Objects, Anomalies, and Actors: The Next Revolution

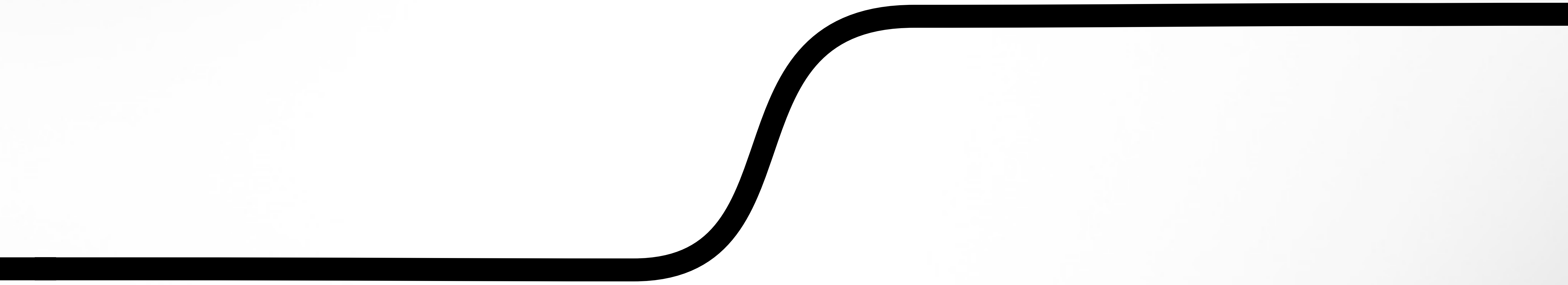
Steve Vinoski
Architect, Basho Technologies
QCon San Francisco 2011
18 Nov 2011
@stevevinoski
http://steve.vinoski.net/
vinoski@ieee.org

- This is actually Kresten Krab Thorup's talk, but he couldn't attend the conference
  - he's CTO of Trifork
  - and an important part of the team behind the QCon and GOTO conferences

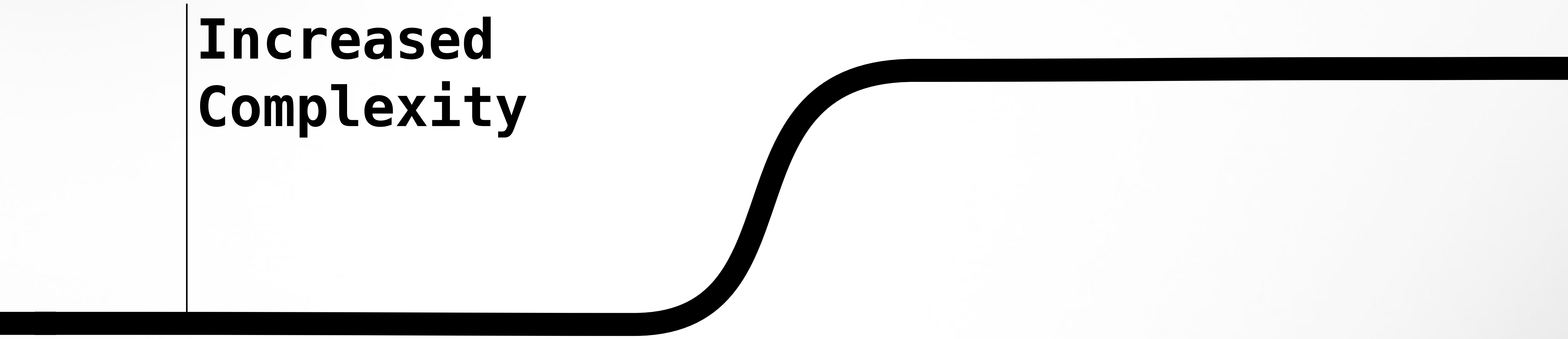
- I'm covering for him here, giving my interpretation of his material
- These are (mostly) his slides, I've changed a few and inserted some of my own

basho

'90s Object Revolution

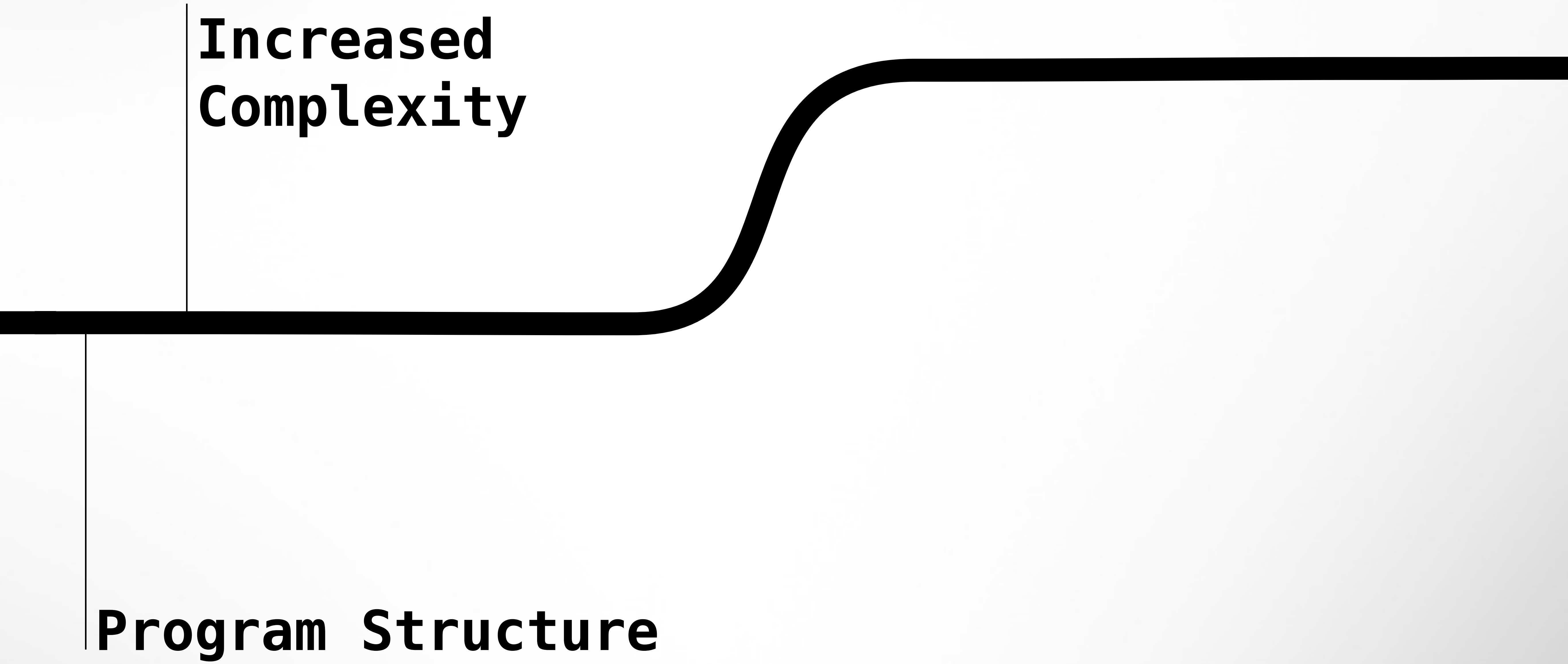


’90s Object Revolution

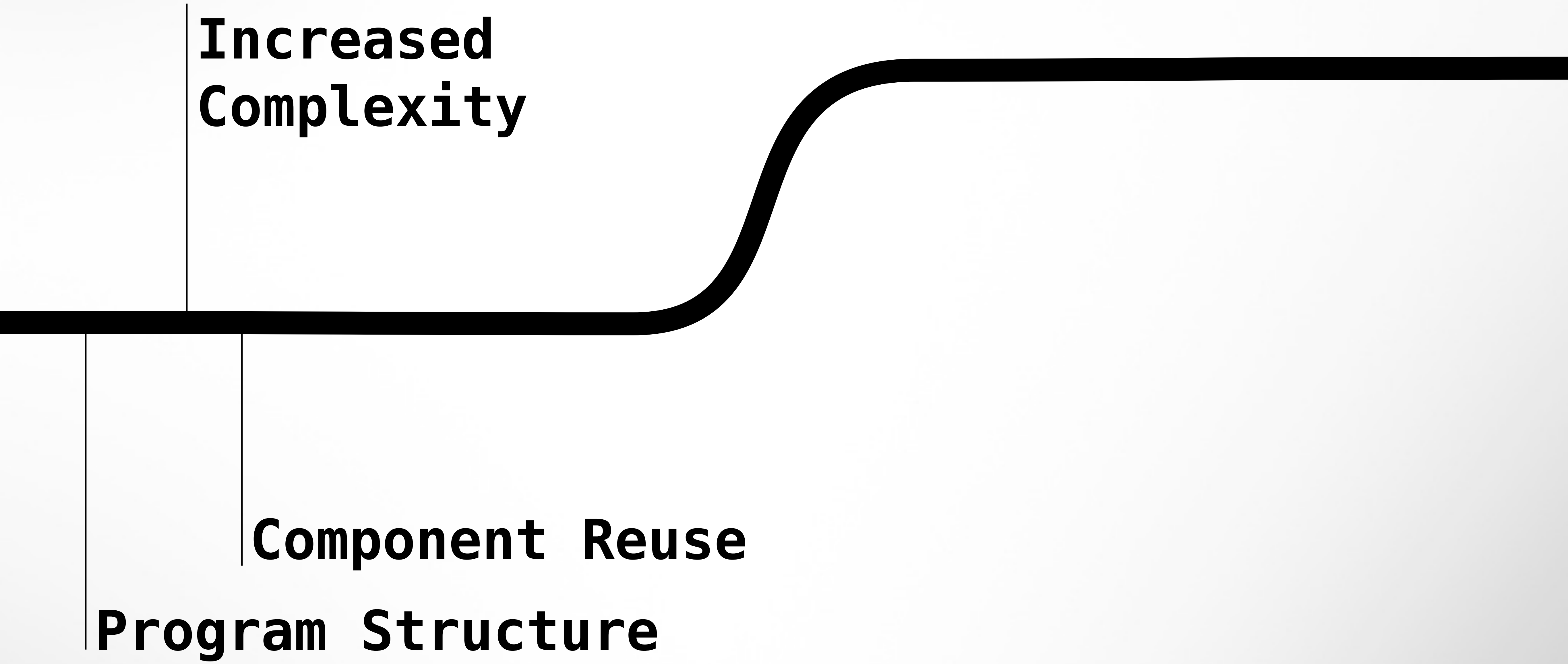


TRIFORK.

'90s Object Revolution

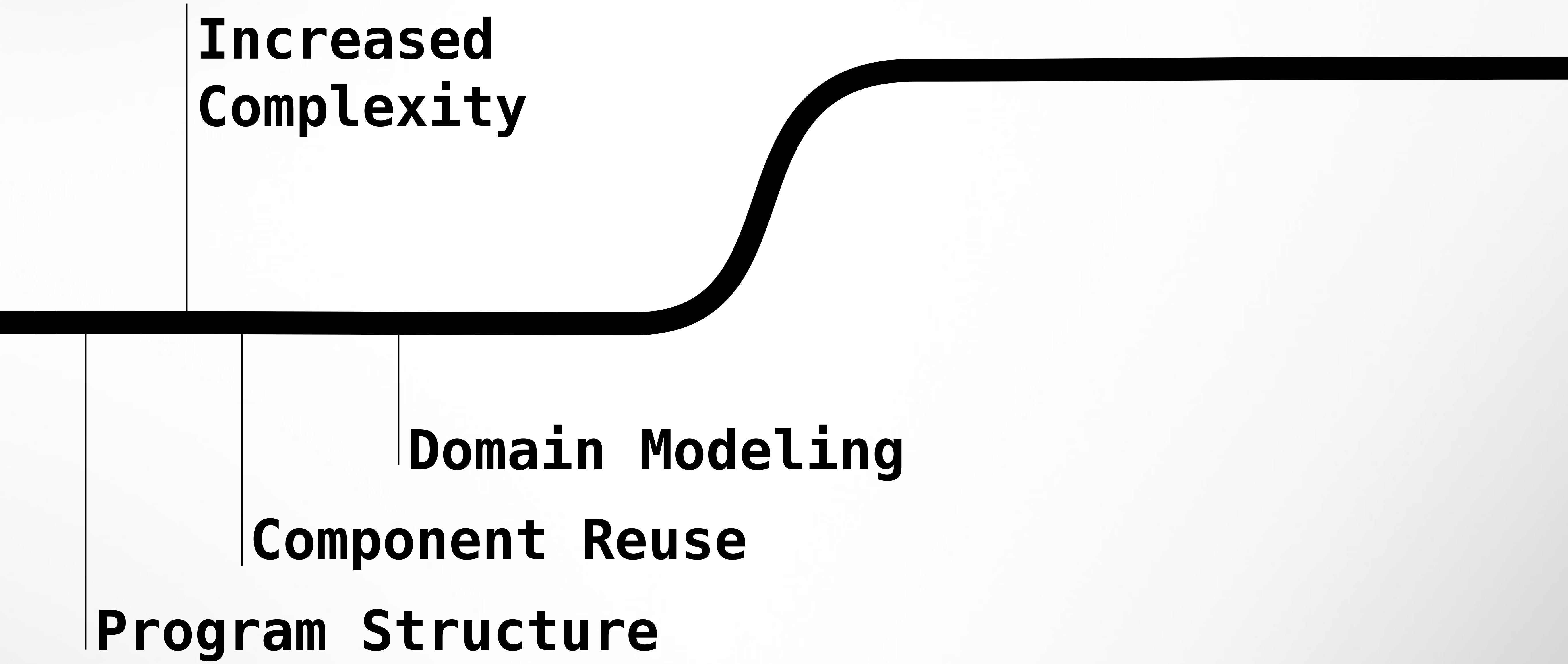


'90s Object Revolution



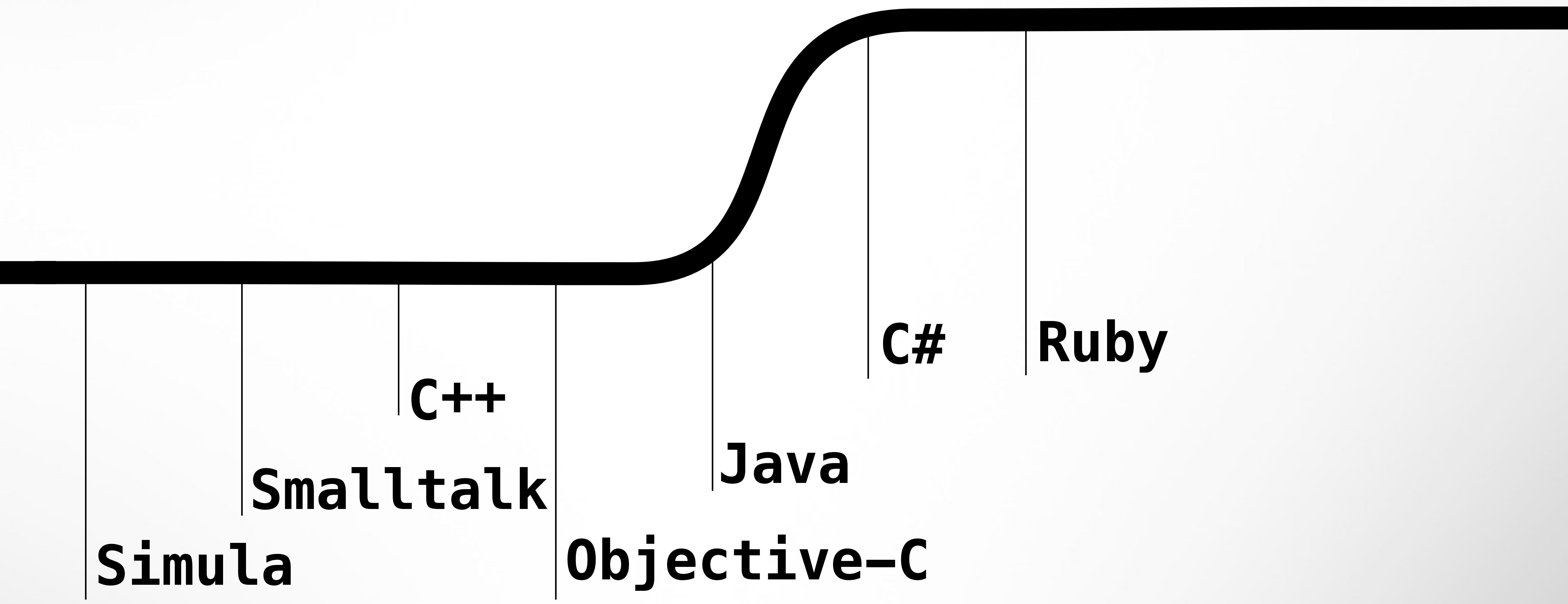
TRIFORK.

'90s Object Revolution



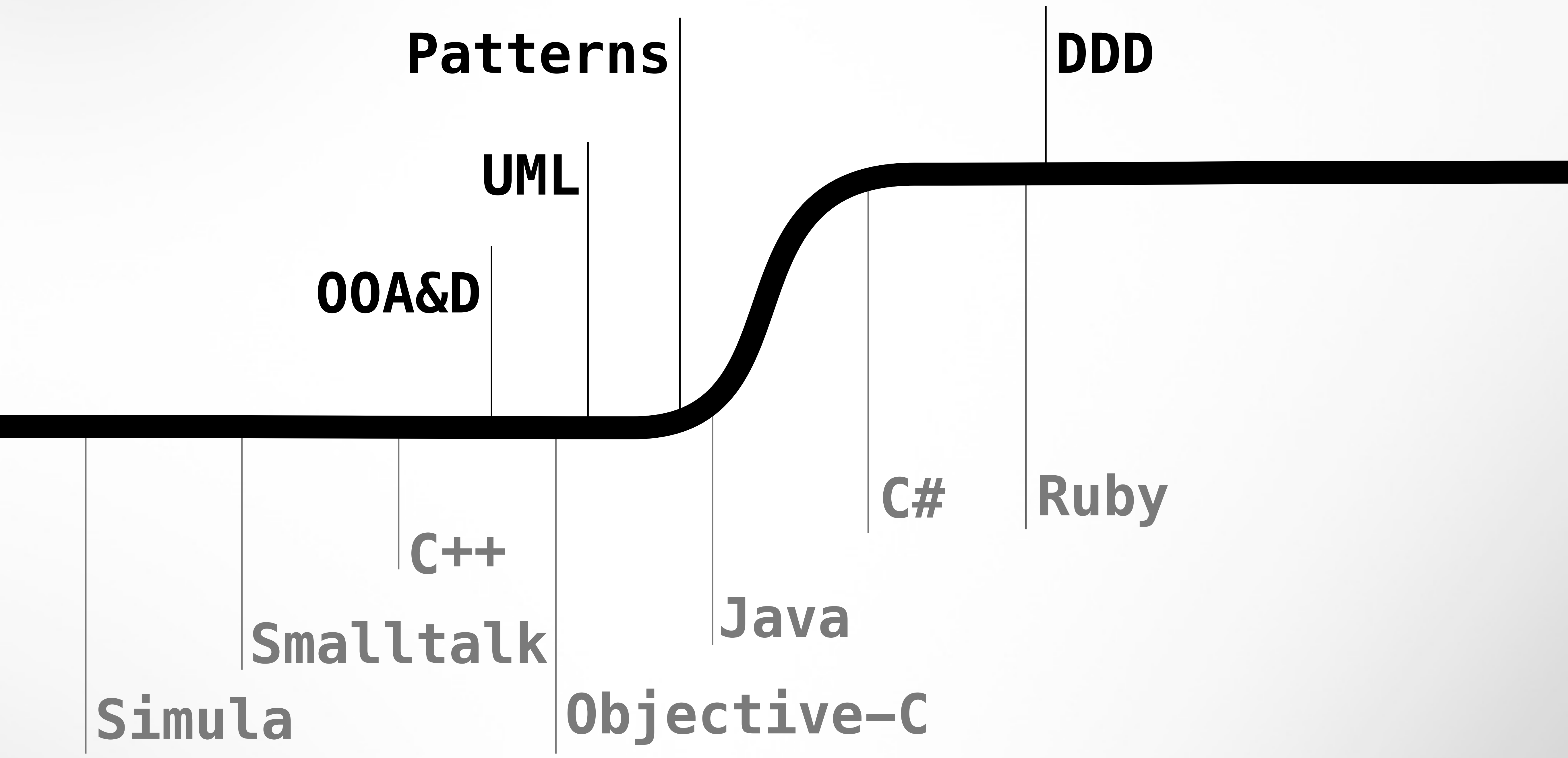
TRIFORK.

Languages

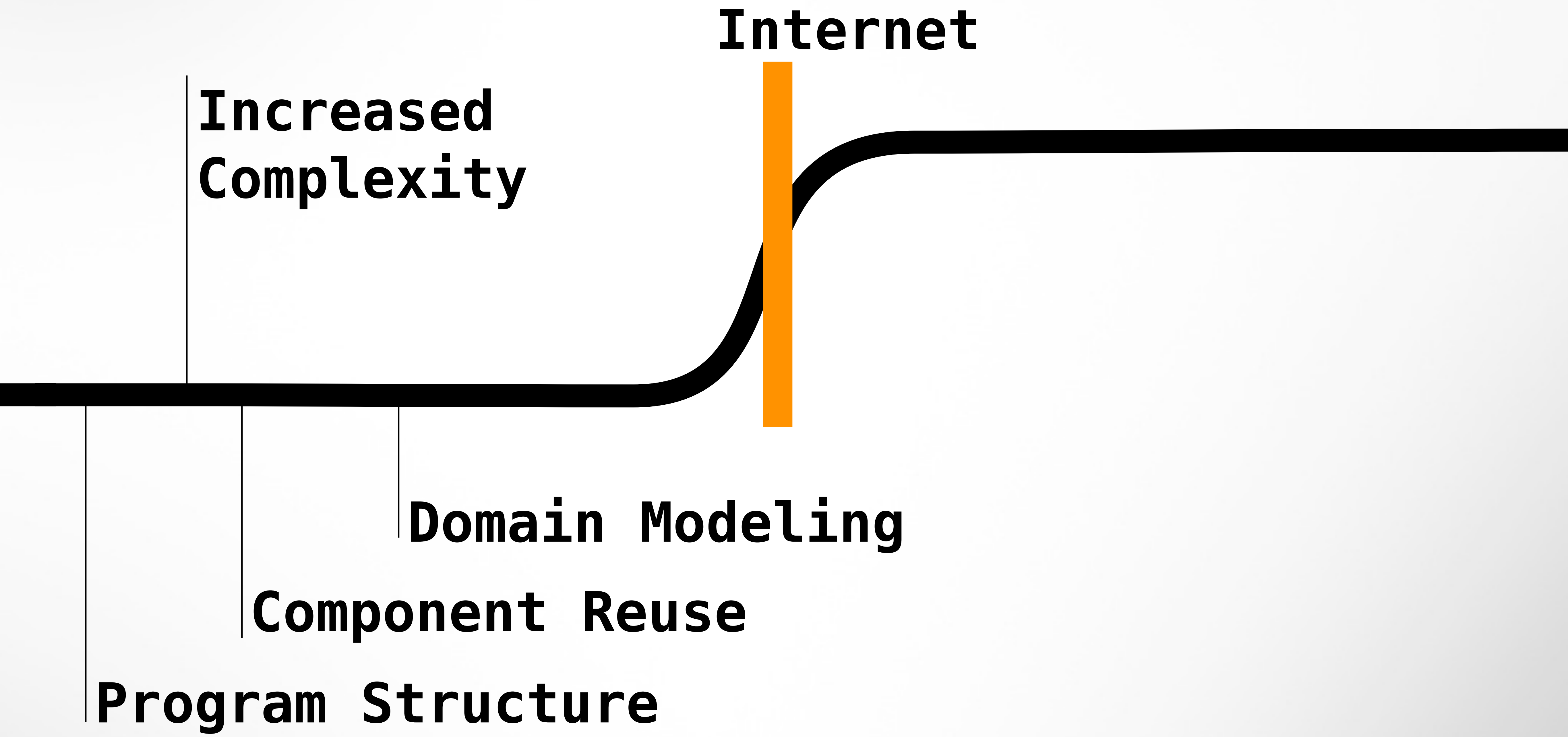


TRIFORK.

Thinking Tools



TRIFORK.


Internet
Increased Complexity
Domain Modeling
Component Reuse
Program Structure

More Complexity

Infrastructure made of Software

More Complexity

Infrastructure made of Software

More Complexity

Infrastructure made of Software

Fault Tolerance, Availability, QoS

More Complexity

Infrastructure made of Software

Integration, Coordination

Fault Tolerance, Availability, QoS

More Complexity

Infrastructure made of Software

Cloud, Multi-Core

Integration, Coordination

Fault Tolerance, Availability, QoS

More Complexity

Infrastructure m

We're struggling to handle these with an object mindset!

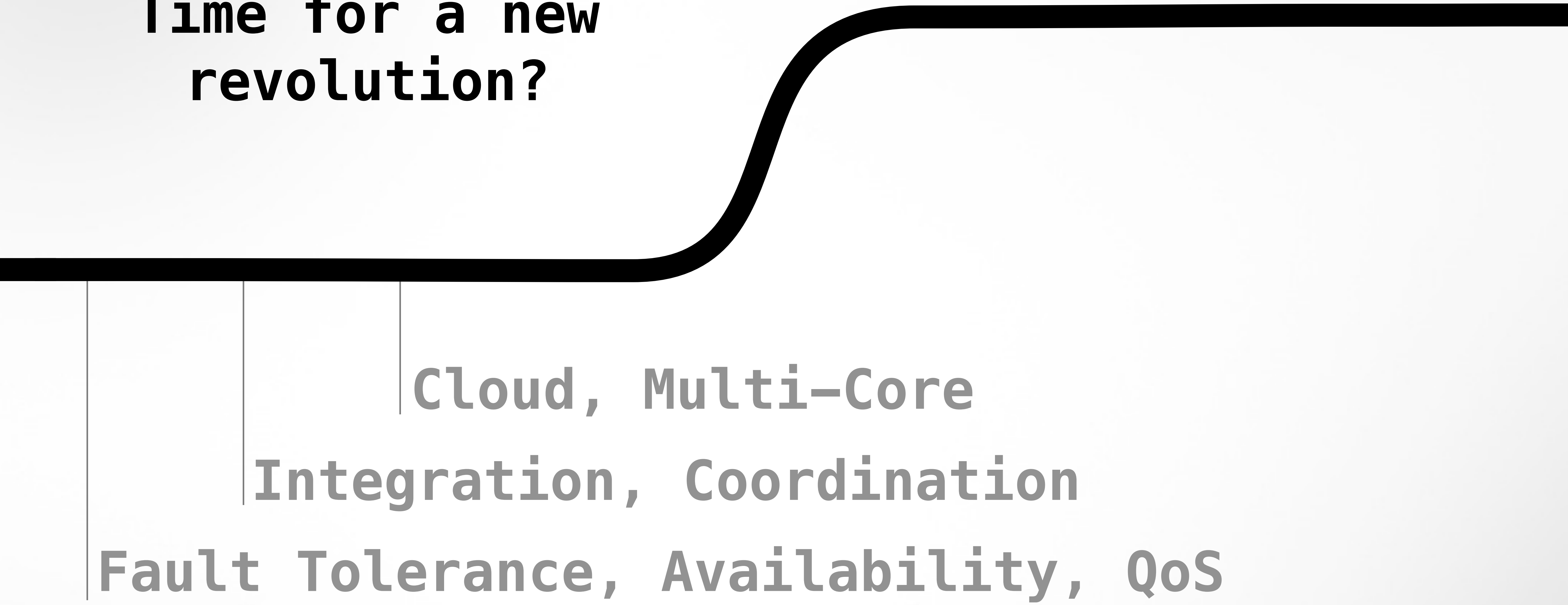
Cloud, Multi-Core

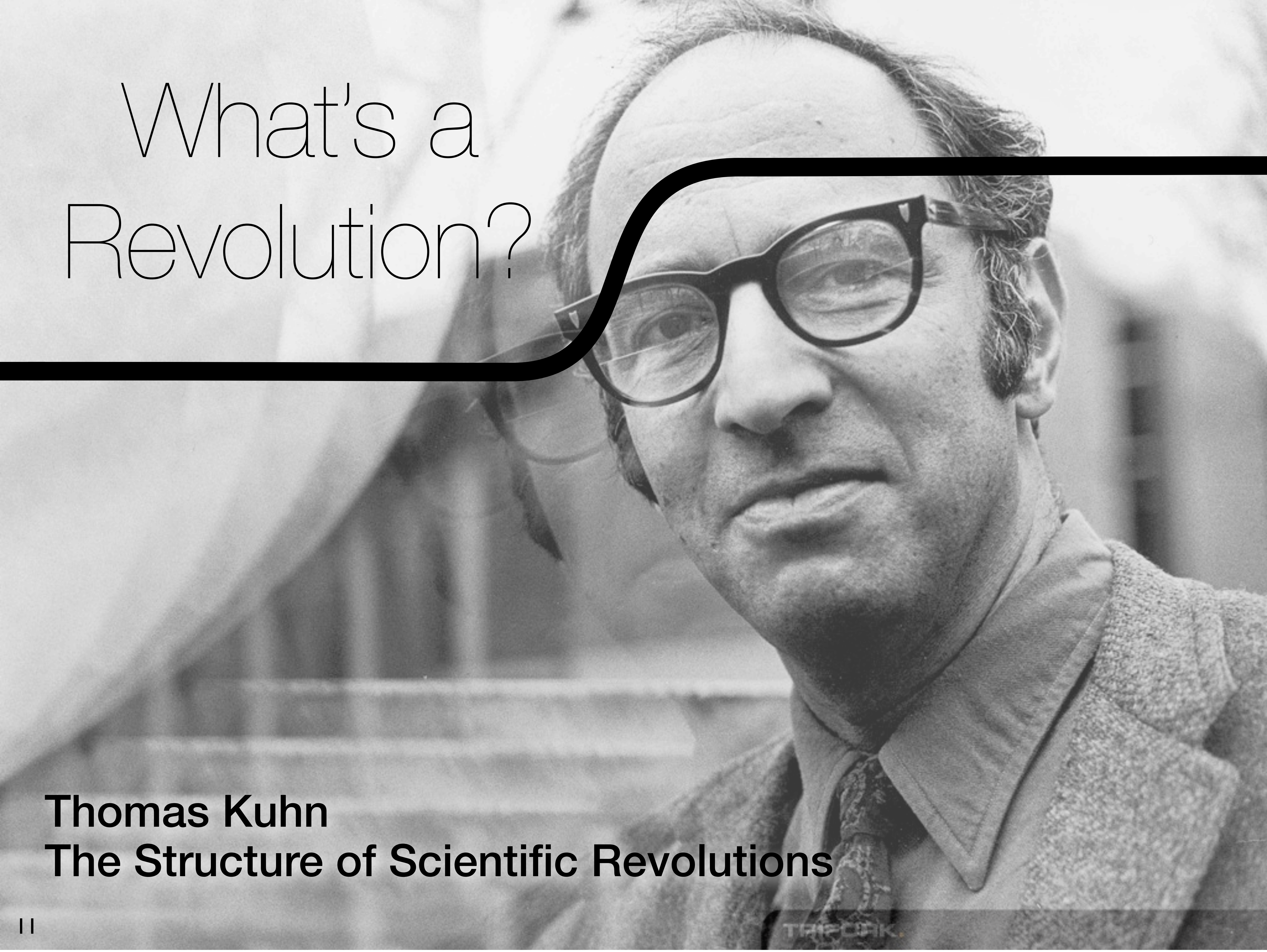
Integration, Coordination

Fault Tolerance, Availability, QoS

TRIFORK.

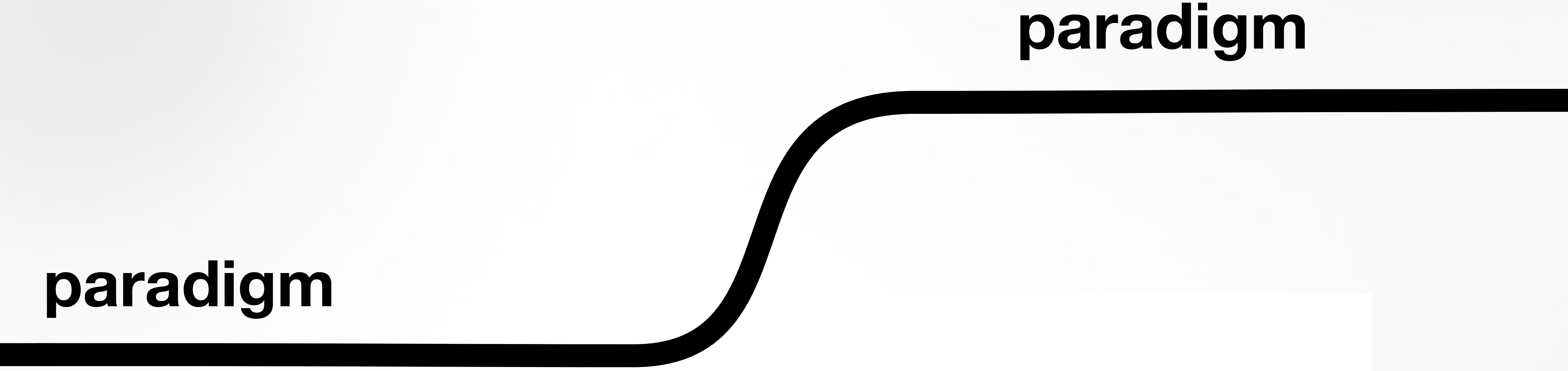
Time for a new revolution?

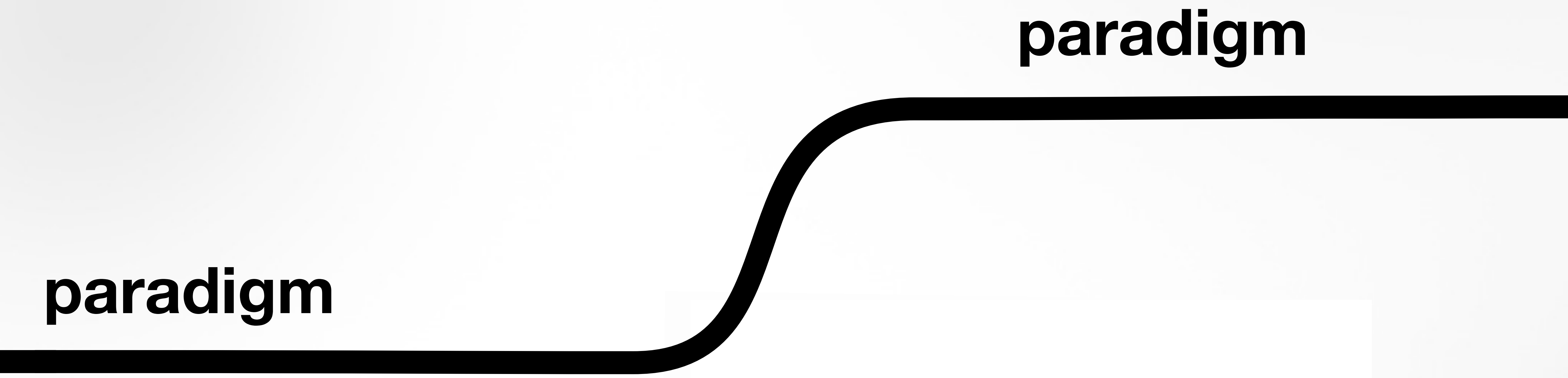


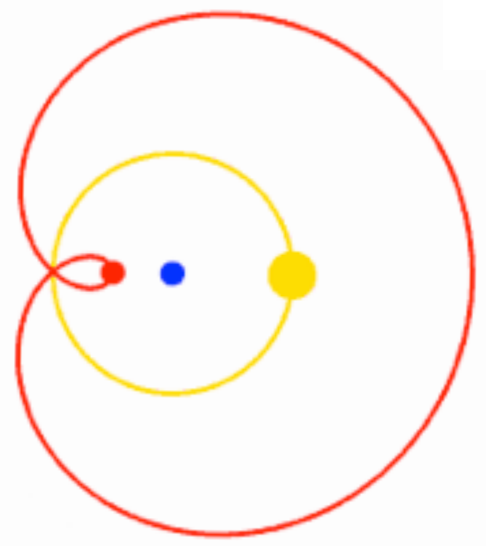


What's a Revolution?

Thomas Kuhn
The Structure of Scientific Revolutions


paradigm
paradigm

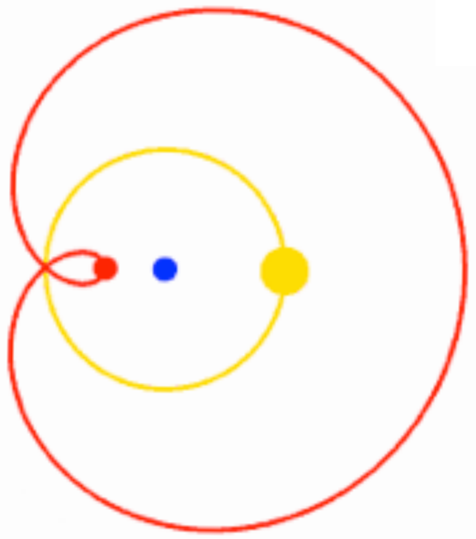

paradigm
paradigm



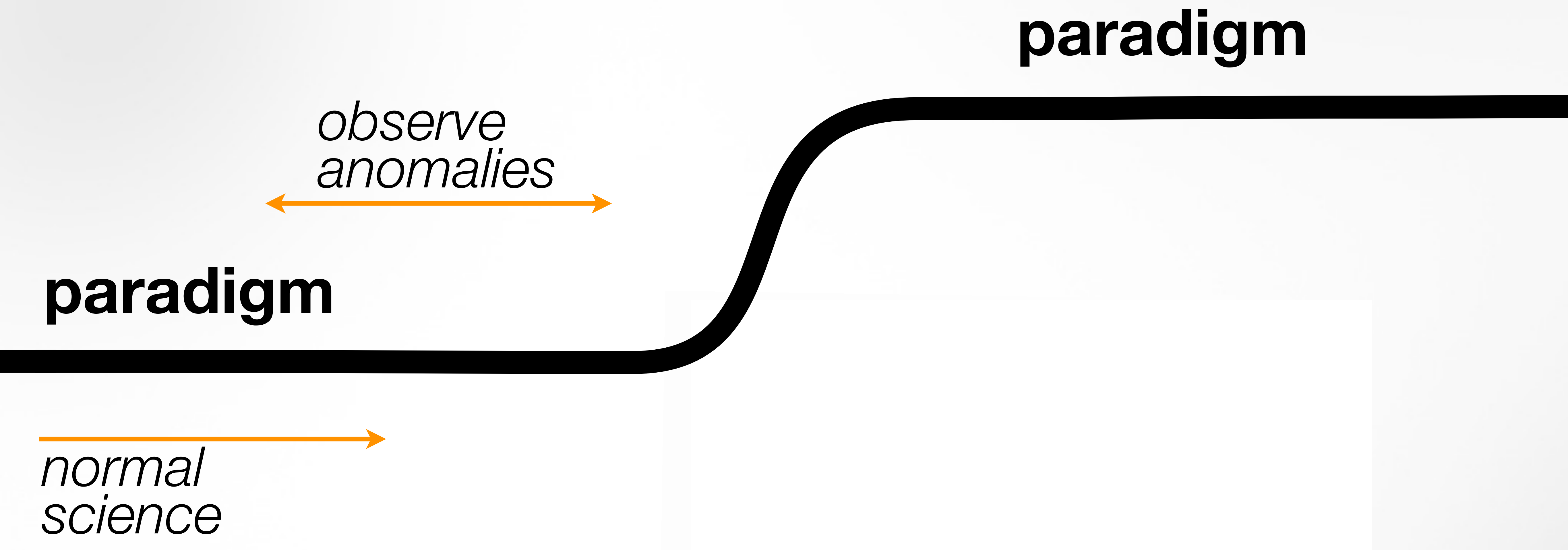
paradigm

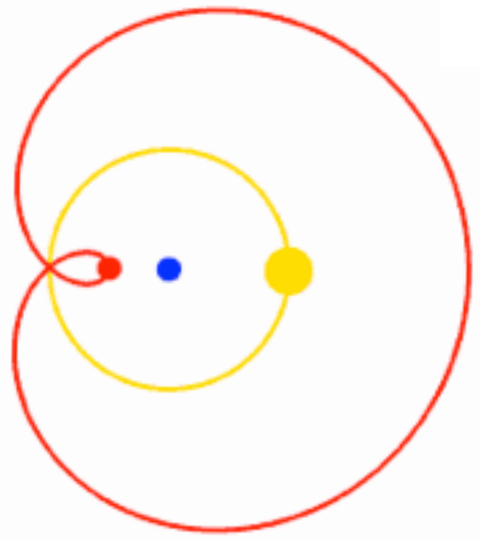
paradigm

normal science

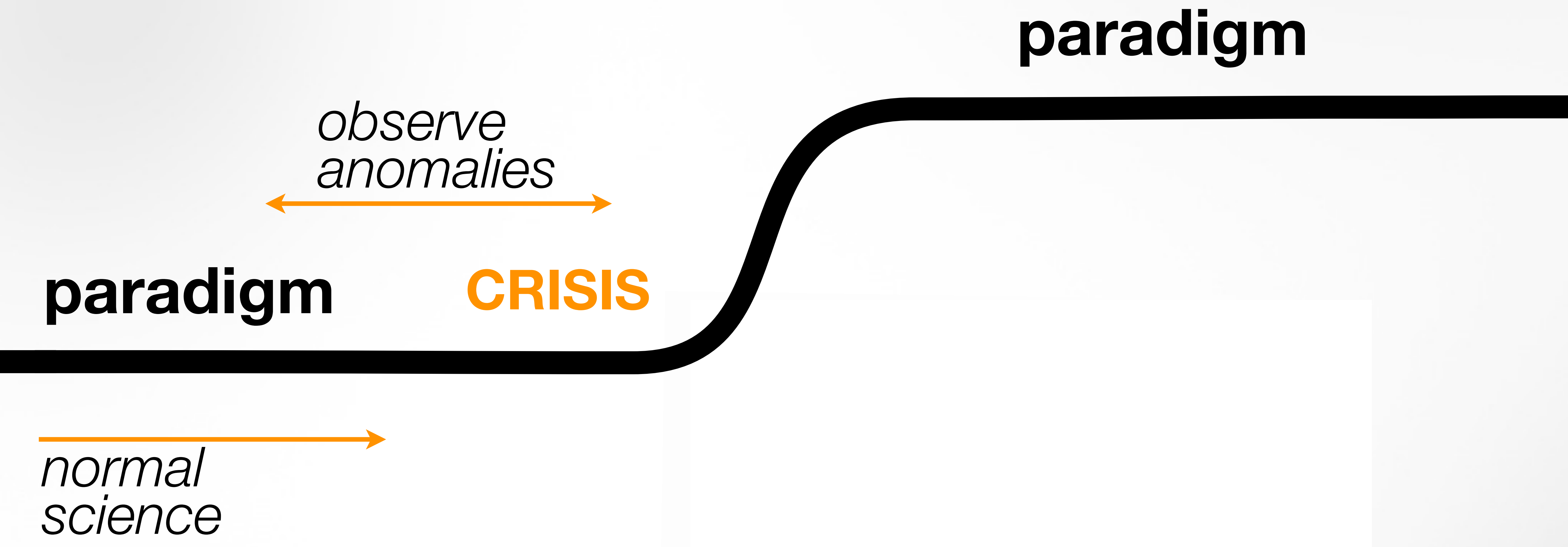


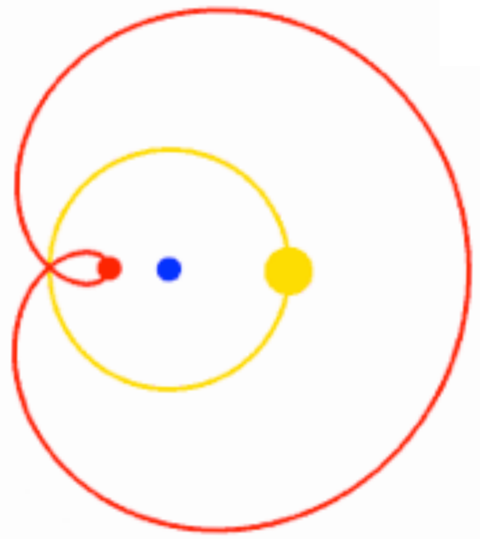
TRIFORK.

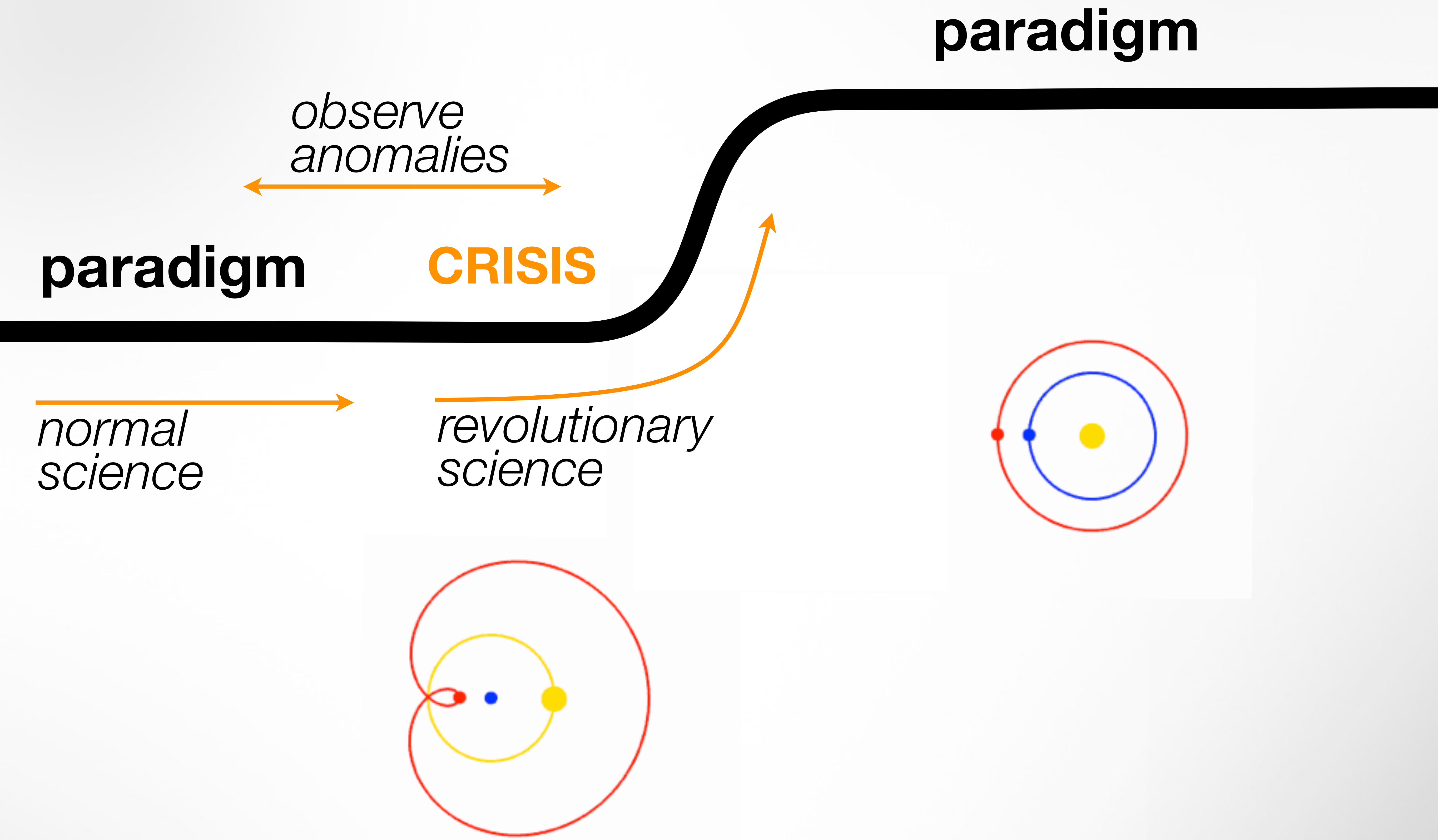


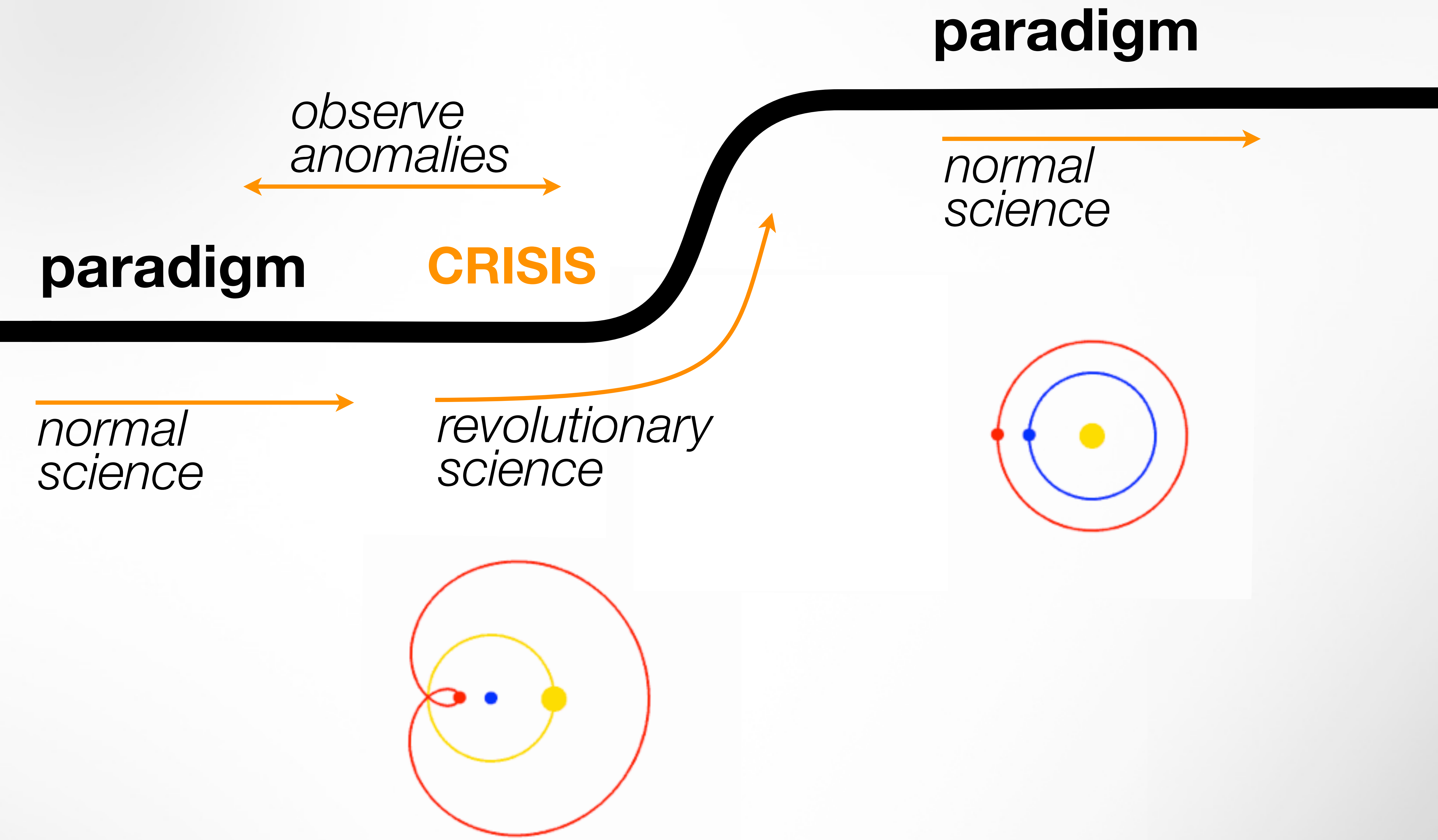


TRIFORK.


paradigm
observe anomalies
paradigm
CRISIS
normal science




paradigm
observe anomalies
paradigm
CRISIS
normal science
revolutionary science


paradigm
observe anomalies
paradigm
CRISIS
normal science
revolutionary science
normal science

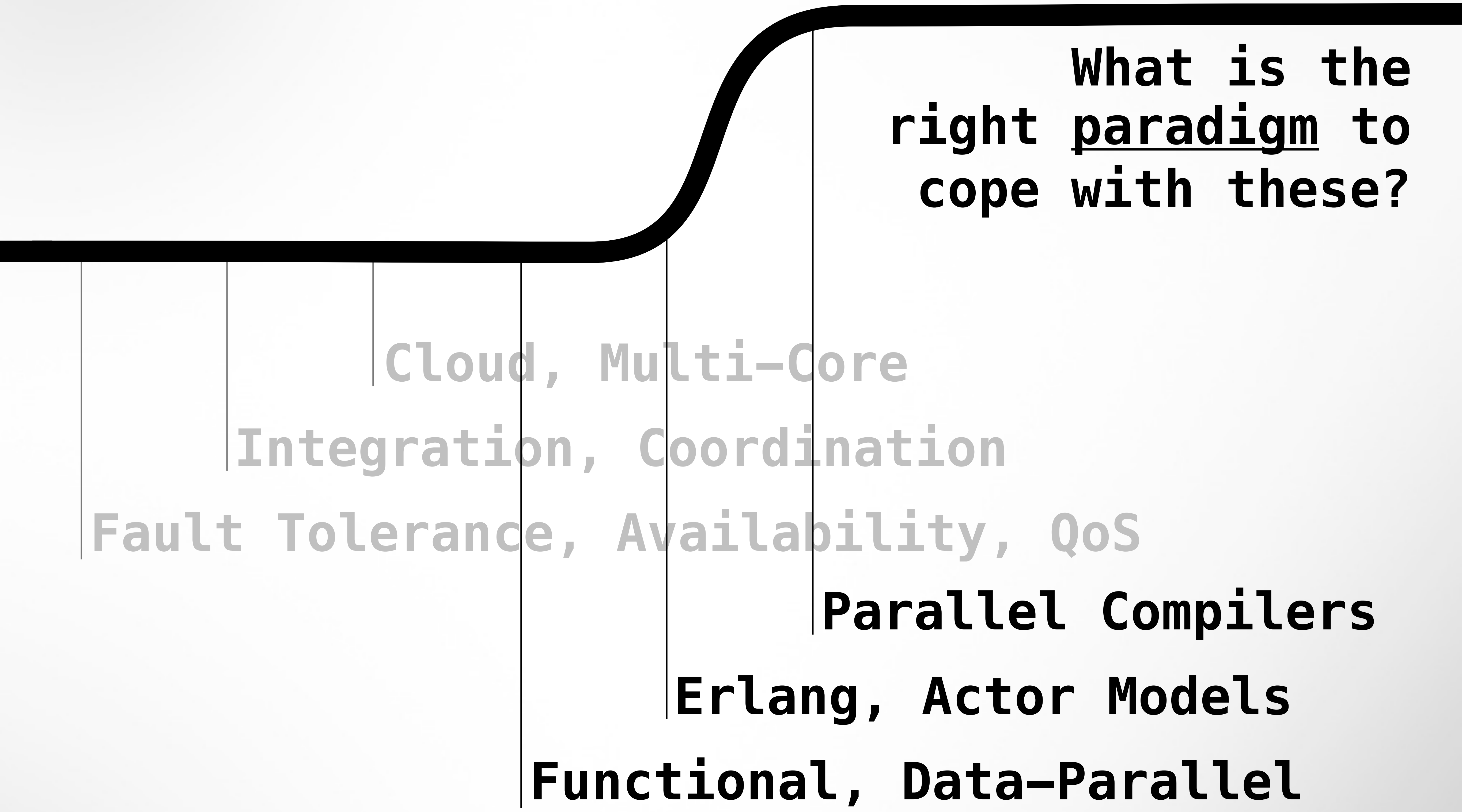
What is the right <u>paradigm</u> to cope with these?

Cloud, Multi-Core

Integration, Coordination

Fault Tolerance, Availability, QoS

What is the right paradigm to cope with these?



Cloud, Multi-Core

Integration, Coordination

Fault Tolerance, Availability, QoS

Parallel Compilers

Erlang, Actor Models

Functional, Data-Parallel

Ralph Johnson's blog
"Erlang, the Next Java"

... Erlang is going to be a very important language ... Its main advantage is that it is perfectly suited for the multi-core, web services future. In fact, it is the ONLY mature, rock-solid language that is suitable for writing highly scalable systems to run on multicore machines.

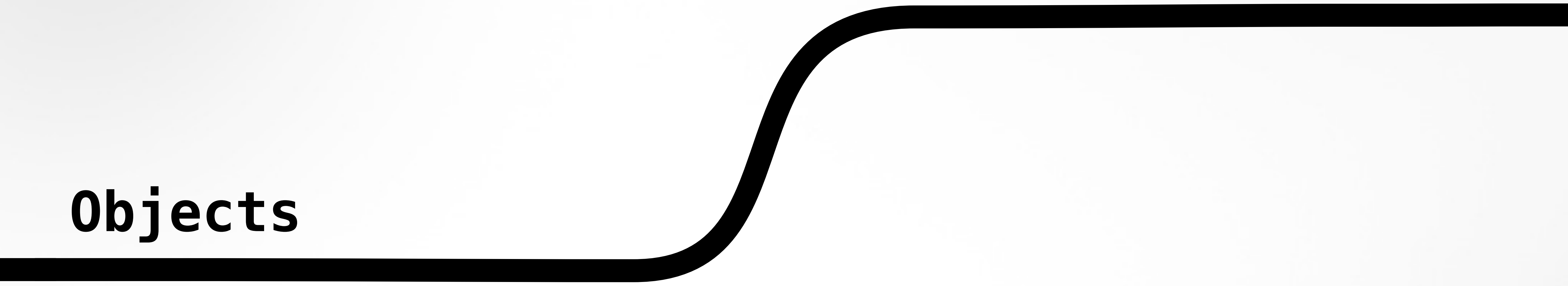
is the
igm to
these?

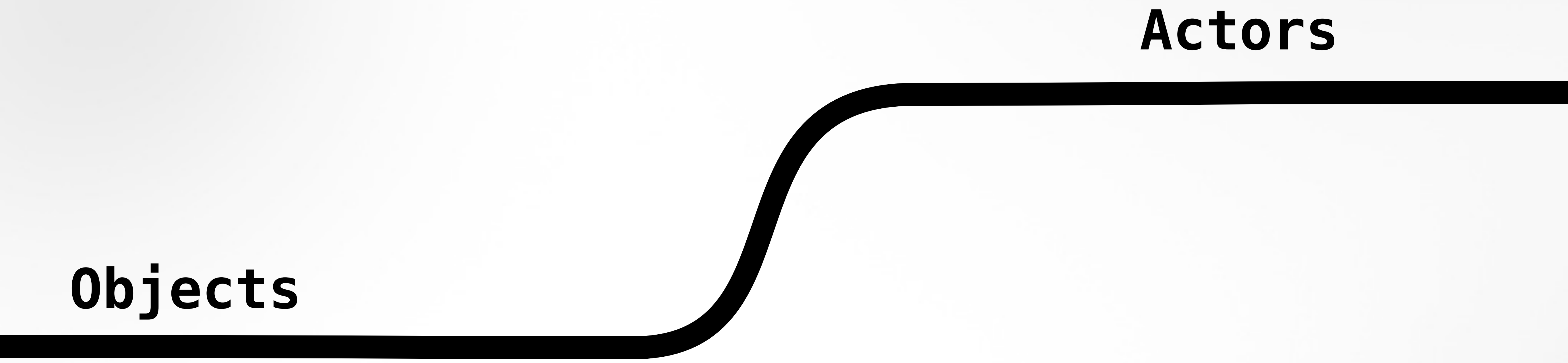
Ir

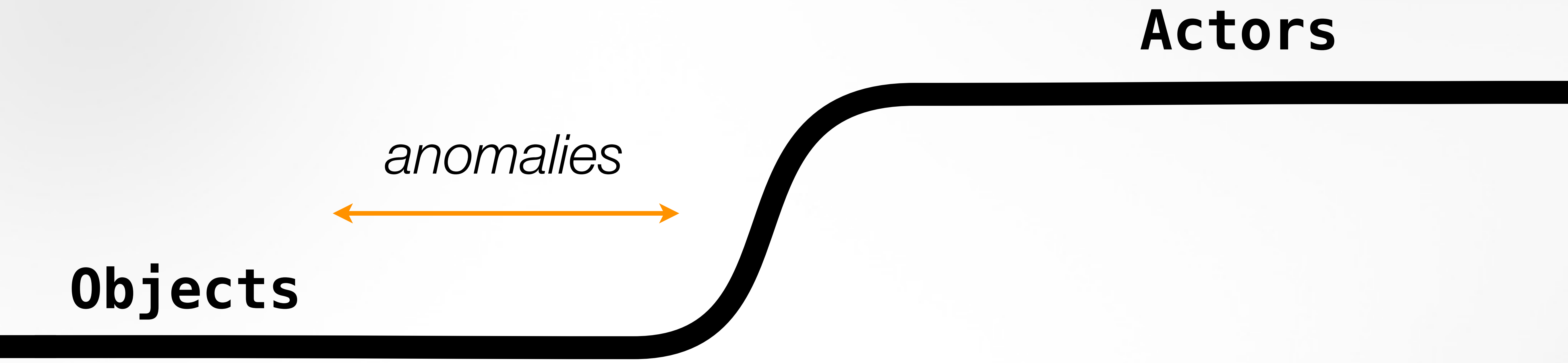
Fault T

ers

Data-Parallel


Objects
Actors

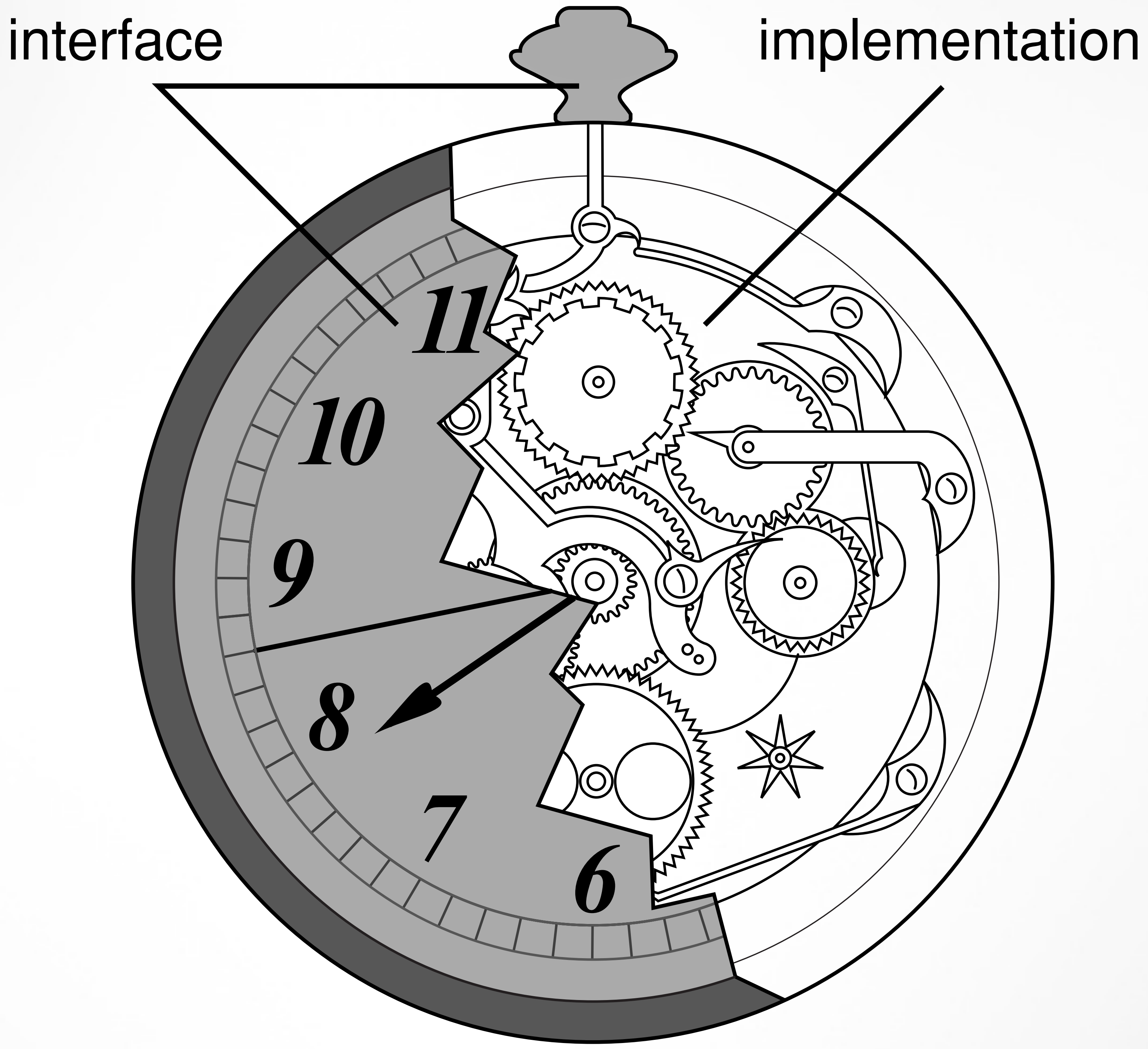

Actors
Objects


Actors
anomalies
Objects

Obj

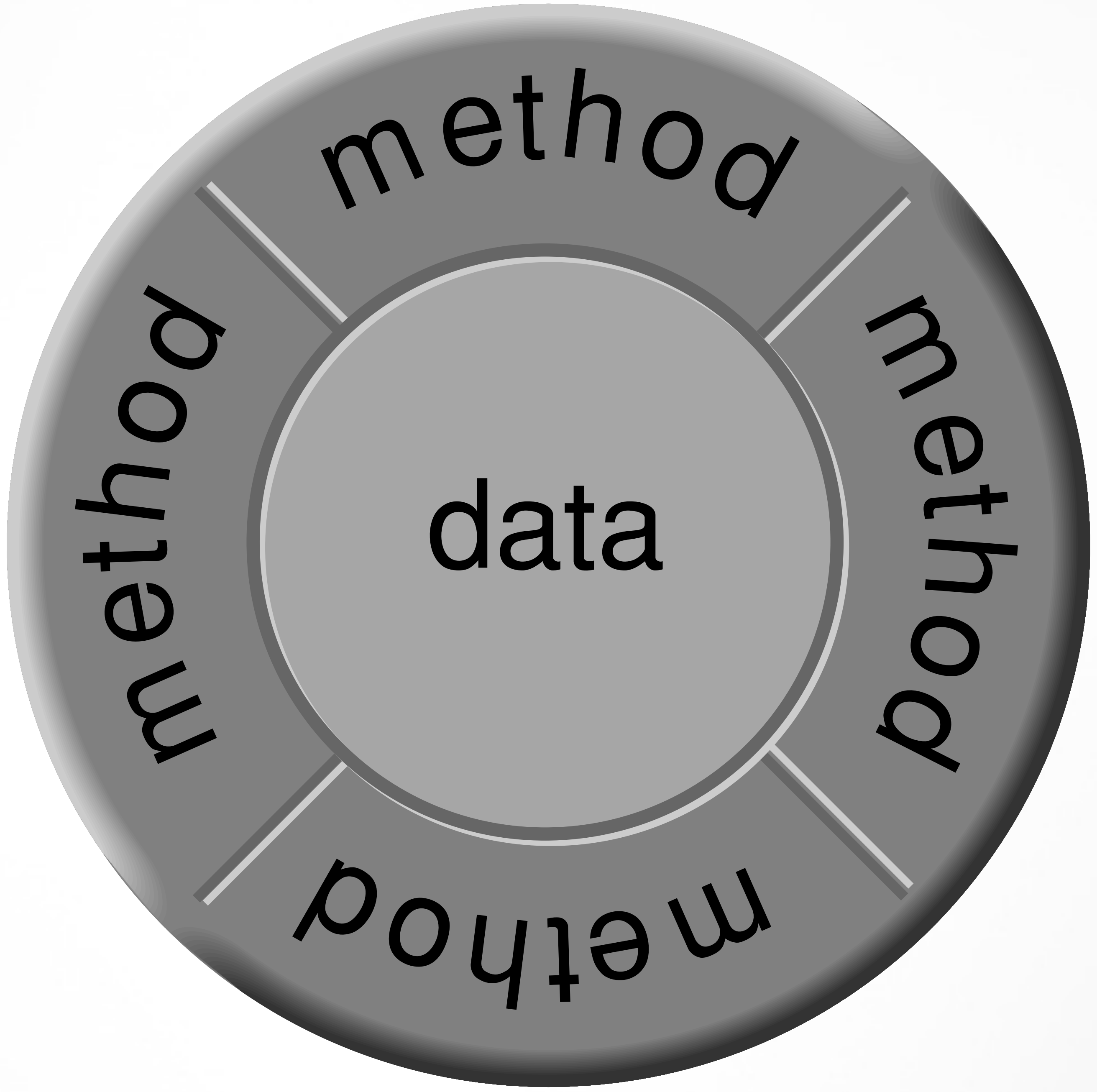
If Actor-Programming is the new Paradigm, What are the anomalies we should see now?

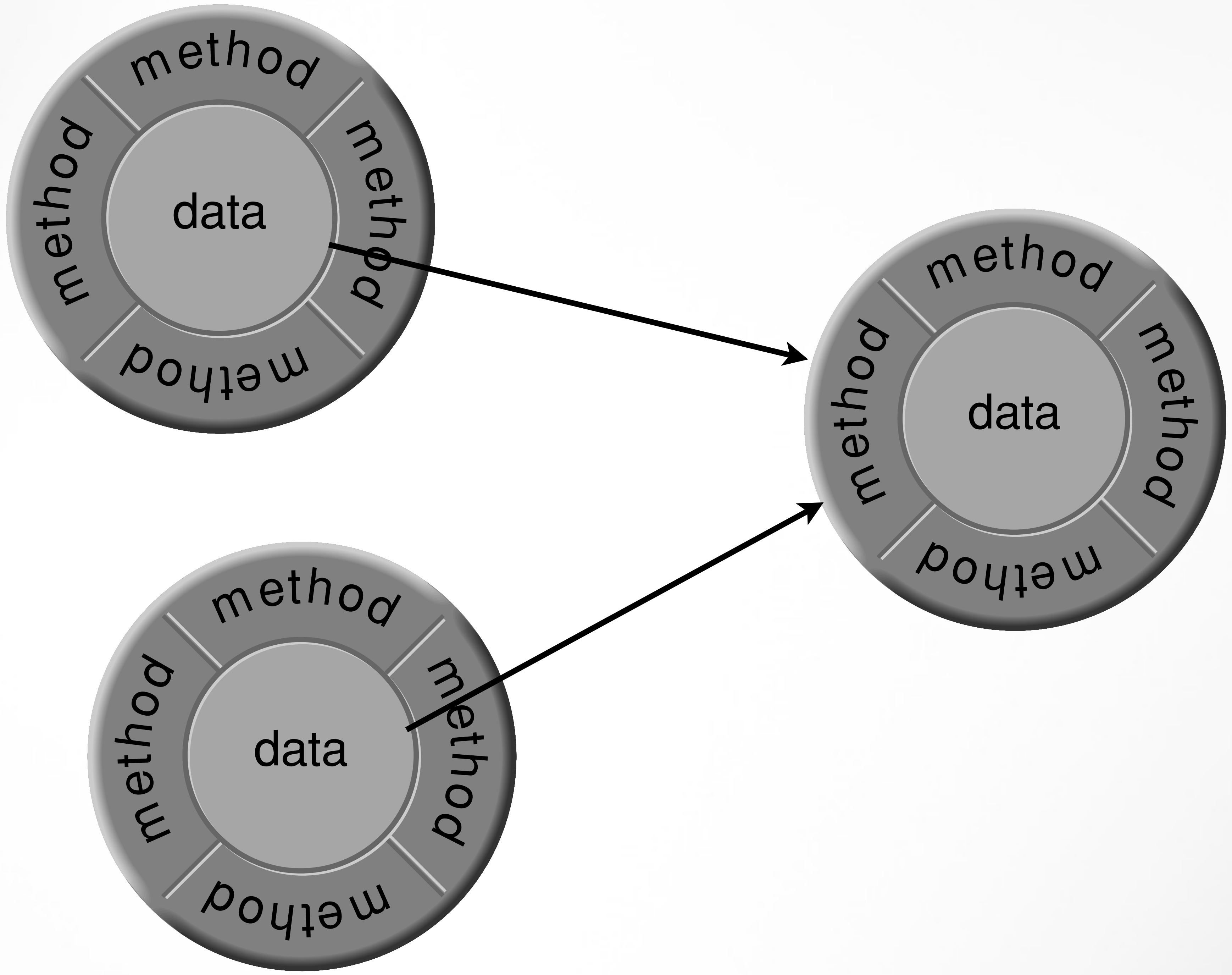
Anomalies in the object-oriented world view

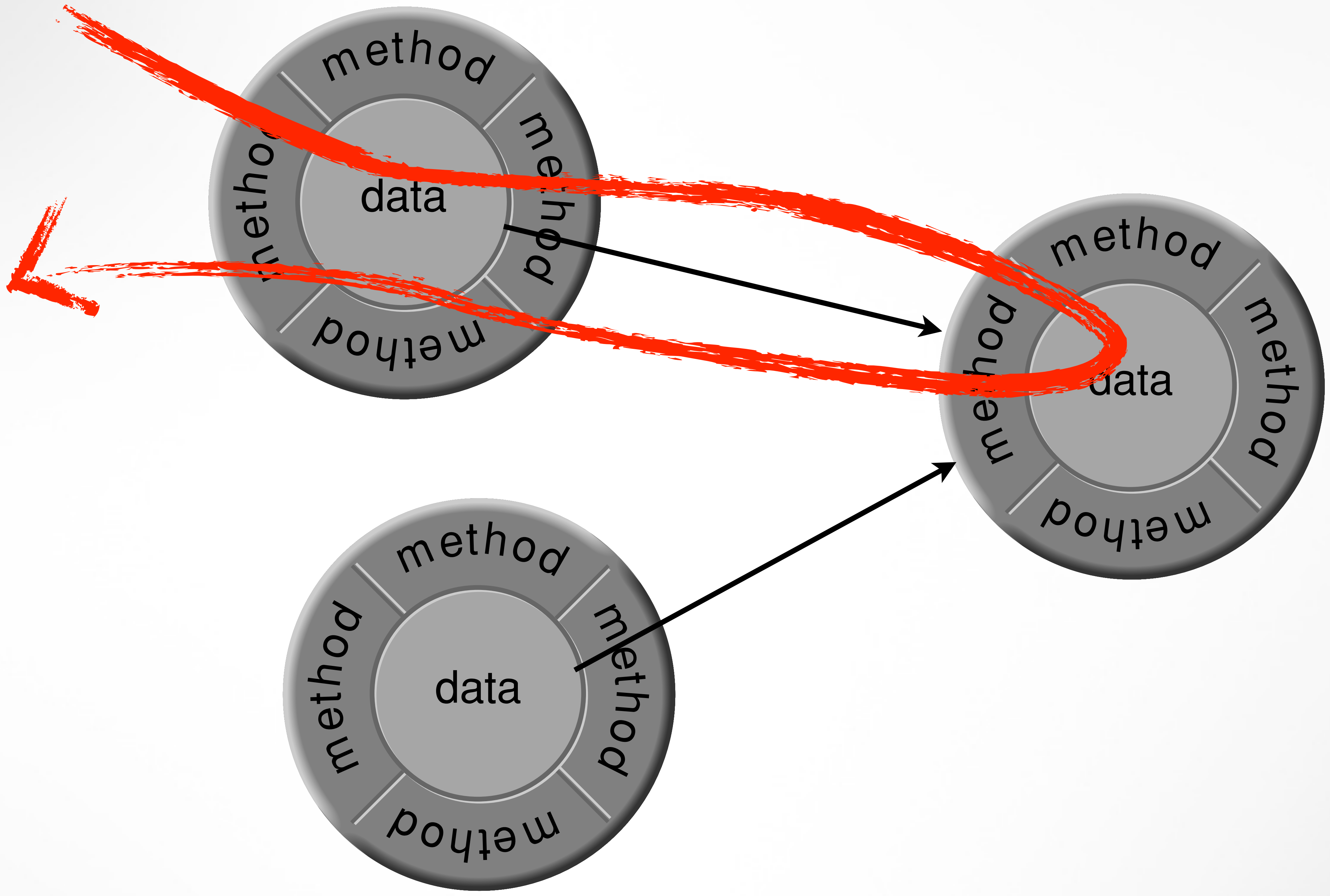


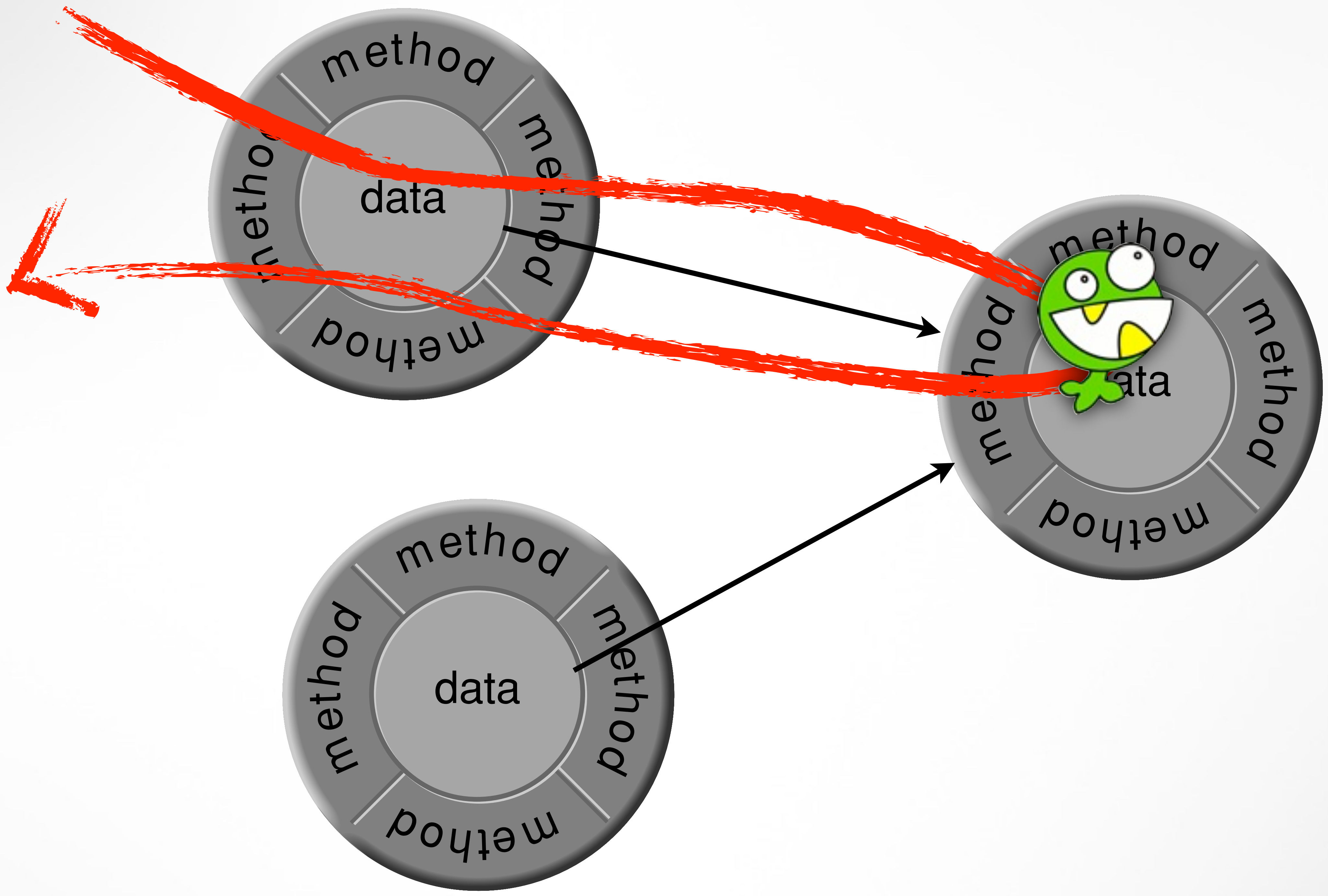
Graphics from *Object-Oriented Programming with Objective C*, Apple, 2011

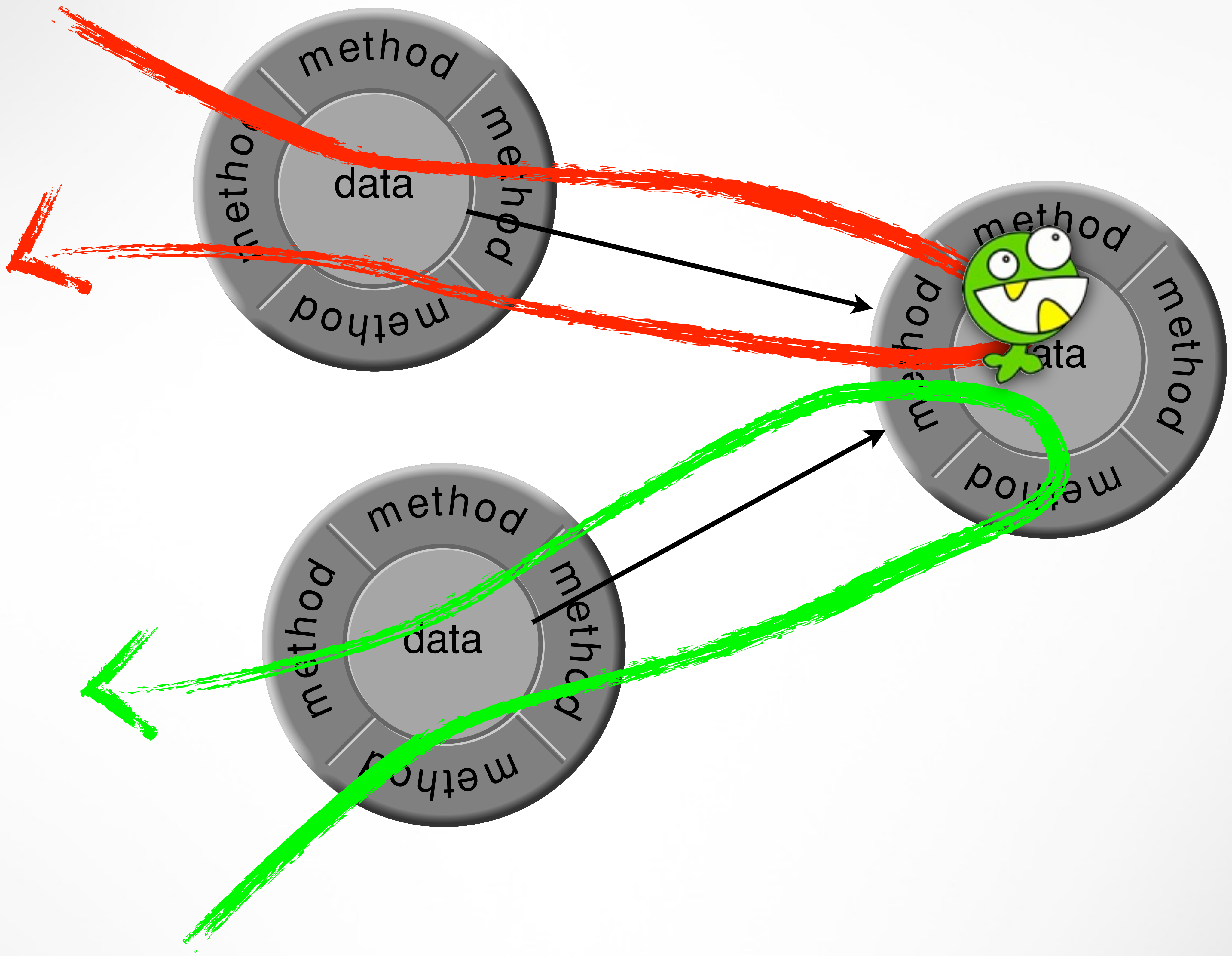
Encapsulation

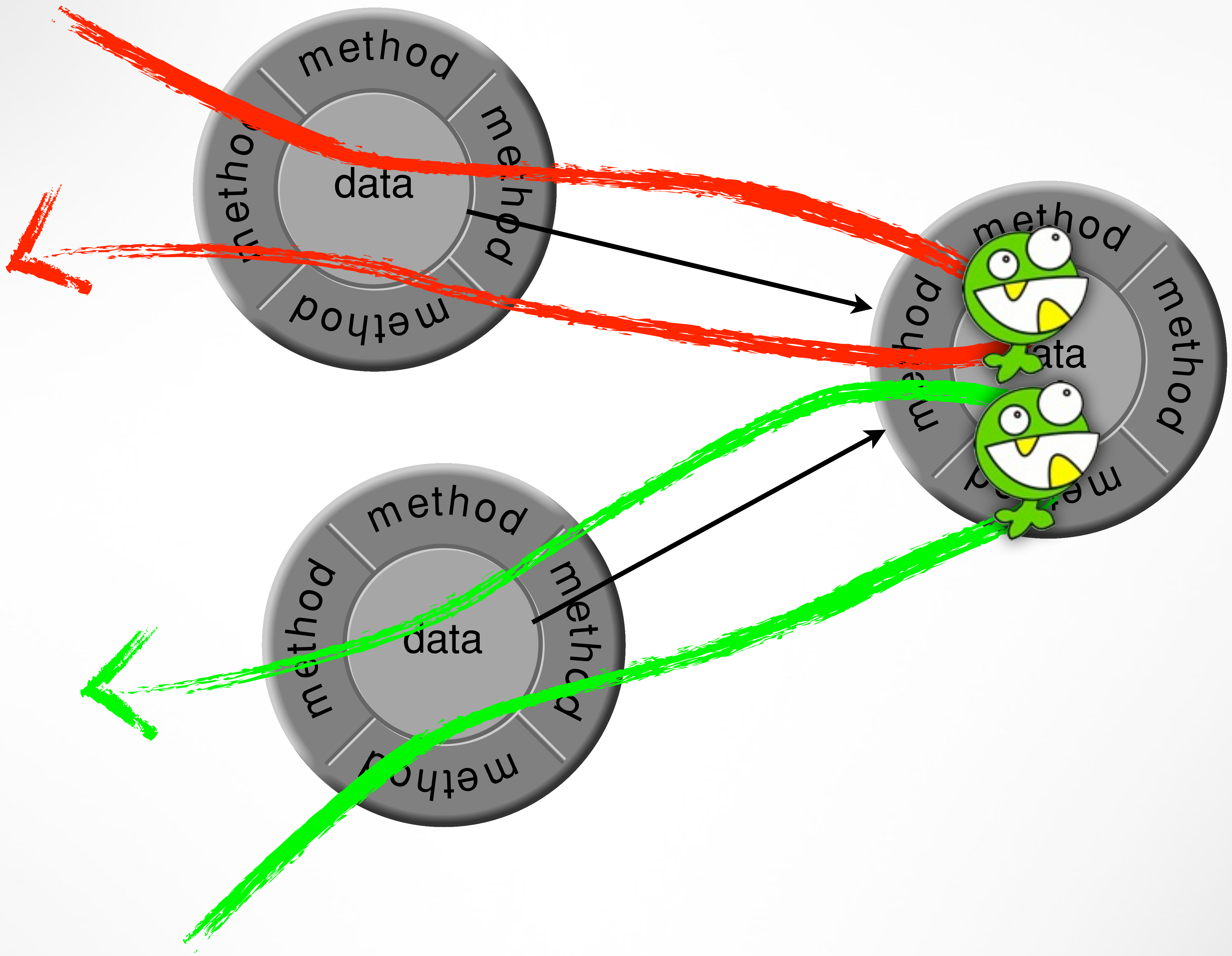


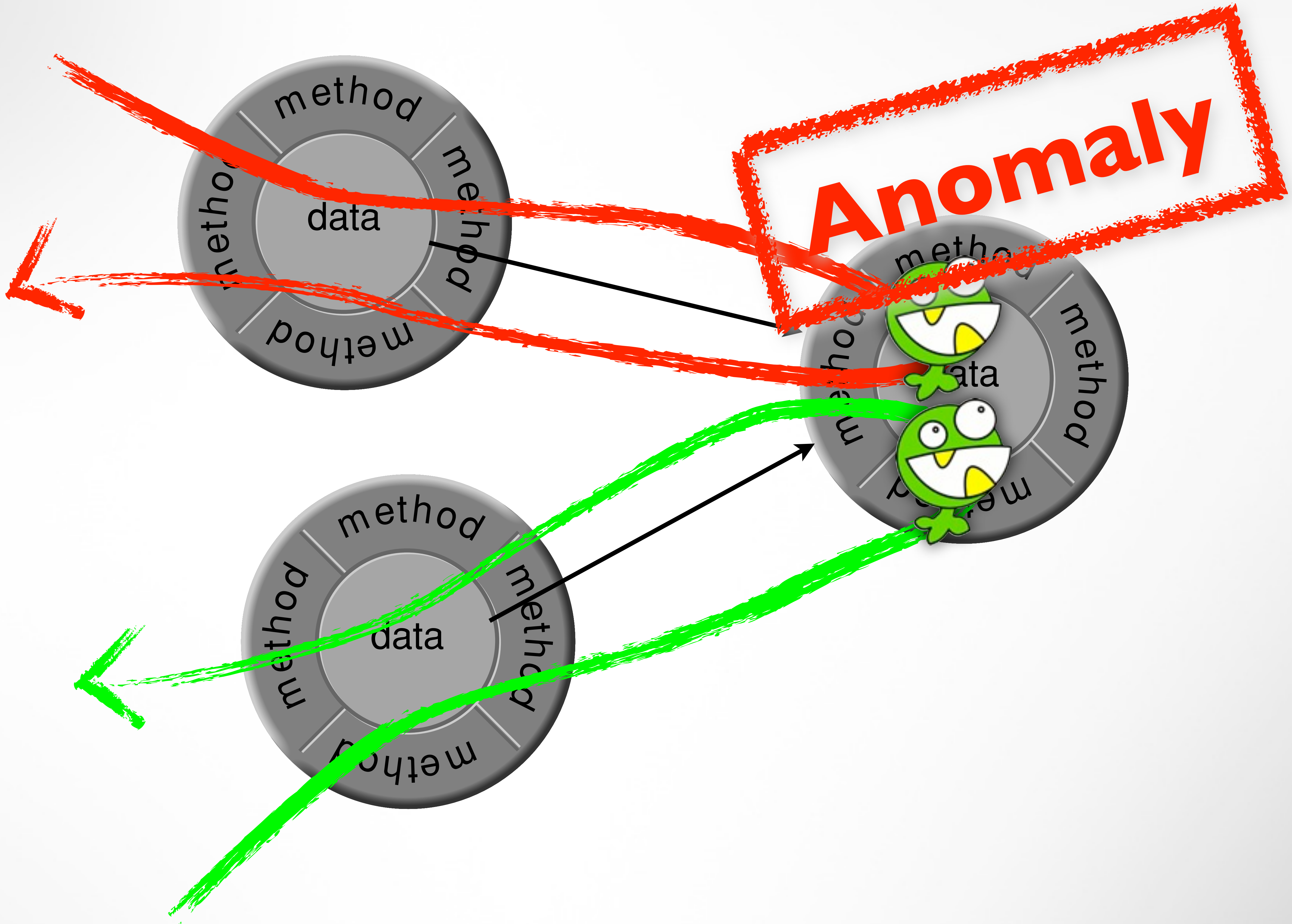

method
method
method
method
data
method
method
method
method
data
method
method
method
method
data

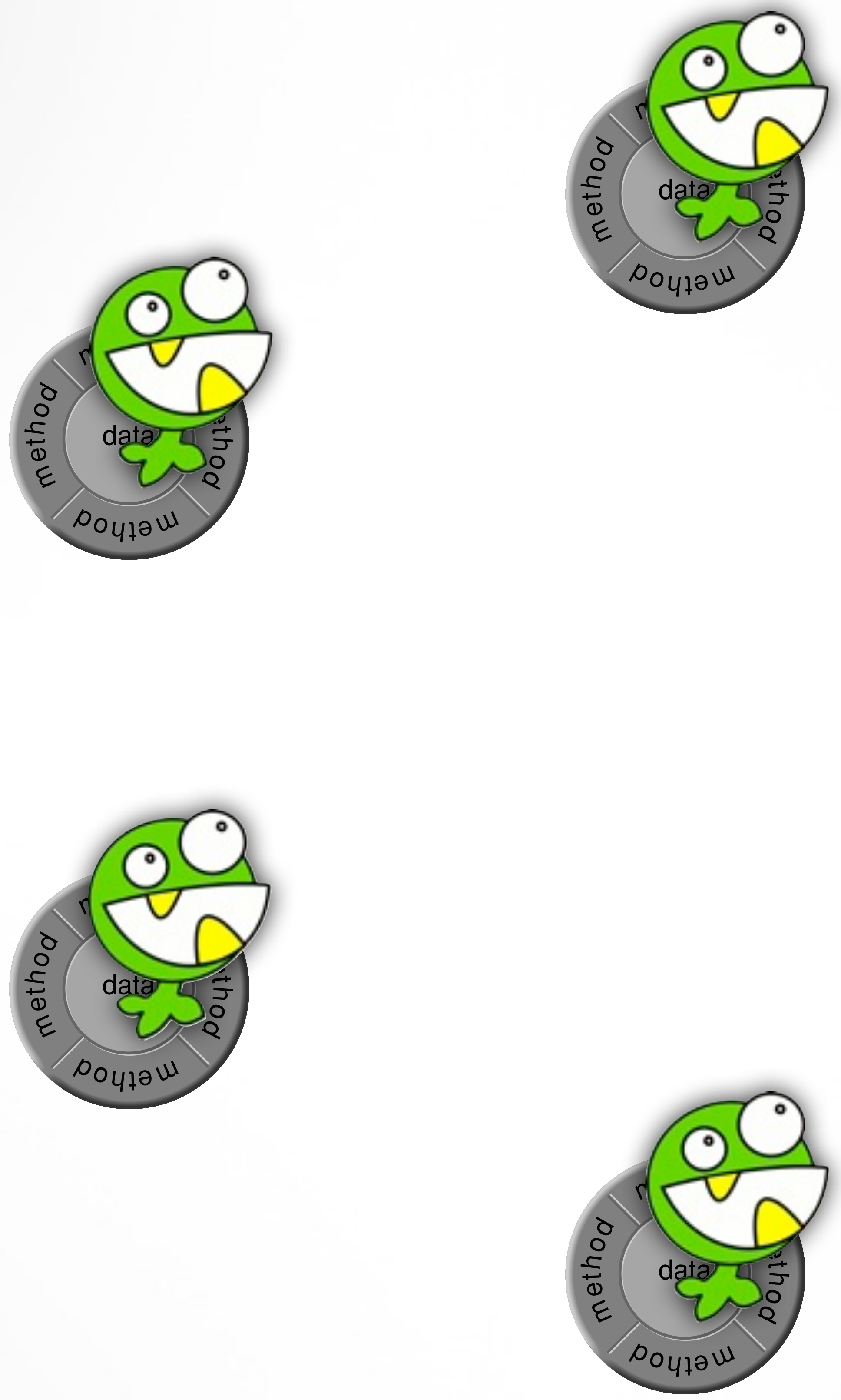

method
method
method
method
data
method
method
method
method
data
method
method
method
method
data

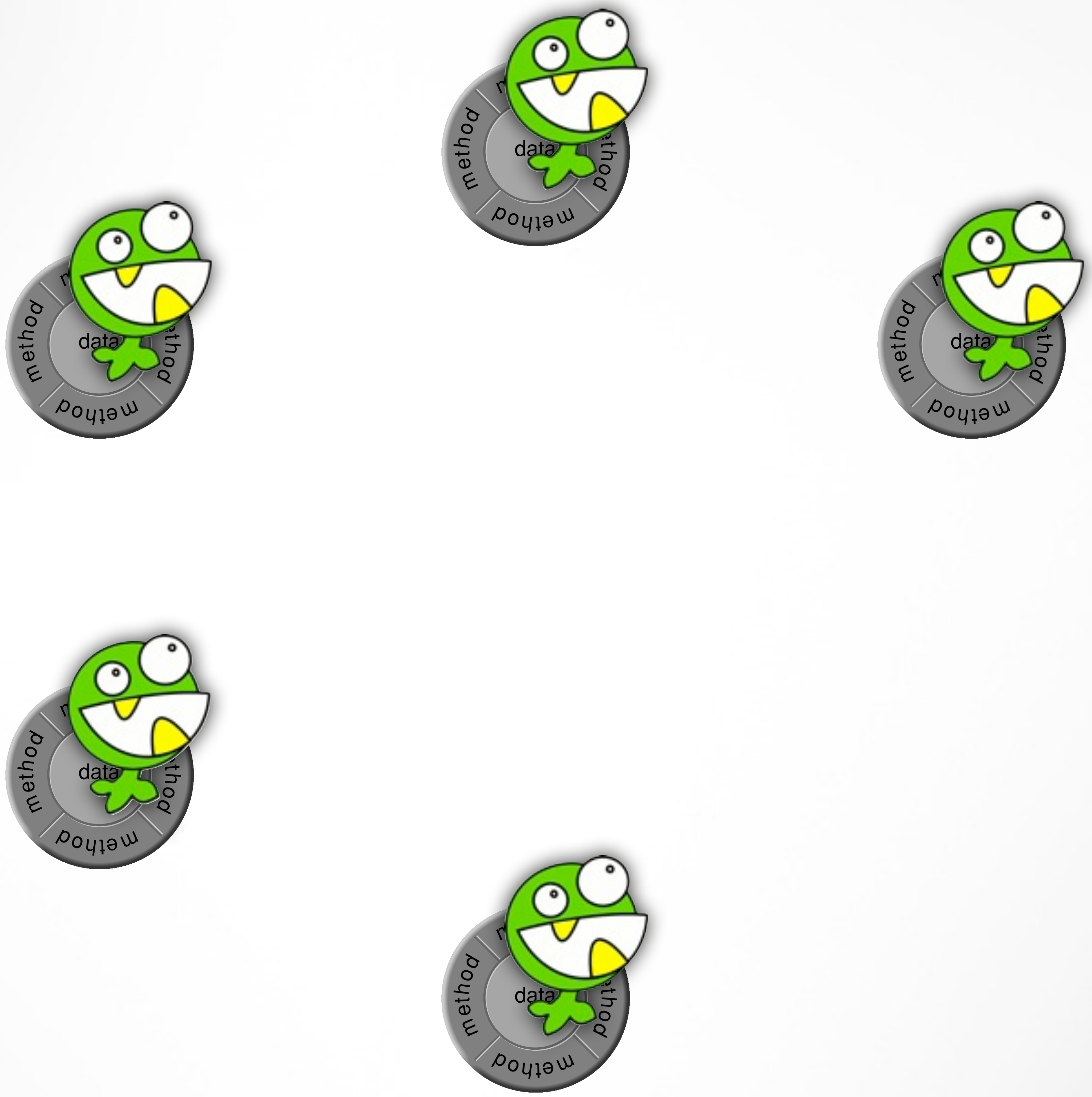

method
method
method
method
data
method
method
method
method
data
method
method
method
method
data

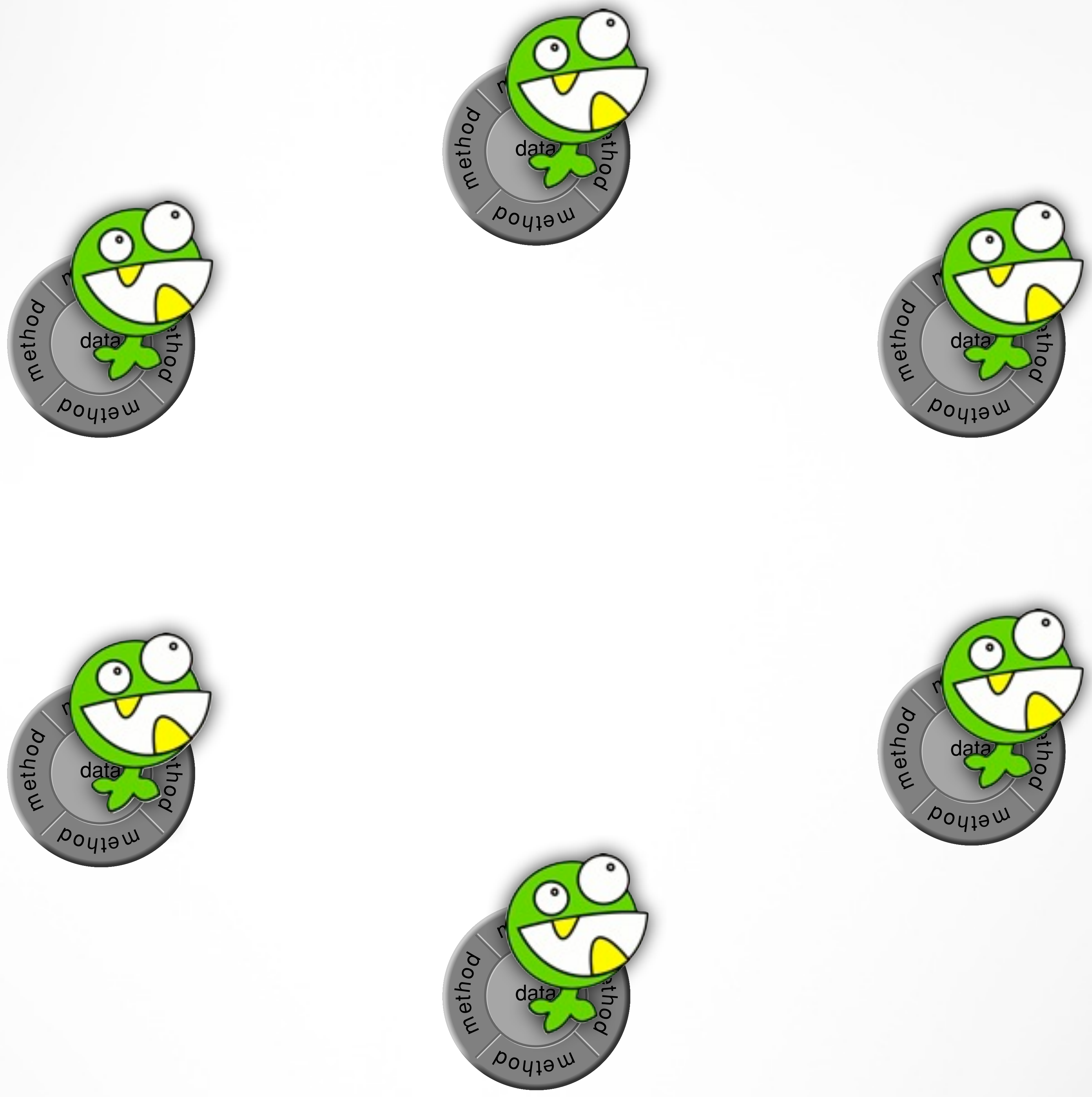

method
method
method
method
data
method
method
method
method
data
method
method
method
method
data


method
method
method
method
data
method
method
method
method
data
method
method
method
method
data

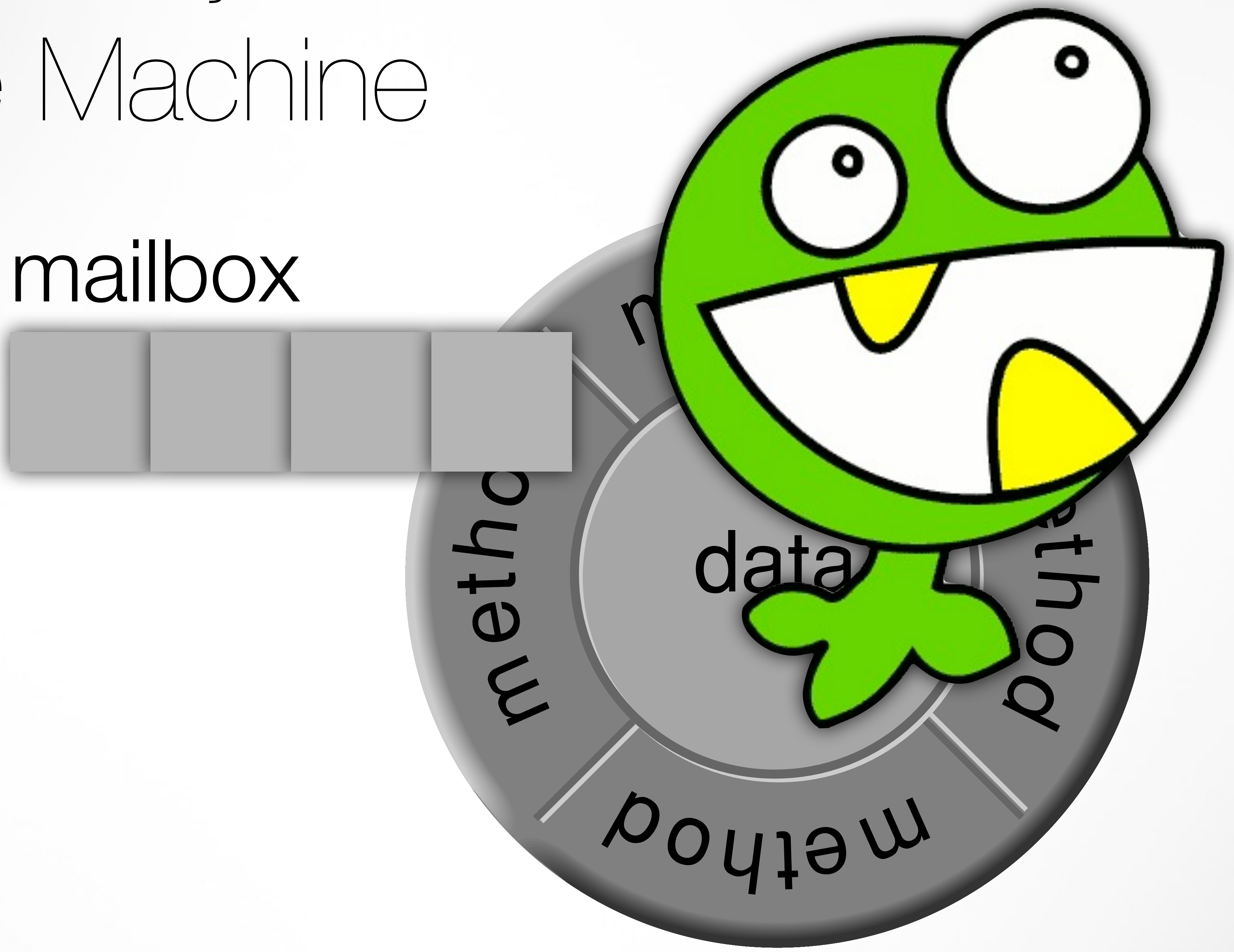

method
method
method
method
data
method
method
method
method
data
method
method
method
method
data
Anomaly


method
data
method
method

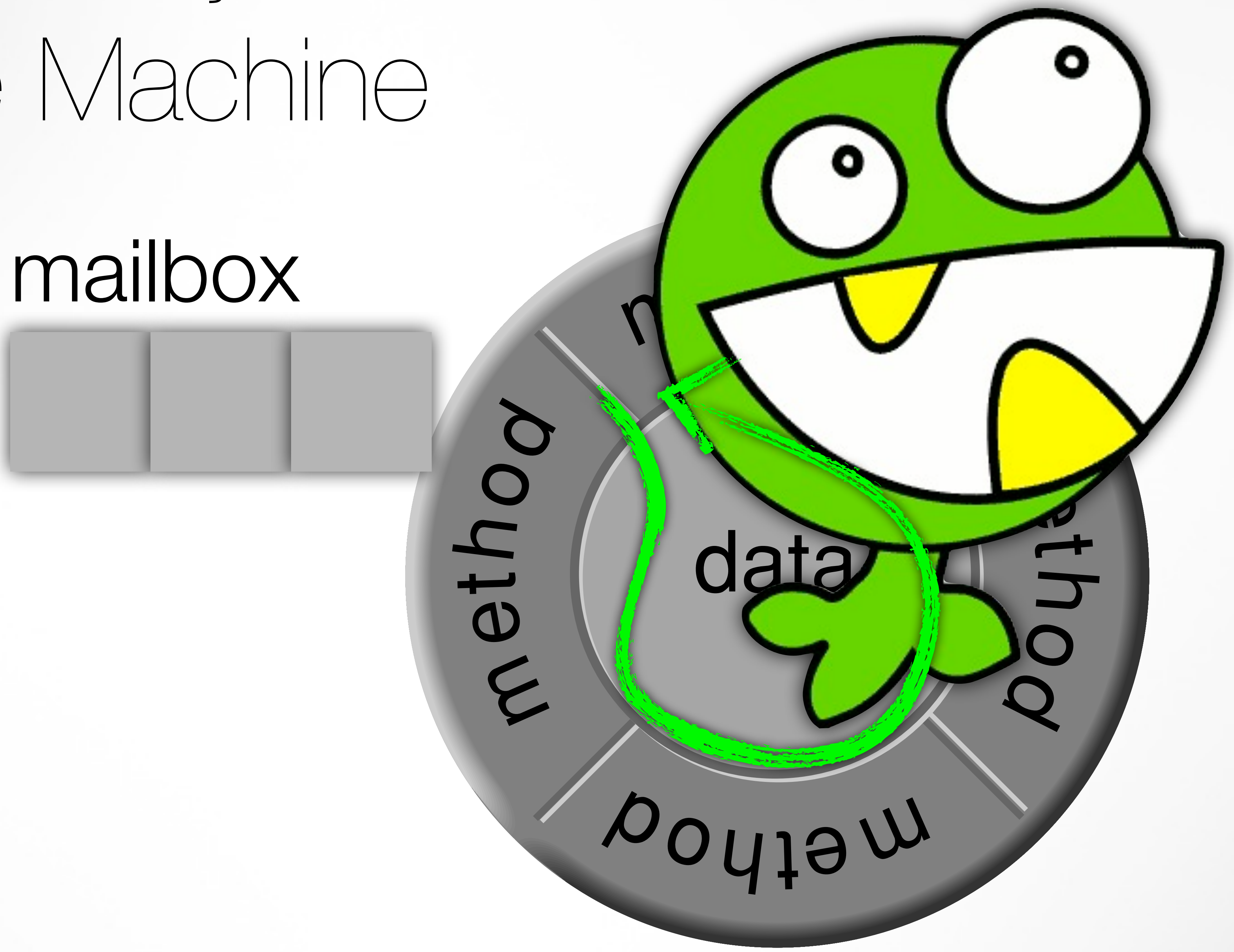

method
data
method
method
method


method
data
method
method

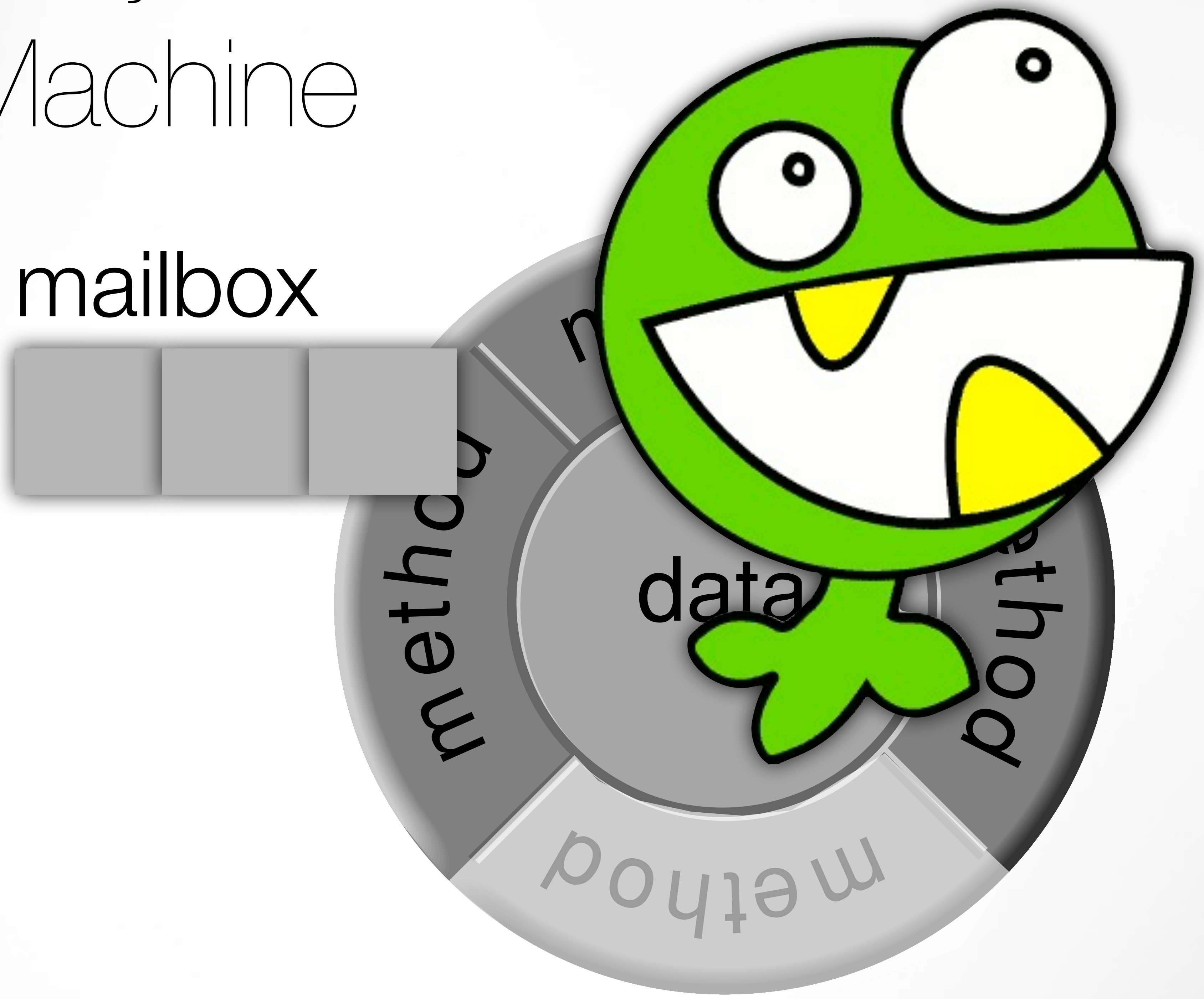
Active Object + State Machine



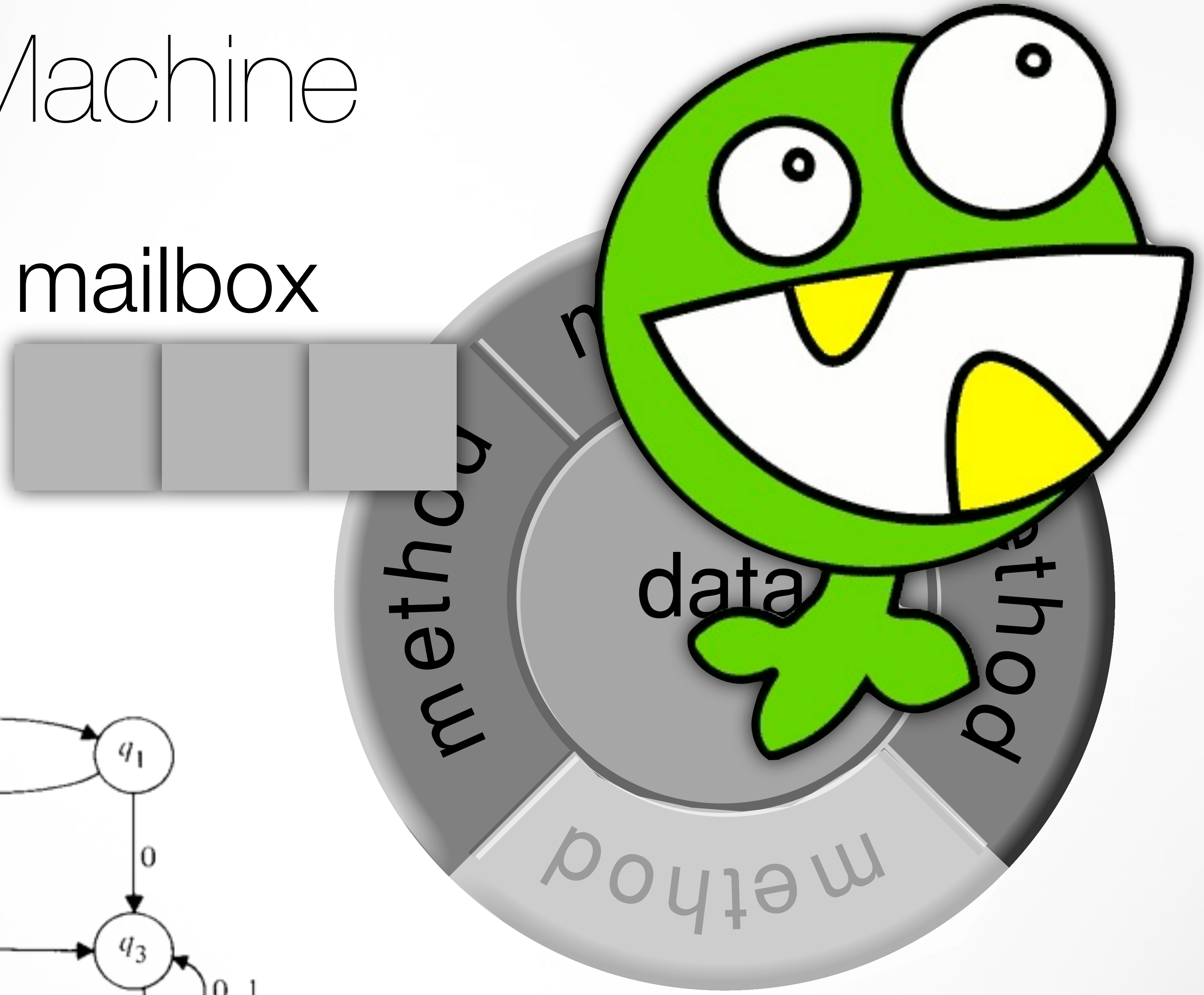
Active Object + State Machine

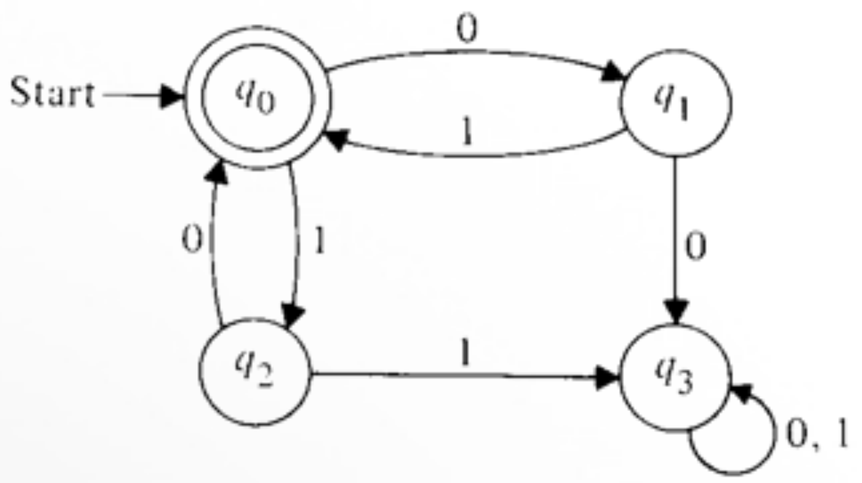


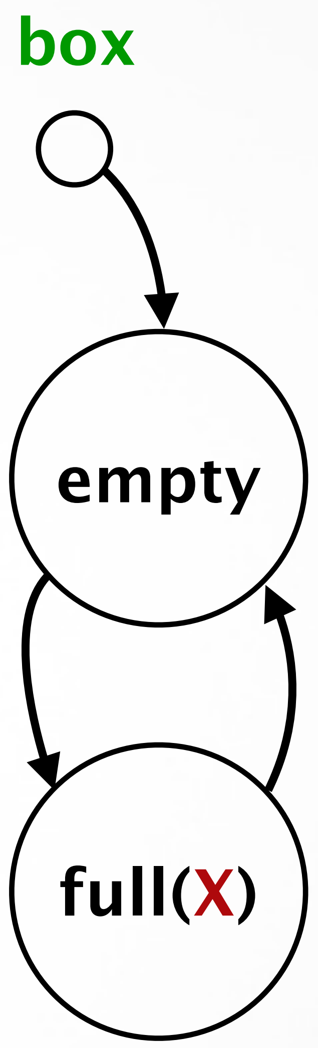
Active Object + State Machine

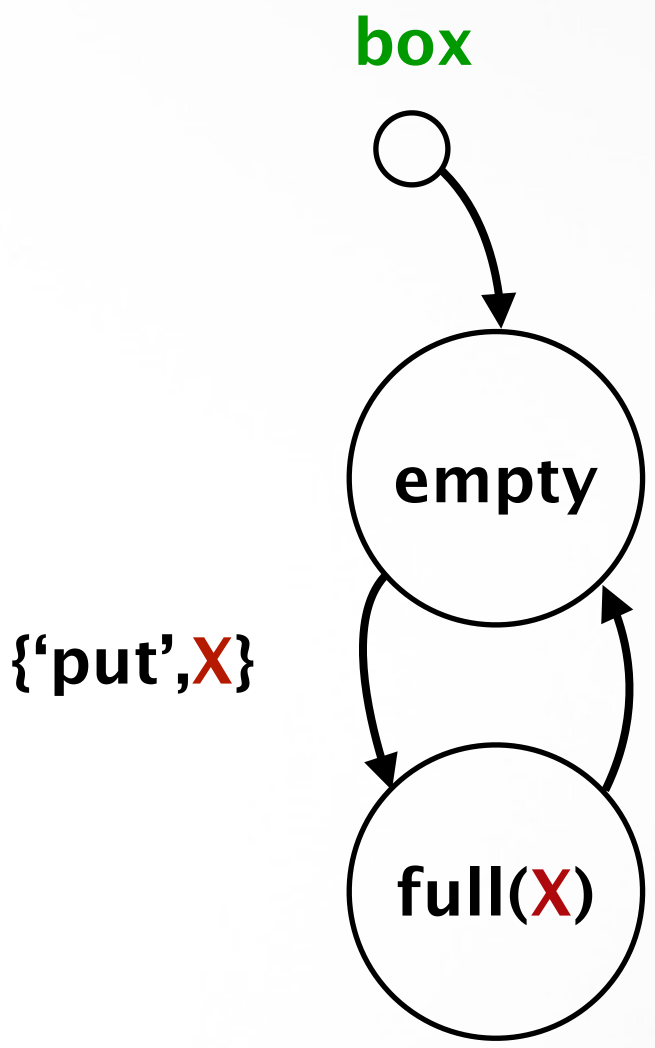


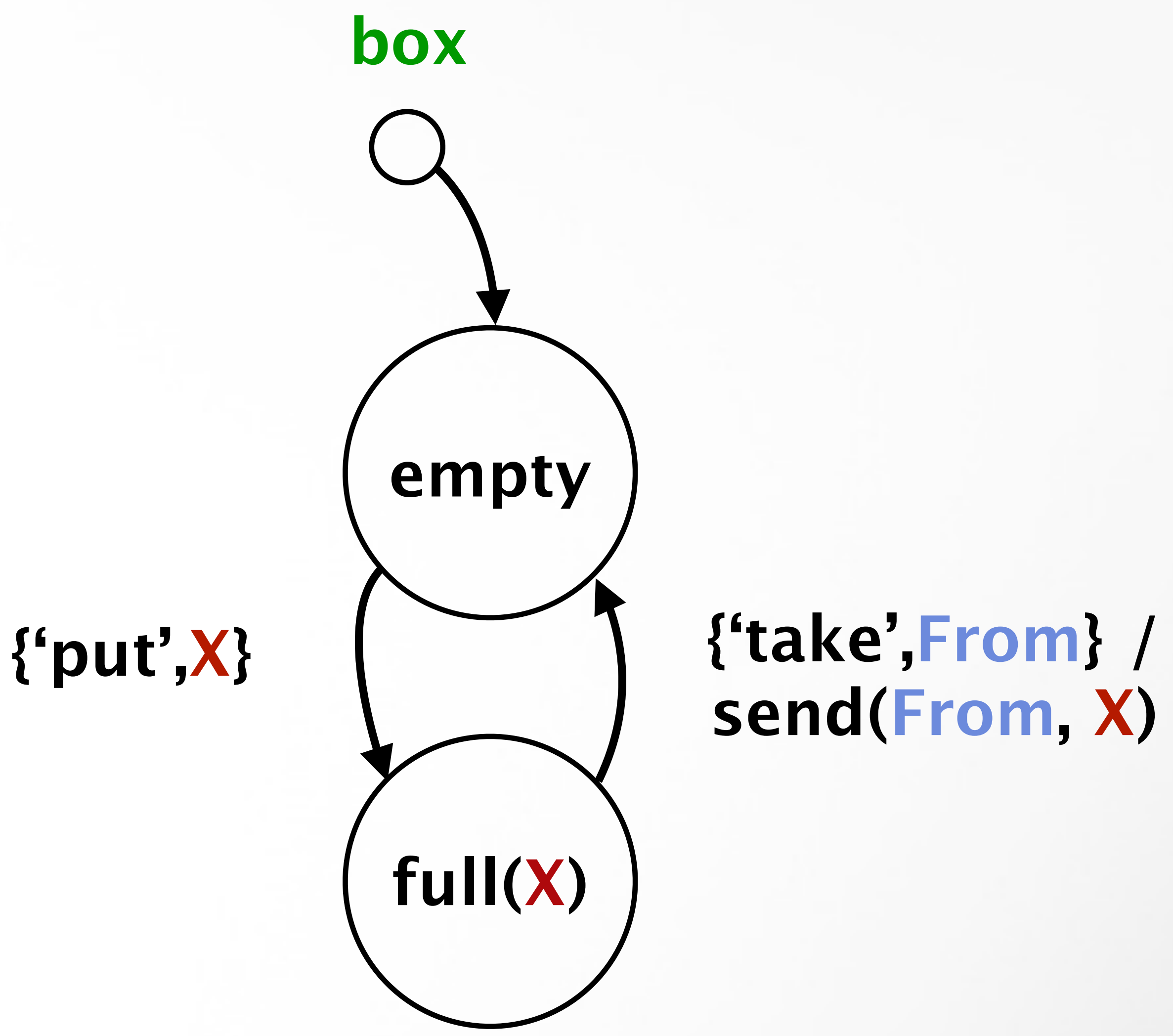
Active Object + State Machine



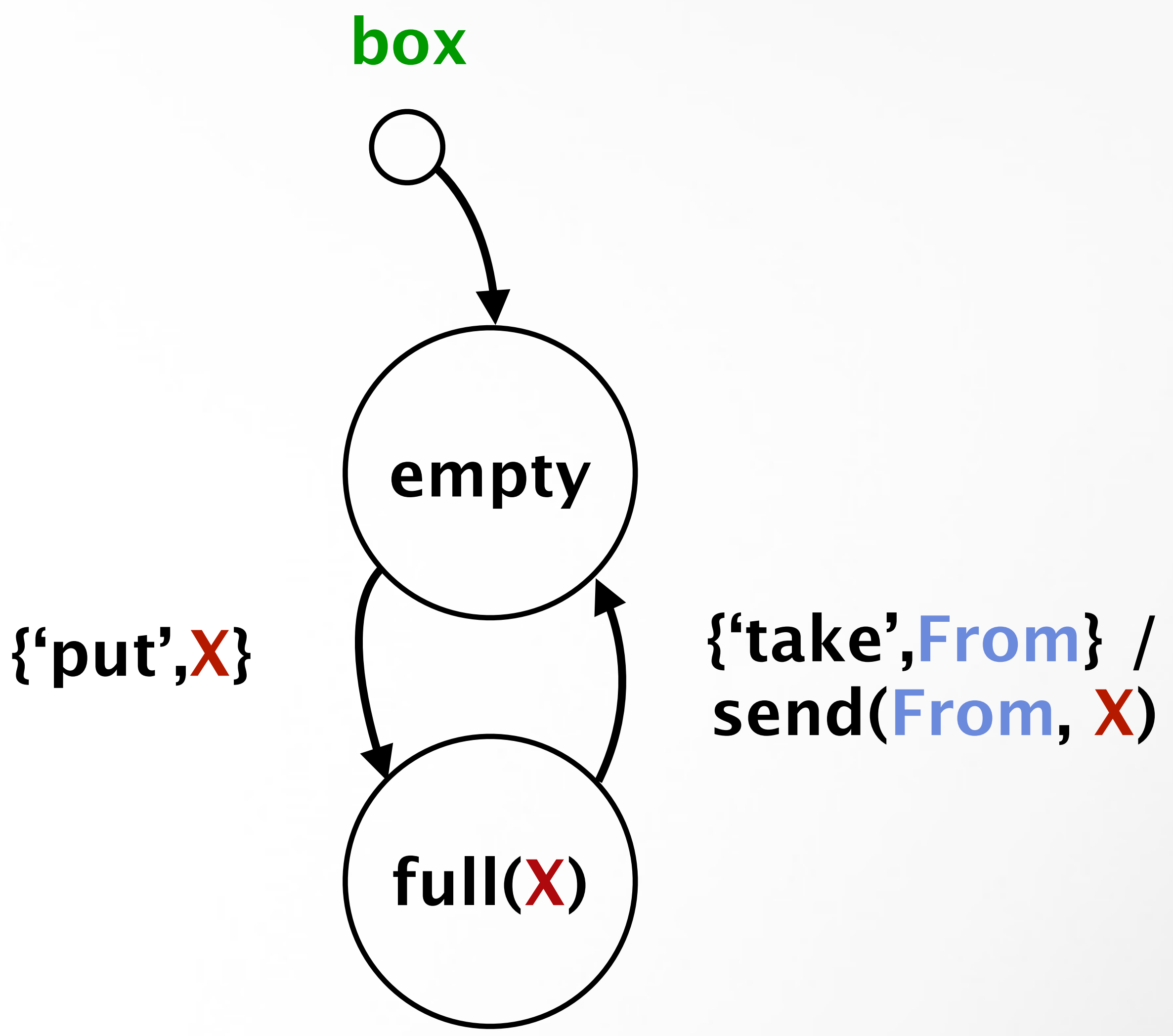



box
empty
full(X)

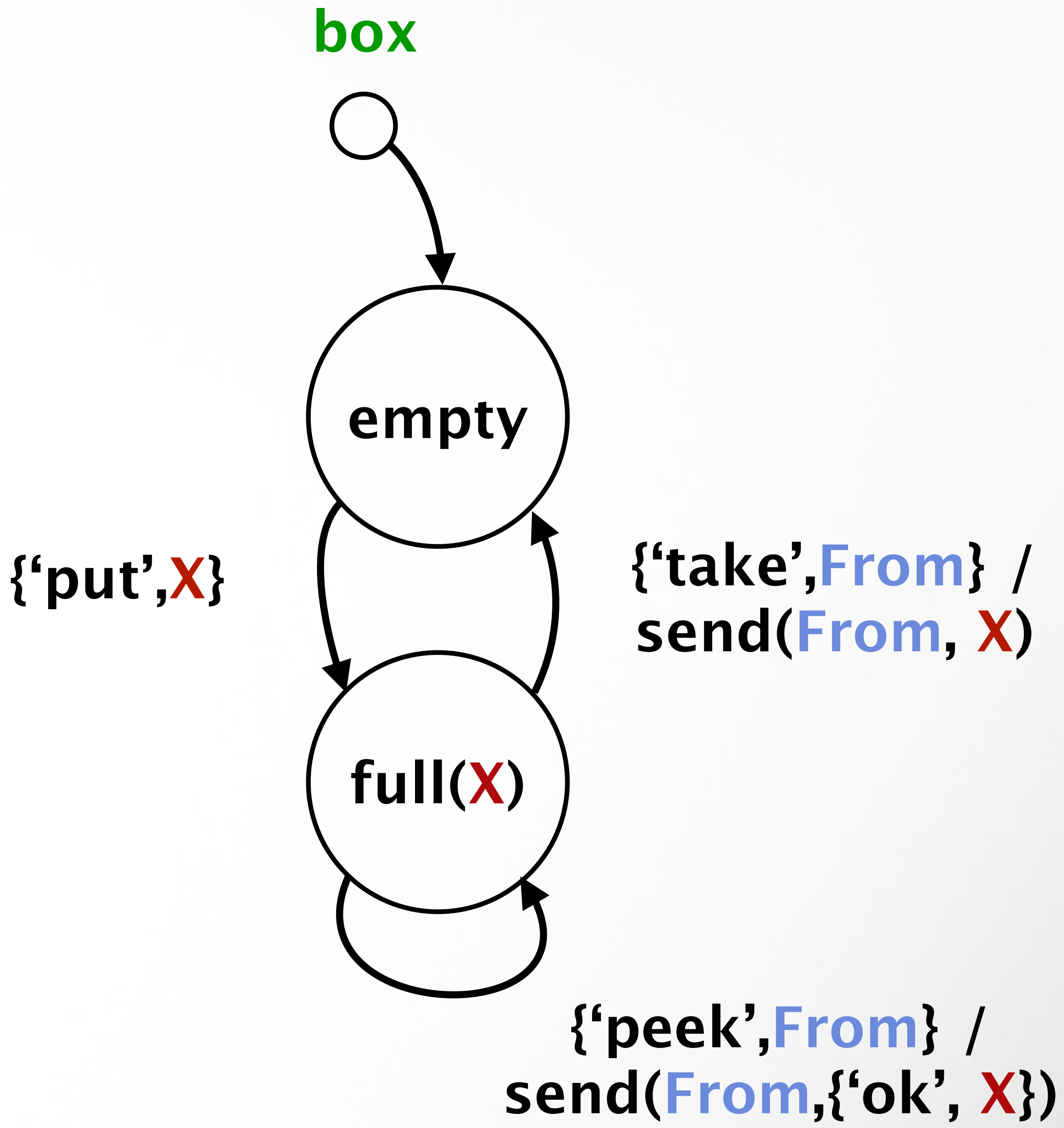

box
empty
{'put',X}
full(X)


box
empty
full(X)
{'put',X}
{'take',From} /
send(From, X)

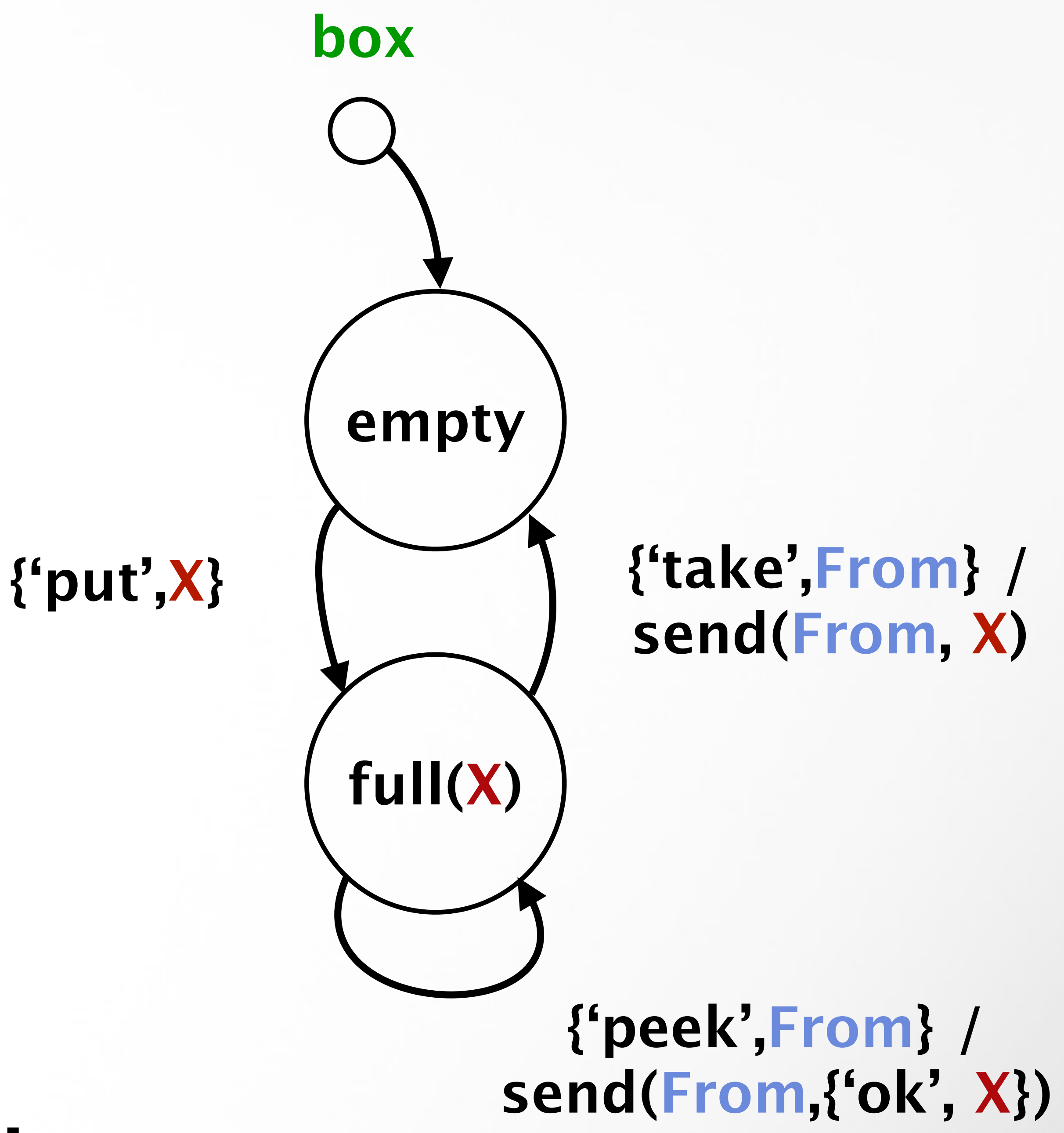
```
box() → empty().

empty() →
  receive
   {'put', X} →
         full(X)
  end.

full(X) →
  receive
   {'take', From} →
         From ! X,
         empty()
  end.
```



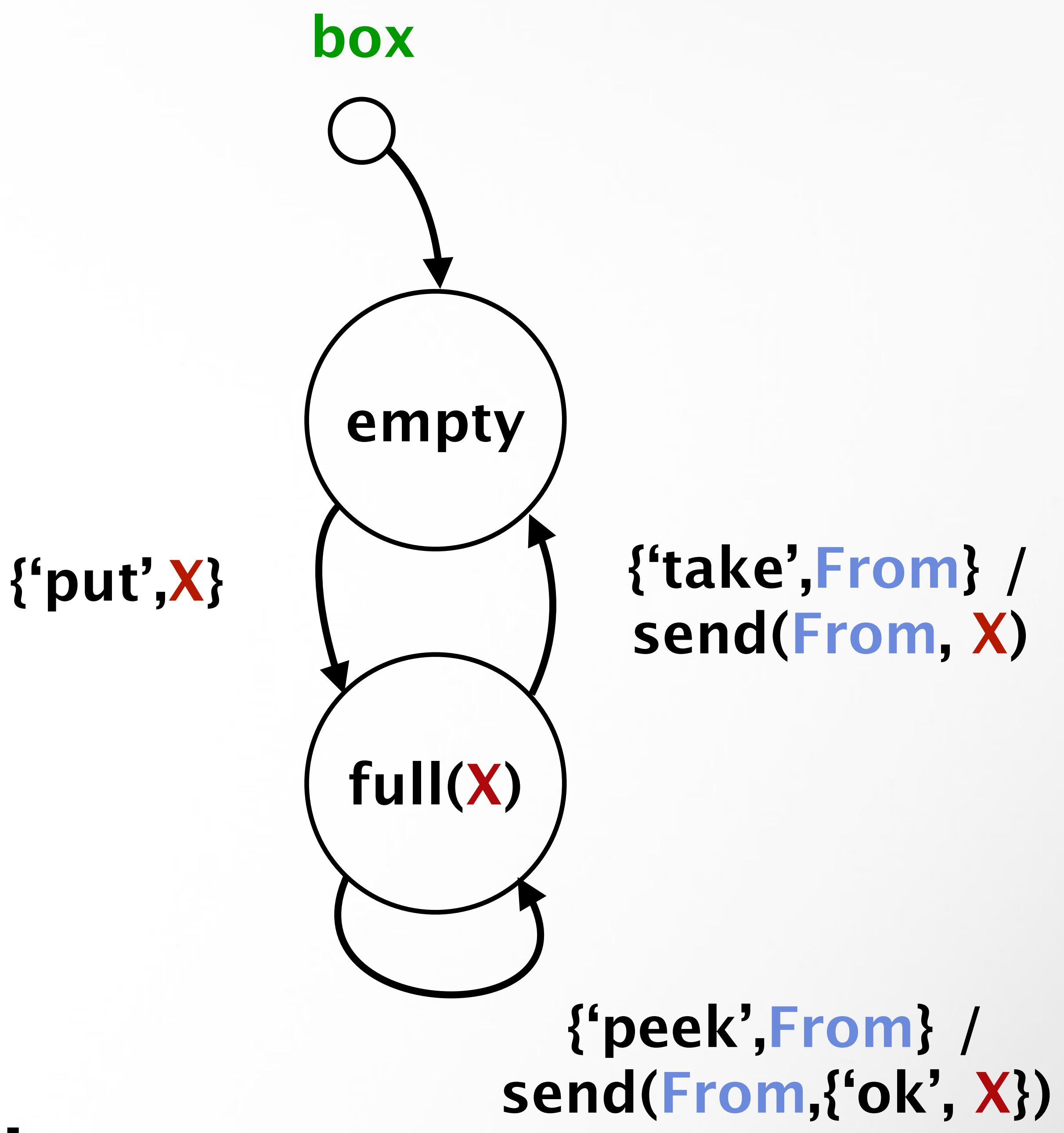
```
box() → empty().

empty() →
  receive
   {'put', X} →
        full(X)
  end.

full(X) →
  receive
   {'take', From} →
        From ! X,
        empty()
  end.
```


box
empty
{'put',X}
{'take',From} / send(From, X)
full(X)
{'peek',From} / send(From,{'ok', X})

```
box() → empty().

empty() →
   receive
    {‘put’, X} →
          full(X)
   end.

full(X) →
   receive
    {‘take’, From} →
          From ! X,
          empty();
    {‘peek’, From} →
          From ! {‘ok’, X},
          full(X)
   end.
```



```
box() → empty().

empty() →
  receive
   {'put', X} →
          full(X)
  end.

full(X) →
  receive
   {'take', From} →
          From ! X,
          empty();
   {'peek', From} →
          From ! {'ok', X},
          full(X)
  end.
```



box
empty
full(X)
{'put',X}
{'take',From} / send(From, X)
{'peek',From} / send(From,{'ok', X})

```
box() → empty().

empty() →
  receive
   {‘put’, X} →
        full(X)
  end.

full(X) →
  receive
   {‘take’, From} →
        From ! X,
        empty();
   {‘peek’, From} →
        From ! {‘ok’, X},
        full(X)
  end.
```

```
box() → empty().                Box = spawn(fun box/0),

empty() →
   receive
    {‘put’, X} →
          full(X)
   end.

full(X) →
   receive
    {‘take’, From} →
          From ! X,
          empty();
    {‘peek’, From} →
          From ! {‘ok’, X},
          full(X)
   end.
```

TRIFORK.

```
box() → empty().

empty() →
   receive
    {‘put’, X} →
         full(X)
   end.

full(X) →
   receive
    {‘take’, From} →
         From ! X,
         empty();
    {‘peek’, From} →
         From ! {‘ok’, X},
         full(X)
   end.
```

```
Box = spawn(fun box/0),

Box ! {‘put’, 27},
```

TRIFORK.

```
box() → empty().

empty() →
   receive
    {‘put’, X} →
         full(X)
   end.

full(X) →
   receive
    {‘take’, From} →
         From ! X,
         empty();
    {‘peek’, From} →
         From ! {‘ok’, X},
         full(X)
   end.
```

```
Box = spawn(fun box/0),

Box ! {‘put’, 27},

Box ! {‘take’, self()},
receive
  Value → print(Value)
end
```

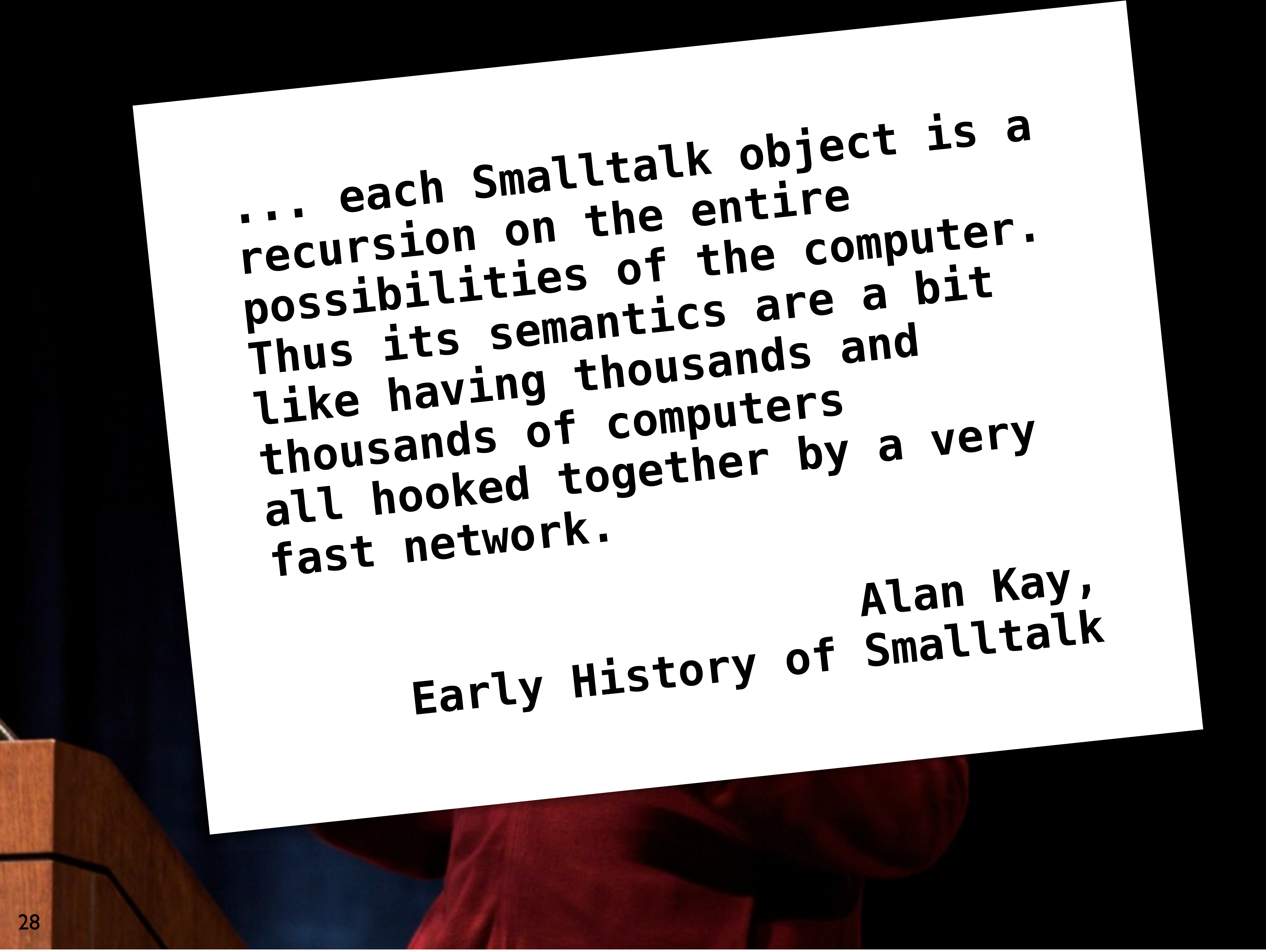
TRIFORK.

Objects	**Interface** Fixed API
Actors	**Protocol** API changes with internal state

Objects	~~**Interface**~~ Anomaly Fixed API
Actors	**Protocol** API changes with internal state

TRIFORK.

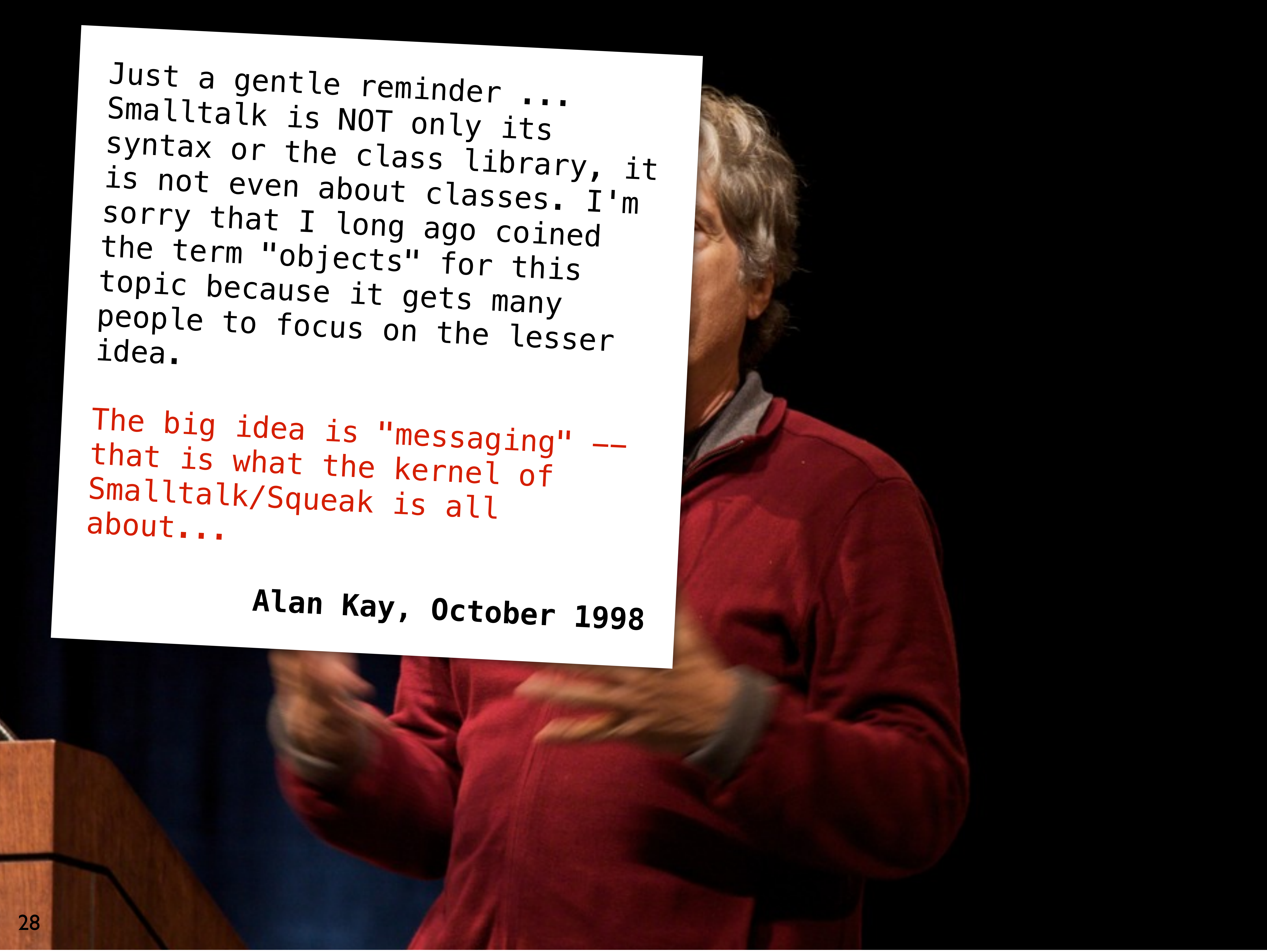
28



... each Smalltalk object is a recursion on the entire possibilities of the computer. Thus its semantics are a bit like having thousands and thousands of computers all hooked together by a very fast network.

Alan Kay,

Early History of Smalltalk



Just a gentle reminder ... Smalltalk is NOT only its syntax or the class library, it is not even about classes. I'm sorry that I long ago coined the term "objects" for this topic because it gets many people to focus on the lesser idea.

The big idea is "messaging" -- that is what the kernel of Smalltalk/Squeak is all about...

Alan Kay, October 1998

But isn't this expensive?

But isn’t this expensive?

... heard in the ‘90s

But isn't this expensive?

... heard in the '90s

Use Objects!

100.000's of objects,
are you crazy?

But isn't this expensive?

... heard yesterday

Use Actors!

100.000's of processes, silly you!

“What if the OOP parts of other languages (Java, C++, Ruby, etc.) had the same behavior as their concurrency support? What if you were limited to only creating 500 objects total for an application because any more would make the app unstable and almost certainly crash it in hard-to-debug ways? What if these objects behaved differently on different platforms?”

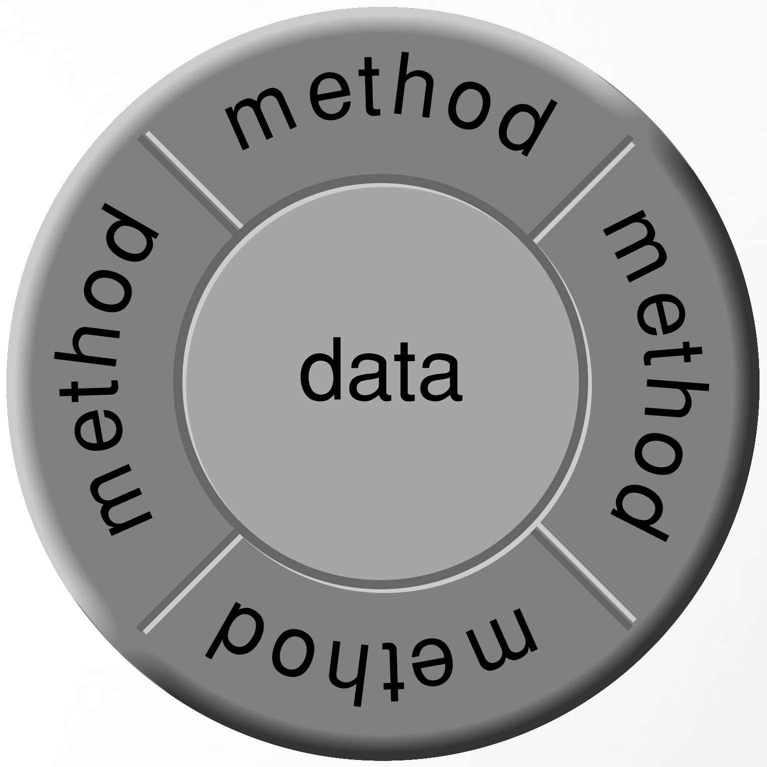
Joe Armstrong, creator of Erlang

as quoted in

http://weblog.hypotheticalabs.com/?p=217

Effect Containment

- Functional languages *disallow effects*
- Many object-oriented styles *encourage side effects*.
- Actors *confine effects*



TRIFORK.

From C++ to Java	Garbage Collection
From Java to Erlang	**State Containment**

Threads and Locks don’t compose well

- Threaded programs are error prone (we already knew that).
- Good thread design requires global knowledge.

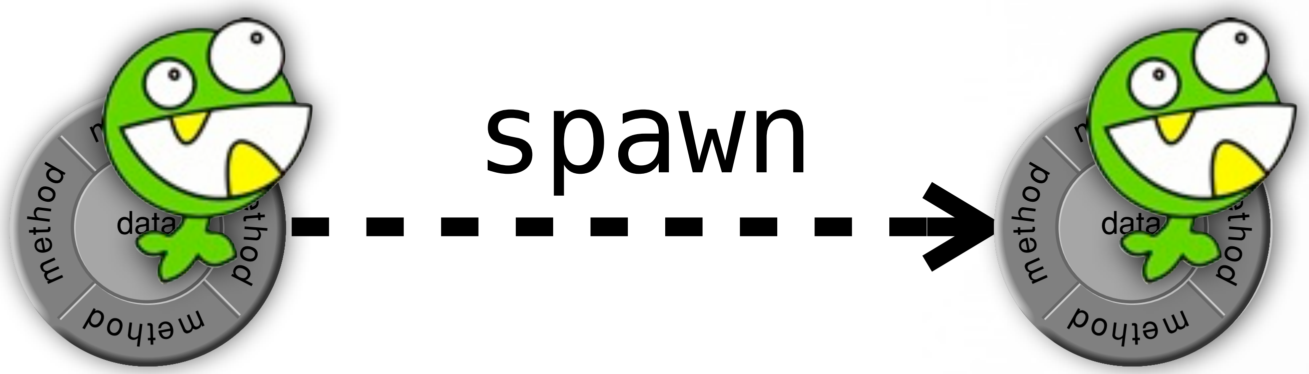
TRIFORK.

Threads and Locks don't compose well

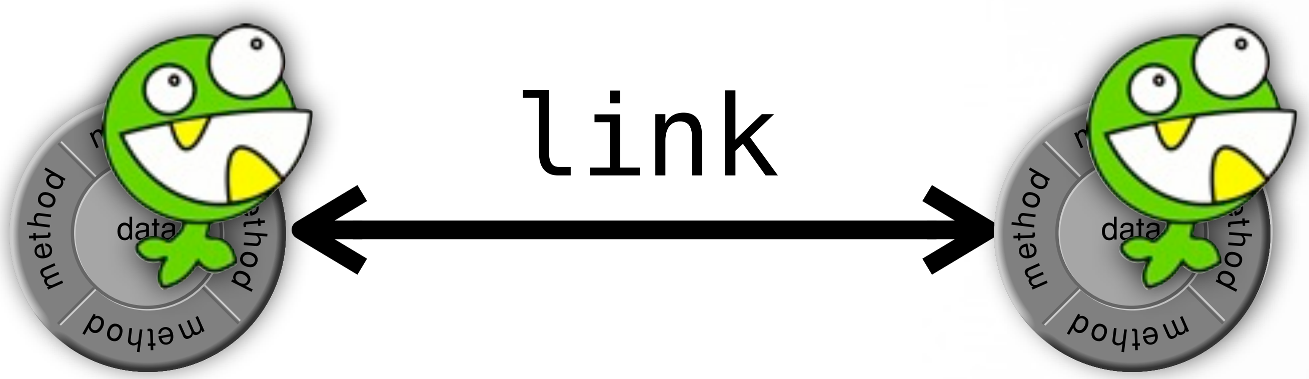
- Threaded programs are error prone (we already knew that).
- Good thread design requires global knowledge.

Anomaly

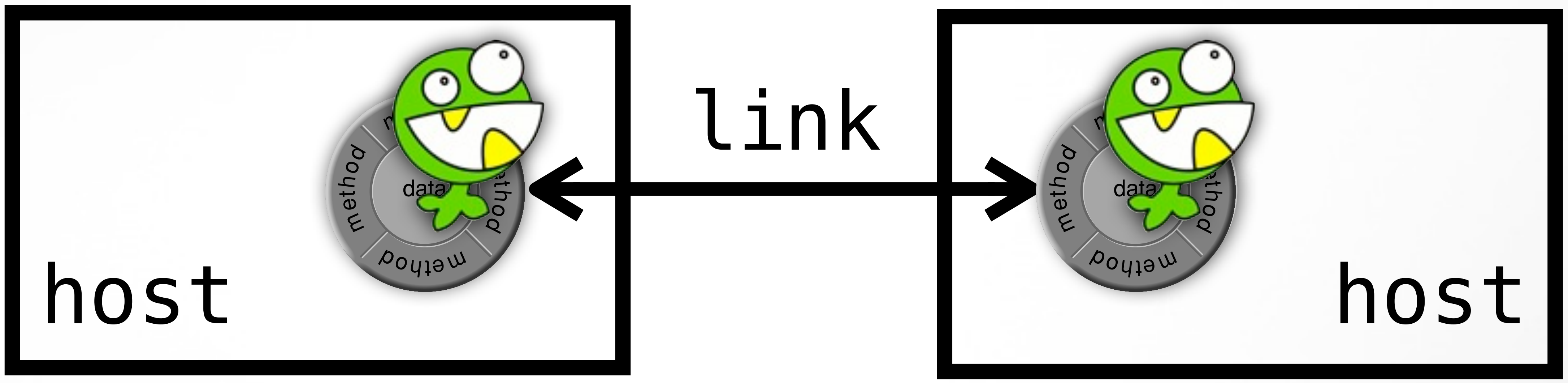
Activity Composition



Activity Composition



Activity Composition

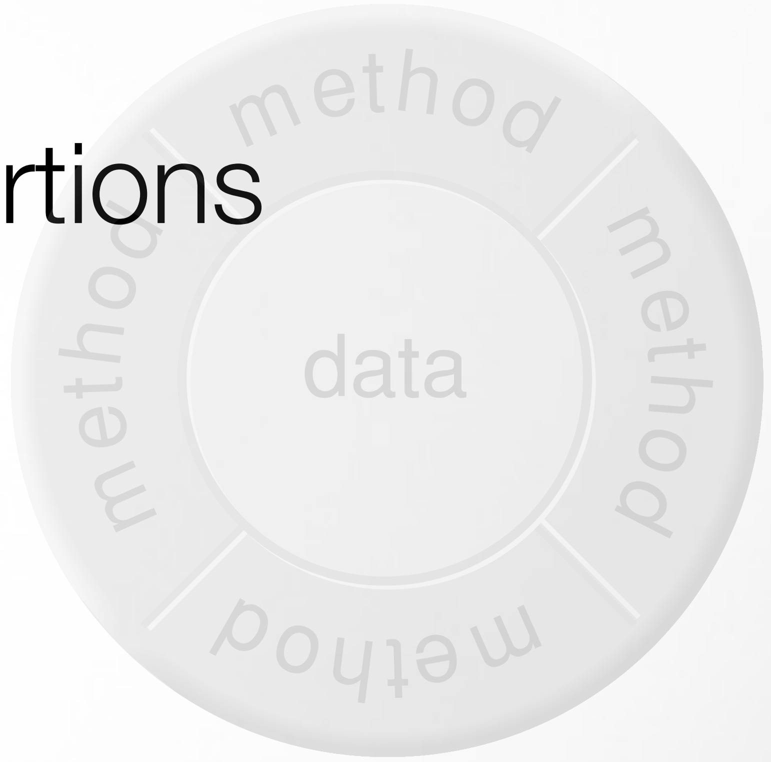


"Let it Fail" philosophy

method
method
method
method
data

“Let it Fail” philosophy

- Write code with lots of assertions



"Let it Fail" philosophy

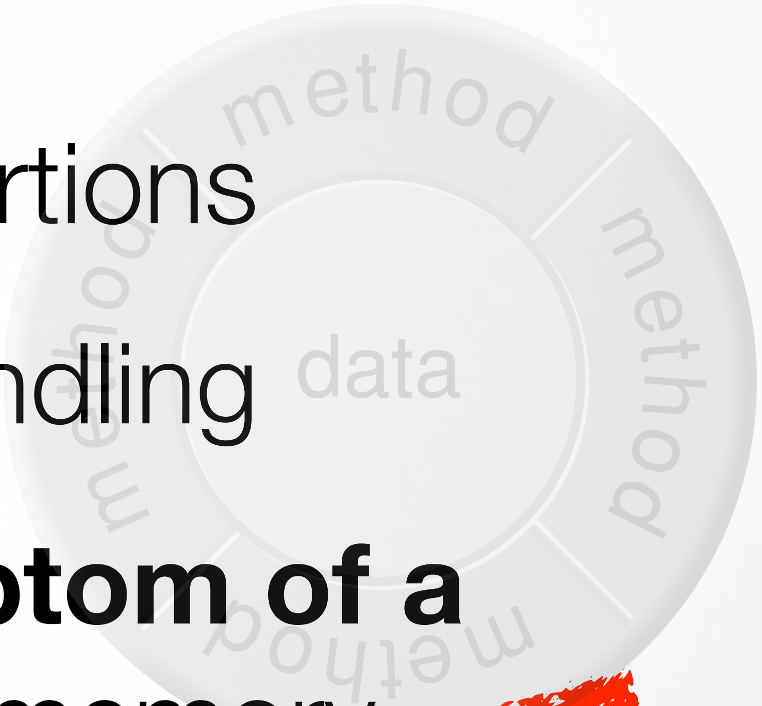
- Write code with lots of assertions
- Let a meta-level do fault handling

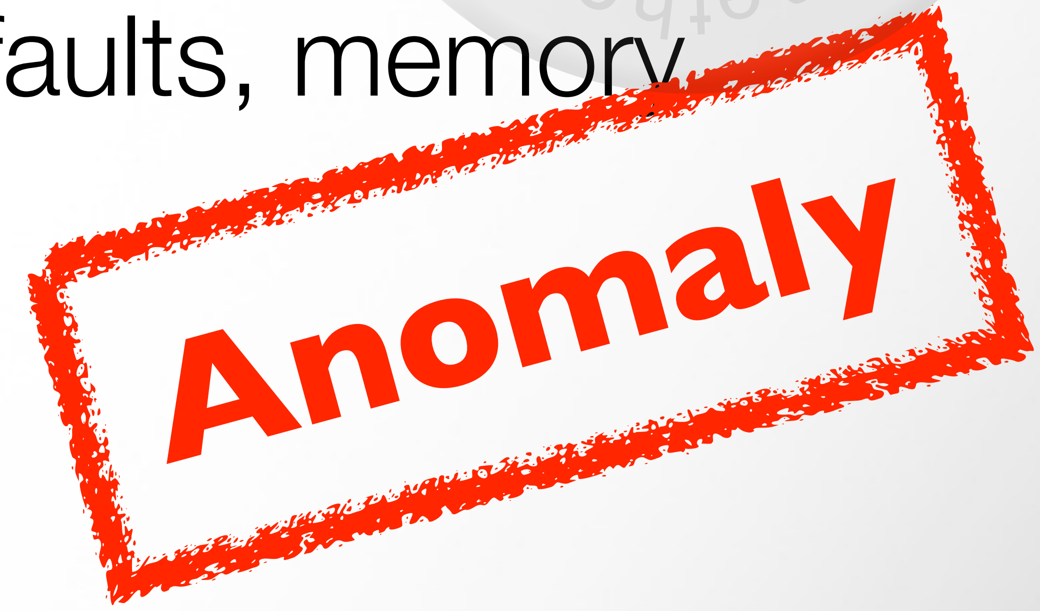
“Let it Fail” philosophy

- Write code with lots of assertions
- Let a meta-level do fault handling
- **Defensive code is a symptom of a weak platform** (segfaults, memory leaks, ...)

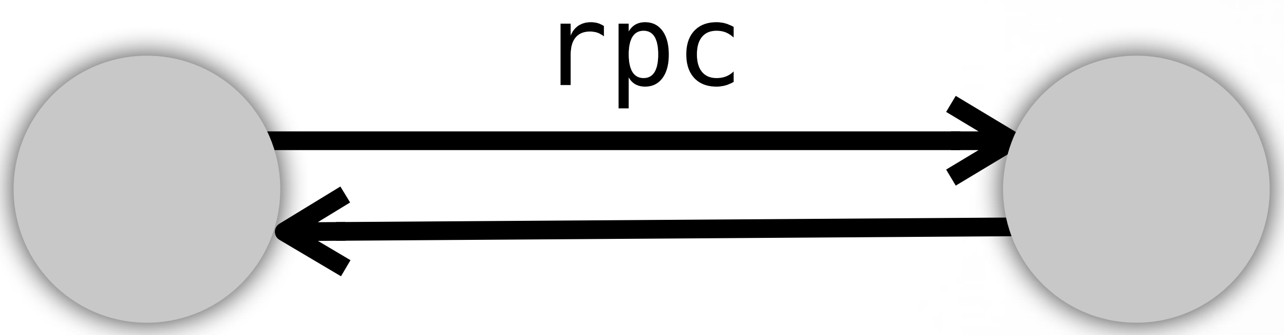
"Let it Fail" philosophy

- Write code with lots of assertions
- Let a meta-level do fault handling
- **Defensive code is a symptom of a weak platform** (segfaults, memory leaks, ...)





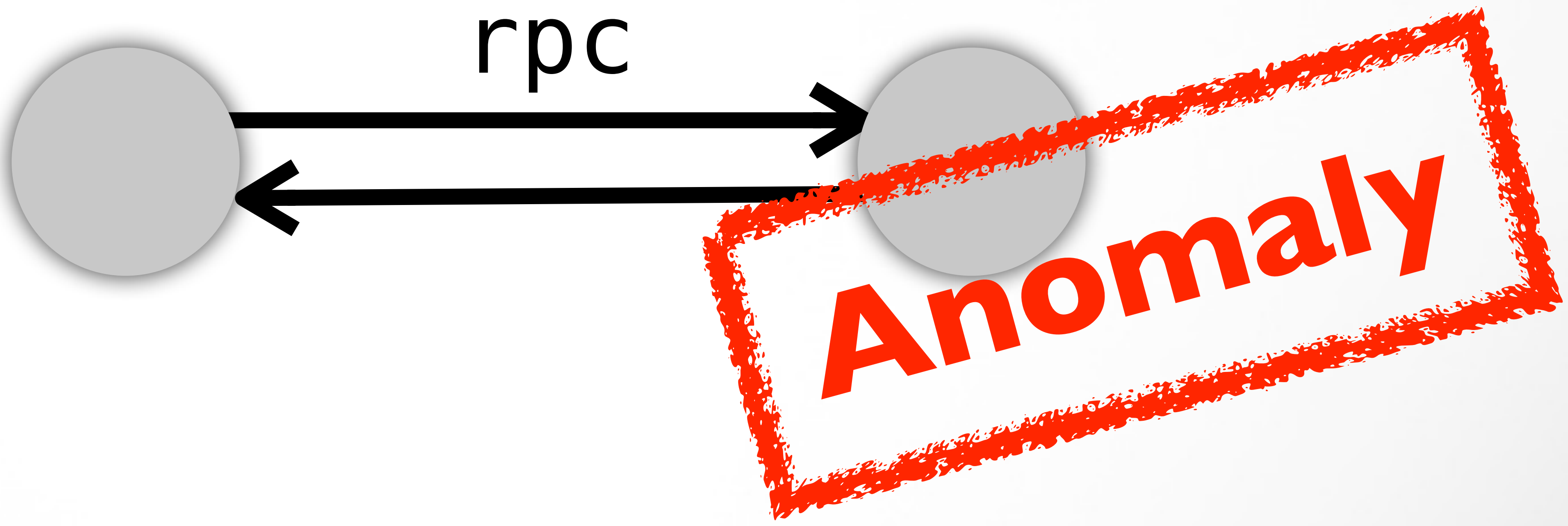
Martin Fowler's First Law of Distributed Objects Design



Starbucks doesn't use transactions

Steve Vinoski: *RPC and its Offspring: Convenient, Yet Fundamentally Flawed*

Martin Fowler's First Law of Distributed Objects Design



Starbucks doesn't use transactions

Steve Vinoski: *RPC and its Offspring: Convenient, Yet Fundamentally Flawed*

Abstractions

TRIFORK.

Abstractions

- Any abstraction (hiding code) is problematic to distribute, persist, etc.

Abstractions

- Any abstraction (hiding code) is problematic to distribute, persist, etc.
- You want to distribute/persist **simple data** (as in sql databases, document stores)

TRIFORK.

Abstractions

- Any abstraction (hiding code) is problematic to distribute, persist, etc.
- You want to distribute/persist **simple data** (as in sql databases, document stores)
- JSON's popularity is a testament to this anomaly.

TRIFORK.

Abstractions

- Any abstraction (hiding code) is problematic to distribute, persist, etc.
- You want to distribute/persist **simple data** (as in sql databases, document stores)
- JSON's popularity is a testament to this anomaly.

Anomaly

Simple Values

Simple Values

- Tuple, List, Record, Number, String, Binary

Simple Values

- Tuple, List, Record, Number, String, Binary
- Pattern matching is polymorphism for values

Simple Values

- Tuple, List, Record, Number, String, Binary
- Pattern matching is polymorphism for values
- Erlang data stores (like Mnesia) just store values, not bytes or objects.

Simple Values

- Tuple, List, Record, Number, String, Binary
- Pattern matching is polymorphism for values
- Erlang data stores (like Mnesia) just store values, not bytes or objects.
- Too much of my Java programs are boilerplate code.

TRIFORK.

“Object” Model Anomalies

- Thread & Locks
- Interfaces with Fixed API
- Defensive Code
- RPC/RMI
- Boilerplate code for persistence

TRIFORK.

Actor “Solutions”

- Processes w/ state containment
- Protocols
- Let it Fail
- Async Messaging
- Send & store simple Data

TRIFORK.

ERLANG

Making reliable distributed control systems in the presence of errors

Making reliable distributed control systems in the presence of errors

ERLANG

TRIFORK.

Making reliable distributed control systems in the presence of errors

- The “secret weapon” for Ericsson’s market leading, real-time telephony systems.

ERLANG

TRIFORK.

Making reliable distributed control systems in the presence of errors

- The “secret weapon” for Ericsson’s market leading, real-time telephony systems.
- <u>20+ years of experience</u> to learn from.

ERLANG

TRIFORK.

Erlang/OTP

Open Telecommunications Platform

- Embedded Distributed Systems
- High Availability
- In-Production Upgrades

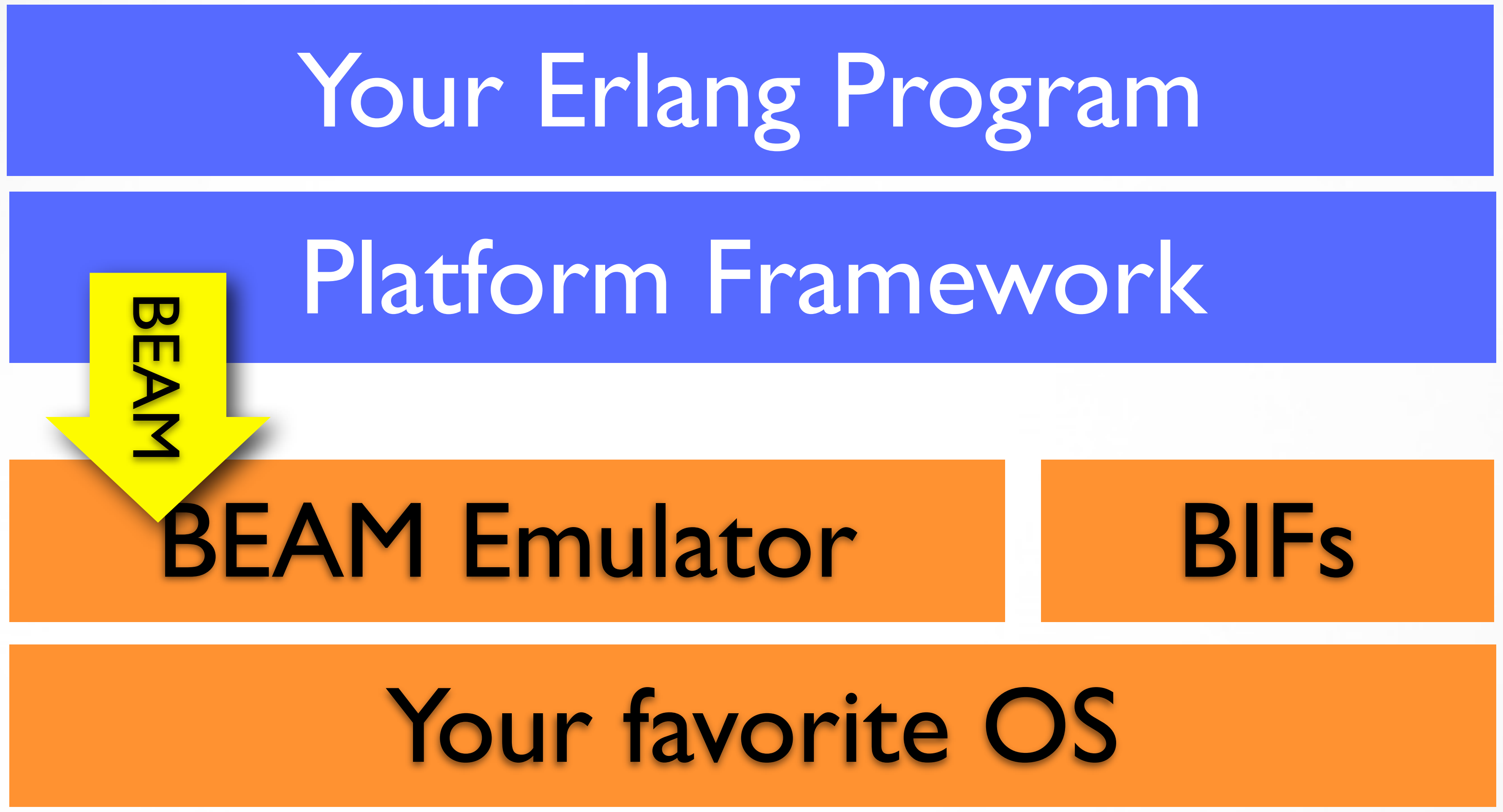
How to Learn Erlang

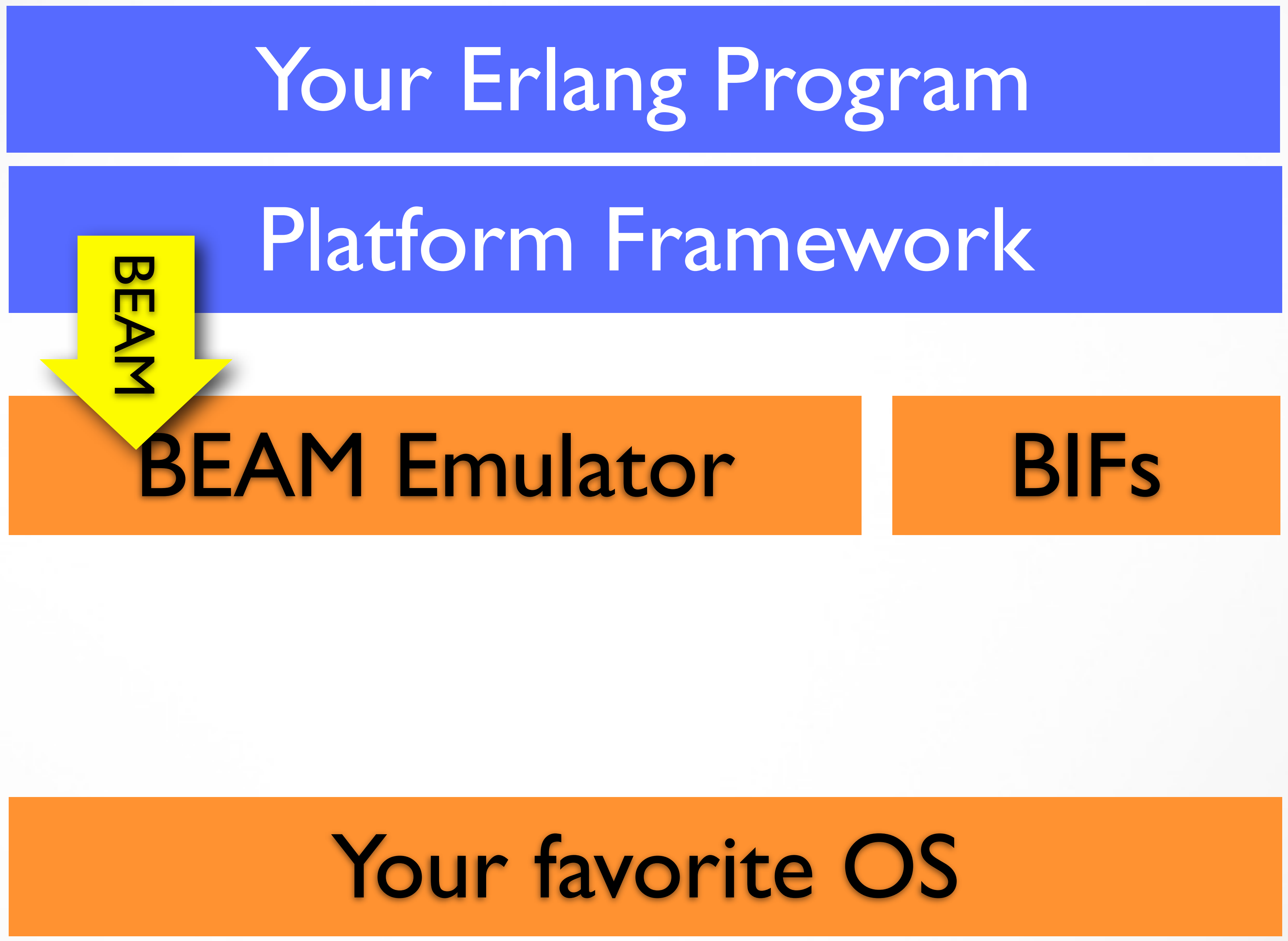
TRIFORK.

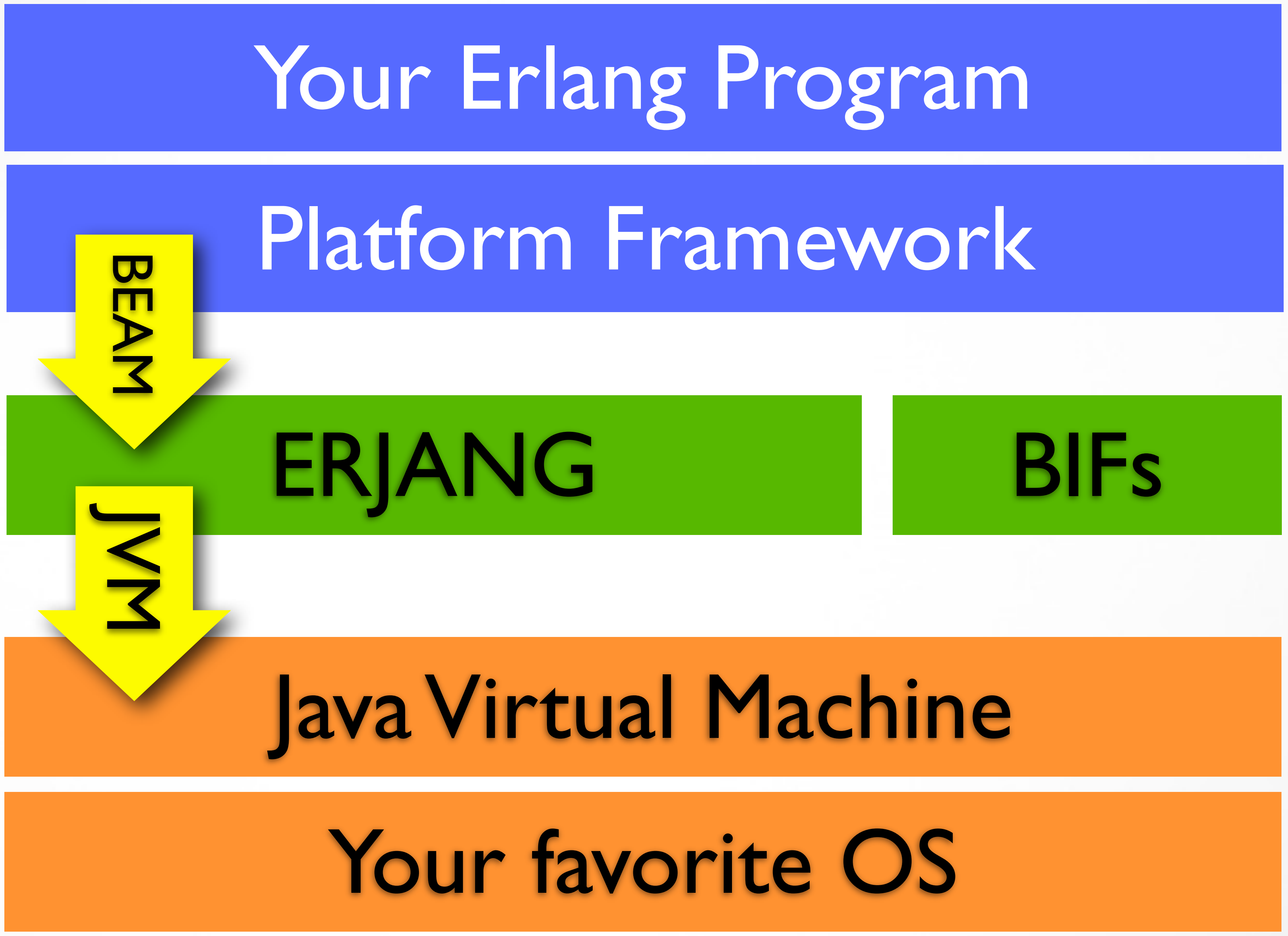
How to Learn Erlang

Dont dissect a frog,
Build One!

Nicolas Negroponte


Your Erlang Program
Platform Framework
BEAM
BEAM Emulator
BIFs
Your favorite OS


Your Erlang Program
Platform Framework
BEAM
BEAM Emulator
BIFs
Your favorite OS


Your Erlang Program
Platform Framework
BEAM
ERJANG
BIFs
JVM
Java Virtual Machine
Your favorite OS

Kresten Krab Thorup
Erlang Person of the Year, 2010

basho

OTP Actor Behaviors

- Servers
- Event Handlers
- Finite State Machine
- Supervisors
- Networking: TCP, HTTP

Tail Recursion

- Tail recursion is critical to programming Erlang processes
  - process enters a function
  - function acts on an incoming message
  - as last action in the function, it calls itself to handle the next message

Essence of OTP Behaviors

```
loop(State) ->
```

Essence of OTP Behaviors

```
loop(State) ->
    receive
        % handle messages here,
```

Essence of OTP Behaviors

```
loop(State) ->
    receive
        % handle messages here,
        % messages may affect State
    end,
```

Essence of OTP Behaviors

```
loop(State) ->
    NState = receive
                 % handle messages here,
                 % messages may affect State
             end,
```

Essence of OTP Behaviors

```
loop(State) ->
    NState = receive
                     % handle messages here,
                     % messages may affect State
                 end,
    loop(NState).
```

Essence of OTP Behaviors

```
loop(State) ->
    NState = receive
                    % handle messages here,
                    % messages may affect State
                end,
    loop(NState).
```

Essence of OTP Behaviors

```
loop(Callbacks, State) ->
    NState = receive
                   Msg ->
                       Callbacks:handle(Msg, State)
               end,
    loop(Callbacks, NState).
```

Essence of OTP Behaviors

```
loop(Callbacks, State) ->
    NState = receive
                    M1 ->
                        Callbacks:handle1(M1,State);
                    M2 ->
                        Callbacks:handle2(M2,State);
                    M3 ->
                        Callbacks:handle3(M3,State)
                end,
    loop(Callbacks, NState).
```

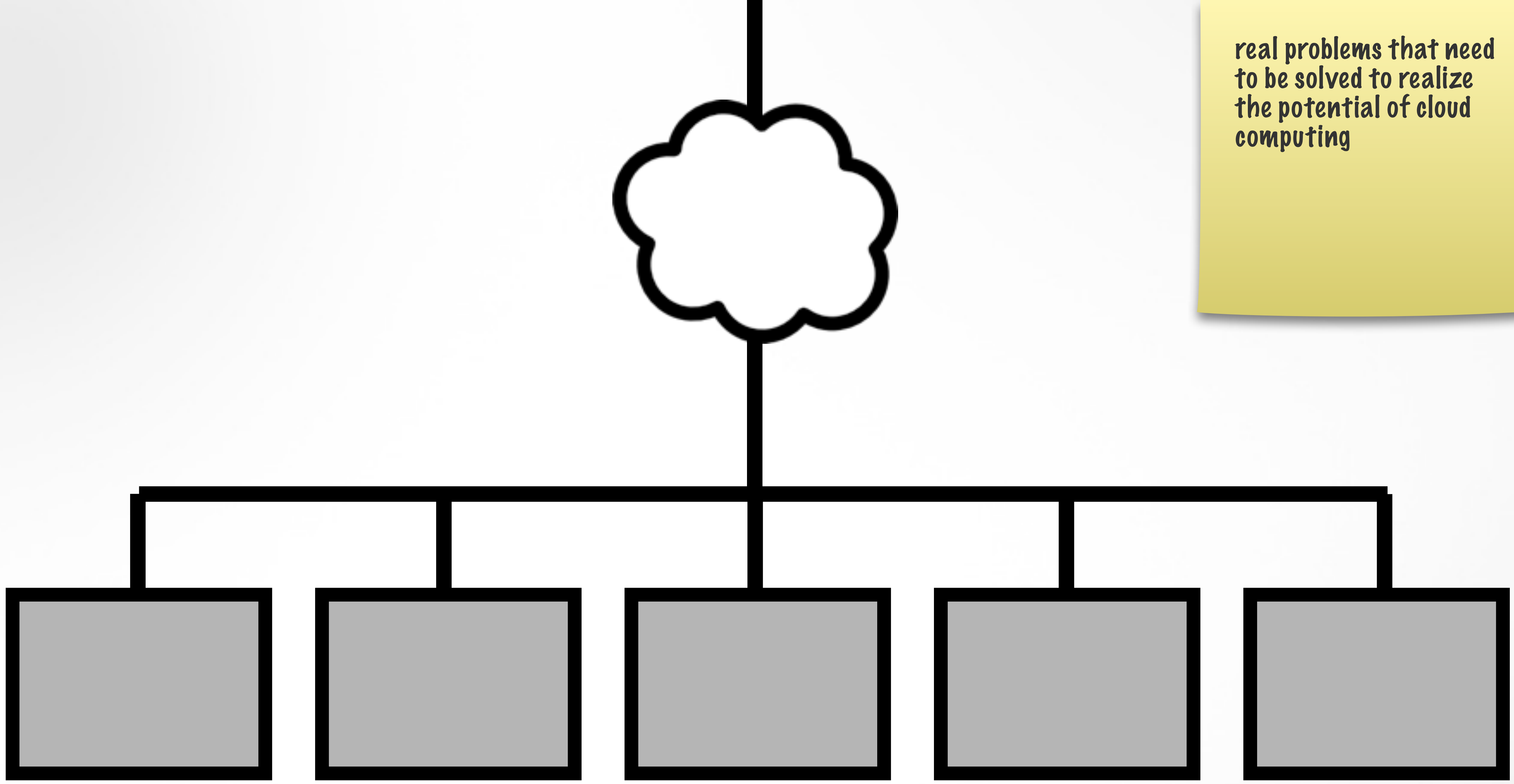
Essence of OTP Behaviors

```
loop(Callbacks, State) ->
    {Next, NState} =
                receive
                    M1 ->
                        Callbacks:handle_m1(M1,State);
                    M2 ->
                        Callbacks:handle_m2(M2,State);
                    M3 ->
                        Callbacks:handle_m3(M3,State)
                end,
    case Next of
        stop -> ok;
        _ -> loop(Callbacks, NState) end.
```

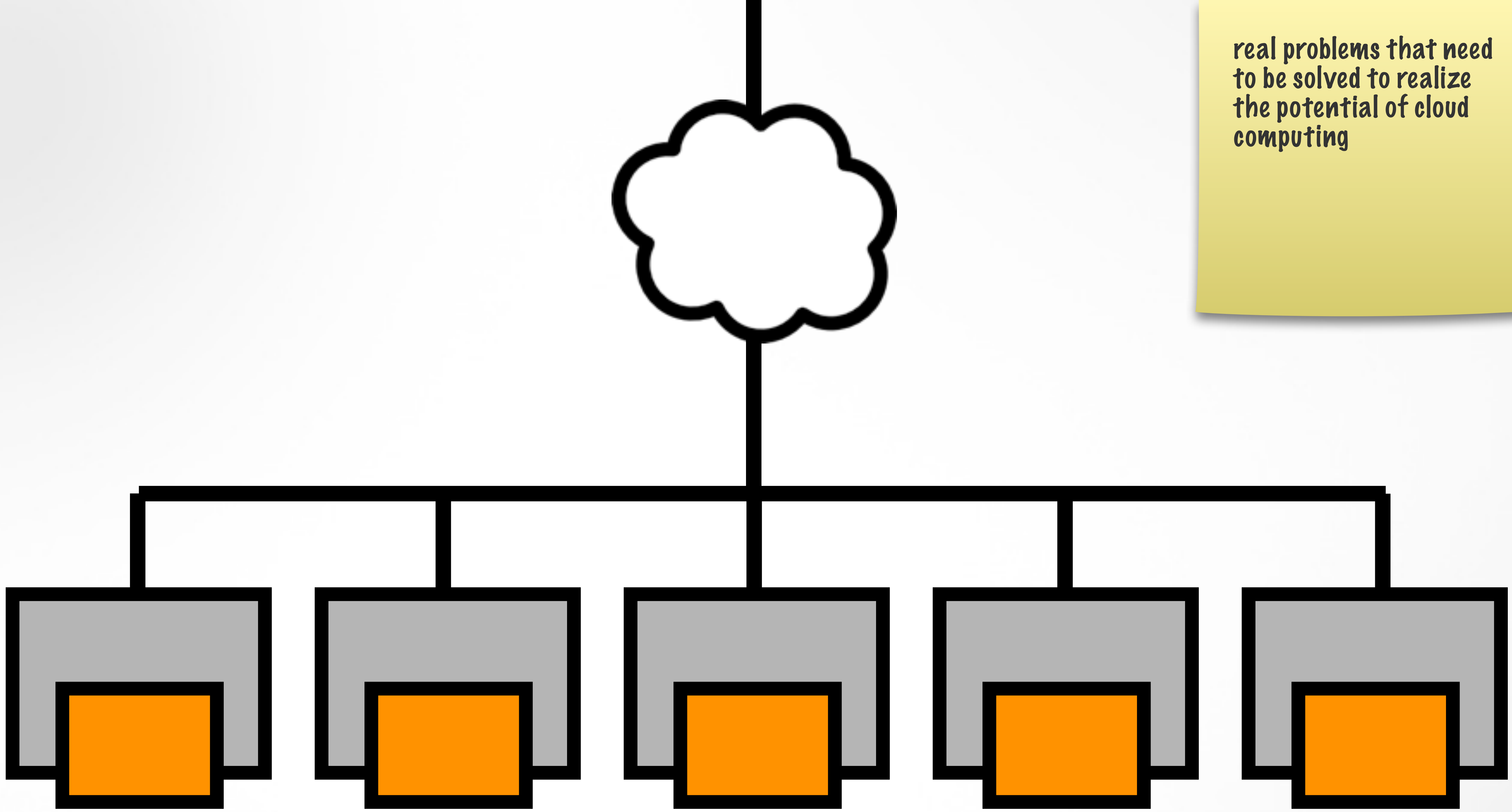
Behavior Loops

- These are the basics
- Actual OTP behavior loops are much more sophisticated
- gen_server, gen_fsm, supervisor, etc. all assume specific callback functions, checked by the compiler

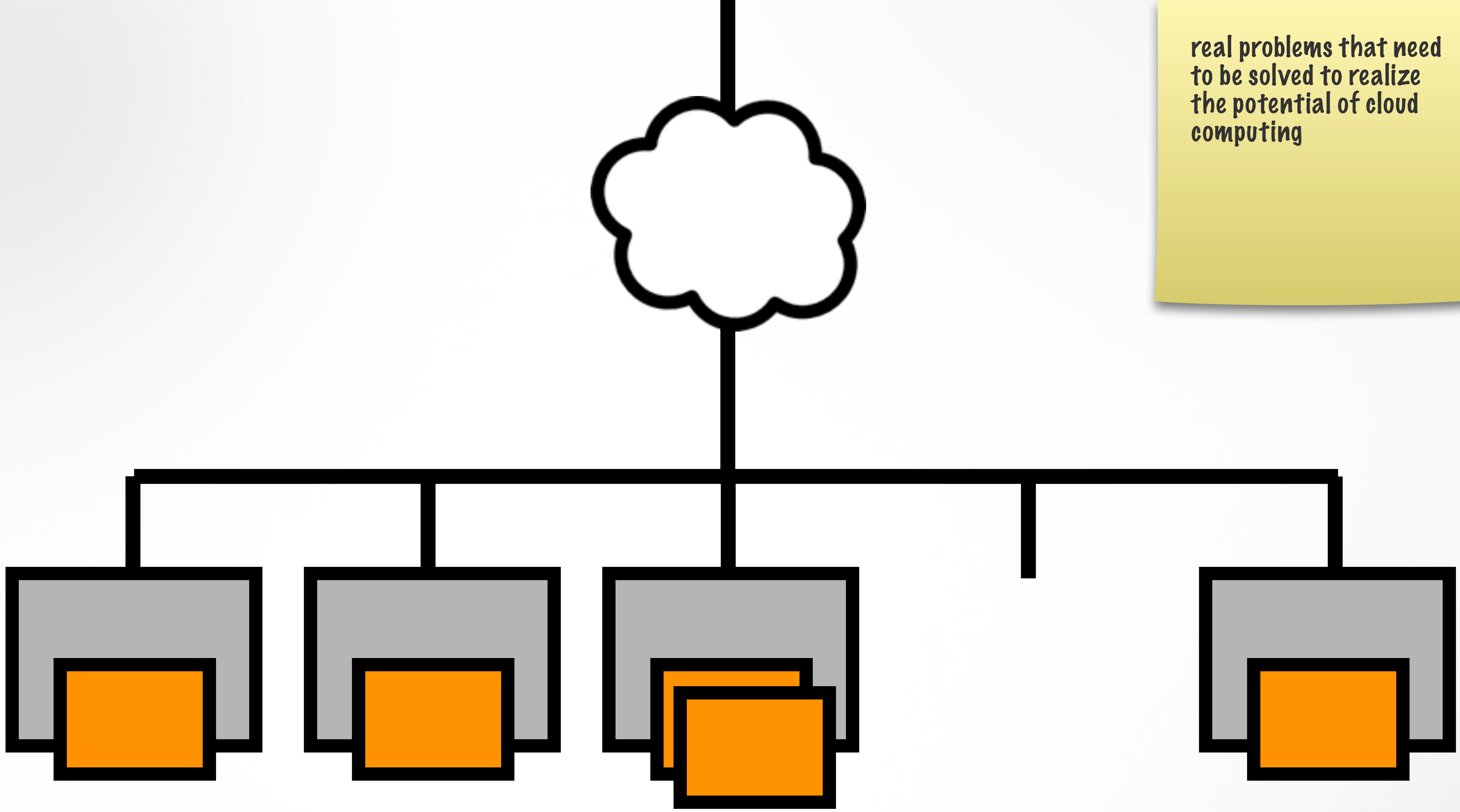
Erlang Systems



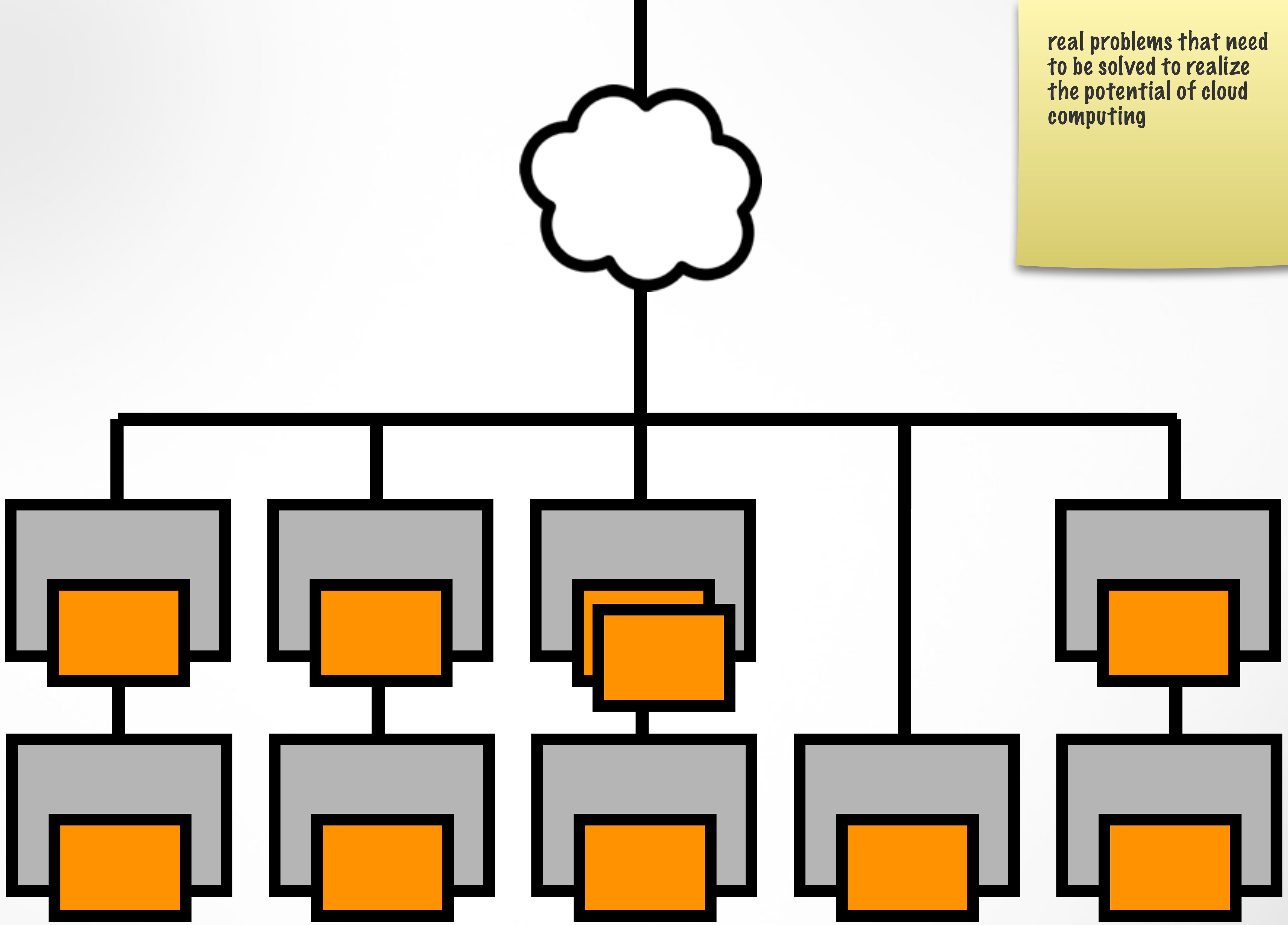
real problems that need to be solved to realize the potential of cloud computing



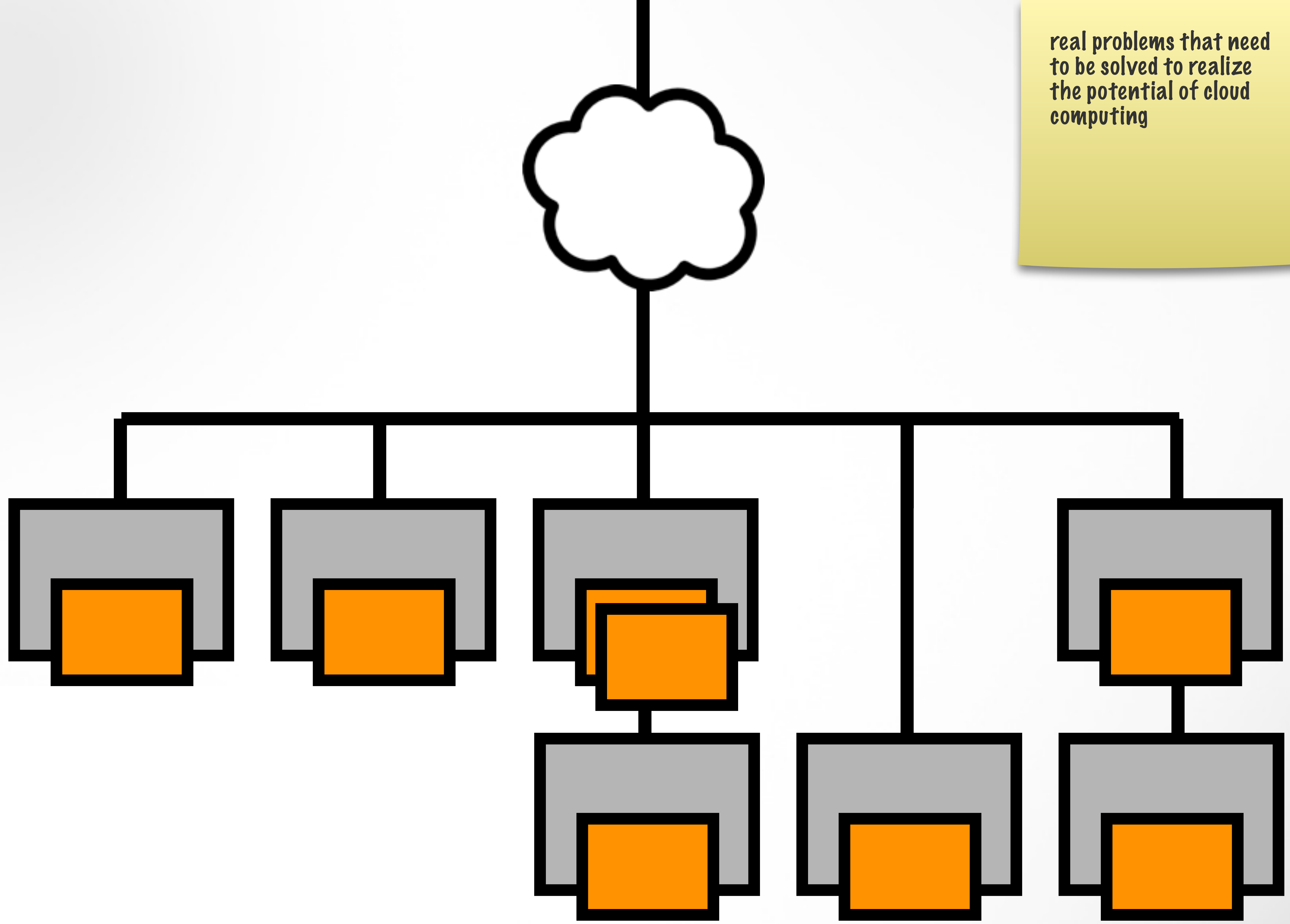
real problems that need to be solved to realize the potential of cloud computing



real problems that need to be solved to realize the potential of cloud computing

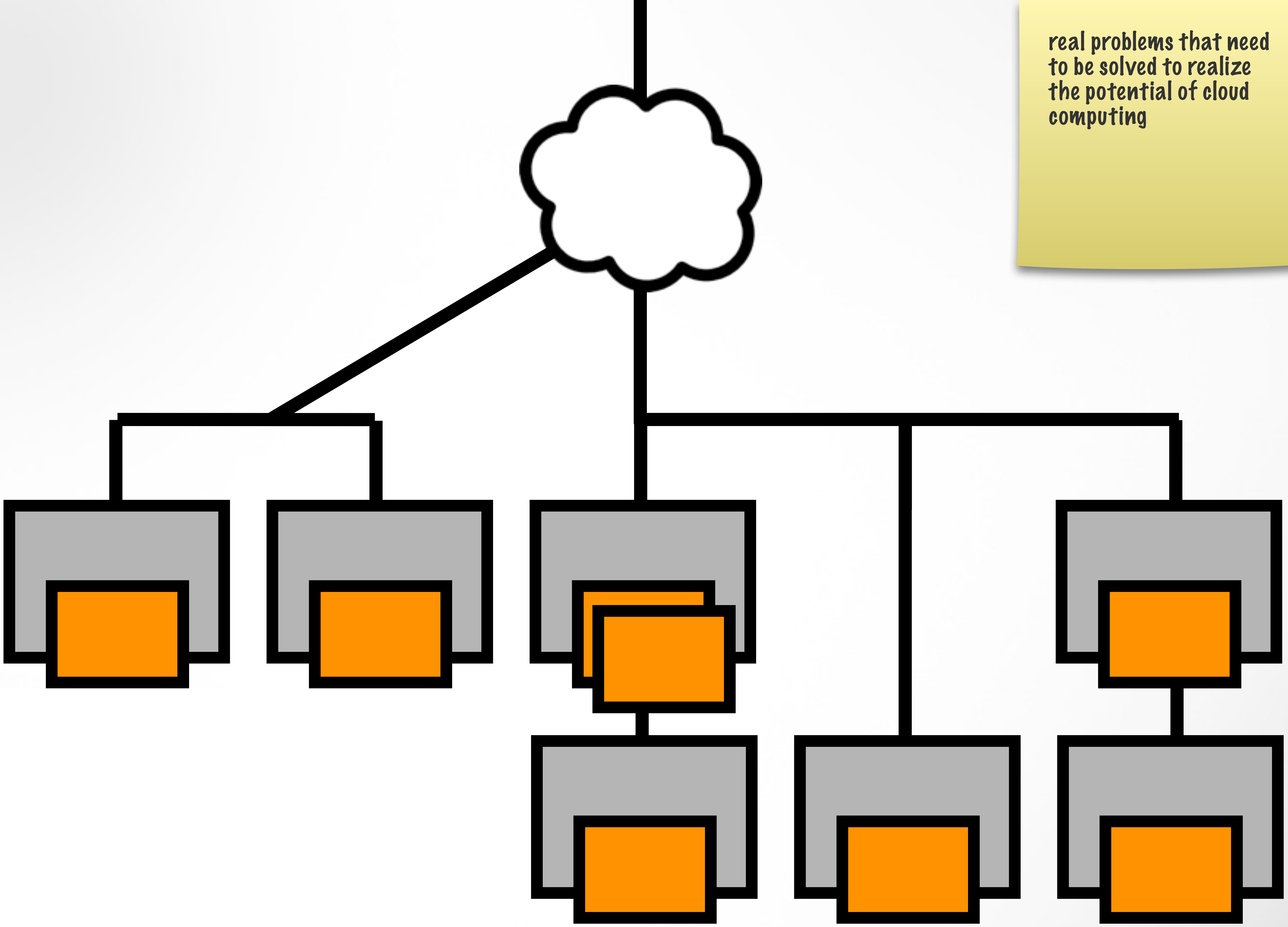


real problems that need to be solved to realize the potential of cloud computing

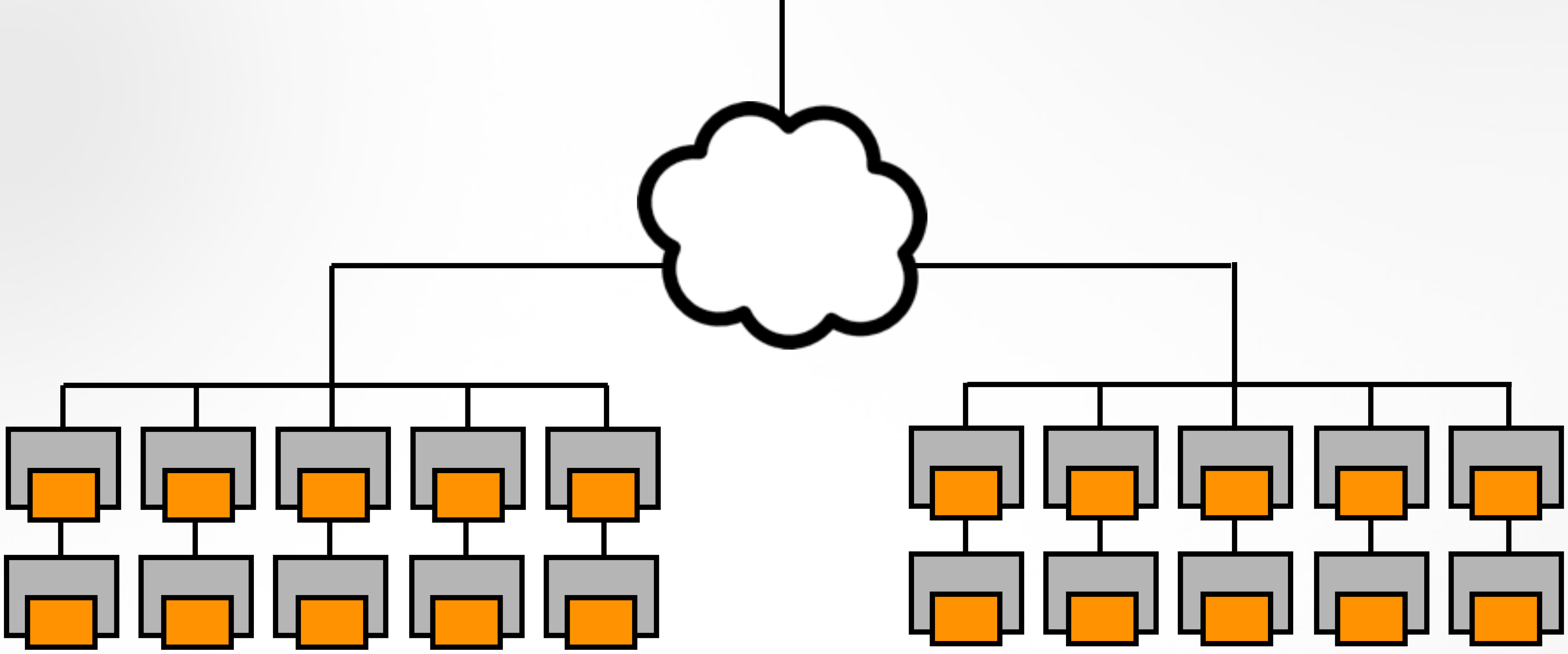


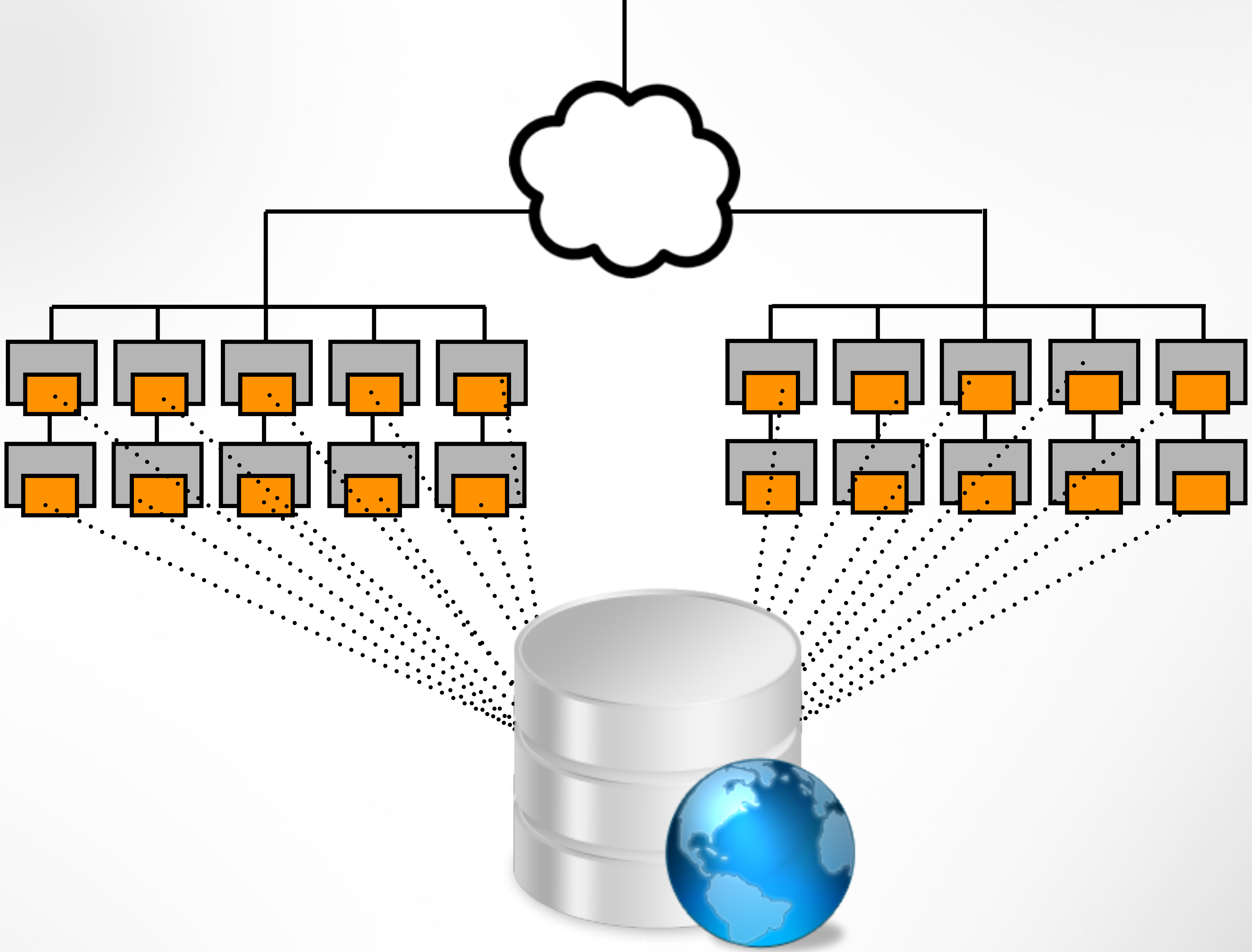
real problems that need to be solved to realize the potential of cloud computing

TRIFORK.

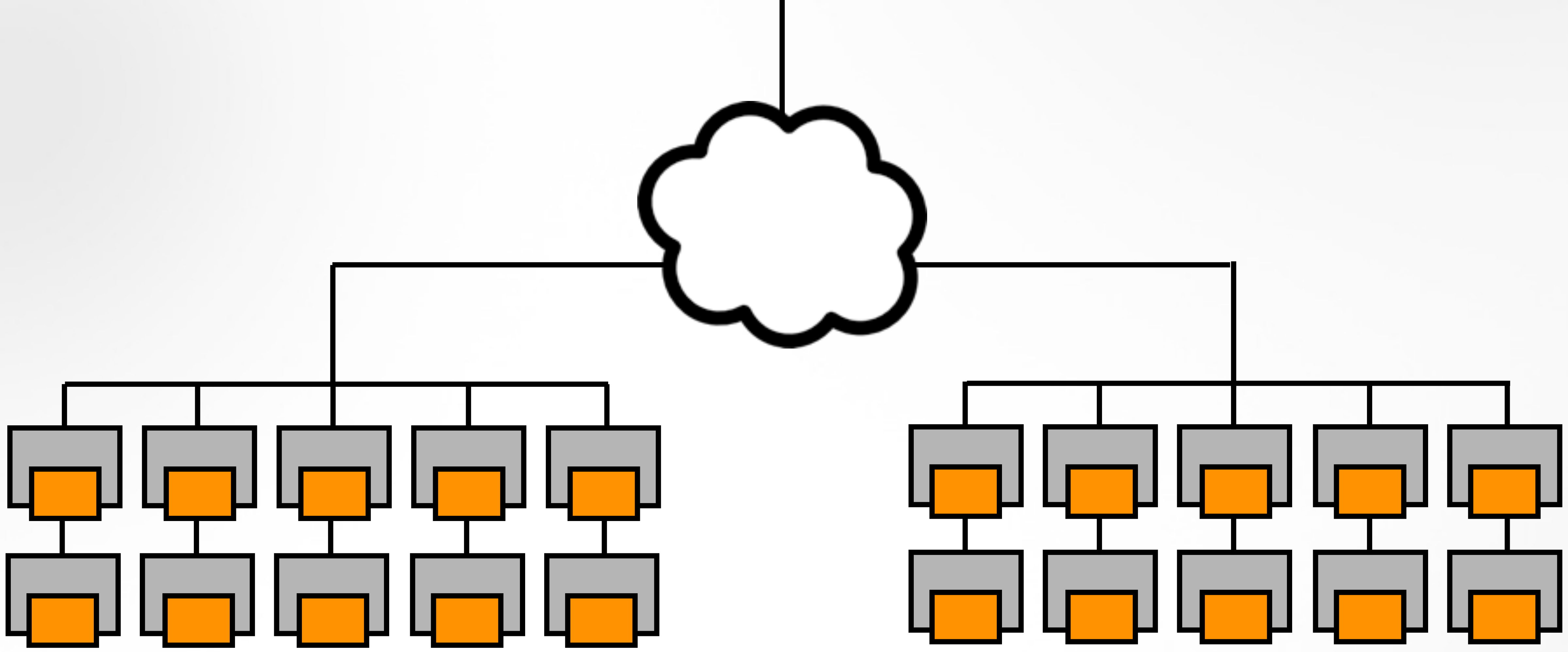


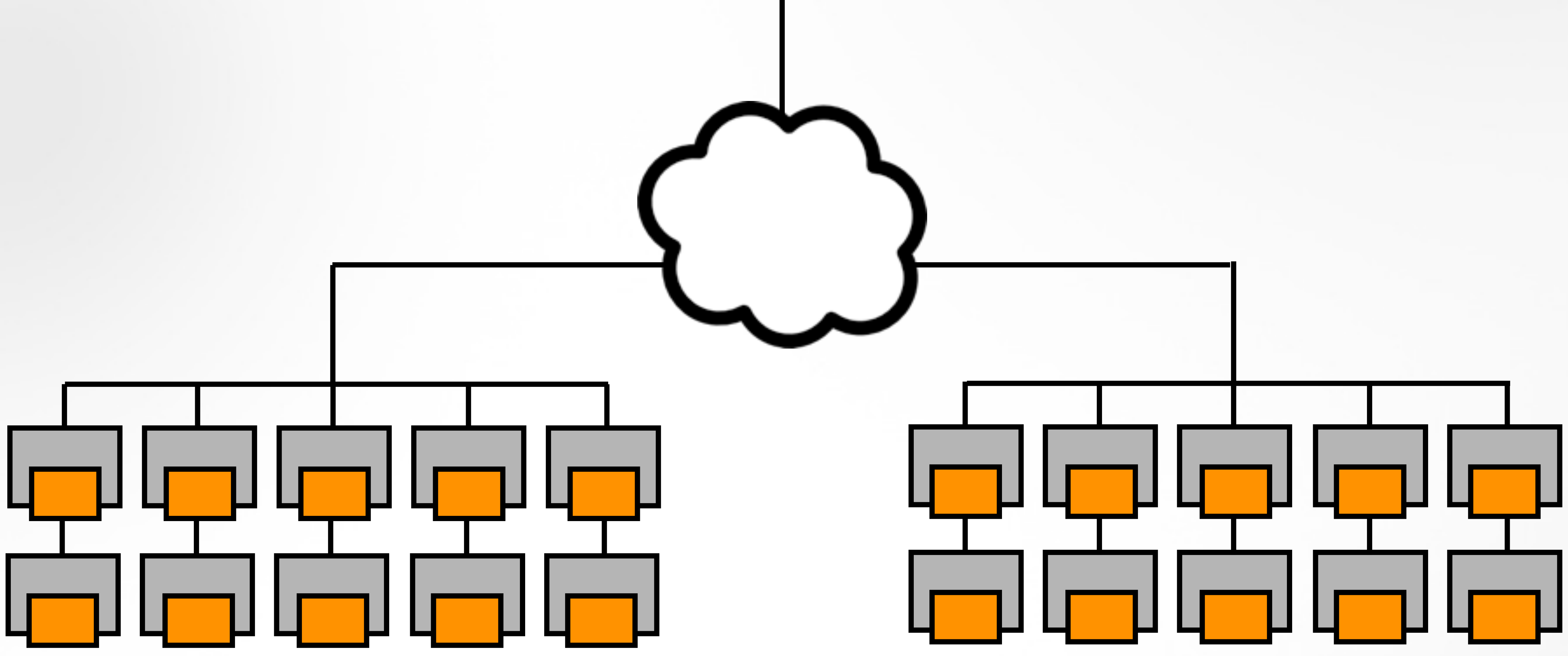
real problems that need to be solved to realize the potential of cloud computing



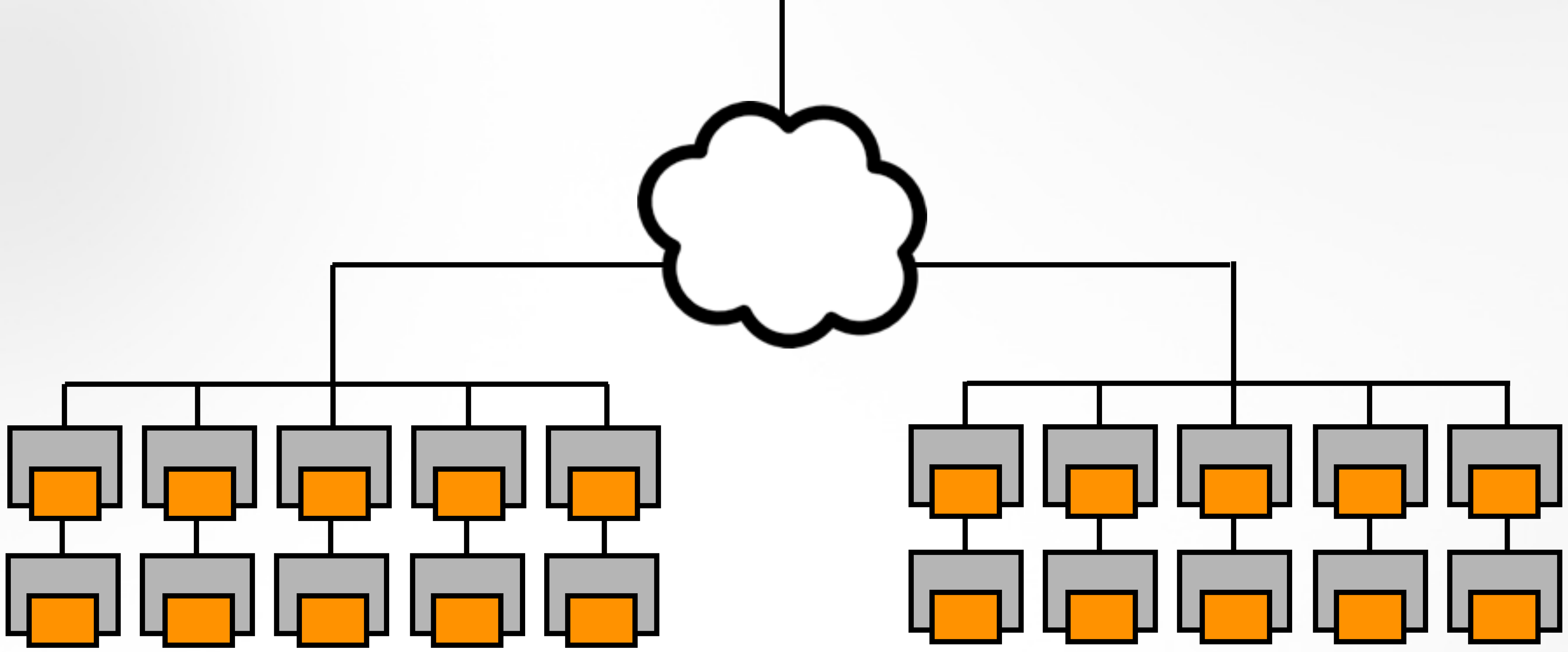


TRIFORK.





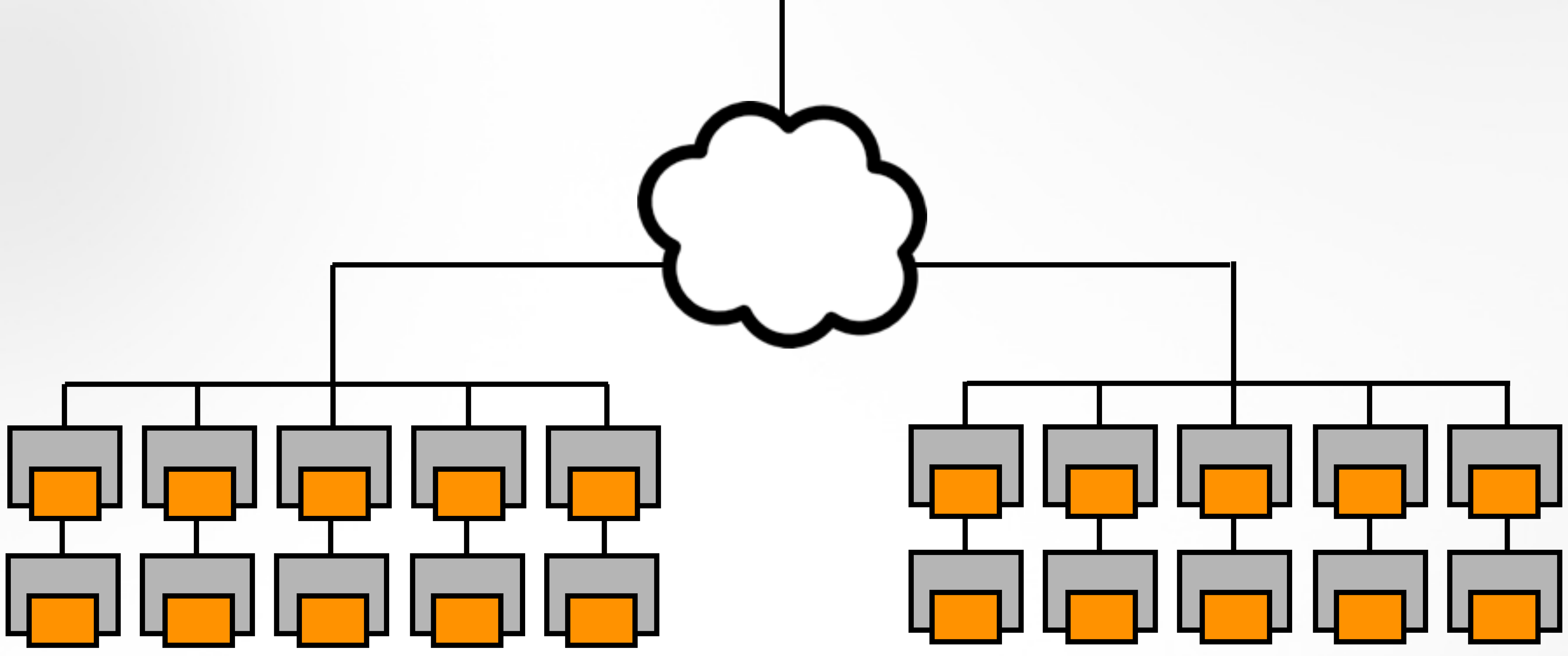
Async Messaging



Async Messaging

State [Fault] Containment

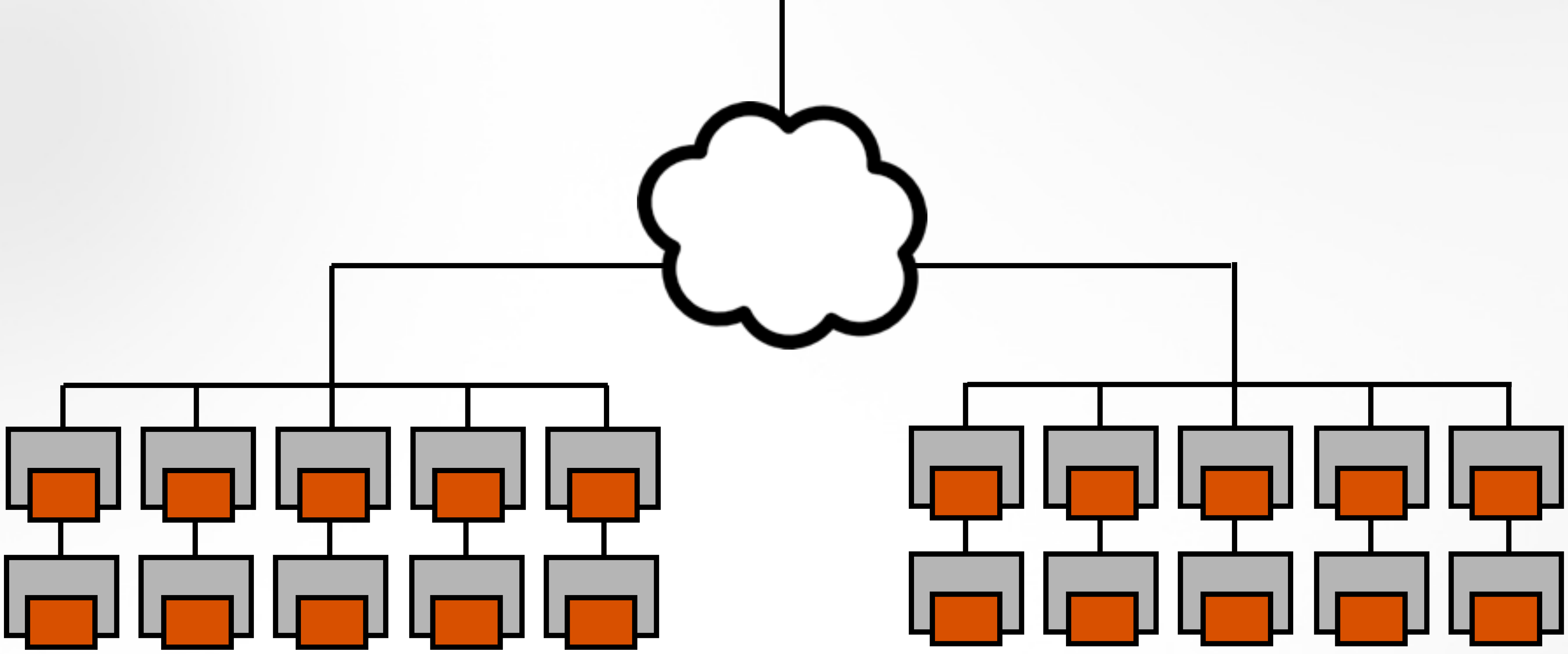
TRIFORK.



Async Messaging

State [Fault] Containment

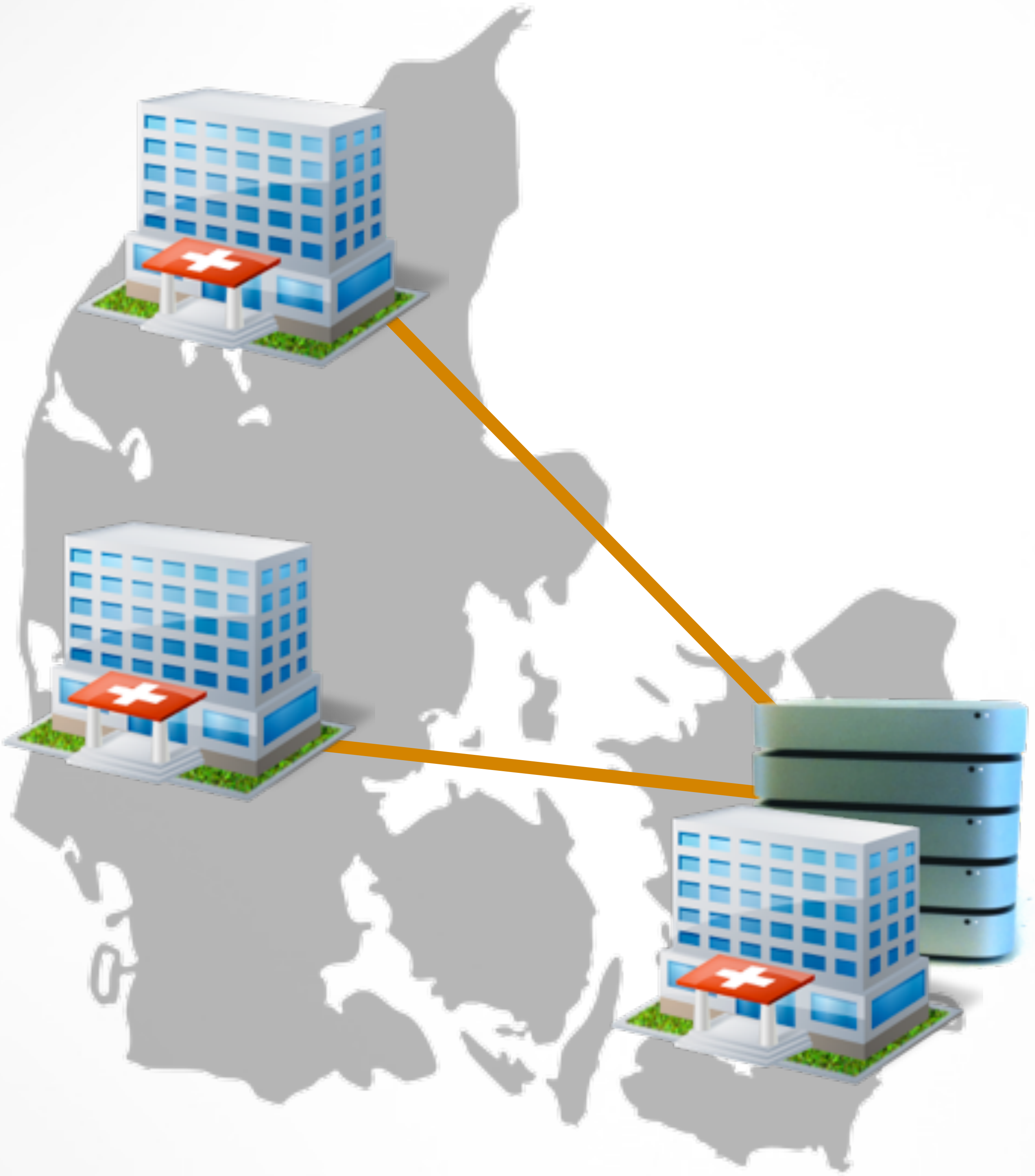
Fault Monitoring



riak

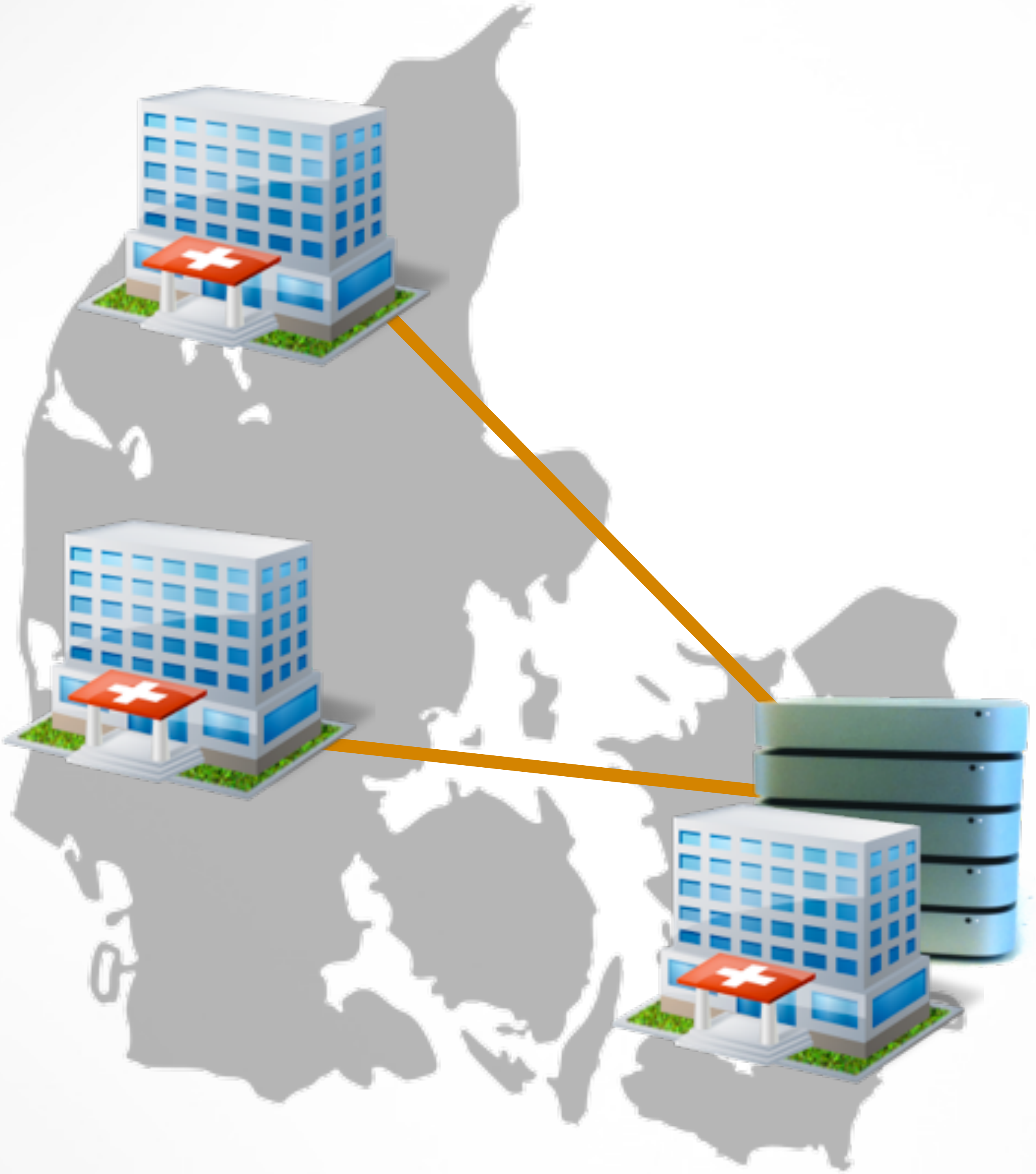
TRIFORK.

Common Medicine Card

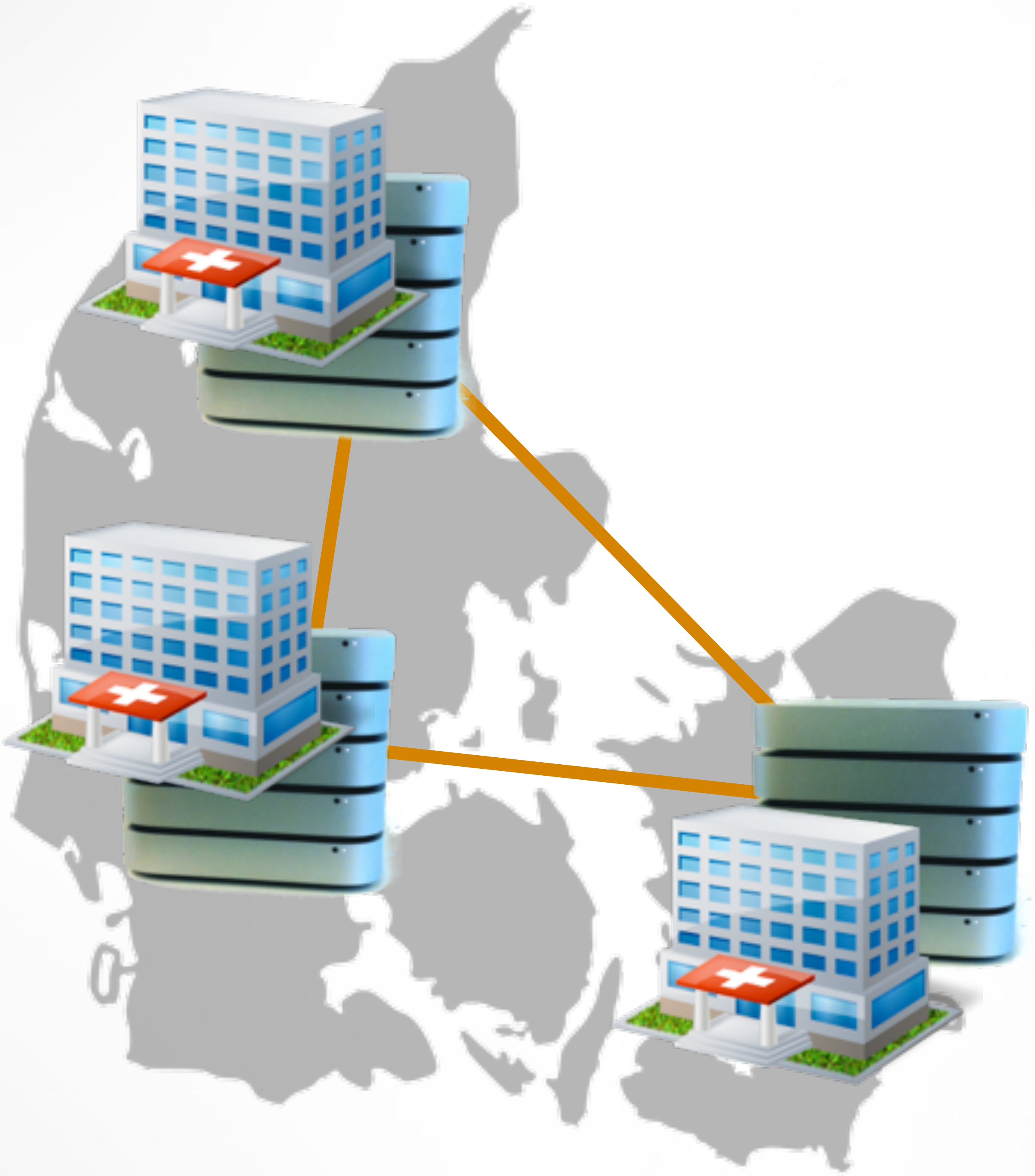


TRIFORK.

Common Medicine Card



Common Medicine Card



TRIFORK.

Who Else Uses Erlang?

facebook

ejabberd

riak

T · · Mobile · · ·

klarna

CouchDB
relax

RabbitMQ
Open Source Enterprise Messaging

Advice for New Erlang Users

Integration Using Erlang

Integration Using Erlang

- Integration often involves distribution

Integration Using Erlang

- Integration often involves distribution
- Dealing with data: bit syntax, built-in packet decoders (HTTP, FCGI, CDR)

Integration Using Erlang

- Integration often involves distribution
- Dealing with data: bit syntax, built-in packet decoders (HTTP, FCGI, CDR)
- Trivial access to TCP, UDP

Integration Using Erlang

- Integration often involves distribution
- Dealing with data: bit syntax, built-in packet decoders (HTTP, FCGI, CDR)
- Trivial access to TCP, UDP
- Sync or async, easy event handling

Integration Using Erlang

- Integration often involves distribution
- Dealing with data: bit syntax, built-in packet decoders (HTTP, FCGI, CDR)
- Trivial access to TCP, UDP
- Sync or async, easy event handling
- Application protocol handlers built using gen_server or gen_fsm

basho

Integration Using Erlang

- Often write little networked clients and servers directly in the erl shell
- Packet decoding and bit syntax sets Erlang apart from netcat, perl, etc. in this regard
- It’s like a middleware/coordination DSL

dbg and Tracing

- Erlang’s tracing is one of its most amazing features
- Learn the dbg module, you’ll use it every day
- I have needed the Erlang debugger only once, I always use dbg instead

Advice for New Users

- All that great stuff you've heard about Erlang? It's true
- Simple concurrency and coordination
- Hot code loading
- Always-available code tracing
- Sound, practical reliability
- Easy integration
- Enables "production prototypes"
- Open source at github.com
- Language and docs available at erlang.org

Warning: "Let It Crash"

- This philosophy can be hard for non-Erlangers to buy into
- QA sees a crash in the log, they treat it as something bad. Always.
  - explaining it was designed that way doesn't always fly
- Programmers new to Erlang (or sometimes not so new) always want to try to handle the errors instead

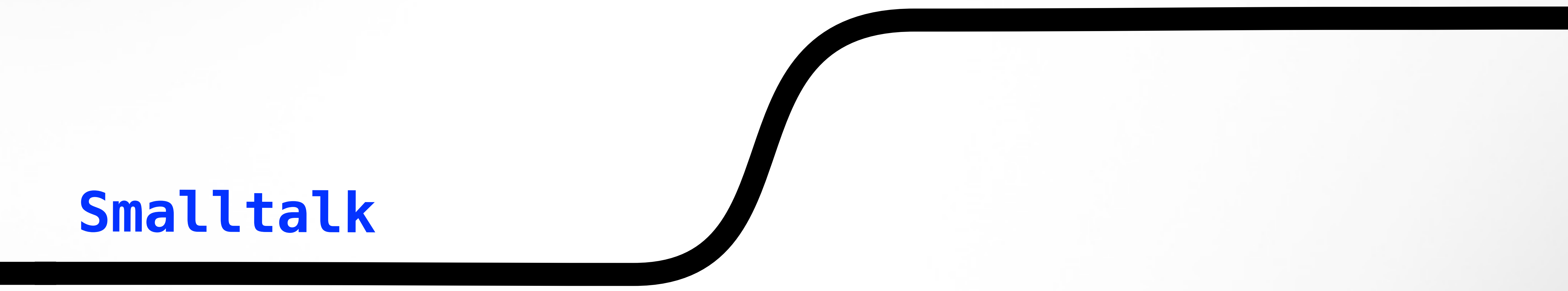
But “Let It Crash” Works

- Crash and recovery is invaluable for early adopter customers
- They keep using the system even if something goes wrong
- Most of the time, they’re unaware of the crash/recovery
- With dbg and hot code loading, you can debug and repair live systems

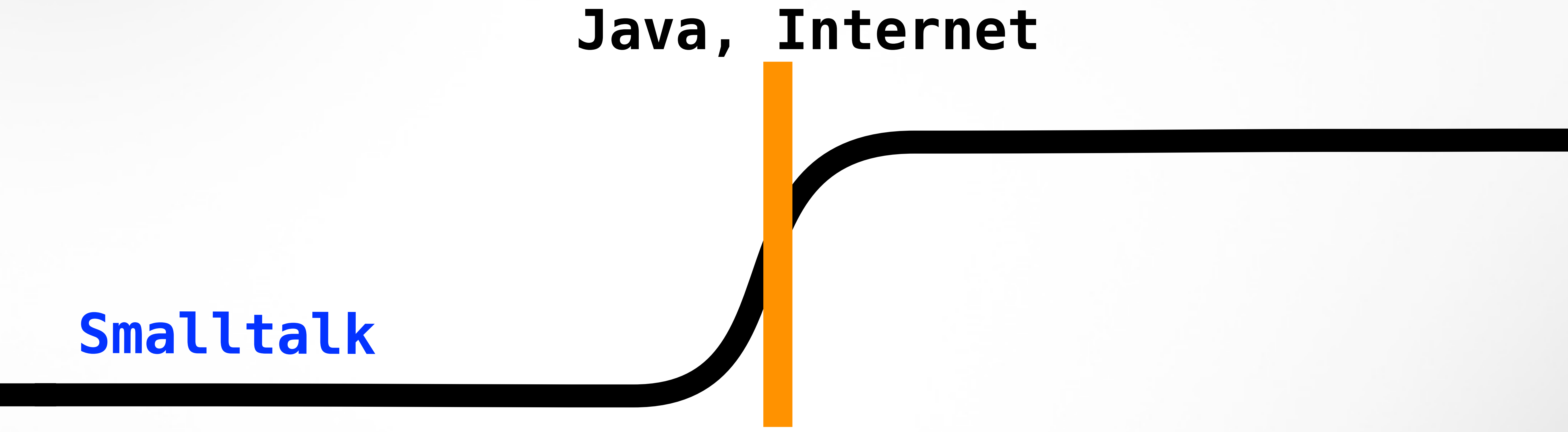
But Wait! There's More

- extensive library of useful modules
- details of OTP applications, supervision, code upgrade, hot code loading
- ets and dets: Erlang Term Storage (in memory) and Disk ets (persistent)
- mnesia: distributed transactional database
- rebar: open source build and dependency management system

Revolution?

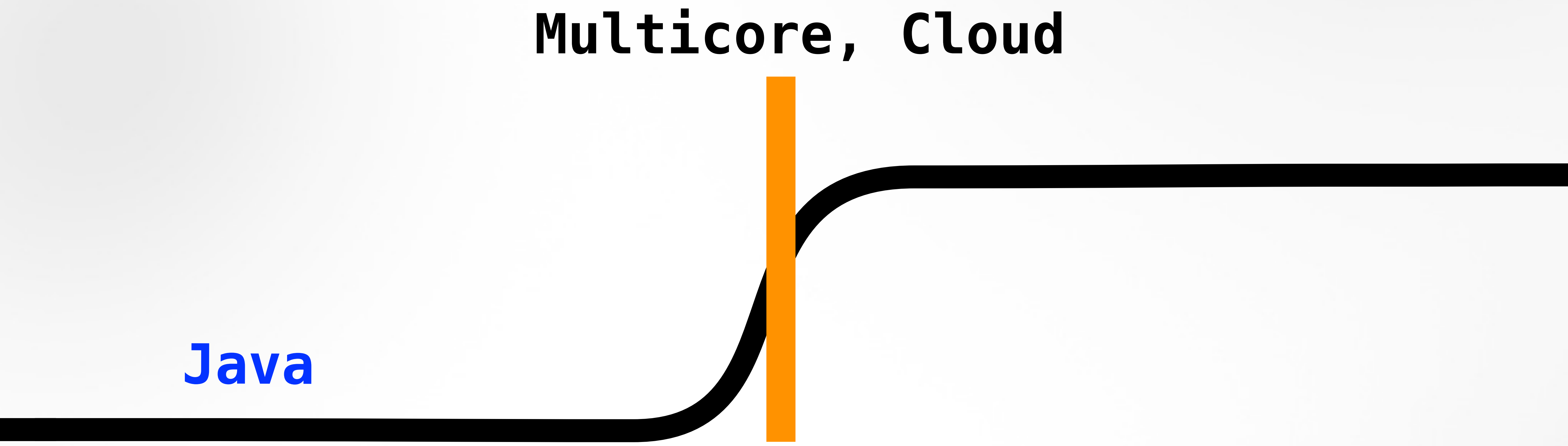


Revolution?

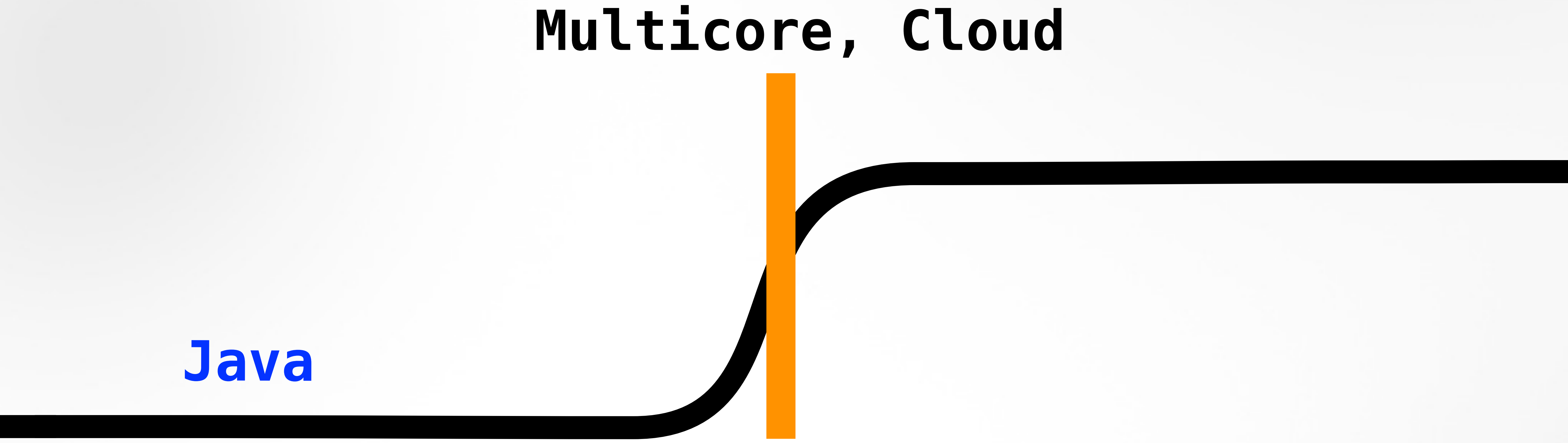


Java

Multicore, Cloud



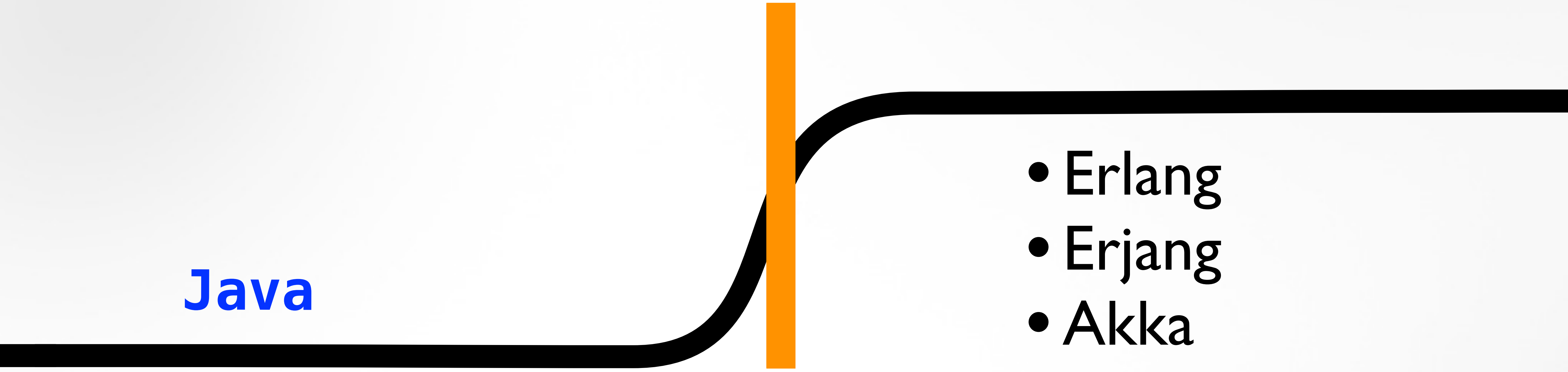
- Thread & Locks
- Interfaces with Fixed API
- Defensive Code
- RPC/RMI
- Boilerplate code for persistence



- Thread & Locks
- Interfaces with Fixed API
- Defensive Code
- RPC/RMI
- Boilerplate code for persistence

Anomalies

Multicore, Cloud, Actors

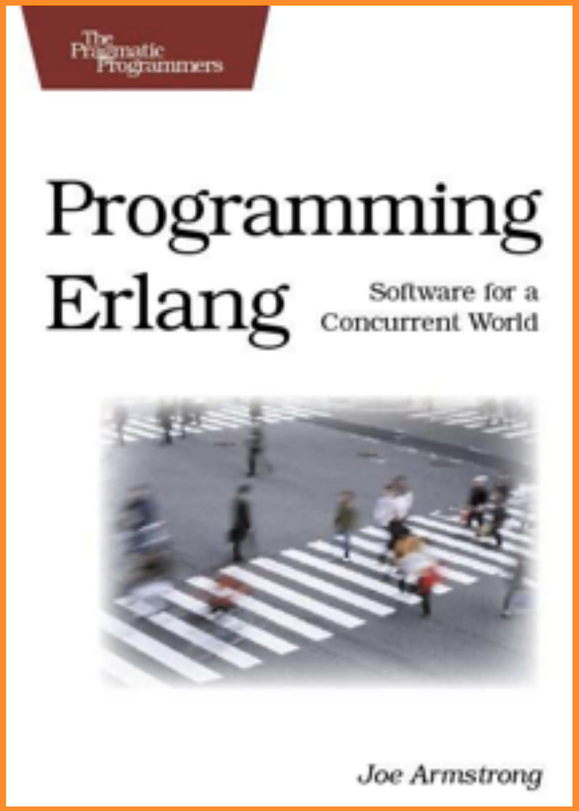


- Erlang
- Erjang
- Akka

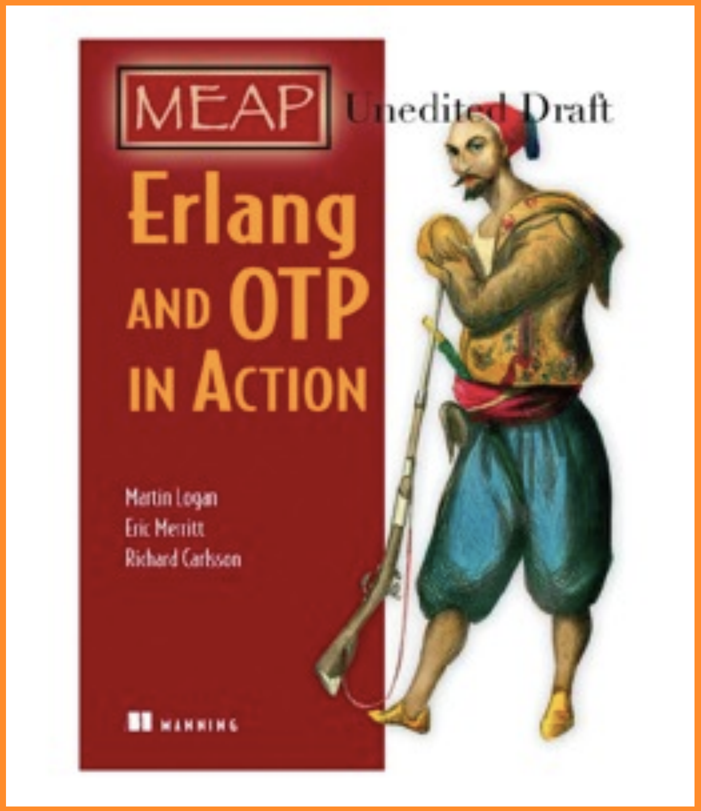
- Processes w/ state containment
- Protocols
- Let it Fail
- Async Messaging
- Send & store simple Data

TRIFORK.

Read These



A Concurrent Approach to Software Development
Erlang Programming
O'REILLY
Francesco Cesarini & Simon Thompson



Also: http://learnyousomeerlang.com/

Thanks