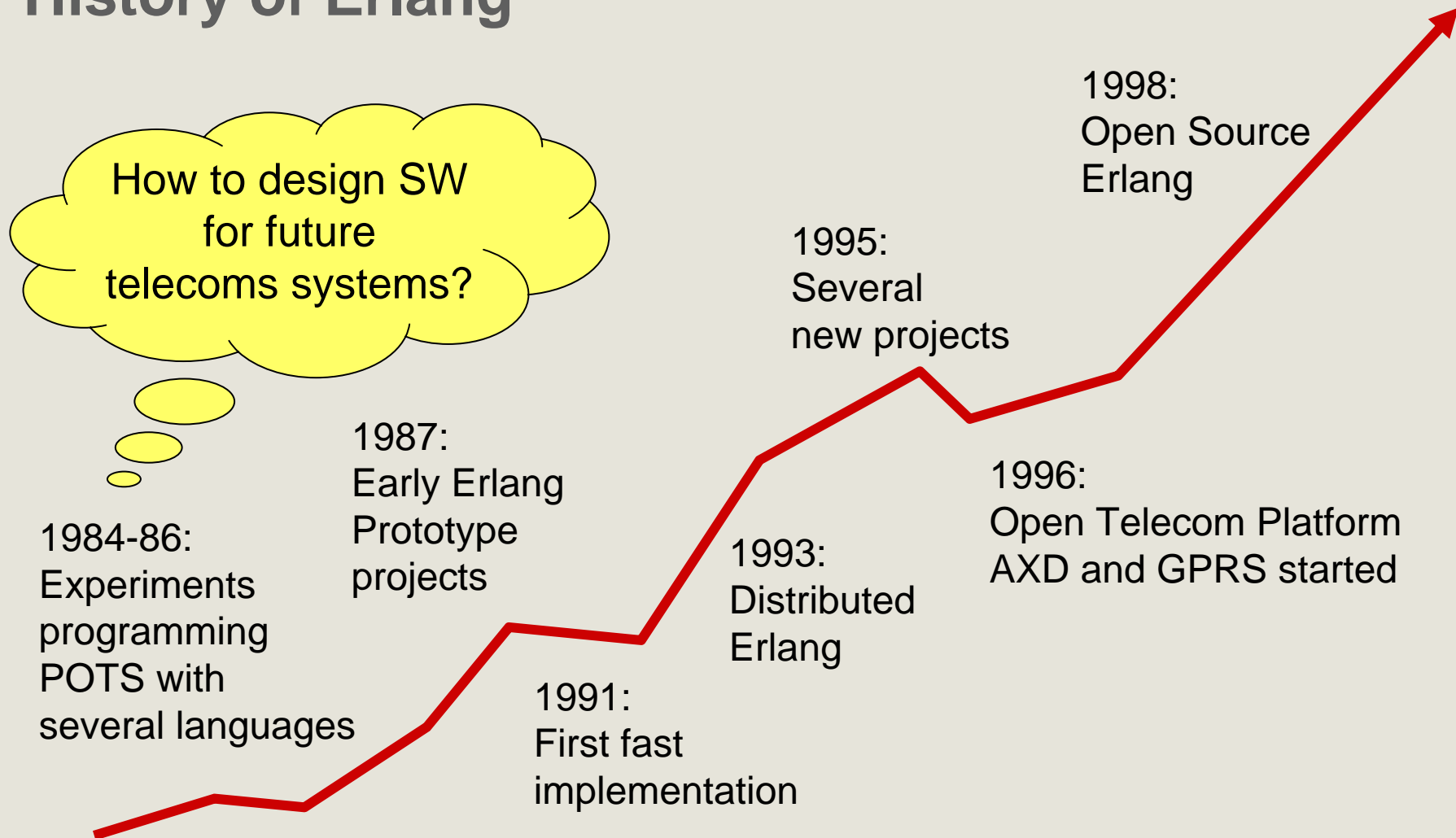


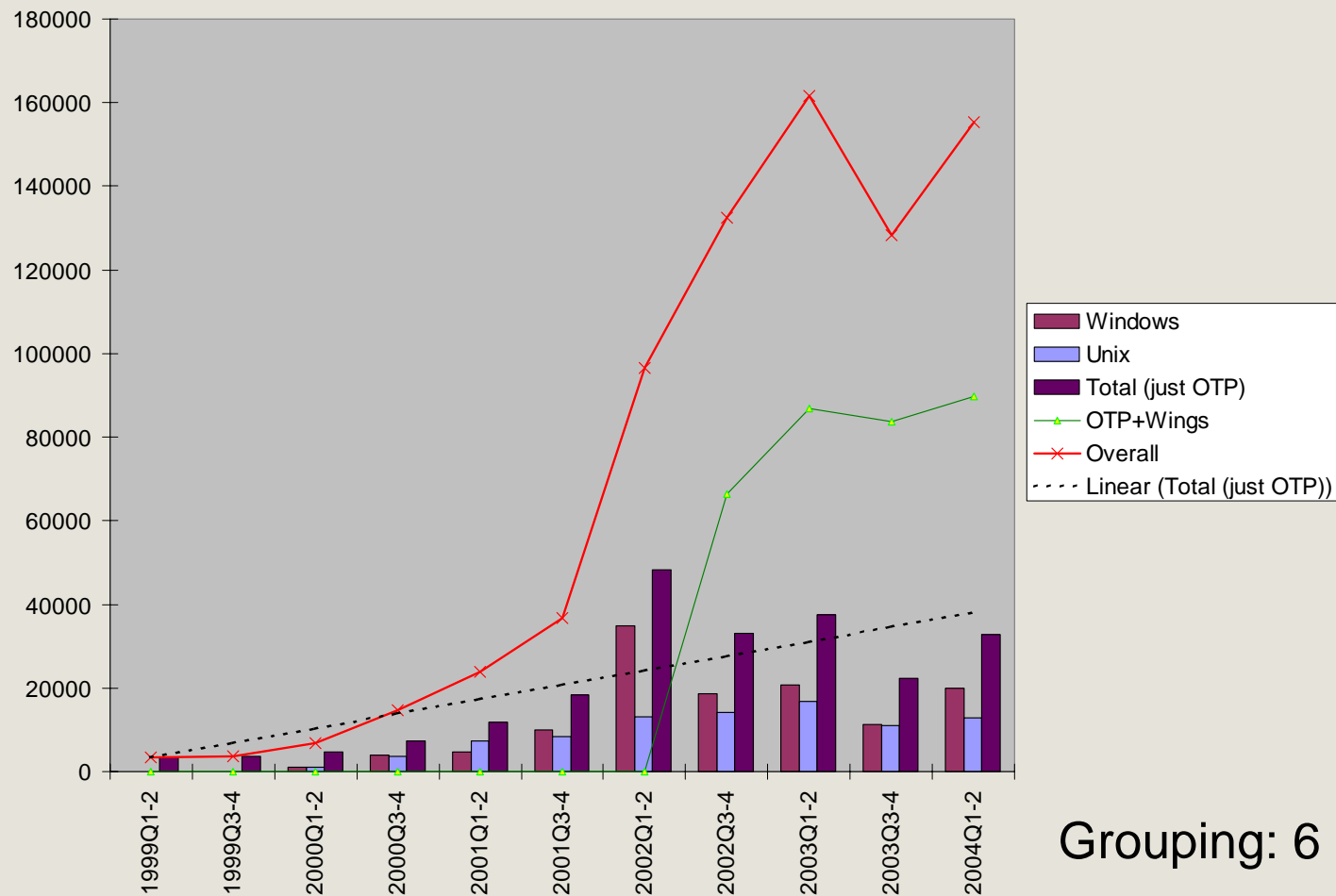
# 20 Years of Commercial Functional Programming

Ulf Wiger  
Senior Software Architect  
Ericsson AB

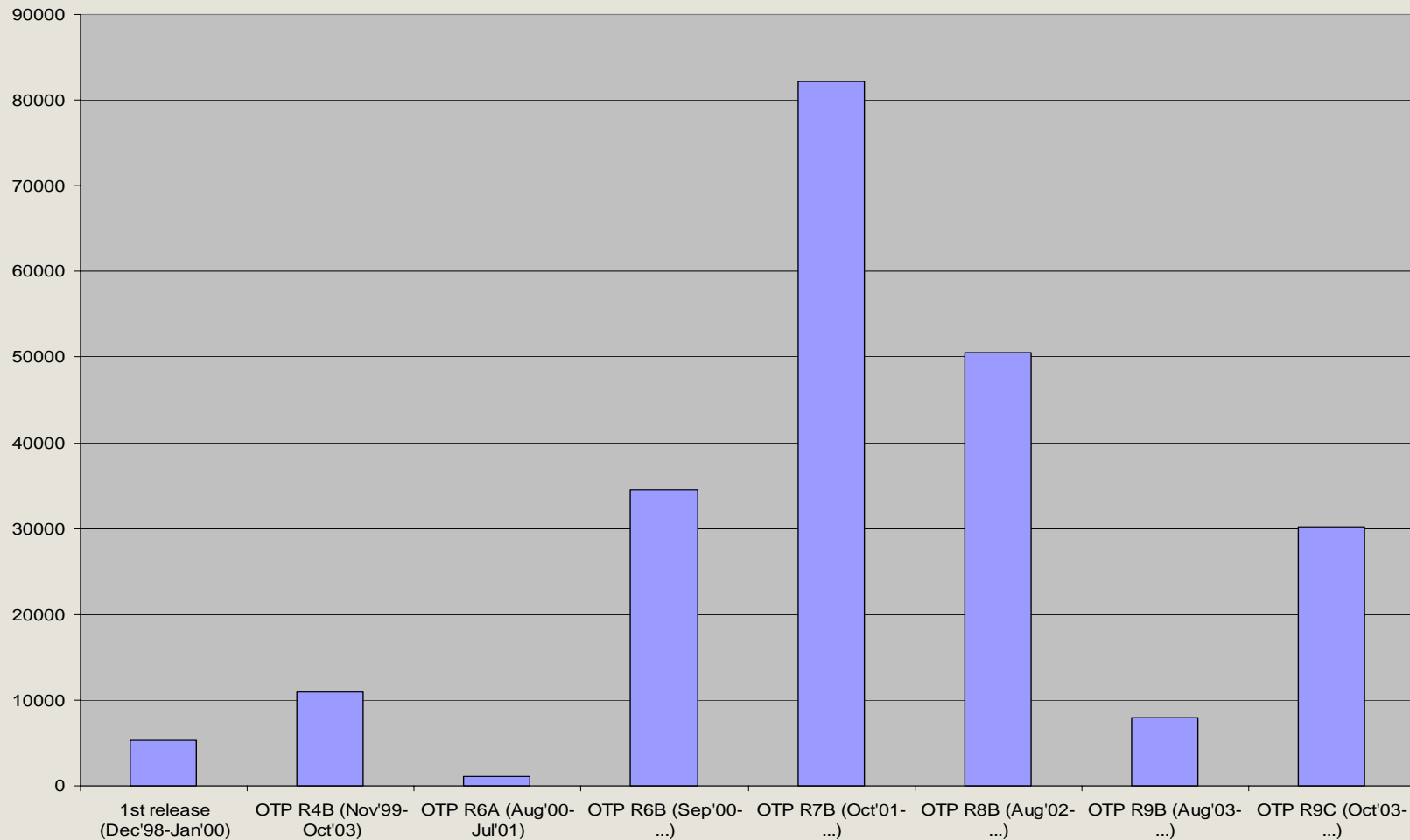
# History of Erlang



# Downloads since Open Source Launch '98



# Total Downloads per Erlang/OTP Release



## Known uses of Erlang

- Start-ups using Erlang to pull off what bigger companies cannot – **small competent teams**
- Nortel using Erlang because they bought a start-up (former Ericsson people) – **small expert team**
- Ericsson using Erlang in their core business (challenging the mainstream) – **large projects with "average" programmers**

## Challenging the Mainstream

- Most difficulties facing technology introduction have little to do with the actual technology.
- The perfect objection to new technology is one that:
  - Sounds quite technical and well considered.
  - Is difficult/impossible to prove or disprove.
- Instead of looking past the actual objection, engineers will spin their wheels trying to refute it.
  - ... and even if they succeed, few people will care.

## How to get off the ground

- The only things that might save you in the end, are commercial success, predictability and high quality.
- Better to do a few things well, than to do lots of things and risk doing some poorly.
- Make sure that you are actually solving an *identified* problem.
- Learn what decision makers care about, and learn to speak their language.
- Be patient!

## Case in point: Erlang

- Everyone "knows" that:
  - Erlang isn't fast enough.
  - Erlang isn't OO (well, this is fortunately true!)
  - It would be too hard to recruit Erlang programmers.
  - If Erlang is indeed better at something, the Mainstream will catch up soon enough (always in the Next Release™.)
  - If an Erlang project succeeds, it's due to excellent project management or unusually skilled designers.
  - If an Erlang project fails, it's proof that Erlang isn't better.



## Objection: Erlang isn't fast enough

- No competing product outperforms Ericsson's ENGINE solution for Telephony over packet-based networks.
- No competing product outperforms Ericsson's Erlang-based GPRS Signalling Support Node (SGSN).
- No competing product outperforms Nortel's Erlang-based SSL Offload Accelerator.
- **Lesson learned:** it's nearly impossible to predict system performance based on low-level "micro" benchmarks.
- Failure to manage complexity often kills performance.

## Objection: It's too hard to hire Erlang programmers

- **Lesson learned:** Well, you need to work at it, but not at all impossible.
- Difficult to hire expert programmers in general.
- Hiring expert C++ (or Java) programmers has proven quite difficult (due to their exceptionally high market value.)
- Establishing relationships with local universities (even in the U.S.) has proven fruitful for Erlang-based projects.
- Companies should not try to recruit programmers who know/accept only one language – look for solid Comp. Sci. competence instead. Erlang itself is not very difficult.

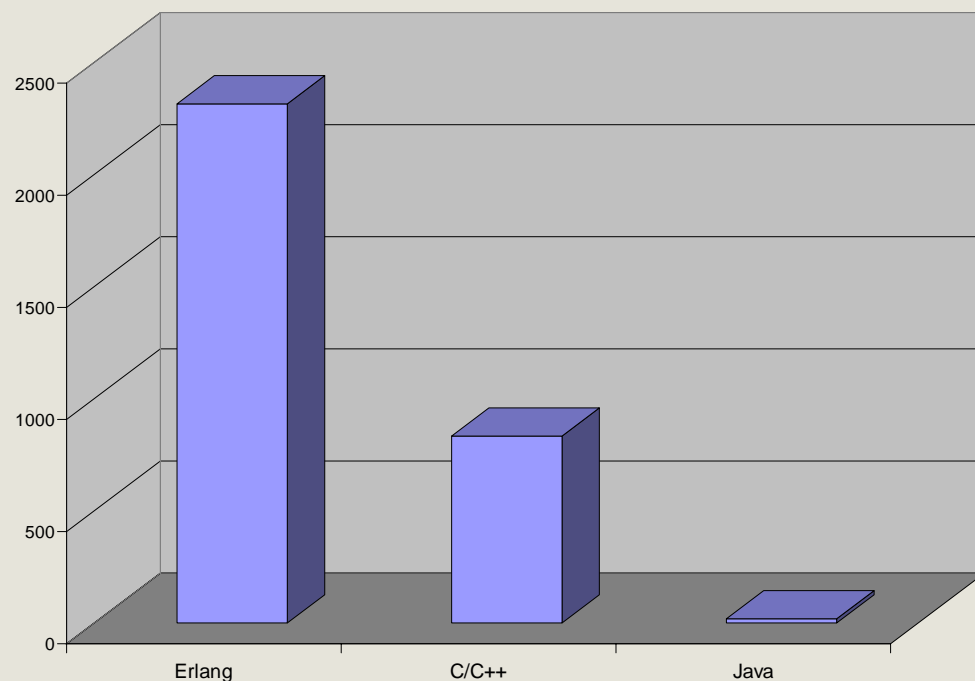
## Objection: Mainstream will catch up

- 12 years...
- Still no serious contender to Erlang in the area of concurrency-oriented programming.
- Still waiting for the next Java compiler that will solve all problems.
- Waiting for UML 2.0 to address issues with concurrency & exception handling (then 3.0 for Executable UML, etc.)
- And Erlang is improving at a good pace.

# Erlang in AXD 301

- Erlang is ideal for:
  - Complex control logic
  - Concurrency/state machines
  - Program supervision
  - Distribution/redundancy
- The whole system comprises:
  - Hardware
  - Hardware control (drivers)
  - Bootstrap logic (C, shell scripts)
  - Application logic
  - Often, 3<sup>rd</sup> party software
  - Operator GUI (HTML, Java, ...)

KLOCs, AXD 301 Control System



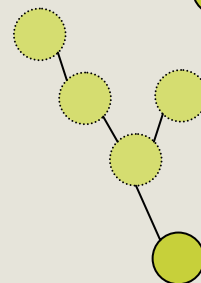
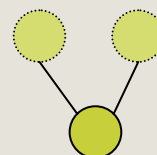
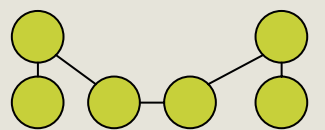
## Enter the positive spiral

- AXD 301 claimed 4x productivity & quality edge over conventional approaches
  - But initially, designers didn't know Erlang, OTP was in beta, etc.
- The project was fun, Erlang was fun
  - designers stayed – low personnel churn.
- This allowed us to mature as an organization
- Now, we have better methods, better tools, more experience.
  - Our delivery precision is second only to that of Erlang/OTP
- Products getting more complex, which works in our favour.

# The Erlang edge in large projects

- Intuitive abstractions.
- Loose coupling between modules gives good scalability.
- Few side-effects – easier to test.
- Outstanding trace and debug facilities.
- Relatively easy to rewrite stuff.
- Easy to support other people's code.
- Easy for a small SWAT team to work across the board with system improvements.

## Example: AXD301 process model



1<sup>st</sup> vsn:

6 processes/call



2 processes/call



1 process/all calls



2 processes/  
call transaction



4-5 processes/  
call transaction

## How can we improve even more?

- Designing tools is not Ericsson's core business
- Mainstream tool makers not that interested in FP or Erlang
- Alternative venues:
  - Open Source collaboration
  - Industry/research collaboration
- We are willing to assist research projects!

# When will we assist research projects?

*("we" = a large industrial development project)*

- Continuously, on a small scale, e.g. with code samples, information, suggestions – e.g. Erlang Verification Project, various type checking work, etc. (pro-bono)
- Now and then, we host a researcher and assist in running a small prototype – e.g. the HiPE team's new type analyser. (low-risk, fun)
- Willing to support a small project if it's likely to produce tangible results (=cost savings or improved product) within, say, 2 years. (exciting potential)



# Research Challenge: How to track messages?

- Hiding message passing inside a function is a wonderful feature, esp. for client-server calls.
- You can use Emacs Tags to jump around, using the Module:Function structure.
- But sometimes, you want a message-passing interface
- This makes it hard to know where to jump to – there is no Process:ReceiveClause structure in the code.

```
%% client code
call(P, Request, Timeout) ->
  Ref = erlang:monitor(process, P),
  P ! {call, self(), Ref, Request},
  receive
    {Ref, Reply} ->
      erlang:demonitor(Ref),
      Reply;
    {'DOWN', process, Ref, Reason} ->
      exit({server_died, Reason})
  after Timeout ->
    exit(timeout)
end.

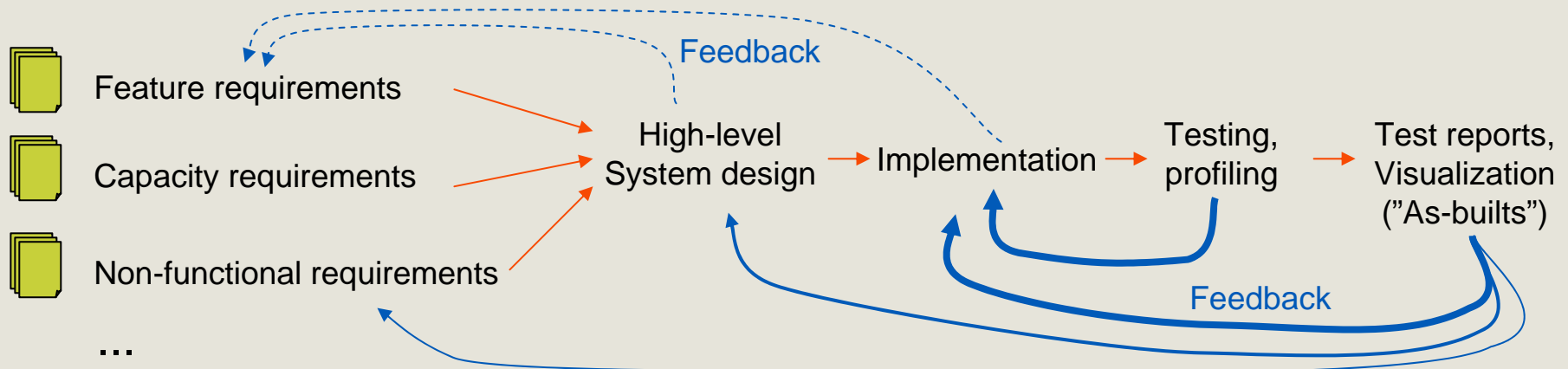
%% server code
loop(State) ->
  receive
    {call, From, Ref, Request} ->
      {Reply, State1} =
        handle_call(Request, State),
      From ! {Ref, Reply},
      loop(State1)
    ... -> ...
  end.
```

## Research Challenge: How to determine a component's "footprint"?

- When a component has changed – how much do you need to re-test?
  - The programmer: “if it compiles, it works!”
  - The project manager: “we have to re-test everything!”
  - The truth normally lies somewhere in between, but where exactly?
- Testing is inevitable in industry, and very expensive.
- What parameters can affect testing outcome?
  - API changes (of course)
  - Message passing sequence & timing changes
  - Algorithmic complexity, timing, CPU & memory consumption
  - ...

# Research Challenge: Specifications, implementation, and "as-builts"

- SW design involves several orthogonal activities/teams
- How to exchange relevant info & close feedback loops?
- (Suggestion: Expressive programming languages coupled with different visualization tools and a non-formal to semi-formal requirements notation.)



## Examples of Visualization

- Jan Nyström, PhD, used static analysis to draw the (static) process tree of Erlang applications.
- Thomas Arts, PhD, hooked into Erlang's trace support and generated state transition diagrams + exported trace analyses to model checking tools.
- Would like to see weighted block dependency diagrams generated through static analysis of code.

