Dean Wampler
dean.wampler@typesafe.com
@deanwampler
polyglotprogramming.com/talks

Typesafe

Reactive Design, Languages, & Paradigms

LambdaJam
Chicago
July 22, 2014

Monday, July 21, 14

Image: Gateway Arch, St. Louis, Missouri, USA.
Special thanks to my fellow speakers at ReactConf 2014 for great feedback on the original version of this talk.

Four Traits of Reactive Programming

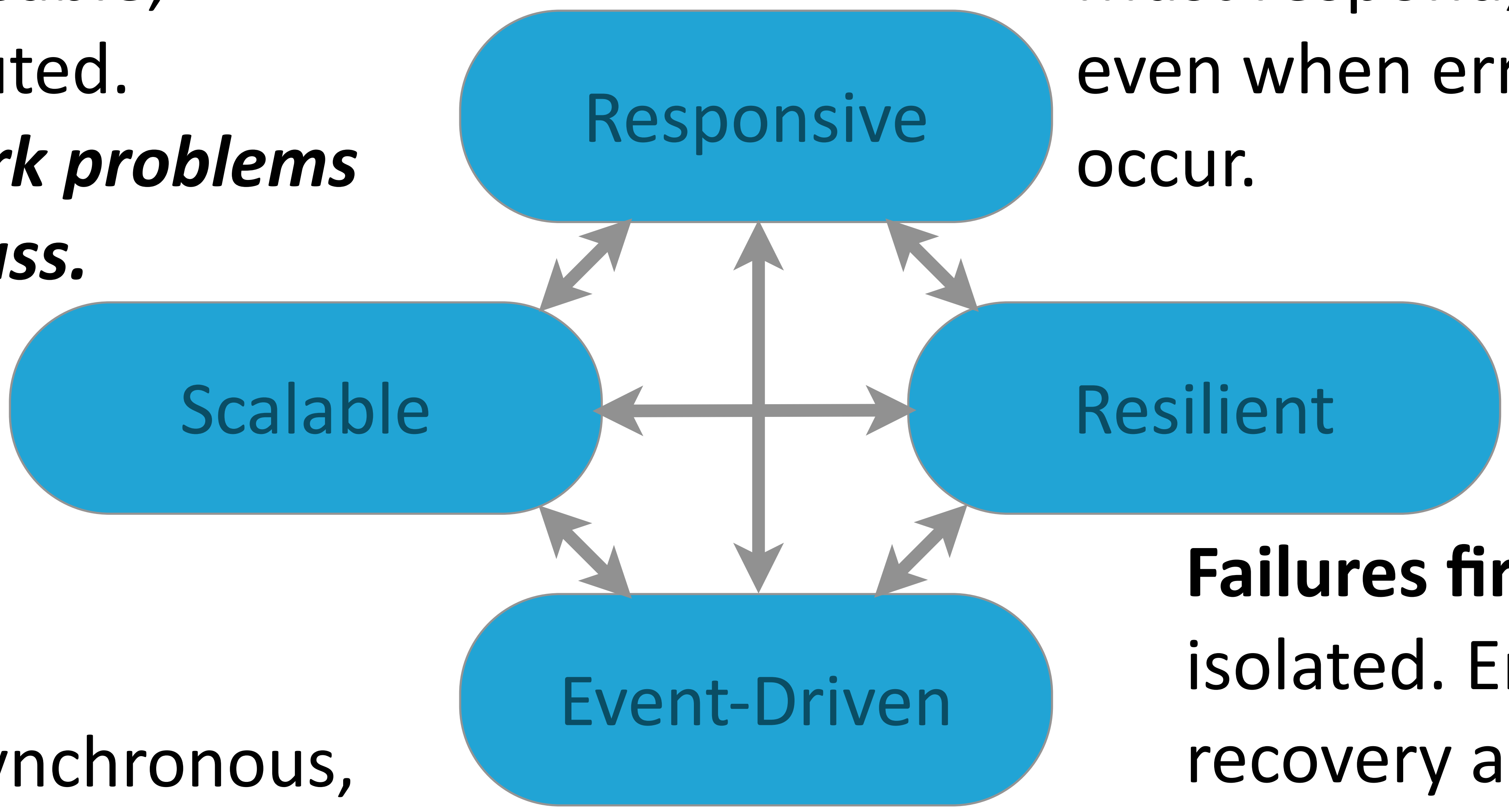
reactivemanifesto.org

Monday, July 21, 14

Photo: Foggy day in Chicago.

Loosely coupled, composable, distributed. ***Network problems first-class.***

Must respond, even when errors occur.



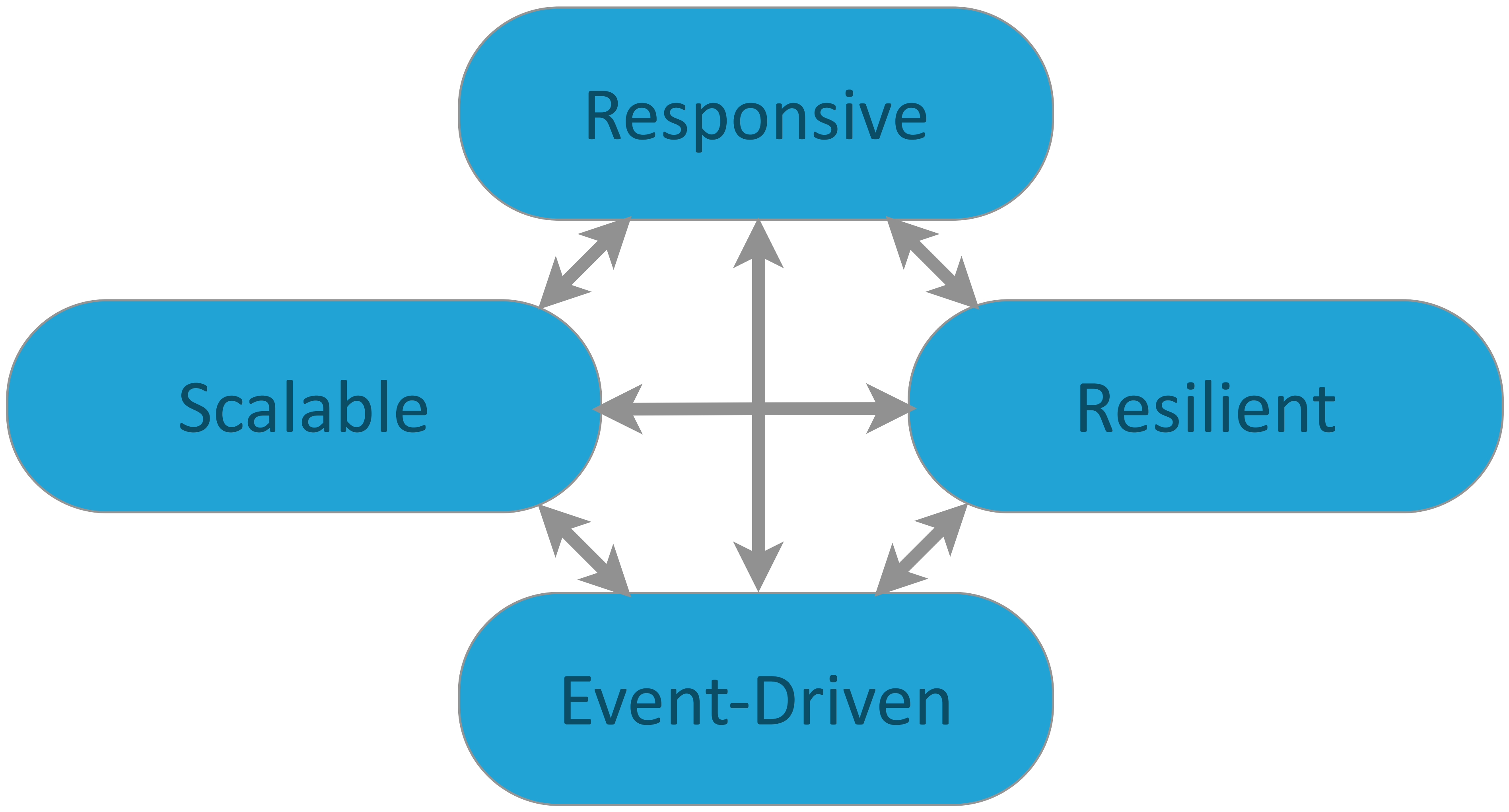
Failures first-class, isolated. Errors/ recovery are just other events.

Asynchronous, non-blocking. Facts as events are pushed.

reactivemanifesto.org

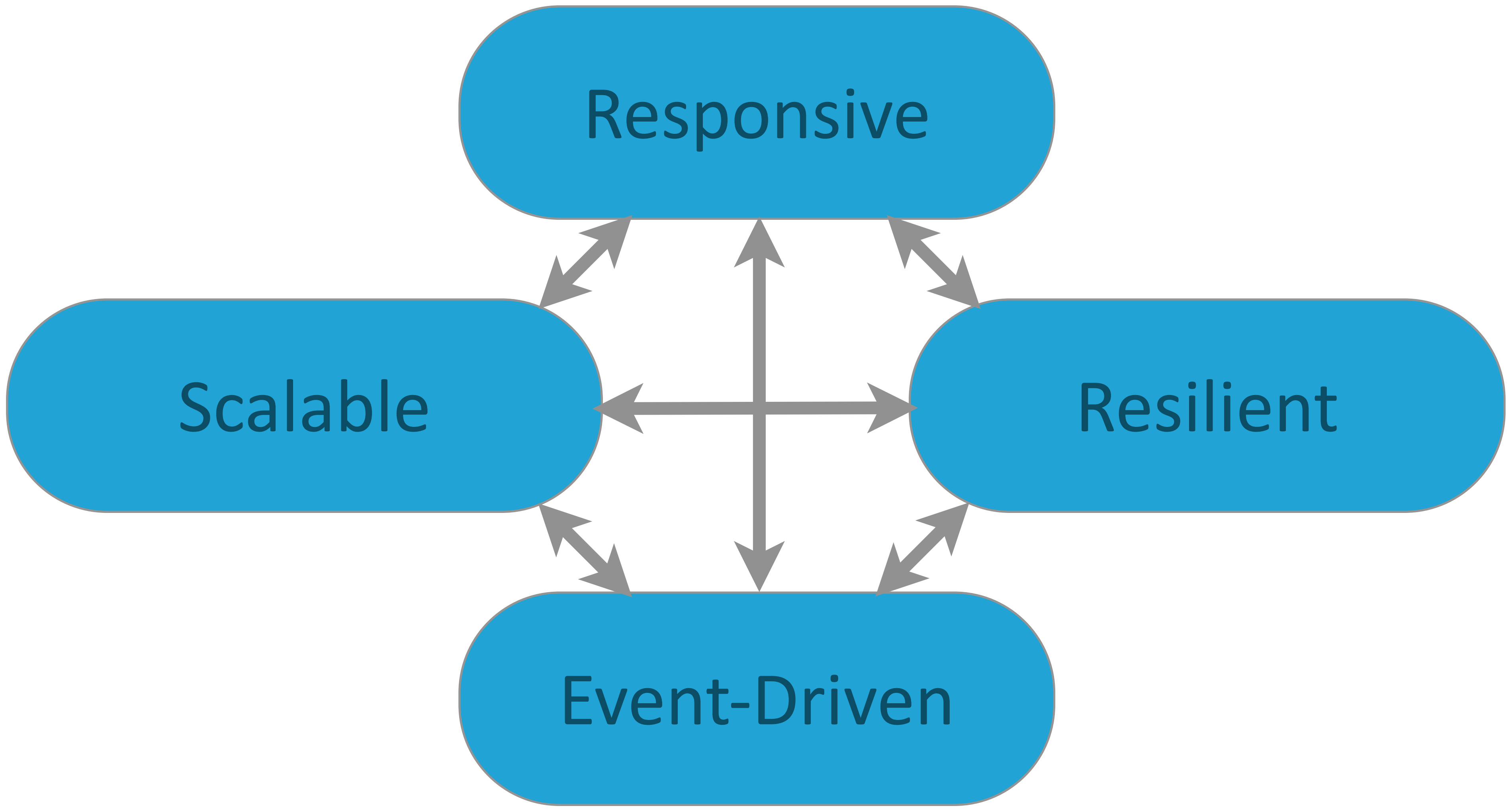
Monday, July 21, 14

The bonus slides walk through the traits in more details.



We'll use this graphic to assess how well different "paradigms" and tools support these traits.

The bonus slides walk through the traits in more details.

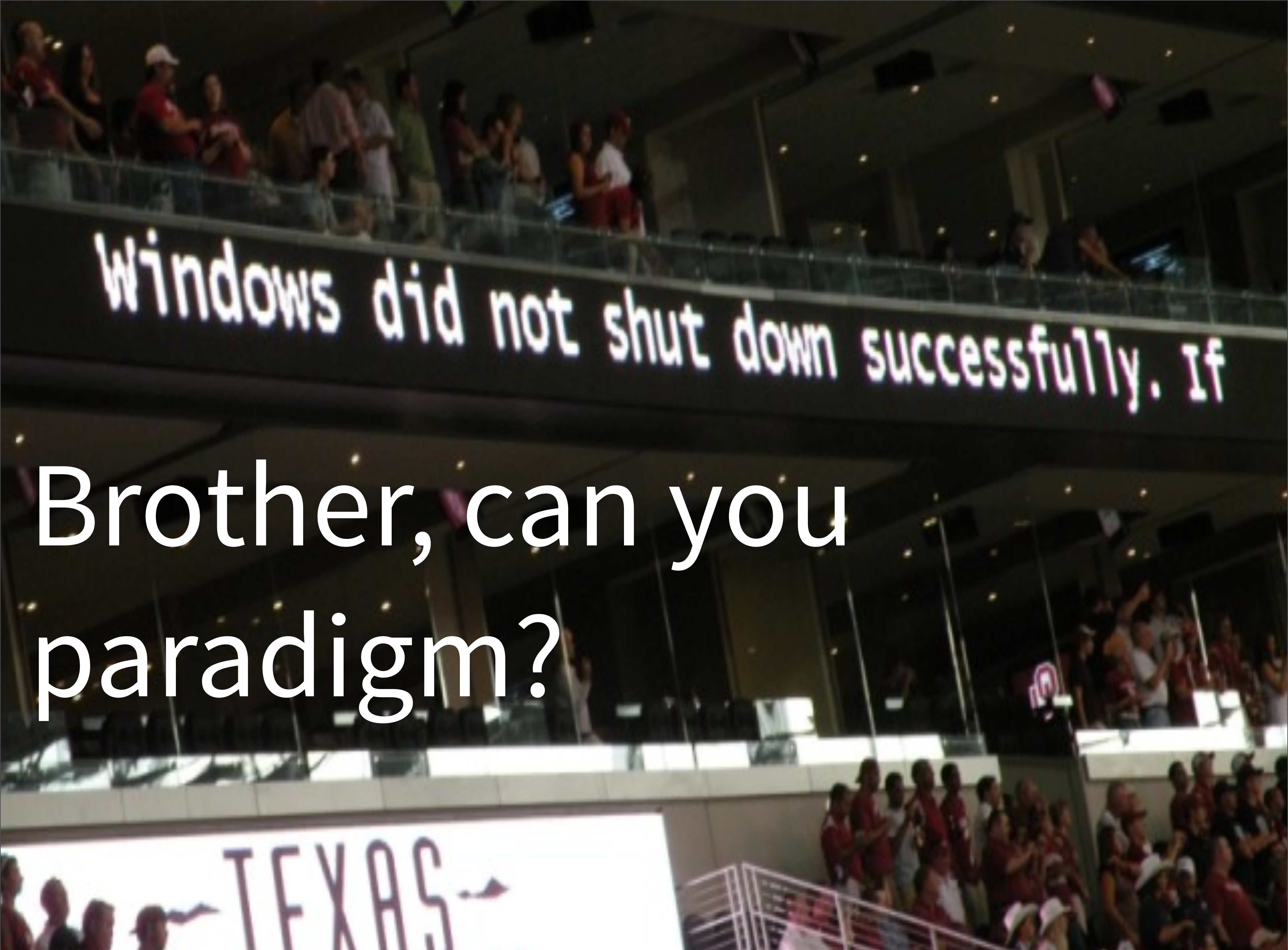


Color code:

Good
Missing Pieces
Problematic

Monday, July 21, 14

The bonus slides walk through the traits in more details.



Brother, can you paradigm?

Let’s start with programming paradigms. How well do they promote reactive programming??

Dallas football stadium (credit: unknown)

OOP

Photo: Frank Gehry-designed apartment complex in Dusseldorf, Germany.

State Mutation Is Good

Preferred over immutability, which requires constructing new objects. FP libraries, which prefer immutable values, have worked hard to implement efficient algorithms for making copies. Most OOP libraries are very inefficient at making copies, making state mutation important for performance. Hence, in typically OOP languages, even “good-enough” performance may require mutating state.
However, “unprincipled” mutable state is a major source of bugs, often hard to find, “action at a distance” bugs. See next slide.
Event-driven: Supports events and state changes well.
Scalable: Mutating state can be very fast, but don’t overlook the overhead of lock logic. Use a lock-free datastructure if you can.
Responsive: Faster performance helps responsiveness, but not if bugs dues to mutation occur.
Resilient: Unprincipled mutation makes the code inherently less resilient.

State and Behavior Are Joined

Joined in objects. Contrast with FP that separates state (values) and behavior (functions).
Event-driven: A benefit because events make natural objects, but event-handling logic can be obscured by object boundaries.
Scalability: Bad, due to the tendency to over-engineer the implementation by implementing more of the domain model than absolute necessary. This makes it harder to partition the program into “microservices”, limiting scalability. For high-throughput systems, instantiating objects for each “record” can be expensive. Arrays of “columns” are better if a lot of “records” are involved per event (or batches of events).
Responsive: Any code bloat and implementation logic scattered across class boundaries slows down the performance, possibly obscures bugs, and thereby harms responsiveness.
Resilient: Harder to reify Error handling, since it is a cross-cutting concern that cuts across domain object boundaries. Scattered logic (across object boundaries) and state mutation make bugs more likely.

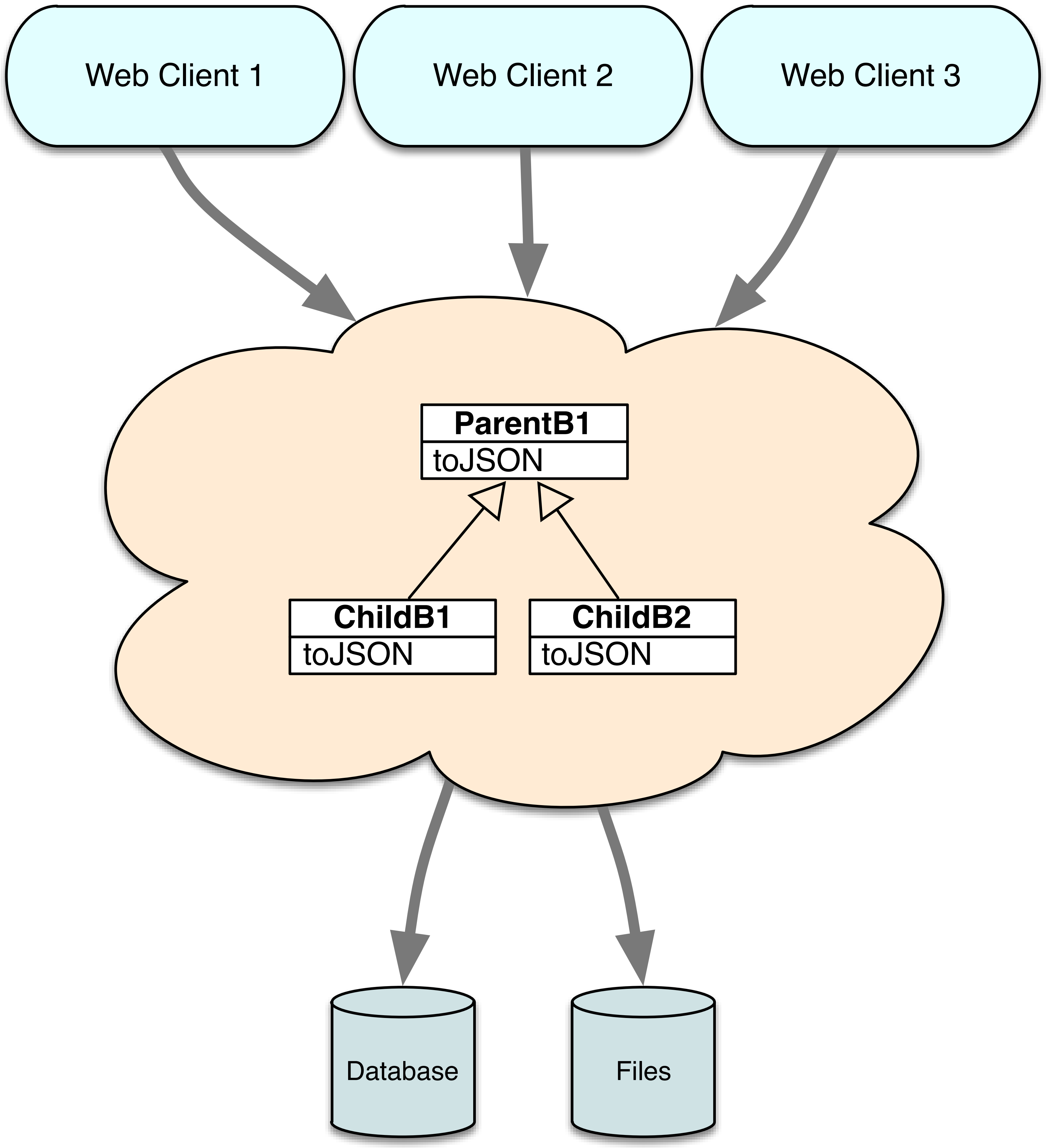
Claim: OOP's biggest mistake: believing you should *implement* your domain model.

This leads to ad-hoc classes in your code that do very little beyond wrap more fundamental types, primitives and collections. They spread the logic of each user story (or use case, if you prefer) across class boundaries, rather than put it one place, where it's easier to read, analyze, and refactor. They put too much information in the code, beyond the "need to know" amount of code. This leads to bloated applications that are hard to refactor in to separate, microservices. They take up more space in memory, etc.
The ad-hoc classes also undermined reuse, paradoxically, because each invents its own "standards". More fundamental protocols are needed.

Example: What should be in a Customer class?

Monday, July 21, 14

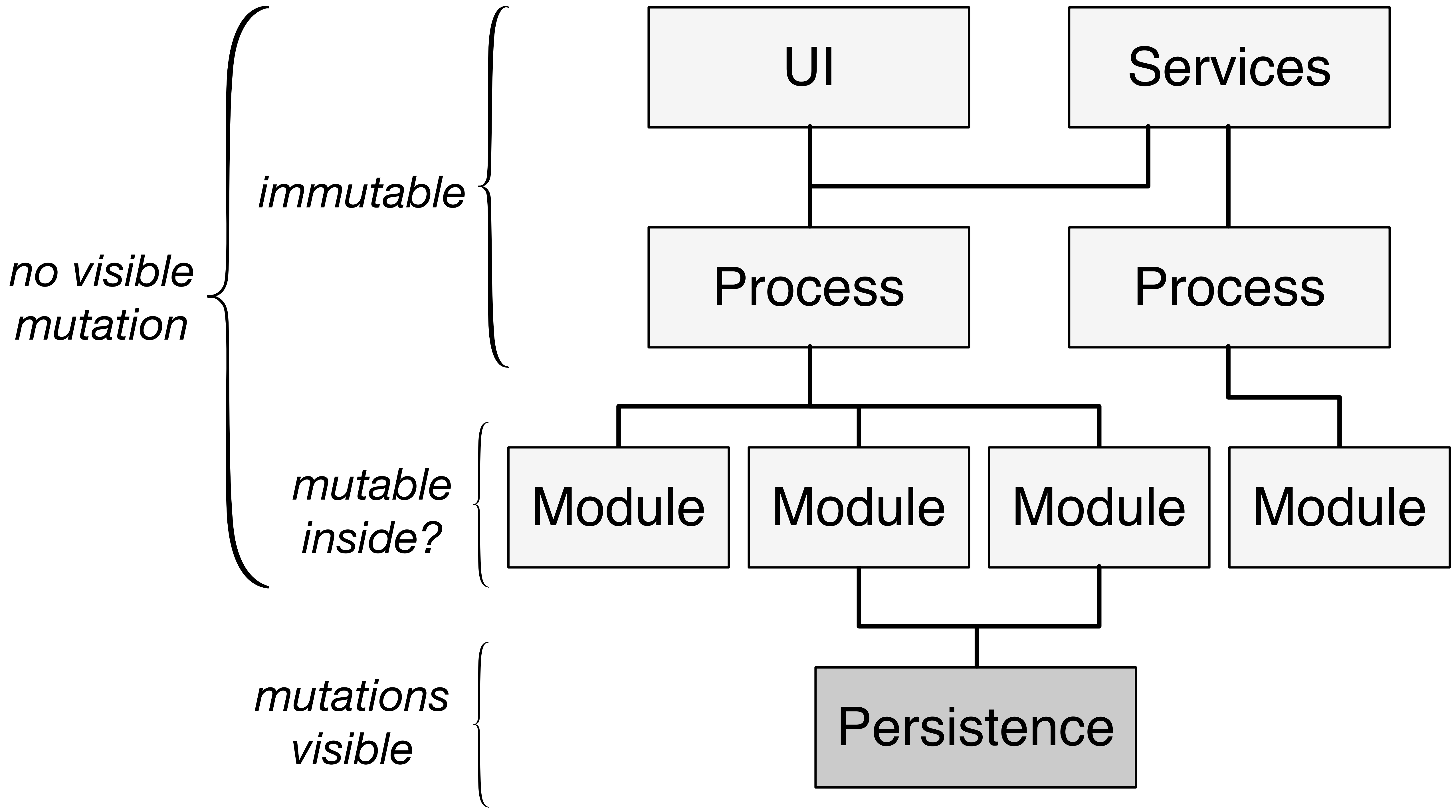
What fields should be in this class? What if you and the team next door need different fields and methods? Should you have a Frankenstein class, the superset of required members? Should you have separate Customer classes and abandon a uniform model for the whole organization? Or, since each team is actually getting the Customer fields from a DB result set, should each team just use a tuple for the field values returned (and not return the whole record!), do the work required, then output new tuples (=> records) to the database, report, or whatever? Do the last option...



Monday, July 21, 14

What most large OO applications look like that I've ever seen. Rich domain models in code that can't be teased apart easily into focused, lightweight, fast services. For example, if a fat "customer" object is needed for lots of user stories, the tendency is to force all code paths through "Customer", rather than having separate implementations, with some code reuse, where appropriate. (We'll come back to this later.)

Mutability



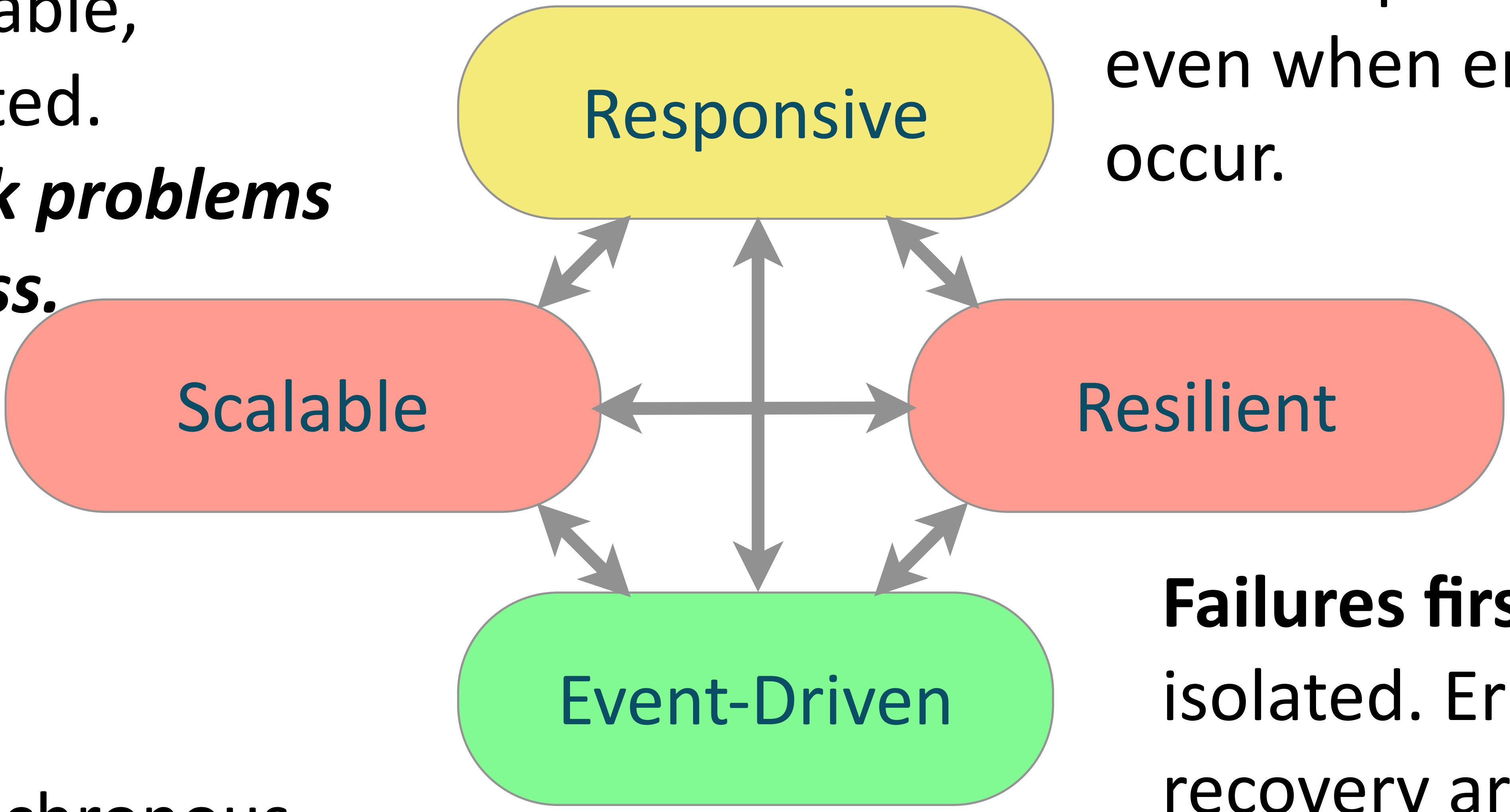
Monday, July 21, 14

There are different levels of granularity. Keep mutation invisible inside modules (& the DB).
Mutation can't be eliminated. Even "pure" FP code, which tries to avoid all mutation, has to have some system-level and I/O mutation somewhere. The key is to do it in a principled way, encapsulated where needed, but remain "logically immutable" everywhere else.
Note that in pure FP, state is expressed at any point in time by the stack + the values in scope.

Critique

Loosely coupled, composable, distributed. ***Network problems first-class.***

Must respond, even when errors occur.



Failures first-class, isolated. Errors/ recovery are just other events.

Asynchronous, non-blocking. Facts as events are pushed.

Monday, July 21, 14

So, how does OOP stand up as a tool for Reactive Programming?
It's good for representing event-driven systems. Actor models have been called object-oriented, matching Kay's description of what he intended. But they suffer in the other traits. Mutation makes loose coupling and scaling very difficult; e.g., it's hard to separate the "model of the world" into separate services. Worse, mutation undermines resilience. A failure requires nontrivial recovery of state.

Alan Kay

"OOP to me means only messaging, local retention and protection, hiding state-process, and extreme late-binding of all things."

http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en and http://en.wikiquote.org/wiki/Alan_Kay

Domain Driven Design

A system-level approach to OOP

Monday, July 21, 14

DDD is very OOP-centric, although efforts are being made to make it applicable to alternative “paradigms”, like functional programming.
Photo: Hancock Building in Chicago on a foggy day.

Ubiquitous Language

All team members use the same domain language. It sounded like a good idea, but leads to bloated, inflexible code. Instead, developers should only put as much of the domain in code as they absolute need, but otherwise, use an appropriate “implementation language”.
Event Driven: It’s important to understand the domain and DDD has events as a first-class concept, so it helps.
Scalable: Modeling and implementing the model in code only makes the aforementioned scaling problems worse.
Responsive: Doesn’t offer a lot of help for more responsiveness concerns.
Resilient: Does model errors, but doesn’t provide guidance for error handling.

Model the Domain

You spend a lot of time understanding the domain and modeling sections of it, relative to use cases, etc. I.e., a domain model for payroll calculation in an HR app. has different concepts than a model for selecting retirement investment options in the same app.
Event Driven: It's important to understand the domain and DDD has events as a first-class concept, so it helps.
Scalable: Modeling and implementing the model in code only makes the aforementioned OOP scaling problems worse. DDD is really an OO approach that encourages implementing objects, which I've argued is unnecessary, although attempts are being made to expand DDD to FP, etc.
Responsive: Doesn't offer a lot of help for more responsiveness concerns.
Resilient: Does model errors, but doesn't provide guidance for error handling.

Claim:
Models *should* be Anemic.

Monday, July 21, 14

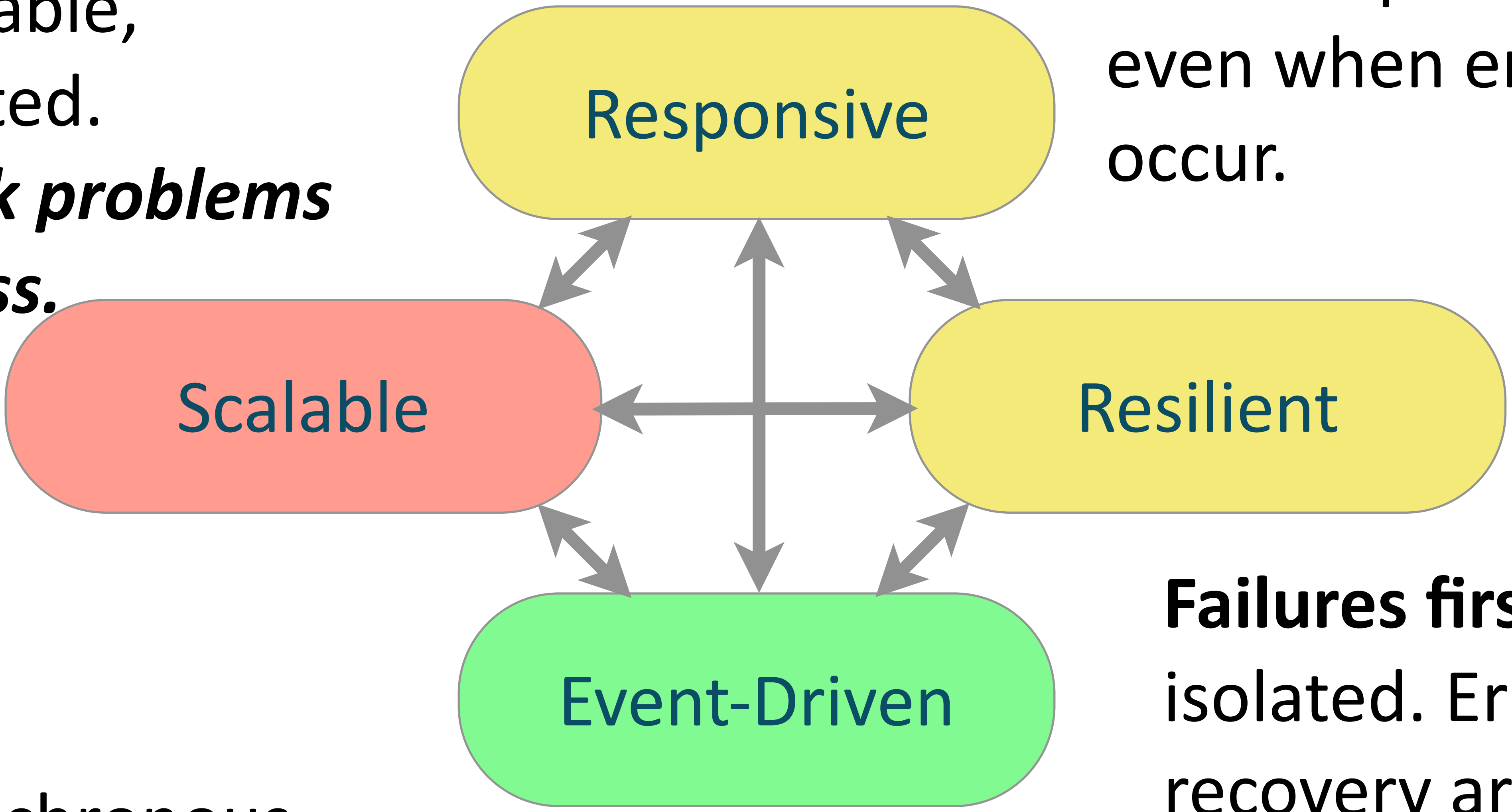
In DDD, models should fully encapsulate state and behavior. Anemic models separate the two concepts, where the class instances are just “structures” and static methods are used to manipulate the data, an “anti-pattern” in DDD. Instead, I’ll argue in the functional programming section that state and behavior should be separated, so Anemic models are preferred!

Model
the domain, but
don’t implement
the models!

Critique

Loosely coupled, composable, distributed. ***Network problems first-class.***

Must respond, even when errors occur.



Failures first-class, isolated. Errors/ recovery are just other events.

Asynchronous, non-blocking. Facts as events are pushed.

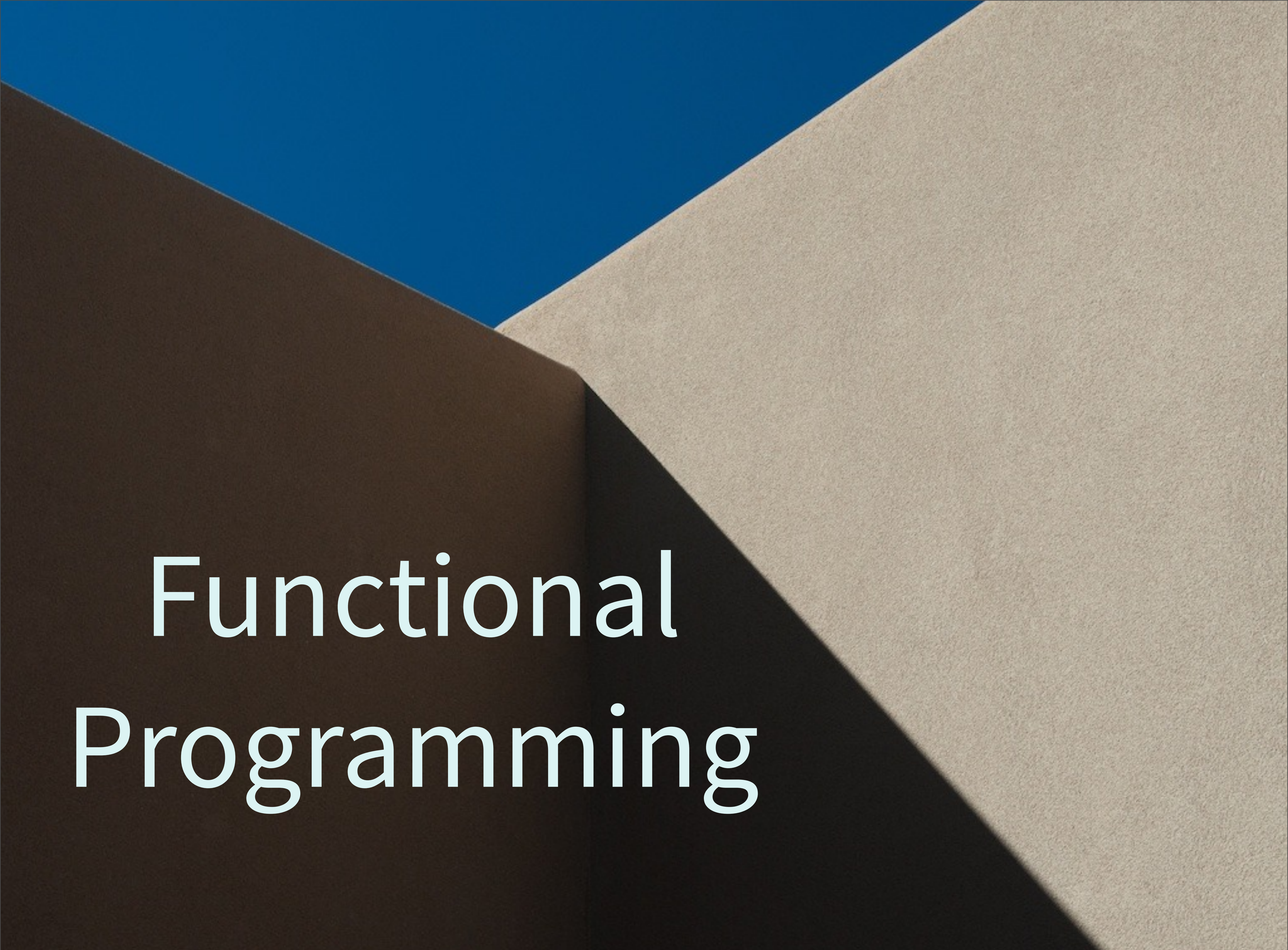
Monday, July 21, 14

Since DDD is primarily a design approach, with more abstract concepts about the implementation, the critique is based on how well it helps us arrive at a good reactive system. I think DDD concepts can be used in a non-OOP way, but few practitioners actually see it that way. Instead, I see DDD practitioners forcing a model onto reactive systems, like Akka, that add complexity and little value. There's a better way...

For a more functional approach to DDD:

debasishg.blogspot.com.au

Start with this blog by Debasish Ghosh, such as this recent post: http://debasishg.blogspot.com.au/2014/04/functional-patterns-in-domain-modeling.html (his most recent at the time of this writing. Some older posts also discuss this topic.
In general, his blog is excellent.



Functional Programming

Pure Functions

Immutable Values

Data “cells” are immutable values.
Event-Driven: The stream metaphor with events that aren’t modified, but replaced, filtered, etc. is natural in FP.
Scalable: On the micro-level, immutability is slower than modifying a value, due to the copy overhead (despite very efficient copy algos.) and probably more cache incoherency (because you have two instances, not one). At the macro scale, mutable values forces extra complexity to coordinate access. So, except when you have a very focused module that controls access, immutability is probably faster or at least more reliably (by avoiding bugs due to mutations).
Resilient: Minimizes bugs.

Referential Transparency

Replace function calls with the values they return. Reuse functions in any context. Caching ("memoization") is an example.
This is only possible with side-effect free functions and immutable values.
Resilient: Side-effect-free functions are much easier to analyze, even replace, so they are less likely to be buggy.
Scalability and responsiveness: Memoization improves performance.

Function Composition

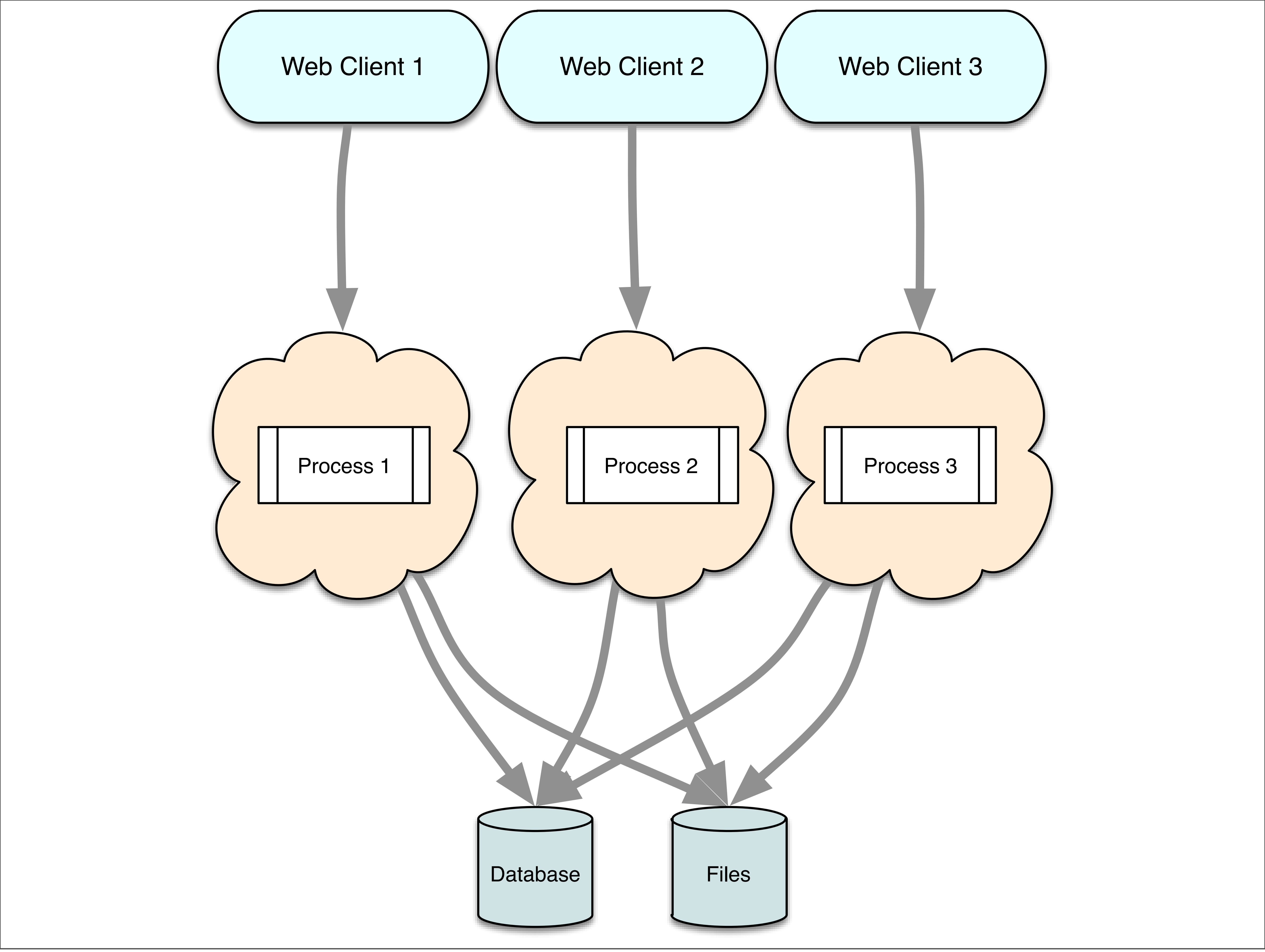
(but we need modules)

Complex behaviors composed of focused, side-effect free, fine-grained functions.
Event-driven: Easy to compose handlers.
Scalable: Small, concise expressions minimize resource overhead. Also benefits the SW-development burden, also if the API is fast, then concise code invoking it makes it easier for users to exploit that performance.
Responsive: Uniform handling of errors and normal logic (Erik Meijer discussed this in his talk).
In general, promotes natural separation of concerns leading to better cohesion and low coupling.

Separation of State and Behavior

Anemic models, for the win...

Functions are separate from values representing state. Functions are *applied* to data. The same operations can be used with any collection, for example. Greatly reduces code bloat through better, more fine-grained reuse. Greater flexibility to compose behaviors. Contrast with Object-Oriented Programming.
Event-Driven: New events can be handled by existing logic. New logic can be added to handle existing events.
Scalable & Responsive: Smaller code base improves resource utilization.
Resilient: Easier to reify exceptions and implement recovery logic.
:



... makes it easier to construct *microservices* that can be sharded (for load scaling) and replicated (for resilience).

Claim:
SW systems are just data-processing systems.

This seems like a trivial statement, but what I mean is that all programs, at the end of the day, just open input sources of data, read them, perform some sort of processing, then write the results to output syncs. That's it. All other "ceremony" for design is embellishment on this essential truth.
If they are data systems, mathematics is our best approach, and FP is our best programming model.

```
import org.apache.spark.SparkContext

object WordCountSpark {
  def main(args: Array[String]) {
    val ctx = new SparkContext(...)
    val file = ctx.textFile(args(0))
    val counts = file.flatMap(
      line => line.split("\\W+"))
                  .map(word => (word, 1))
                  .reduceByKey(_ + _)
    counts.saveAsTextFile(args(1))
  }
}
```

Spark

Monday, July 21, 14

Spark is emerging as the de facto replacement for the Java API, in part due to much drastically better performance, but also LOOK AT THIS CODE! It's amazingly concise and to the point. It's not just due to Scala, it's because functional, mathematical idioms are natural fits for dataflows.

Note the verbs - method calls - and relatively few nouns. The verbs are the work we need to do and we don't spend a lot of time on structural details that are besides the point.

```
import org.apache.spark.SparkContext

object WordCountSpark {
  def main(args: Array[String]) {
    val ctx = new SparkContext(...)
    val file = ctx.textFile(args(0))
    val counts = file.flatMap(
      line => line.split("\\W+"))
                  .map(word => (word, 1))
                  .reduceByKey(_ + _)
    counts.saveAsTextFile(args(1))
  }
}
```

Spark

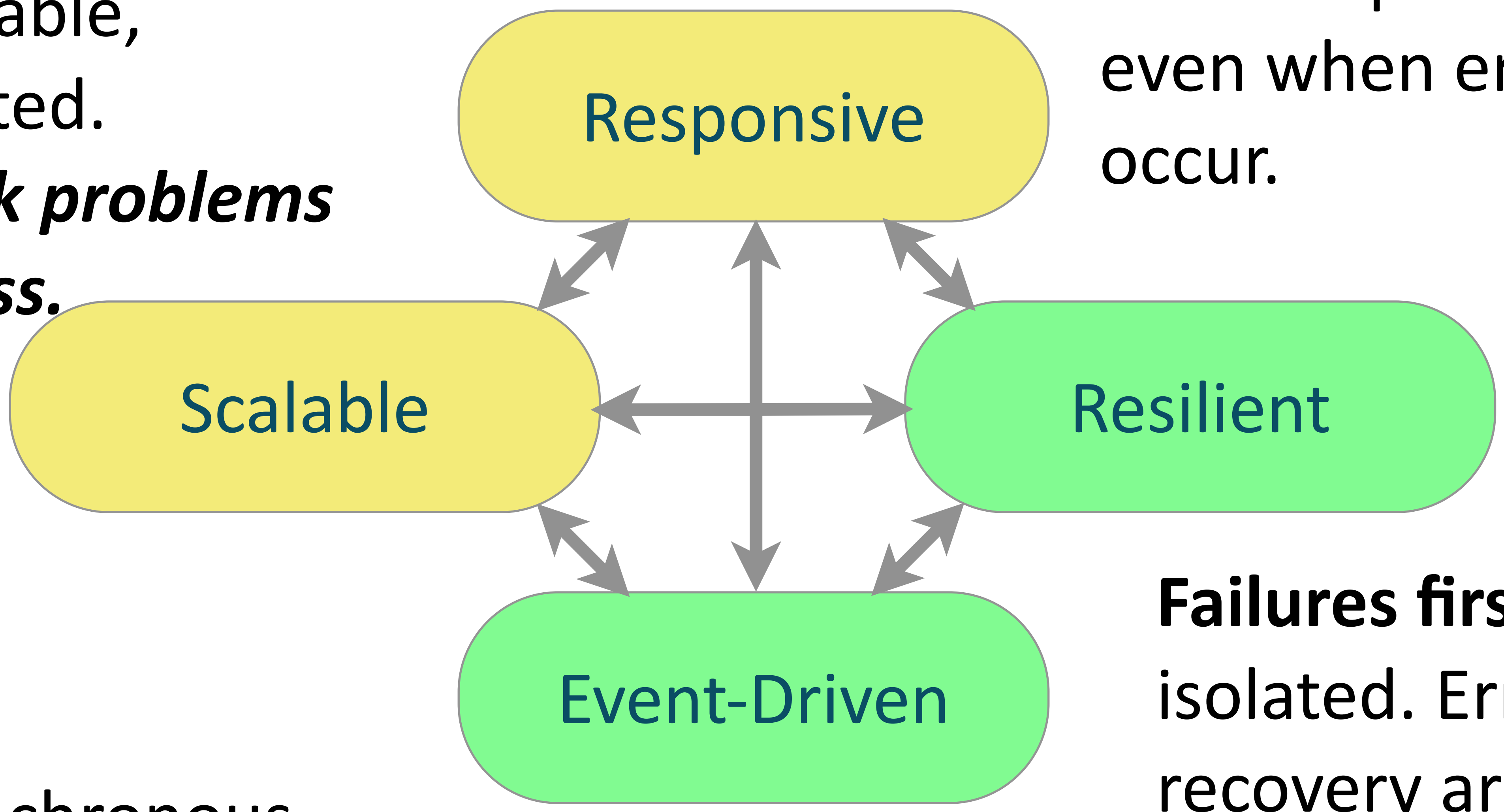
Monday, July 21, 14

Because it's so concise, it reduces to a “query on steroids”, a script that's no longer a “program”, requiring all the usual software development process hassles, but a script we tweak and try, use when it's ready, and discard when it's no longer needed.
I want to return to the simple pleasures of bash programming.

Critique

Loosely coupled, composable, distributed. ***Network problems first-class.***

Must respond, even when errors occur.



Failures first-class, isolated. Errors/ recovery are just other events.

Asynchronous, non-blocking. Facts as events are pushed.

Monday, July 21, 14

FP is the most natural basis for the Reactive Programming.
I claim it provides the core concepts that are best for addressing the 4 traits, but specific libraries are still required to implement the particulars, especially scalability and responsiveness. Also, there will be exceptions that you make for performance, such as mutable, lock-free queues. See Martin Thompson's talk at React Conf 2014!

Small Tools

Monday, July 21, 14

Now let’s look at “small” tools; idioms and design patterns we’ve had for a while.

photo: Serra sculpture in the Seattle Art Museum outdoor sculpture garden.

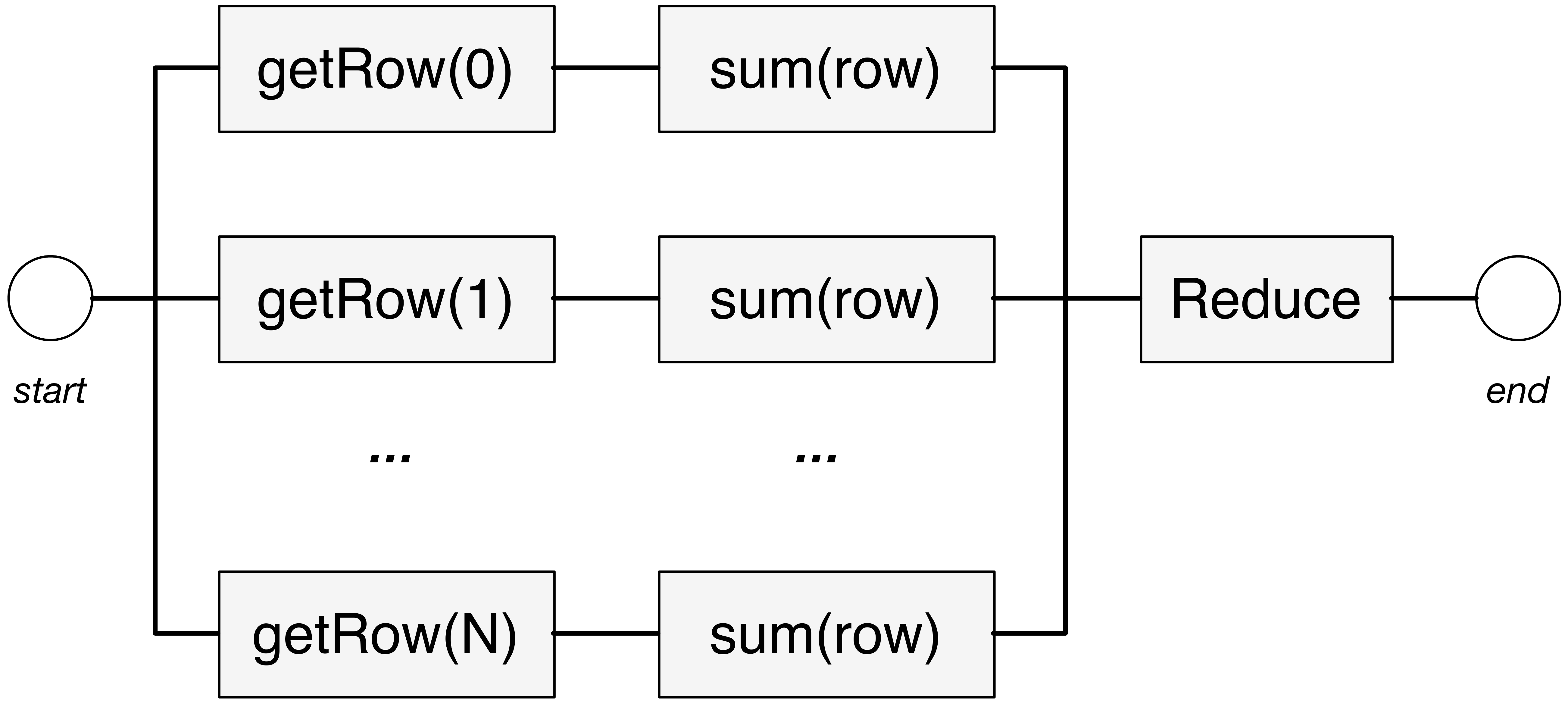
Futures

Monday, July 21, 14

Photo: Grand Canyon National Park

Ex: Scatter/Gather

- Sum large matrix of numbers, using divide and conquer:



Monday, July 21, 14

Dataflow graph construction!
Note that each row is synchronous, but the rows run in parallel. They “rendezvous” at the fold. In other words, this is a dataflow graph.
This is a “batch-mode” example; assumes all the data is present, it’s not a data pipeline.
Can also use futures for event streams, but you must instantiate a new dataflow for each event or block of events.

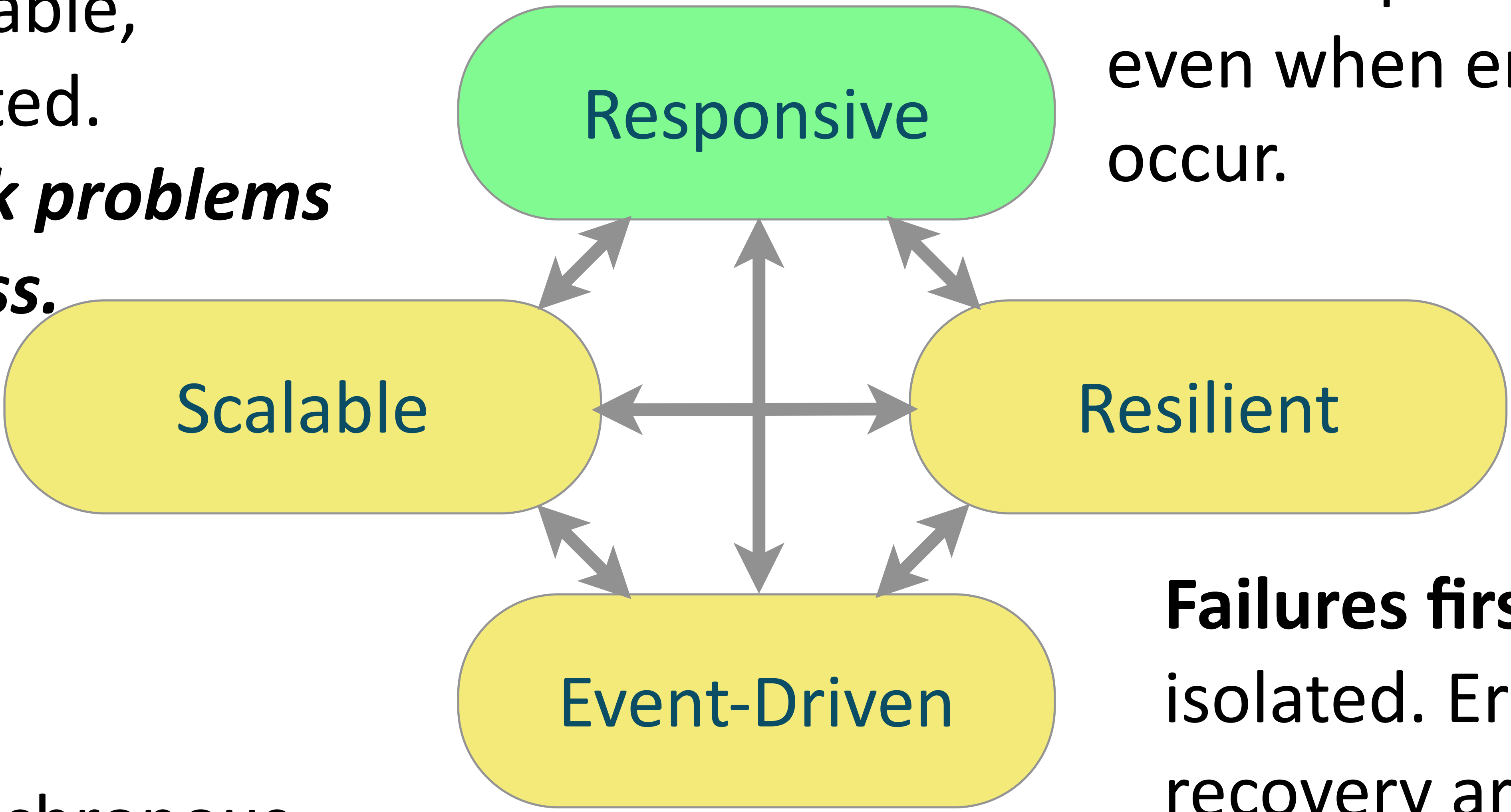
```
def sumRow(i: Int): Future[Long] =
  Future(getRow(i)).map(row => sum(row))

val rowSumFutures: Seq[Future[Long]] =
  for (i <- 1 to N) yield sumRow(i)

val result = Future.reduce(rowSumFutures) {
  (accum, rowSum) => accum + rowSum2
}  // => Future[Long]

println(result.value)
```

This example assumes I have an NxN matrix of numbers (longs) and I want to sum them all up. It's big, so I'll sum each row in parallel, using a future for each one, then sum those sums. Note that "sumRow" sequences two futures, the first to get the row (say from some slow data store), then it maps the returned row to a call to "sum(row)" that will be wrapped inside a new Future by the "map" method.
"sum" (sum the Long values in a row) and "getRow" (get a row from a matrix) functions not shown, but you can guess what they do.
This is a "batch-mode" example; assumes all the data is present.
Can also use futures for event streams.

Critique

Loosely coupled, composable, distributed. ***Network problems first-class.***

Must respond, even when errors occur.



Failures first-class, isolated. Errors/ recovery are just other events.

Asynchronous, non-blocking. Facts as events are pushed.

Monday, July 21, 14

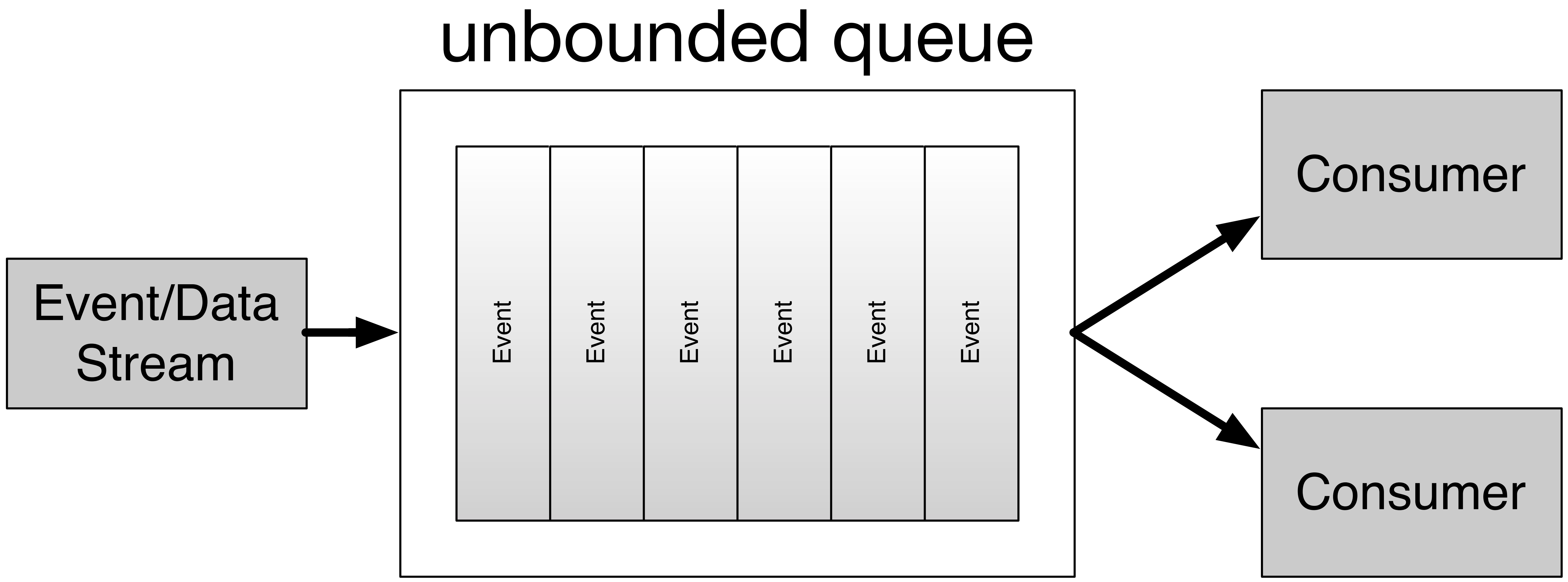
Event-Driven: You can handle events with futures, but you'll have to write the code to do it.
Scalable: Improves performance by eliminating blocks. Easy to divide & conquer computation, but management burden for lots of futures grows quickly and by themselves, they don't address network issues.
Resilient: Model provides basic error capturing, but not true handling and recovery. If there are too many futures many will wait for available threads. The error recovery consists only of stopping a sequence of futures from proceeding (e.g., the map call on the previous page). A failure indication is returned. However, there is no built-in retry, and certainly not for groups of futures, analogous to what Actor Supervisors provides.
Responsive: Very good, due to nonblocking model, but if you're not careful to avoid creating too many futures, they'll be "starved" competing for limited threads in the thread pool.

Streams

Monday, July 21, 14

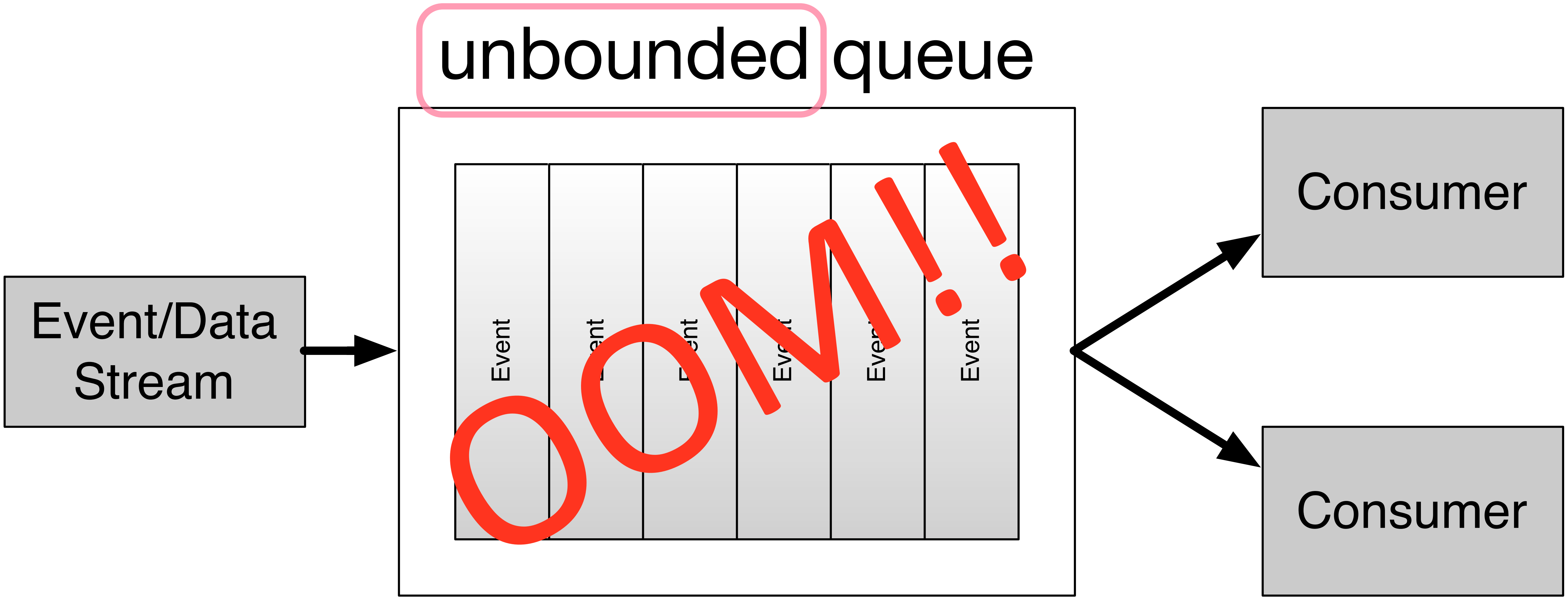
Photo: Near Sacramento, California

Generic Streams...



Simple streams, modeled as a queue. Should it be bounded? What happens if it isn't?
Eventually, in a long-running system, the consumers won't be able to keep up with the producers. If the queue grows without bound, an inevitable out-of memory error (OOM) will happen.

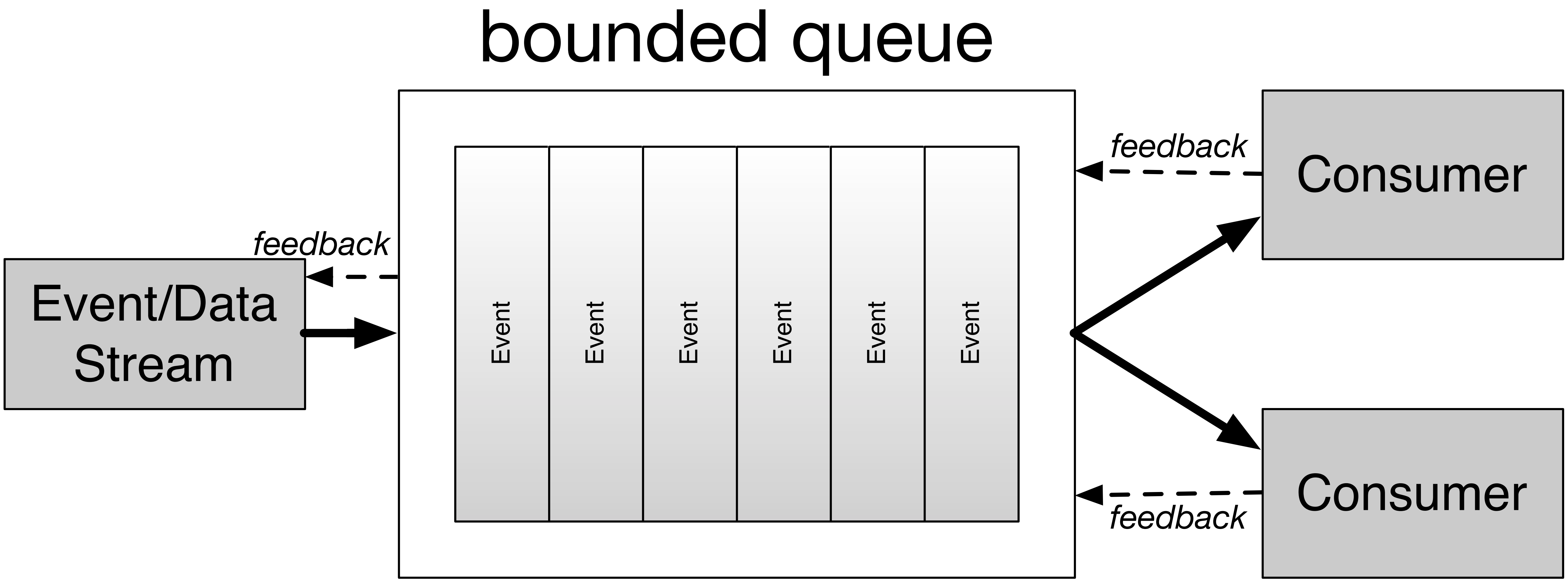
Generic Streams...



Monday, July 21, 14

Simple streams, modeled as a queue. Should it be bounded? What happens if it isn't?
Eventually, in a long-running system, the consumers won't be able to keep up with the producers. If the queue grows without bound, an inevitable out-of memory error (OOM) will happen.

Stream with Backpressure



reactive-streams.org

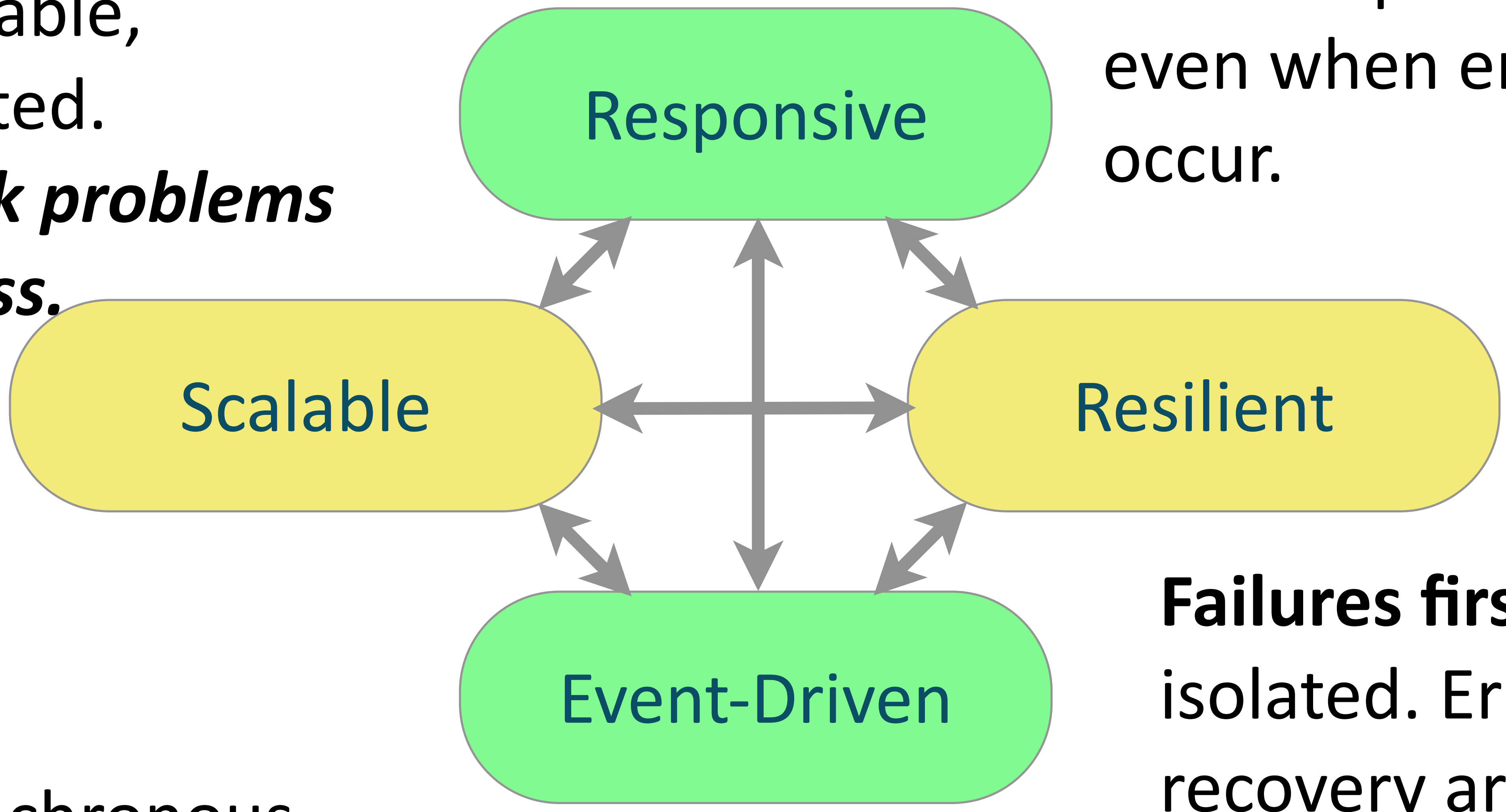
Monday, July 21, 14

Asynchronous streams: Support time-varying values as first class.
Back Pressure first class: No explicit mutation or update end-user logic required. And signaling is effectively out of band (think of a priority queue instead of a regular queue...).
Martin Thompson has talked a lot about how important this is (google: mechanical sympathy).

Critique

Loosely coupled, composable, distributed. ***Network problems first-class.***

Must respond, even when errors occur.



Failures first-class, isolated. Errors/ recovery are just other events.

Asynchronous, non-blocking. Facts as events are pushed.

Monday, July 21, 14

Compared to RX, adding backpressure turns responsive “more” green.
Event-Driven: First class.
Scalable: Designed for high performance, but for horizontal scaling, need independent, isolated instances.
Resilient: Doesn’t provide failure isolation, error recovery, like an authority to trigger recovery, but back pressure eliminates many potential problems.
Responsive: Excellent, due to nonblocking, push model, support for back pressure.

Tools for Reactive

Monday, July 21, 14

Let’s start with programming paradigms. How well do they promote reactive programming??

Photo: Rethink Robots’ Baxter (at SolidCon 2014)

Functional Reactive Programming

Monday, July 21, 14

Original idea: “Functional Reactive Animation”, by Conal Elliott and Paul Hudak
Photo: Building, San Francisco.

Functional Reactive Programming

- ***Behaviors*: Datatypes of values over time:**
  Support time-varying values as first class.

```
x = mouse.x
y = mouse.y
```

- ***Events*: Drive changes over time.**
- **Derived expressions update automatically:**

```
a = area(x,y)
```

- **Deterministic, fine-grained, and concurrent.**

It's a dataflow *system.*

Monday, July 21, 14

Invented in Haskell ~1997. Recently implemented and spread outside the Haskell community as part of the Elm language for functional GUIs, Evan Czaplicki's graduate thesis project (~2012).
Time-varying values are first class. They could be functions that generate "static" values, or be a stream of values. They can be discrete or continuous.
User's don't have to define update logic to keep derived values in sync, like implement observer logic.

A Scala.React example

```
Reactor.flow { reactor =>
  val path = new Path(
   (reactor.await(mouseDown)).position)
  reactor.loopUntil(mouseUp) {
    val m = reactor.awaitNext(mouseMove)
    path.lineTo(m.position)
    draw(path)
  }
  path.close()
  draw(path)
}
```

From *Deprecating the Observer Pattern with Scala.React*.

It's a prototype DSL for writing what looks like imperative, synchronous logic for the the "state machine" of tracking and reacting to a mouse drag operation, but it runs asynchronously (mostly).
I've made some minor modifications to the actual example in the paper.

Encapsulates Evolving State Changes

Nicely let's you specify a dataflow of evolving state, modeled as events.

Each Reactive Element Has Its Own Thread

Monday, July 21, 14

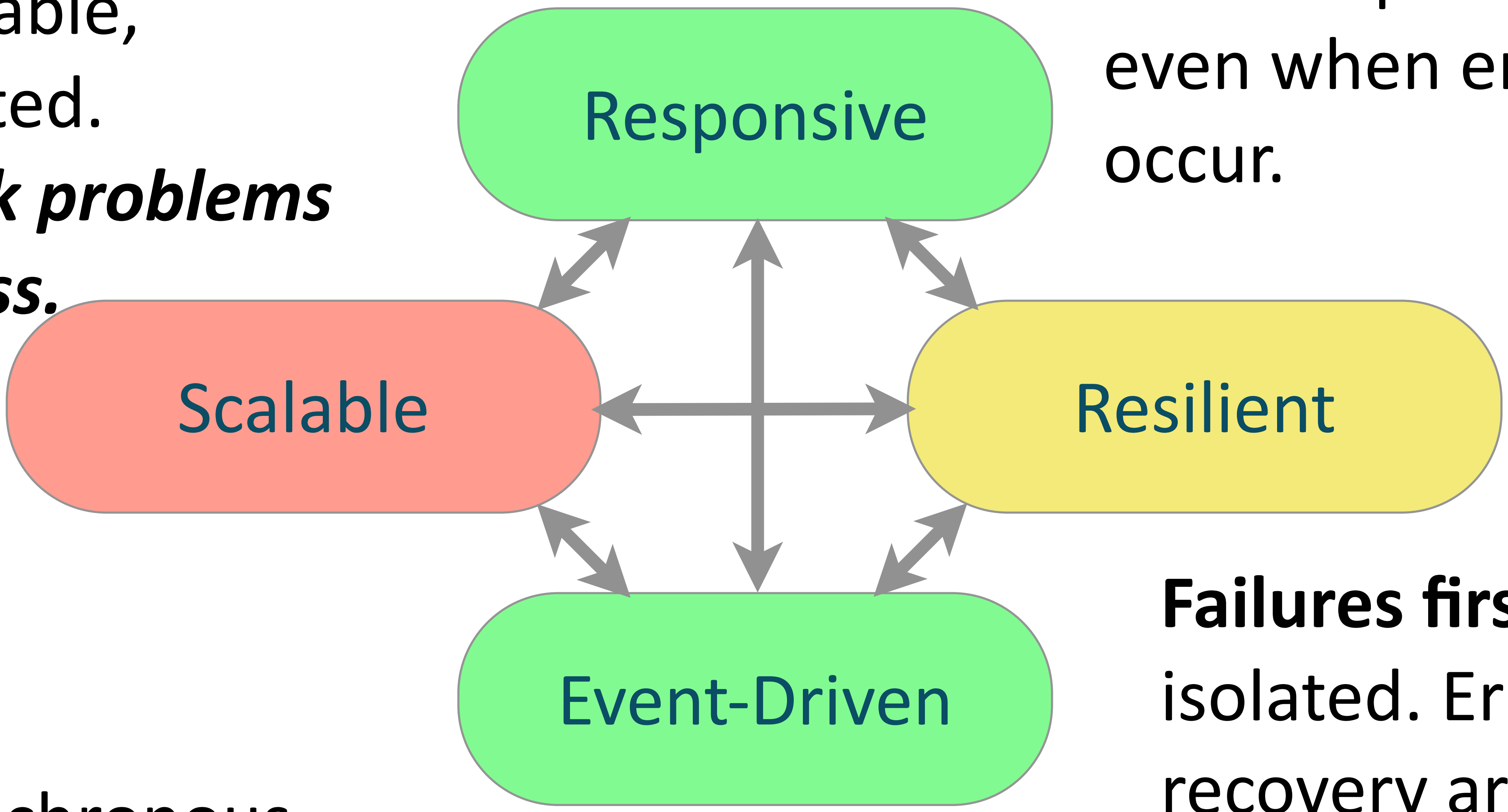
Unfortunately, most UI toolkits have a (single) UI event loop. This makes it difficult to have concurrent dataflows that intersect, in part because of the challenge of universal time synchronization. However, separate UI elements could have many, completely-independent threads of control.

So, horizontal scalability is not solved in many implementations. Also resiliency is a concern as no error recovery mechanism is provided. See Evan Czalplicki's Thesis on Elm for one solution.

Critique

Loosely coupled, composable, distributed. ***Network problems first-class.***

Must respond, even when errors occur.



Failures first-class, isolated. Errors/ recovery are just other events.

Asynchronous, non-blocking. Facts as events are pushed.

Monday, July 21, 14

FRP is mostly about encapsulation of evolving state in a functional manner. Implementations don't typically define robust error handling, such as reconstructing the dataflow on error. Scalability is a practical concern, because many don't implement concurrent constructs. Elm is one exception, but it's also limited by JavaScript.

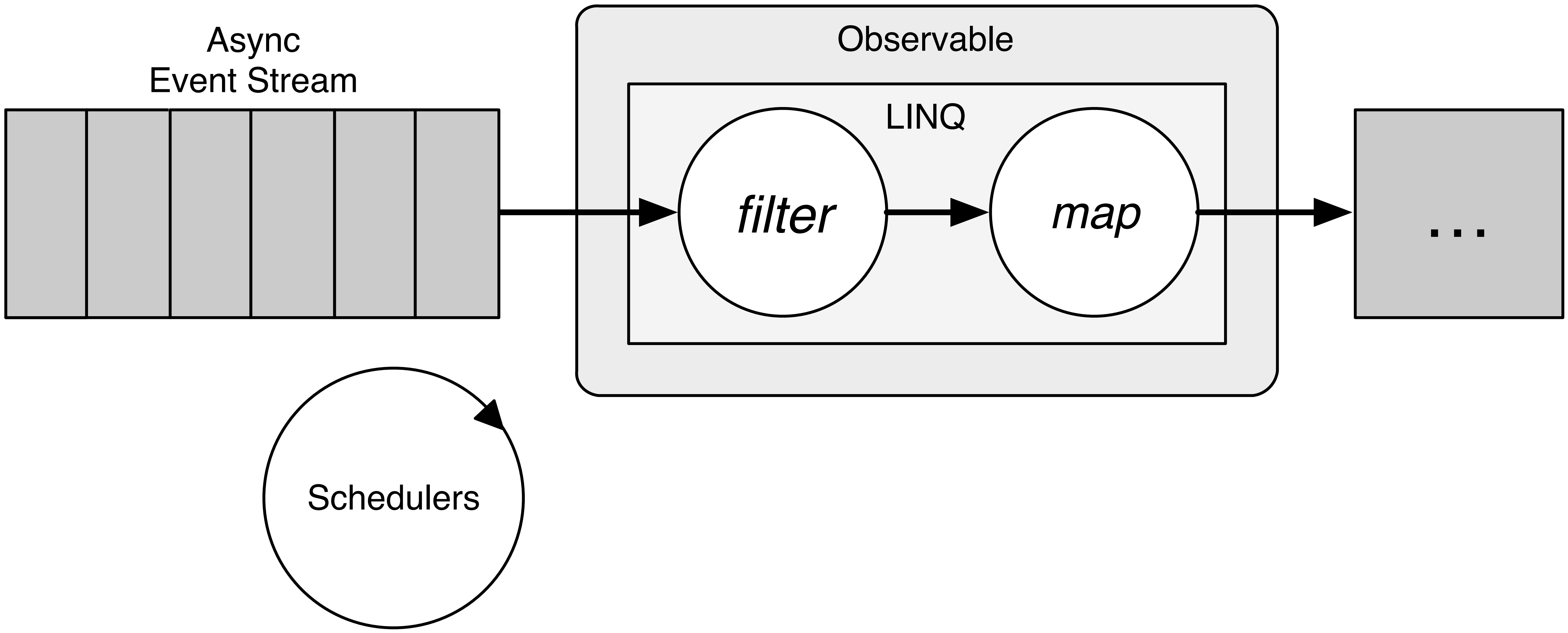
Rx

Not the little blue pill you might be thinking of...

Reactive Extensions

- **Composable, event-based programs:**
- **Observables:** Represent async. event streams.
- **LINQ:** The streams are queried using LINQ (language integrated query).
- **Schedulers:** *parameterize* the concurrency in the streams.

https:///rx.codeplex.com - This is the original Microsoft implementation pioneered by Erik Meijer. Other implementations in a wide-variety of languages follow the same model, but differ in various ways, such as a replacement for LINQ.



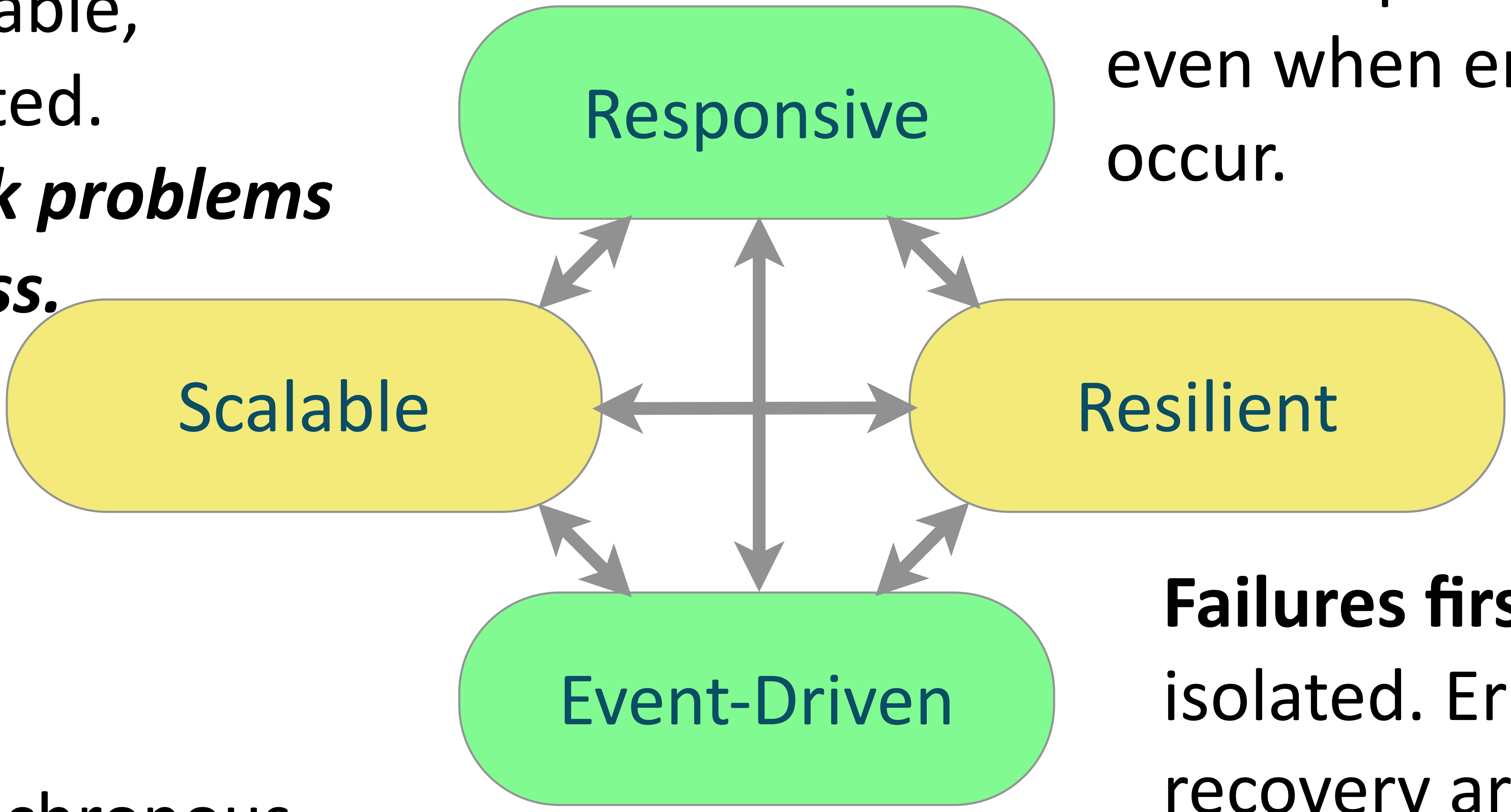
Monday, July 21, 14

https:///rx.codeplex.com - This is the original Microsoft implementation pioneered by Erik Meijer. Other implementations that follow the same model will differ in various ways.

Critique

Loosely coupled, composable, distributed. ***Network problems first-class.***

Must respond, even when errors occur.



Failures first-class, isolated. Errors/ recovery are just other events.

Asynchronous, non-blocking. Facts as events are pushed.

Monday, July 21, 14

This is a specific, popular approach to reactive. Does it meet all our needs?
This looks a bit more negative than it really is, as I'll discuss.
Event-Driven: Represents events well
Scalability: Increased overhead of instantiating observers and observable increases. Not as easy to scale horizontally without single pipelines, e.g. a farm of event-handler "workers".
Responsive and Resilient: Errors handled naturally as events, although an out-of-band error signaling mechanism would be better and there's no built-in support for back pressure.

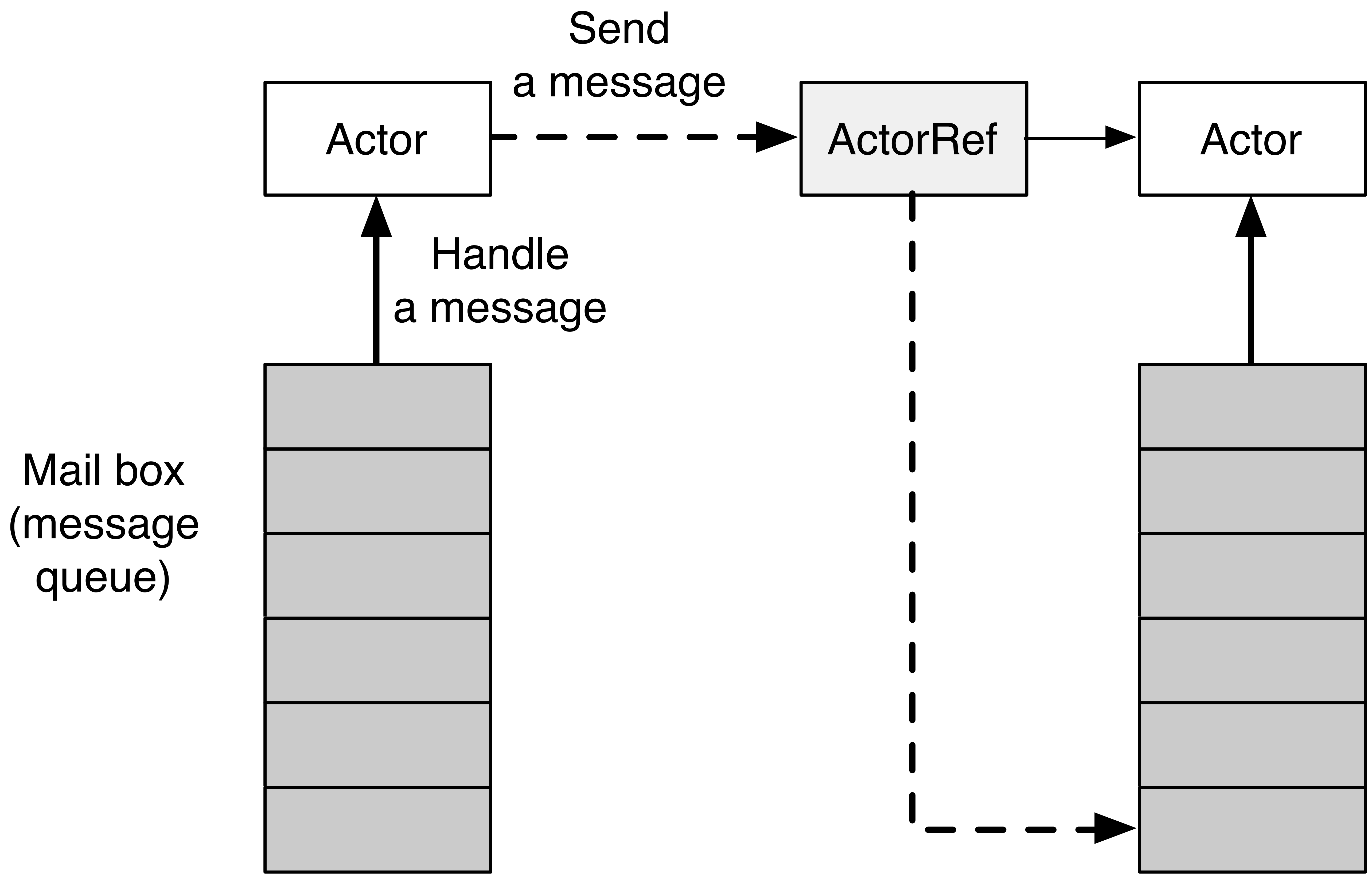
Actors

Monday, July 21, 14

Photo: San Francisco Sea Gull... with an attitude.

Actors

- **Actor Model**
  - Autonomous agents encapsulating state.
  - Actors communicate only through messages.
  - First class concept in Erlang!
  - Implemented with libraries in other languages.
- **Best of breed error handling:**
  - *Supervisor hierarchies of actors* dedicated to lifecycle management of workers and sophisticated error recovery.



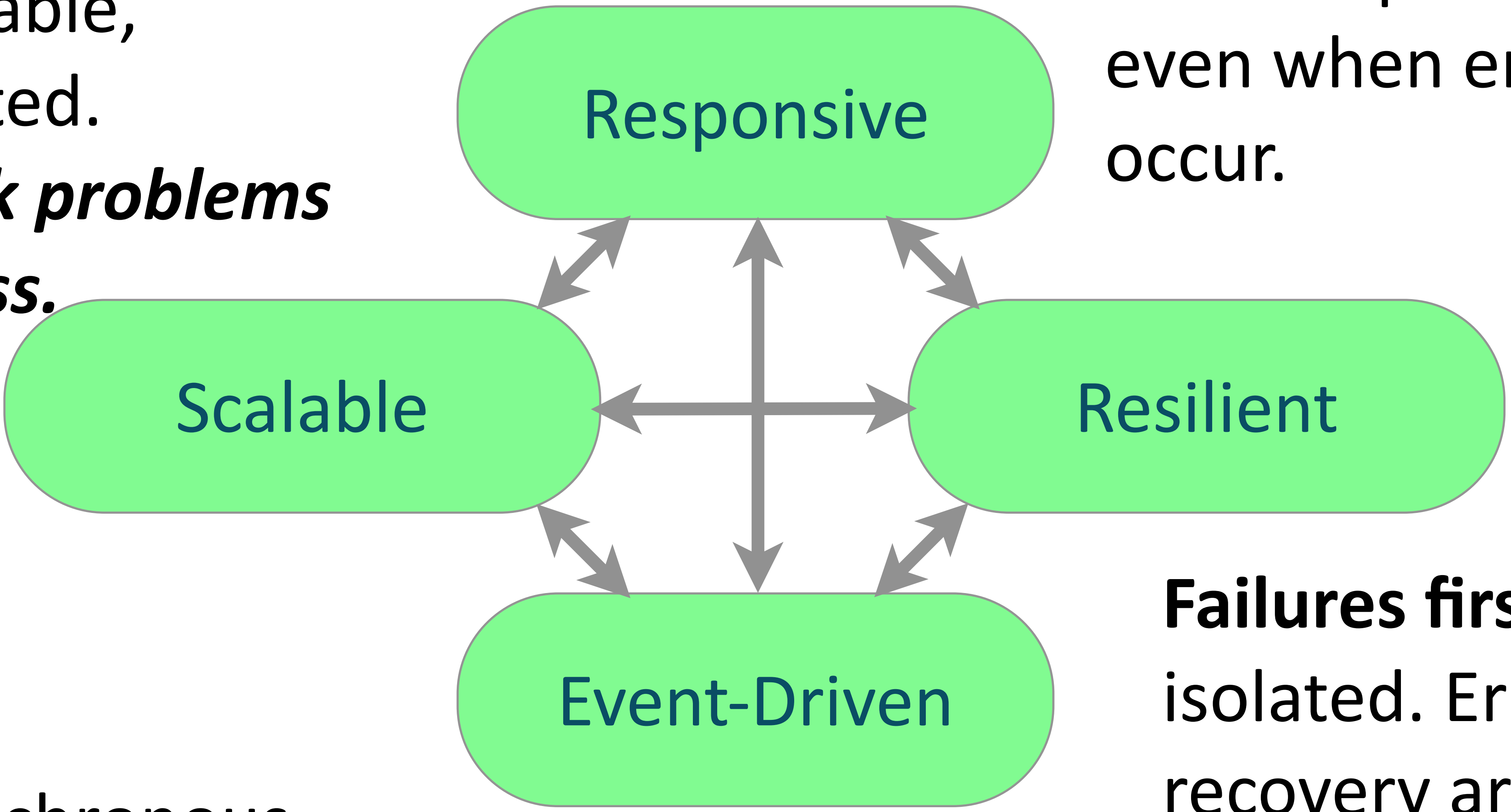
Monday, July 21, 14

Synchronization through nonblocking, asynchronous messages. Sender-receiver completely decoupled. Messages are immutable values.
Each time an actor processes a message: 1) The code is thread-safe. 2) The actor can mutate state safely.
In Akka (http://akka.io), messages sent from one actor are actually received by an ActorRef (reference) that writes the message to the end of the queue). The extra layer of indirection supports restarting a failed actor without breaking client references to the actor, which are actually to the corresponding ActorRef.

Critique

Loosely coupled, composable, distributed. ***Network problems first-class.***

Must respond, even when errors occur.



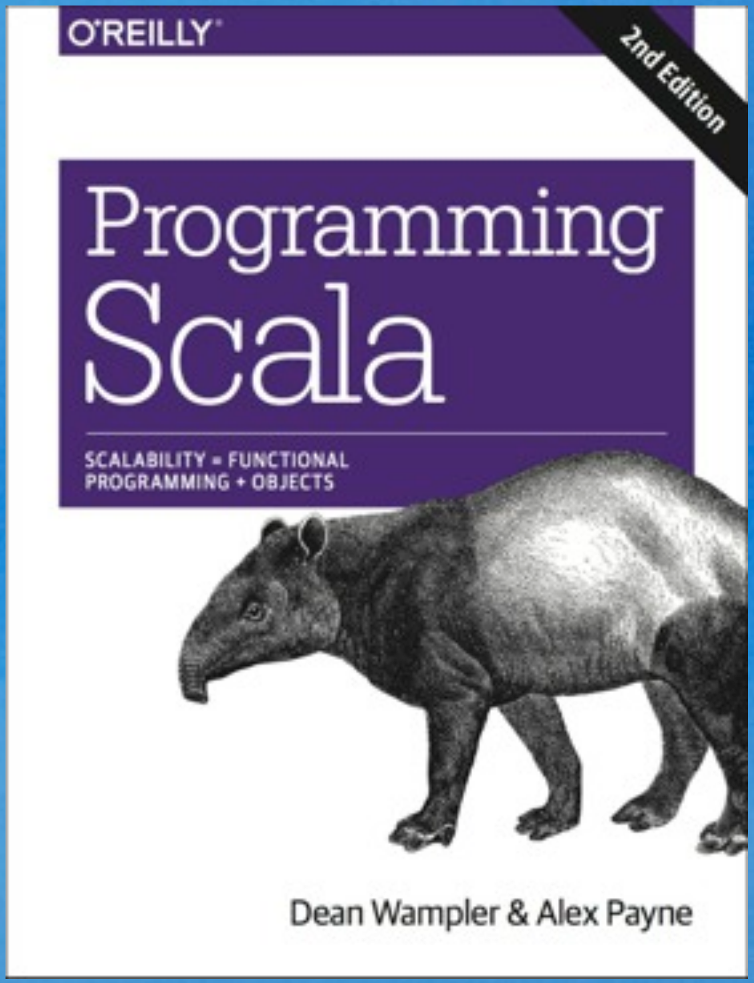
Failures first-class, isolated. Errors/ recovery are just other events.

Asynchronous, non-blocking. Facts as events are pushed.

Monday, July 21, 14

Event-Driven: Events map naturally to messages, stream handling can be layered on top.
Scalable: Improves performance by eliminating blocks. Easy to divide & conquer computation. Principled encapsulation of mutation.
Resilient: Best in class for actor systems with supervisor hierarchies.
Responsive: Good, but some overhead due to message-passing vs. func. calls.

Typesafe

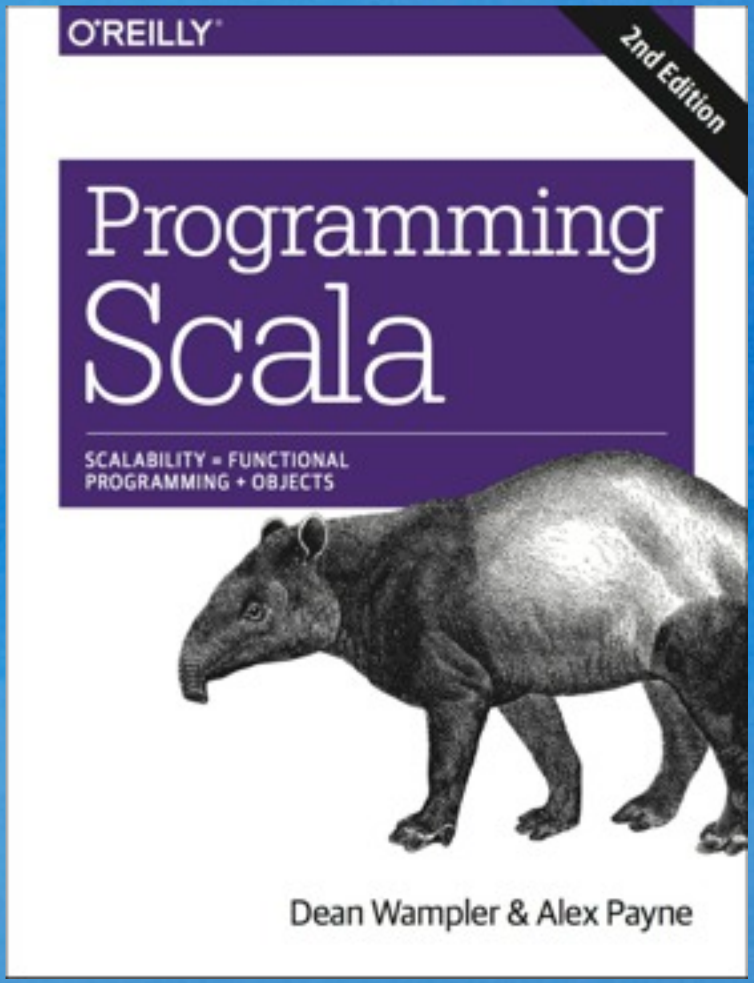


Dean Wampler
dean.wampler@typesafe.com
@deanwampler
polyglotprogramming.com/talks

Monday, July 21, 14

Image: My cat Oberon, enjoying the morning sun...

Typesafe



O'REILLY

2nd Edition

Programming Scala

SCALABILITY = FUNCTIONAL PROGRAMMING + OBJECTS

Dean Wampler & Alex Payne

Dean Wampler
dean.wampler@typesafe.com
@deanwampler
polyglotprogramming.com/talks

Monday, July 21, 14

Image: My cat Oberon, enjoying the morning sun...