The BEAM Programming Paradigm *

Kenji Rikitake | @jj1bdx

#CodeBEAMSTO 2019

* ... Or how I've been struggling to understand the well-designed ideas behind the Erlang/OTP, Elixir, and other BEAM languages and systems, while I still have a very hard time to learn "object-oriented" programming languages

Kenji Rikitake
17-MAY-2019
Code Beam STO 2019
Stockholm, Sweden
@jj1bdx

Programming paradigm?

What is that?

Is it about a programming *paradise*?

Paradigm = pattern + worldview [1]

- A typical example or pattern of something; a model
- A worldview underlying the theories and methodology of a particular scientific subject

[1] New Oxford American Dictionary, macOS 10.14.4

Programming paradigm, shown in Wikipedia

Programming paradigms are a way to classify programming languages based on their features.

— *Wikipedia*

Languages -> paradigms -> concepts

- Many languages belong to one paradigm
- A languages may have many paradigms available
- A paradigm may have many concepts

Peter Van Roy states there are 27 different programming paradigms [2]

---

[2] Peter Van Roy: Programming Paradigms for Dummies: What Every Programmer Should Know, 2009, Section 2

Programming paradigm:

Language patterns, worldview, and features

Simplified characteristics of the features

Design philosophy

Then what is the BEAM Programming Paradigm?

The philosophy of the BEAM languages/systems:

Lagom

Lagom: not too much, not too little, just right

Lagom är bäst

Just the right amount is best / enough is as good as a feast [3]

---

[3] Wikitionary entry of "Lagom är bäst"

Lagom in philosophy

中庸 / Zhōngyōng, Chu-yaw

Confucianism: Doctrine of the Mean

μεσότης / mesotes

Aristotle: Golden Mean

Quote from Programming Erlang [4]

> **Don’t Create Too Many Processes**
>
> Remember that `pmap(F, L)` creates `length(L)` parallel processes. If `L` is very large, you will create a lot of processes. The best thing to do is create a *lagom* number of processes. Erlang comes from Sweden, and the word *lagom* loosely translated means “not too few, not too many, just about right.” Some say that this summarizes the Swedish character.

---

[4] Joe Armstrong, "Programming Erlang", Second Edition, Pragmatic Bookshelf, 2013, Section 26.3, "Parallelizing Sequential Code"

Computer is as greedy as people: anti-lagom

- People want *fast* actions: more speed in less time
- Speed-first programming: cutting corners, less secure
- People want more features (really?)
- Feature bloat: bloatware, software inefficiency
- Less stable, safe, and secure software

Lagom: accuracy transcends speed

- Safety transcends speed
- Simplicity transcends rich features
- Stability transcends convenience

... these targets are more easily actualized by thinking a bit about how lagom your software is

... and these are the phisolophy of *the BEAM programming paradigm*

Erlang's programming paradigms [5]

- Functional programming
- Message-passing concurrent programming
- Multi-agent programming (Erlang processes)
- Some shared states (Process dictionaries, ETS, Mnesia)

---

[5] Peter Van Roy: Programming Paradigms for Dummies: What Every Programmer Should Know, 2009, Figure 2 (Taxonomy of programming paradigms) and Table 1 (Layered structure of a definitive programming language)

A hidden BEAM programming paradigm and design: safety first, speed second [6]

- Strong enforcement of immutability
- deep-copied variables, no references
- ... Programmers still can write dangerous code if needed

---

[6] Kenji Rikitake, Erlang and Elixir Fest 2018 Keynote Presentation, 16-JUN-2018, Tokyo, Japan

Immutability [7]

- Once the value is stored, it cannot be changed
- No mutable variables on either Erlang or Elixir, *unless explicitly stated as an external function (e.g., ETS) or processes*
- Immutability makes debugging easier because all stored values of created objects during actions remain untouched

[7] José Valim, Comparing Elixir and Erlang variables, Plataformatec blog, January 12, 2016

Variable binding strategies between Erlang and Elixir differs with each other

- Erlang: single binding only, with implicit pattern matching
- Elixir: multiple binding allowed as default, pattern matching enforceable with the pin (^) operator

Erlang enforces single binding variables

```
1> A = 10.
10
2> A = 20.
** exception error: no match of right hand side value 20
% Each variable can only be bound *once and only once*
3> B = [1, 2].
[1,2]
4> [_, X] = B, X.
2 % Bindings are equivalent to the pattern matching
```

Advantages of Erlang's single-binding variables

- Debugging gets easier: once a variable is bound, it doesn't change until the function exits
- The meaning attached to every variable must be clearly defined, because no shared meaning is allowed

Erlang's ambiguity on case expression (1)

```
case an_expr() of
  % S is bound to an_expr()'s result
  {ok, S} -> do_when_matched();
  _ -> do_when_unmatched()
end
```

Erlang's ambiguity on case expression (2)

```
S = something, % newly added
case an_expr() of
  % an_expr()'s result is pattern-matched implicitly
  % to the result of previous S instead
  {ok, S} -> do_when_matched();
  _ -> do_when_unmatched()
end
```

Elixir allows variable rebinding [8]

```
iex(1)> a = 10
10
iex(2)> a = 20
20 # a is rebound
# pin operator forces pattern matching without rebinding
iex(3)> ^a = 40
** (MatchError) no match of right hand side value: 40
```

[8] Stack Overflow: What is the “pin” operator for, and are Elixir variables mutable?

Advantages of Elixir's multiple binding

- Aligning well with the default behavior of many other languages
- Pattern-matching is explicitly controllable to remove ambiguity, e.g. for case expressions

Elixir on case expression (1)

```
s = :a_previous_value
case an_expr() do
  # s is bound to an_expr()'s result anyway
  {:ok, s} -> do_when_matched()
  _ -> do_when_unmatched()
end
```

Elixir on case expression (2)

```
s = :a_previous_value
case an_expr() do
  # an_expr()'s result is explicitly pattern-matched
  # with the content of s (:a_previous_value)
  # by the pin operator before s
  {:ok, ^s} -> do_when_matched()
  _ -> do_when_unmatched()
end
```

Erlang's deep-copied variables

```
1> A = 10, B = [A, 30].
[10,30]
2> f(A), A. % f(A): unbind A
* 1: variable 'A' is unbound
3> B.
[10,30] # old A remains in B
```

Elixir's deep-copied variables

```
iex(1)> a = 10; b = [a, 30]
[10, 30]
iex(2)> a = 20; [a, b]
[20, [10, 30]] # old a remains in b
```

Advantage of deep-copied variables

- Immutable, by always creating new object bodies for copying
- The same copy semantics is applied regardless of the data types, especially between simple (integers, atoms) and structured (lists, tuples, maps) types

Disadvantages of shared-nothing / deep-copied variables

- Slow: all assignments imply deep copying
- Much more memory space: *you cannot implicitly share*

… Are they really disadvantages at the age of abundant processing power and memory space?

Many of programming languages work in different ways *as default*

Variables are not necessarily immutable

Copy semantics differ between different data types

LISP is not necessarily immutable, even it's a functional language [9]

```
(defparameter *some-list* (list 'one 'two 'three 'four))
(rplaca *some-list* 'uno)
(rplacd (last *some-list*) 'not-nil)
; result by CLISP 2.49
(ONE TWO THREE FOUR) ; original
(UNO TWO THREE FOUR) ; head replaced
(UNO TWO THREE FOUR . NOT-NIL) ; tail replaced
```

[9] Source code example from Hyperspec Web site, modified by Kenji Rikitake, run on Wandbox with CLISP 2.49

JavaScript has a complicated copy semantics

```
// var a = {first: 1, second: 2}
// b = a // only sharing *references*
{ first: 1, second: 2 }
// a.second = 3
3
// b // changing a also changes b
{ first: 1, second: 3 }
// b == { first: 1, second: 3 }
false // WHY?
// The right-hand side is a *constructor*
```

C# also has a complicated copy semantics

Type int is value copied, List is *reference* copied (why??)

```
using System.Collections.Generic;
int i = 100; List<int> a = new List<int>(){10, 20};
MutableMethod(i, a);
void MutableMethod(int i, List<int> a) {
  i = 200; a.Add(30); }
```

Result: `i = 100, a = {10, 20, 30}`

C++: can you tell the difference?

```
double func(std::vector<double> x);
double func(std::vector<double> &v); // with reference
double func(std::unique_ptr<std::vector<double>> u);
double func(std::shared_ptr<std::vector<double>> s);

std::vector<double> y = x;
std::vector<double> &w = v; // with reference
std::unique_ptr<std::vector<double>> u2 = std::move(u);
// You cannot -> std::unique_ptr<std::vector<double>> u3 = u;
```

... actually, I'm not sure I can accurately explain the difference.

These languages perplex me by: [10]

- Different actions for different data types
- Constructors (and destructors)
- Copy semantics (C#: value type, reference type)
- Shallow-copied objects = no immutability
- Shared state and references as default

[10] Rikitake, K.: Shared Nothing Secure Programming in Erlang/OTP, IEICE Technical Report IA2014-11/ICSS2014-11, Vol. 114, No. 70, pp. 55--60 (2014). (Slide PDF)

Design of these languages

- Avoid object copying
- Creation of objects need explicit actions
- Explicit use of reference
- Object isolation is the programmer's responsibility

… *mostly for speed and cutting corners*

What BEAM languages provide

- Same actions for all data types
- No need for explicit constructors/destructors
- Single copy semantics (deep copy)
- Deep copied objects = immutability
- No shared state, no reference, as default

Design of BEAM languages

- Deep-copying as default
- New objects are always created by assignments
- Prohibit use of reference
- Object isolation is the language's responsibility

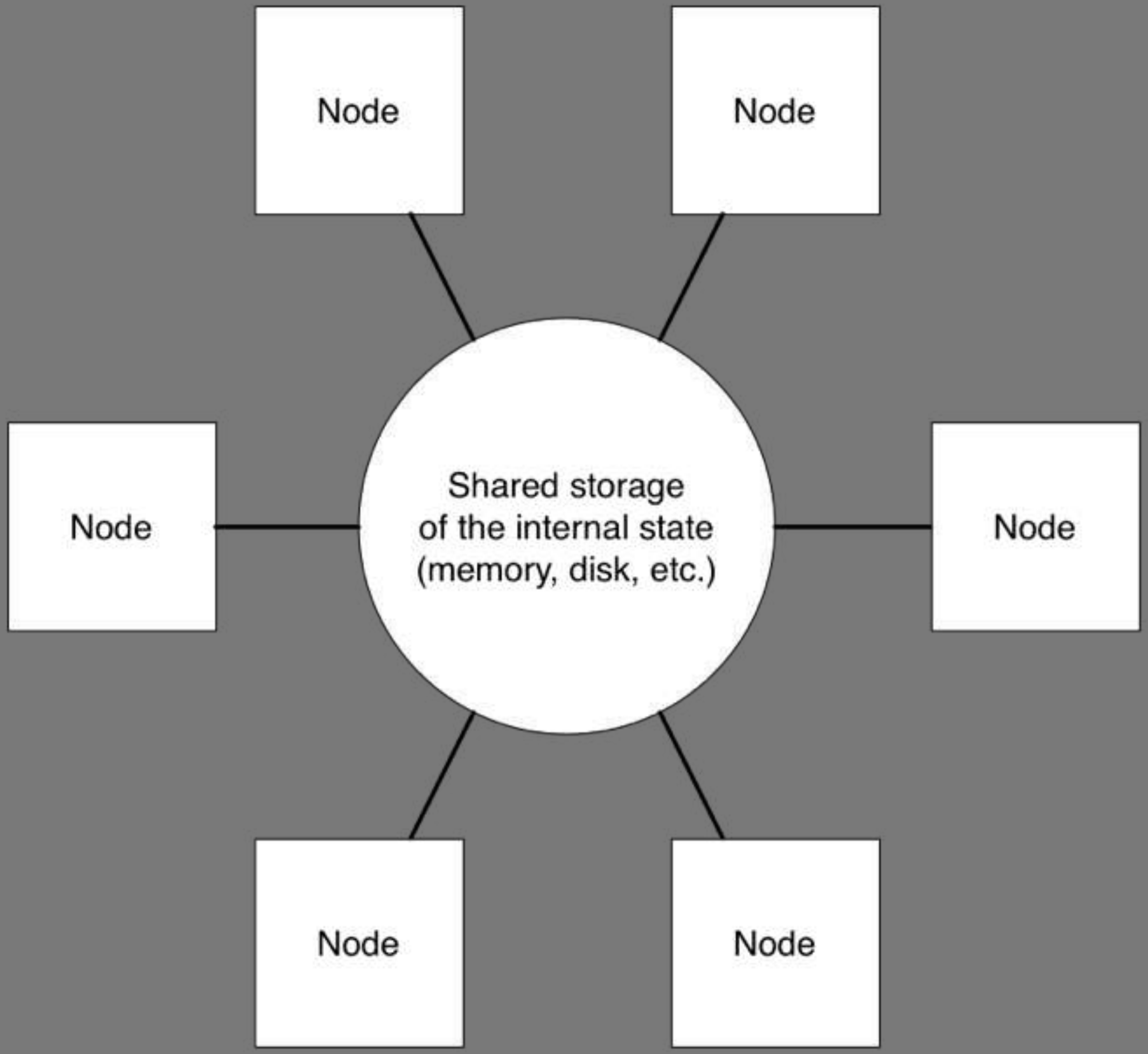
... *for security first, and lagom speed second*

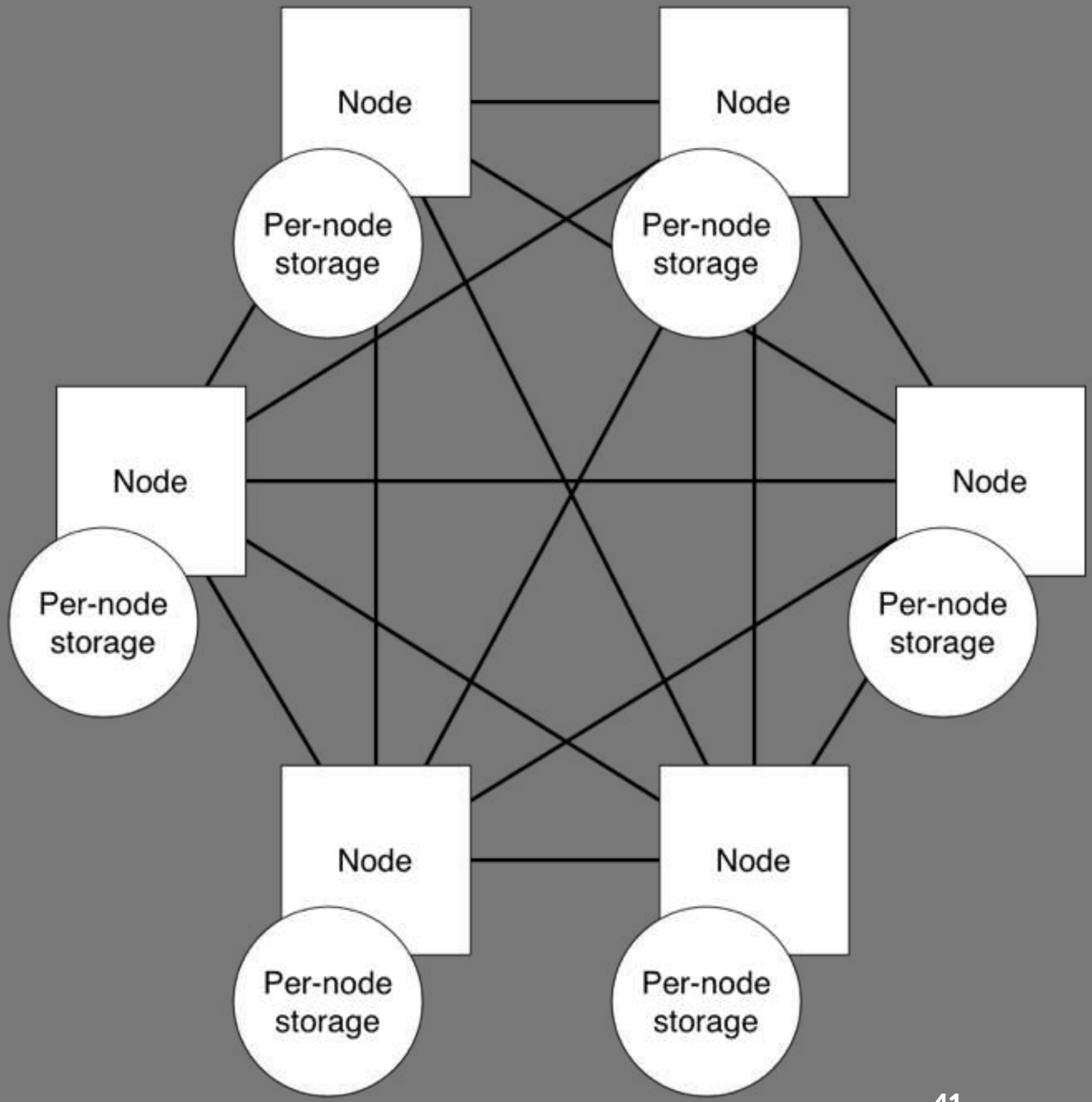
The BEAM Programming Paradigm difference

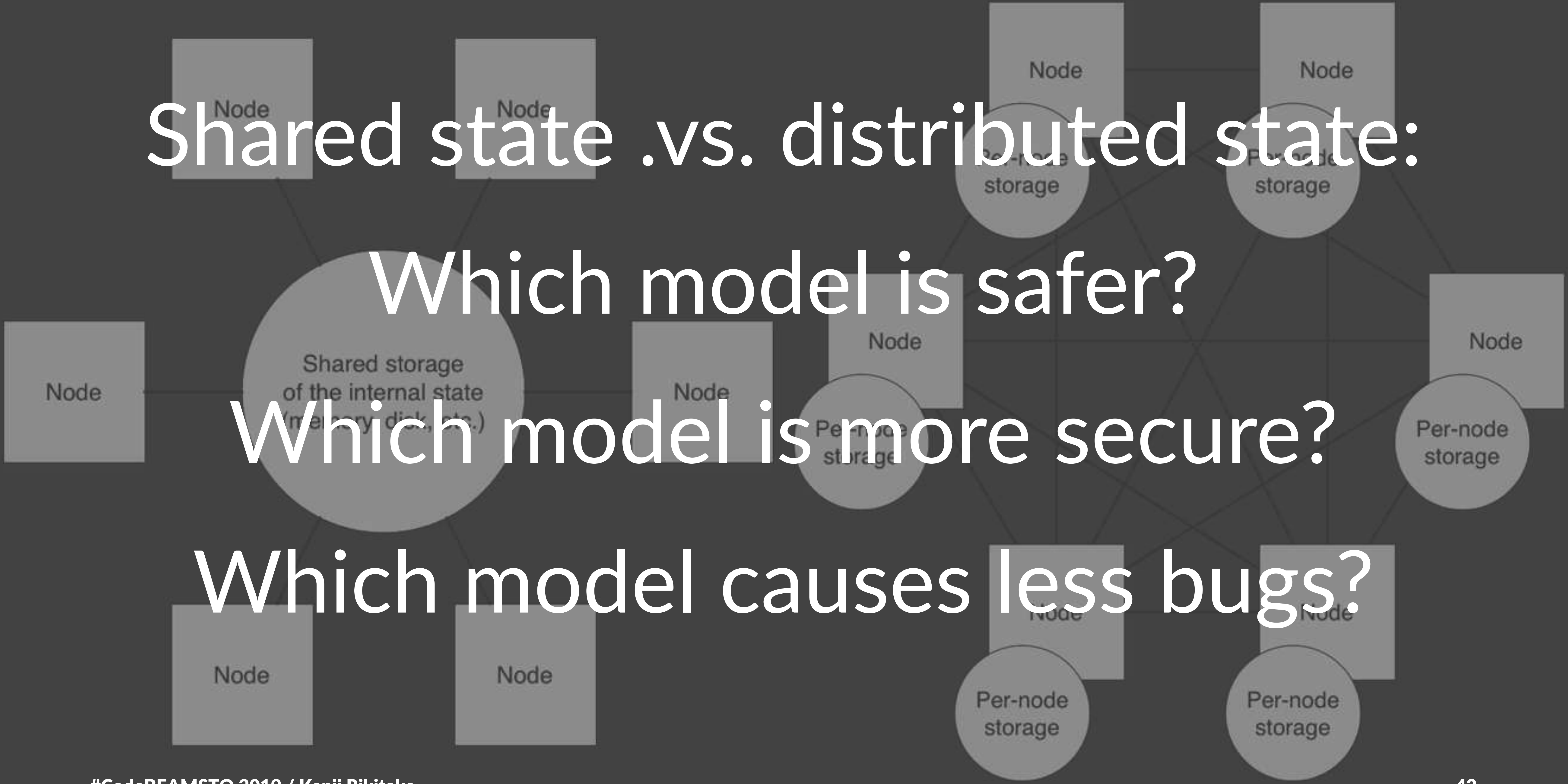
from the popularly-used shared-state object-oriented languages:

Choice of default data copying mode

By choosing *lagom* speed traded in for much more secure programming


Node
Node
Node
Shared storage
of the internal state
(memory, disk, etc.)
Node
Node
Node


Node
Node
Per-node
storage
Per-node
storage
Node
Node
Per-node
storage
Per-node
storage
Node
Node
Per-node
storage
Per-node
storage



Shared state .vs. distributed state:

Which model is safer?

Which model is more secure?

Which model causes less bugs?

Topics excluded from this talk

- BEAM architecture [11]
- Concurrency models
- Process supervision and signals
- How BEAM languages handle shared states

---

[11] Erik Stemman, The Beam Book

Acknowledgment

This presentation is suppored by
Pepabo R&D Institute, GMO Pepabo, Inc.

Thanks to Code BEAM Crew and Erlang Solutions!

... and thank you for being here!

ペパボ研究所
Pepabo R&D Institute, GMO Pepabo, Inc.

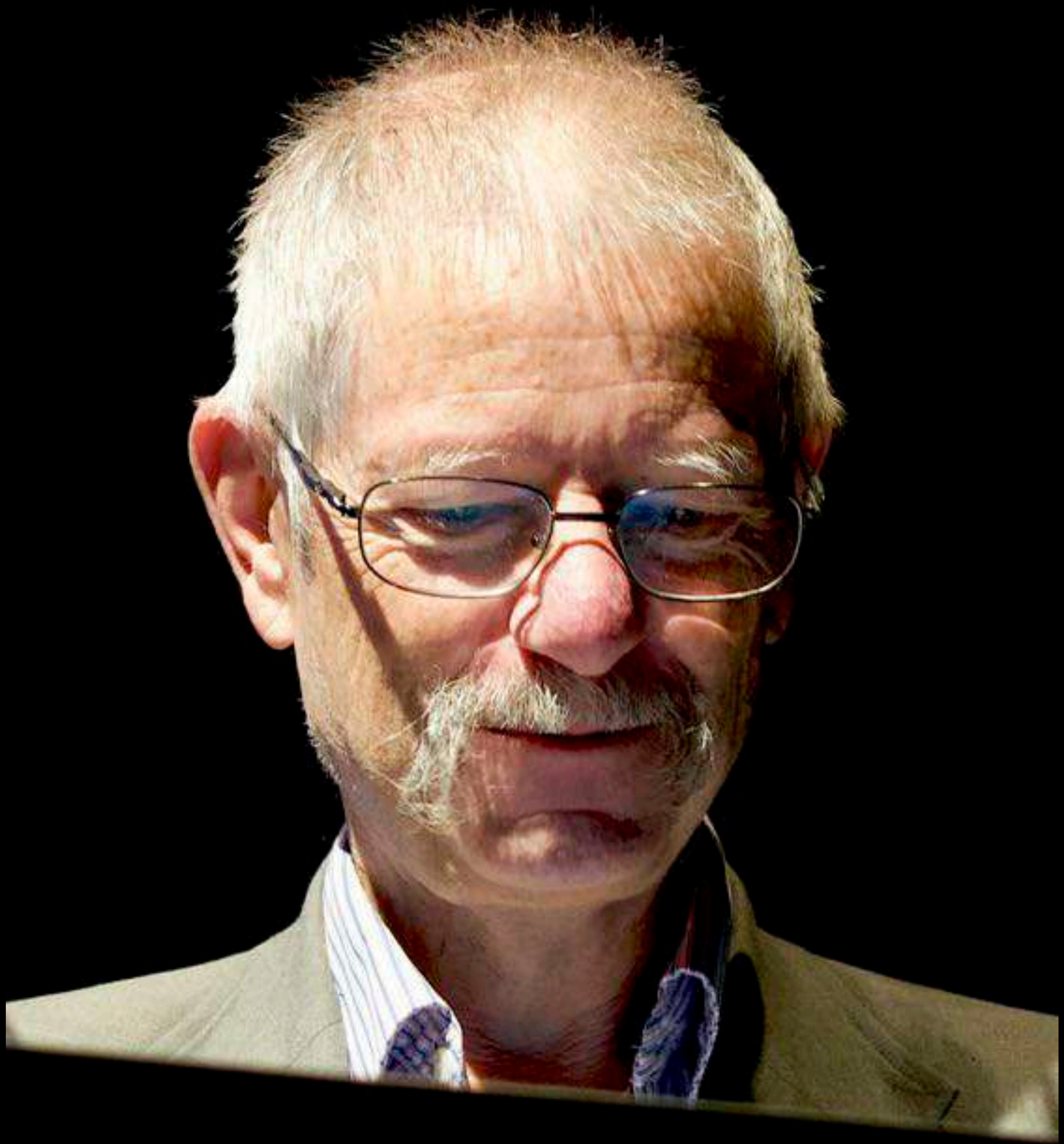
Thanks, Joe.

You taught me how to program in the principle of *lagom är bäst*.

You helped me finding out a new hope for programming, after I got lost in the C header files of ISC BIND 9.4.2 in 2007.

I'm impressed by your hospitality, as well as your creative mind.

We will remember you.



Thank you
Questions?

Photo / graphics credits

- Title: Photo by Masayoshi Yamase on Unsplash
- Lagom: Photo by Jen P. on Unsplash
- Programming Erlang quote: from Pragmatic Bookshelf's EPUB ebook rendered by iBooks on macOS 10.14.4, underline added by Kenji Rikitake
- Joe Armstrong: Photo by Brian L. Troutwine, edited by Kenji Rikitake, licensed CC BY-NC 4.0 International
- Pepabo R&D Institute Logo: GMO Pepabo, Inc.
- Thank you page: Photo by Raphael Andres on Unsplash
- All the other photos: Kenji Rikitake