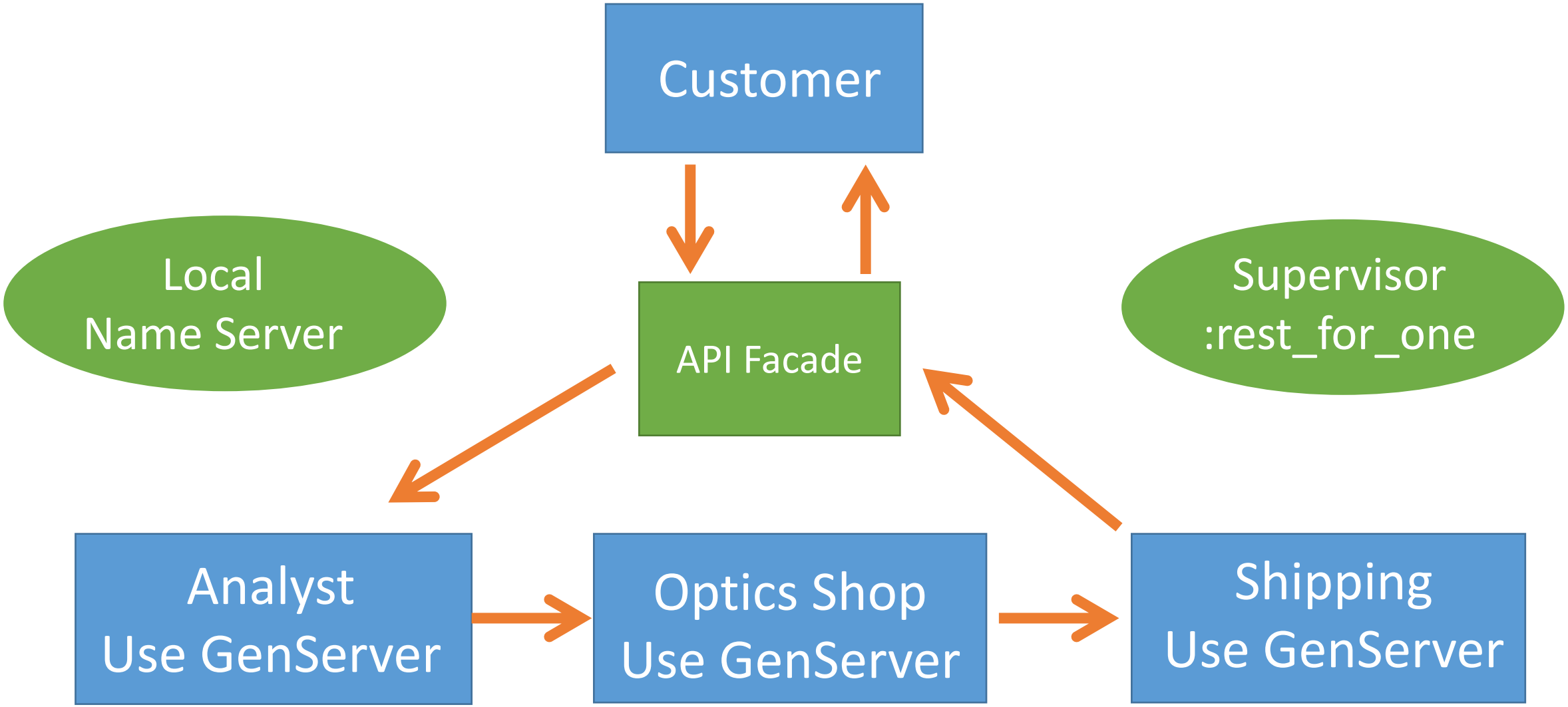
Supervisor Implementation

```
defmodule Watchman do
      use Supervisor

      def start_link(name_server) do
            Supervisor.start_link(__MODULE__, name_server )
      end

      def init(name_server) do
            children = [
                  worker(Shipping, [name_server]),
                  worker(OpticsShop, [name_server]),
                  worker(Analyst, [name_server])
                  ]
            supervise(children, strategy: :rest_for_one)
      end
end
```

Implementation 4: GenServers and Supervisors



GenServer with Supervisors

Can parts of the system crash and reboot without causing the program to fail?

Can the system scale to larger and larger systems effectively?

Cool Fact: GenServer code is hot-swappable

With Elixir’s goal of always being online, they have built methods into behaviours like GenServer to allow for updating the code for the server without having to stop the system.