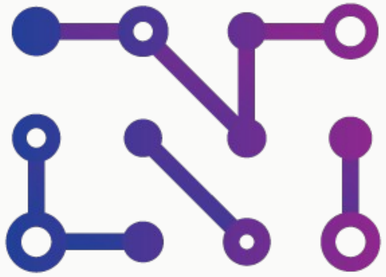
Flexibility Through Immutability

Ricardo J. Méndez
ricardo@numergent.com



NUMERGENT

What we'll talk about

- Quick background on immutable data and FP.
- Advantages and trade-offs. i.e., "why bother?"
- Four simple things to put it in practice in an object-oriented approach.

Getting to know each other

@ArgesRic

NUMERGENT

Anyone working without garbage collection?

Who’s working on a functional programming language?

What are you working on? Python? Ruby? Java? C#?

Who is already using immutable data *somewhere*?

About me

- Software engineer, run Numergent.
- Run project-specific, distributed development teams.
- Work mostly with data-oriented projects, on media, health care information management, and financial companies.
- Doing software development professionally for 20+, hacking around for longer.

@ArgesRic

NUMERGENT

My path here

@ArgesRic

NUMERGENT

Come for the functional way,
stay for the immutable data.

@ArgesRic

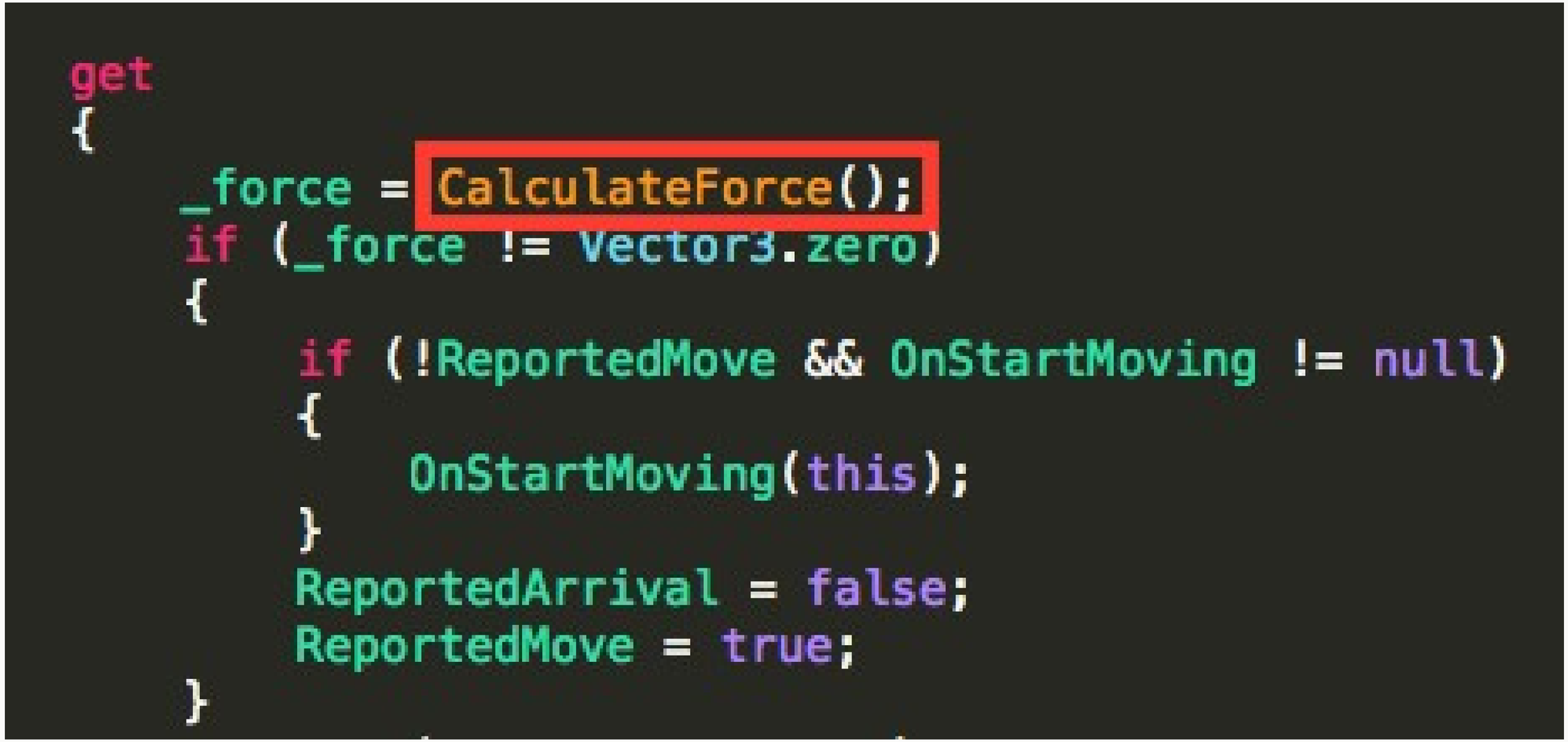
NUMERGENT

Realized immutable data made code easier to refactor.

```
get
{
    _force = CalculateForce();
    if (_force != Vector3.zero)
    {
        if (!ReportedMove && OnStartMoving != null)
        {
            OnStartMoving(this);
        }
        ReportedArrival = false;
        ReportedMove = true;
    }
```

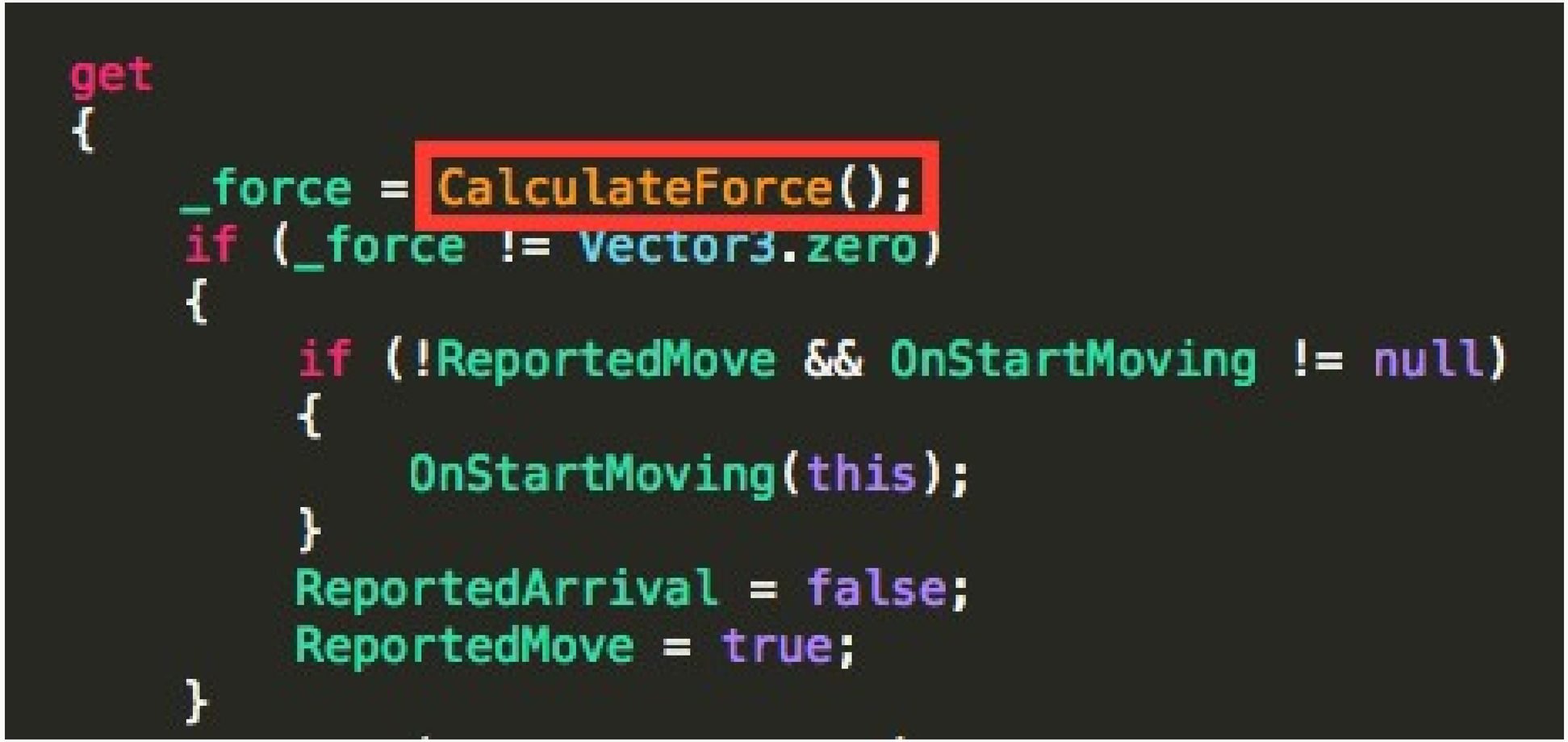
@ArgesRic

NUMERGENT



@ArgesRic

NUMERGENT



@ArgesRic

NUMERGENT

If you have mutable data,
you have to take things on faith.

```
/*
 * Creates two list: one of randomly selected elements, and one
 * that contains all those that were ignored. The first one will
 * be assigned to the user, the second one will be moved along.
 */
var randomizer = new ArrayRandomizer<Domain.Image>(imageList);
var randomList = randomizer.GetRandomSubsetFisherYates(checkNumber);
```

```
/*
 * Creates two list: one of randomly selected elements, and one
 * that contains all those that were ignored. The first one will
 * be assigned to the user, the second one will be moved along.
 */
var randomizer = new ArrayRandomizer<Domain.Image>(imageList);
var randomList = randomizer.GetRandomSubsetFisherYates(checkNumber);
```

@ArgesRic

NUMERGENT

```
/*
 * Creates two list: one of randomly selected elements, and one
 * that contains all those that were ignored. The first one will
 * be assigned to the user, the second one will be moved along.
 */
var randomizer = new ArrayRandomizer<Domain.Image>(imageList);
var randomList = randomizer.GetRandomSubsetFisherYates(checkNumber);
```

Can a long-lived object trust we won’t change its parameters?

Why immutable data?

There is no frictionless movement.

Stop thinking about operations, start thinking about results

@ArgesRic

NUMERGENT

Immutability is not statelessness

@ArgesRic

NUMERGENT

You have a state.
Your state is your world view.

When your state changes,
you don’t discard knowledge.

A functional approach

Many inputs, one single output.

Values are immutable.

Functions do not trigger any state side-effects.

Functional is about semantics, languages just help

@ArgesRic

NUMERGENT

“The most boring things in the universe”

Constantin Dumitrescu @ BucharestFP

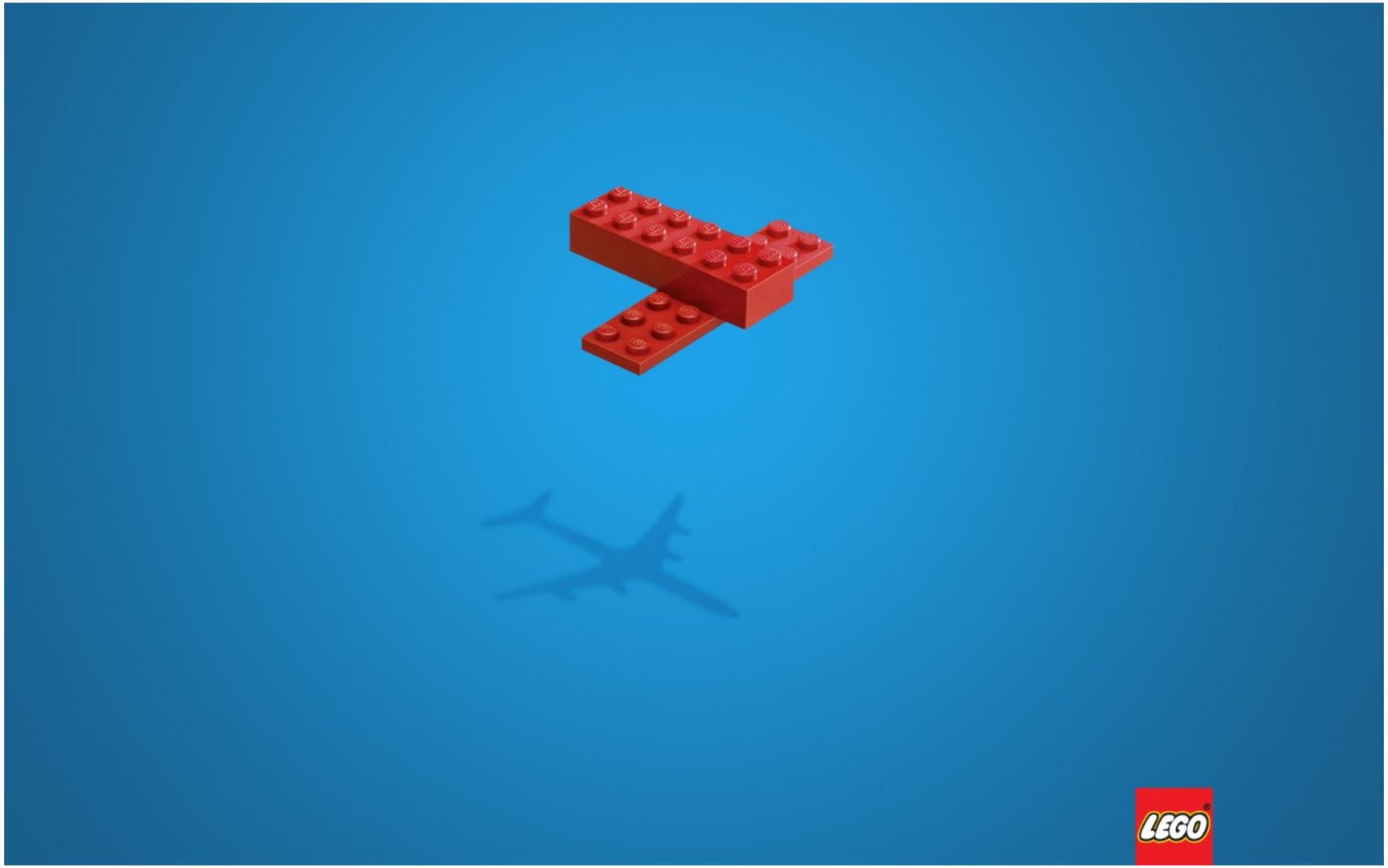
“Clojure is Boring”

https://8thlight.com/blog/colin-jones/2016/10/06/clojure-is-boring.html

@ArgesRic

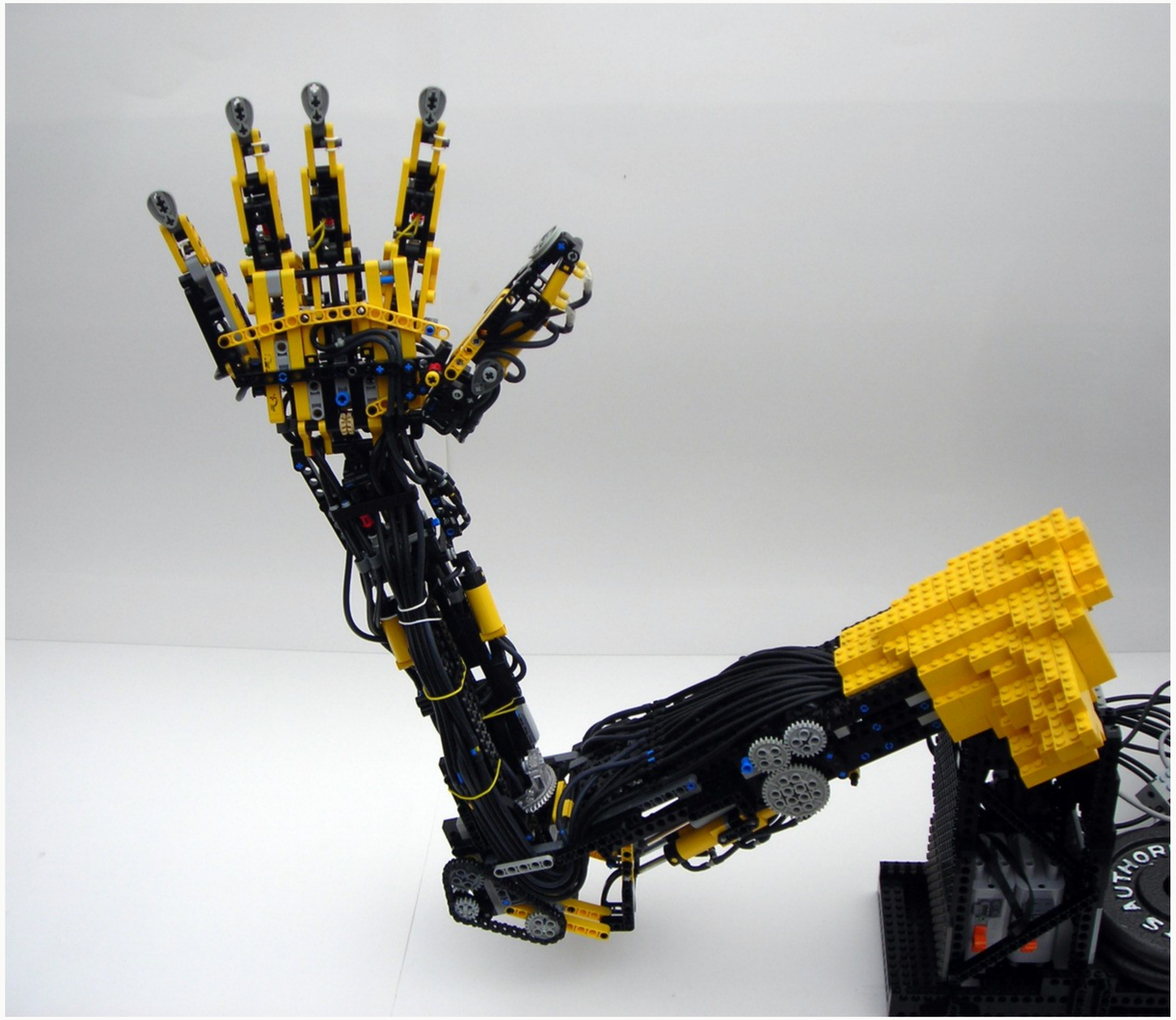
NUMERGENT


LEGO
LEGO
LEGO
LEGO
LEGO
LEGO
LEGO
LEGO



@ArgesRic

NUMERGENT



@ArgesRic

numergent

Show of hands again...
C# / Java users.

Strings!

- Do you have a problem understanding how they work?
- Are you worried that they'll be changed from under you?
- Are you concerned about using it as a key in a dictionary?
- Have you had to check the implementation?
- Do you think they are exciting?

Strings are boring, reliable, *immutable* data items.

@ArgesRic

numergent

void DoSomethingToObject()

In-place Add/Remove

ref and out

Dealing with unknowns

```clojure
(defn migrate
  "Migrates a data set from its version to the next one. Returns the same
  data set if it cannot apply any migration."
  [data]
  (condp = (:data-version data)
    nil (->
          data
          (assoc :instance-id (or (:instance-id data)
                                  (.-uuid (random-uuid))))
          (assoc :data-version 1)
          (assoc :url-times (into {} (map #(vector (key %)
                                                   (dissoc (val %) :favIconUrl :icon))
                                          (:url-times data))))
          (assoc :site-times (accumulate-site-times (:url-times data))))
    1 (->
        data
        (assoc :data-version 2)
        (assoc :site-times (accumulate-site-times (:url-times data))))
```

@ArgesRic

NUMERGENT

```clojure
(defn migrate
  "Migrates a data set from its version to the next one. Returns the same
  data set if it cannot apply any migration."
  [data]
  (condp = (:data-version data)
    nil (->
          data
          (assoc :instance-id (or (:instance-id data)
                                  (.-uuid (random-uuid))))
          (assoc :data-version 1)
          (assoc :url-times (into {} (map #(vector (key %)
                                                   (dissoc (val %) :favIconUrl :icon))
                                          (:url-times data))))
          (assoc :site-times (accumulate-site-times (:url-times data))))
    1 (->
        data
        (assoc :data-version 2)
        (assoc :site-times (accumulate-site-times (:url-times data))))
```

@ArgesRic

NUMERGENT

For an unknown method:

1. Poke it.
2. Read it.

Being fully acquainted with the code is the only option with variable data.

1. Have access to every source involved.

2. Have the time available.

There’s unknowns everywhere.

The larger the team, the more unknowns.

1. Not everyone will understand the subtleties of the language.

2. Not everyone will understand the subtleties of your code base.

But...

Single Responsibility Principle!

Cross-cutting concerns make Single Responsibility non-trivial.

Eventually, you'll encapsulate your herd of methods.

Encapsulation reduces mental clutter.

It also obscures.

Readability is a matter of habit.

Not only *Readability*,
but
Comprehensibility.

Functional, the OOP way

@ArgesRic

NUMERGENT

1. Structs can be a gateway drug.

2. Don’t mutate your objects.

~~Vector.Normalize()~~

Vector.Normalized

~~employee.Salary += 100~~

Employee SalaryChange(float v)

employee.SalaryChange(100)
.SetPosition(newTitle)
.SetSomeProp(true)

@ArgesRic

3. Write to Enumerables, not to Collections.

3.a. Use the functional facilities for result generation (Where, Select, etc).

4. Use immutable collections.

.Net: https://msdn.microsoft.com/en-us/library/system.collections.immutable(v=vs.111).aspx

Java: https://github.com/google/guava/wiki/ImmutableCollectionsExplained

http://clojure.org/

@ArgesRic

NUMERGENT

Where to do this?

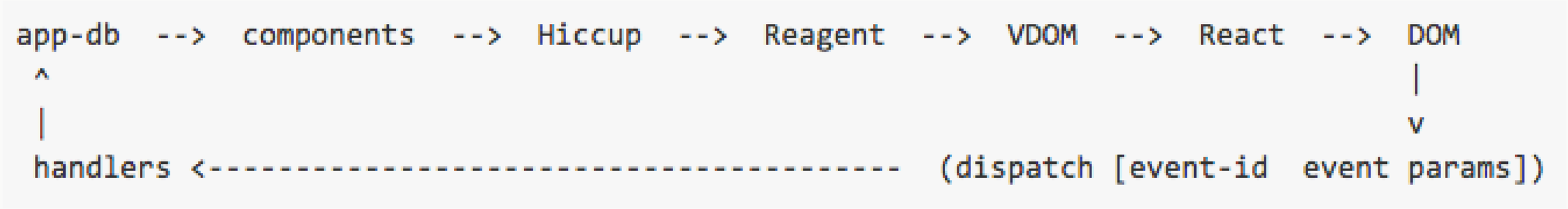
Business logic?

Logic is about reasoning according to strict principles of validity.

UI?

UI should be about
representing state.

re-frame’s event conveyor belt



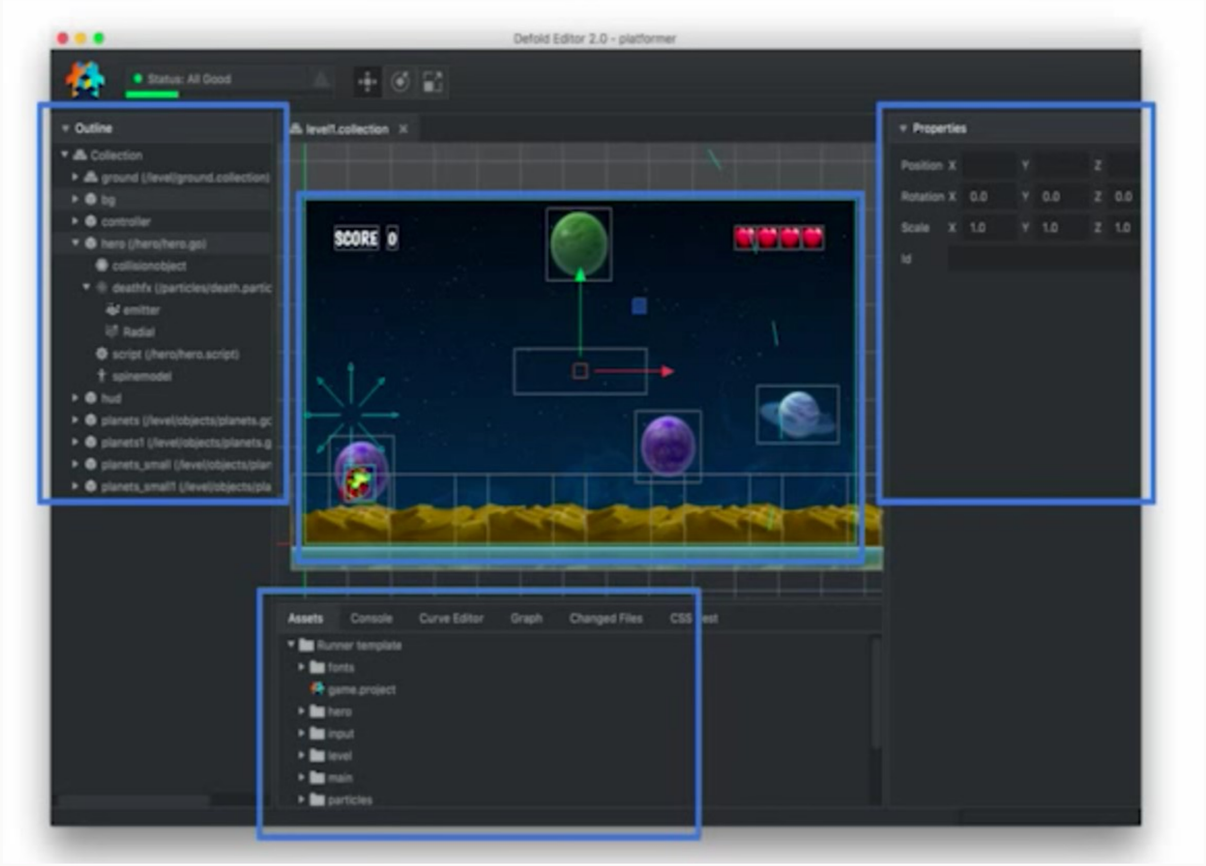
https://github.com/Day8/re-frame

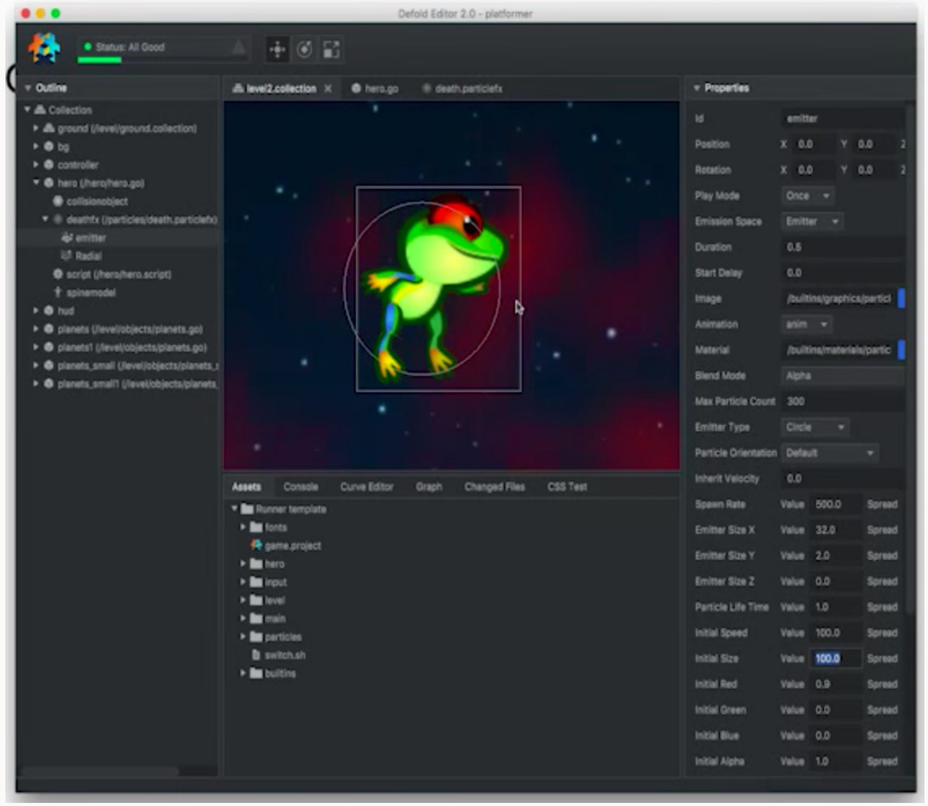
@ArgesRic

NUMERGENT

“Oh well, that’s all fine for two divs and a listbox”

Defold





https://www.youtube.com/watch?v=ajX09xQ_UEg

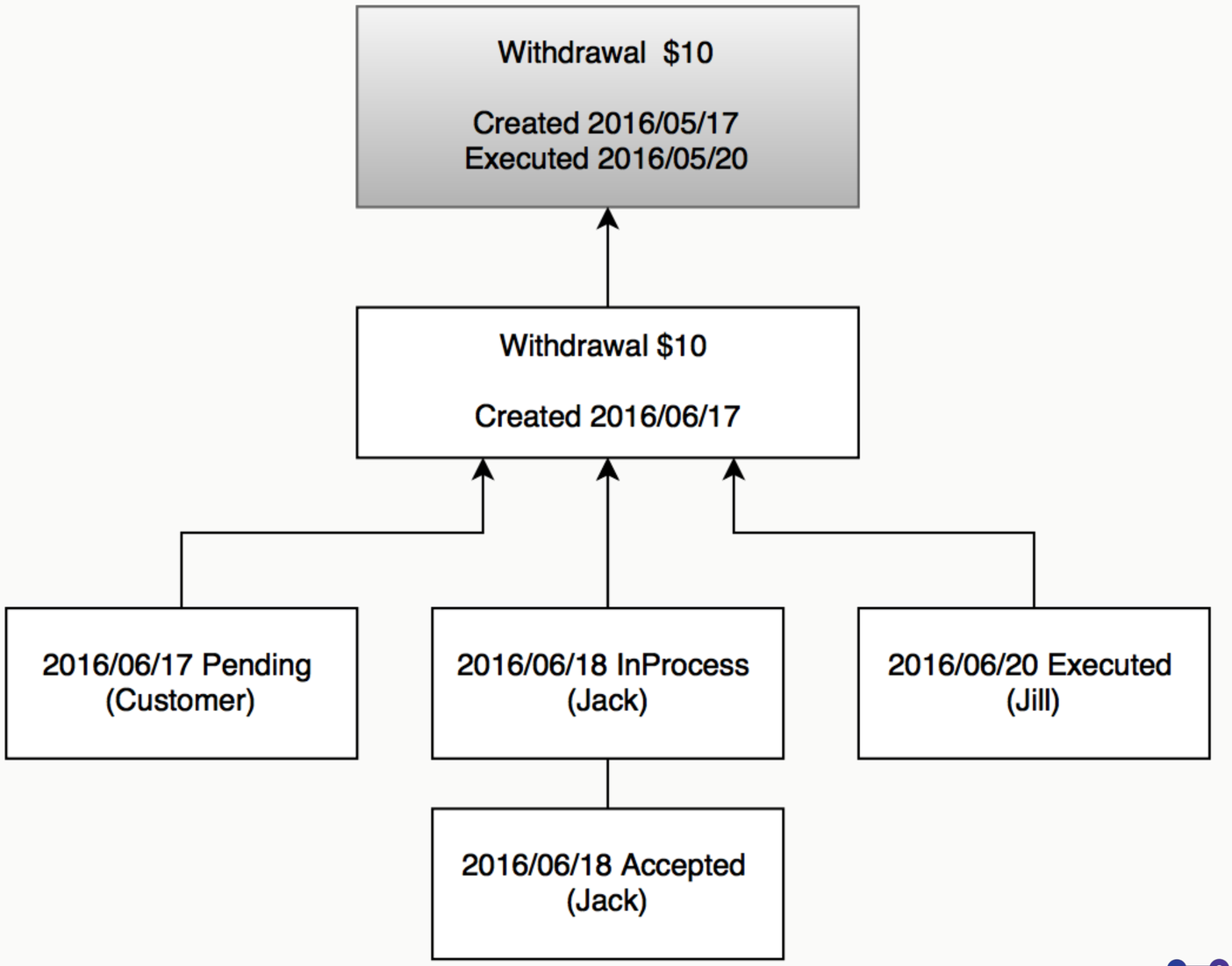
@ArgesRic

NUMERGENT

For a simple UI, anything will do.

For a complex UI,
immutability helps.

Data layer?


Withdrawal $10
Created 2016/05/17
Executed 2016/05/20
Withdrawal $10
Created 2016/06/17
2016/06/17 Pending
(Customer)
2016/06/18 InProcess
(Jack)
2016/06/20 Executed
(Jill)
2016/06/18 Accepted
(Jack)

Where NOT to do this?

Is RAM a concern?
Is the GC hit a concern?
Is raw performance a concern?

Why do this?

Trading off GC hit for a codebase that’s easier to reason about.

@ArgesRic

NUMERGENT

You'll never have to wonder about side-effects when refactoring again.

You'll write code that's easier to delete.

Easier threading.

Easier to offload processing.

"Who’s holding these objects?"

Who cares?

Immutable data lets you focus
on **comprehension**,
not **memorization**.

Conclusions

Immutability frees you to change your mind.

To be in control, you have to know.

Variability demands you take things on faith.

Try some functional patterns.

Replace trust with certainty.

Questions?

https://speakerdeck.com/ricardojmendez/flexibility-through-immutability

@ArgesRic

NUMERGENT

Thank you!

Ricardo J. Méndez
ricardo@numergent.com

https://speakerdeck.com/ricardojmendez/flexibility-through-immutability

@ArgesRic