Out of the Tar Pit

by Ben Moseley and Peter Marks

Nikolas Göbel

niko@clockworks.io

ETH zürich

Systems Group

clockworks

All problems with software stem from our limited ability to understand systems.

Complexity destroys the ability to understand a system.

Complexity :=

Number of things we have to keep in mind concurrently.

Methods of Understanding

black-box testing | **informal reasoning**

more errors found | **less errors created**

State := *What is the case?*

Control Logic := *When is it the case?*

Behaviour := *State over time.*

Complexity Causes

(1) possible states ~ scenarios to consider

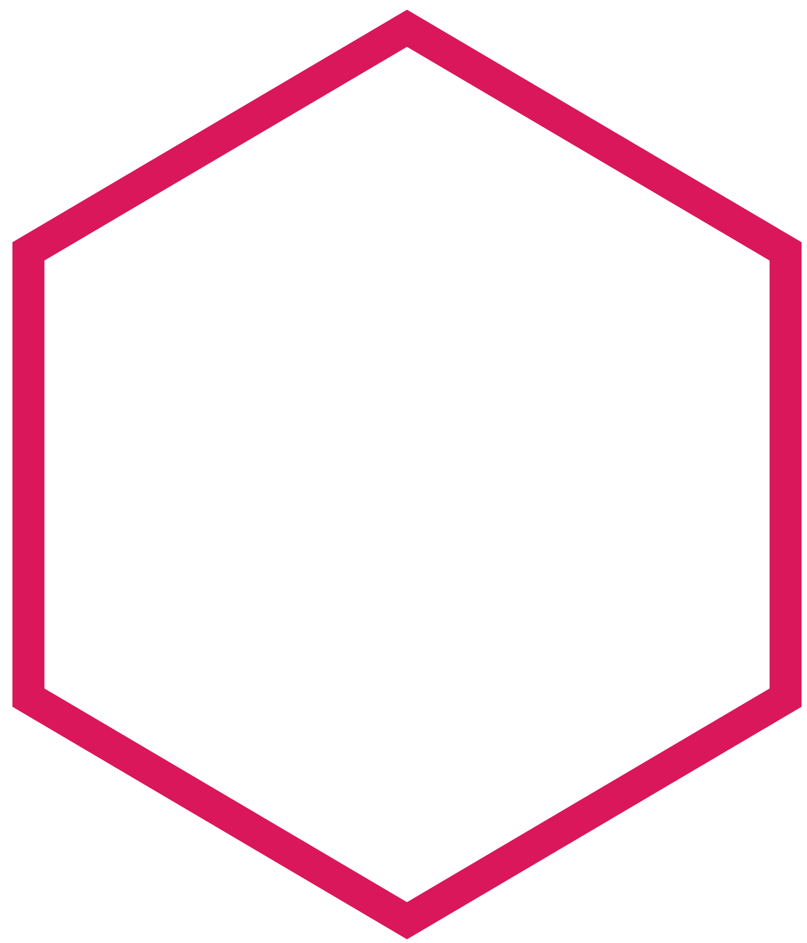
(2) control logic ~ interactions to consider

2nd Order Complexity

- State contaminates whatever it touches
- Over-specified control causes mismatch w/ runtime
- In general: all language powers will seep in eventually

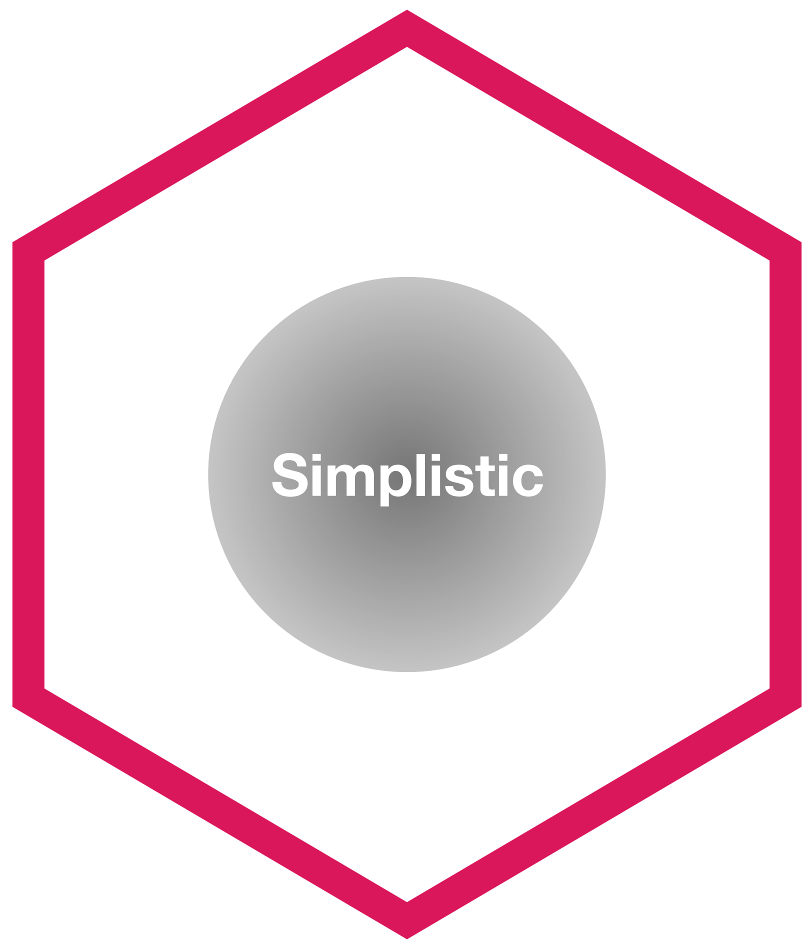
(1)

Essential vs Accidental

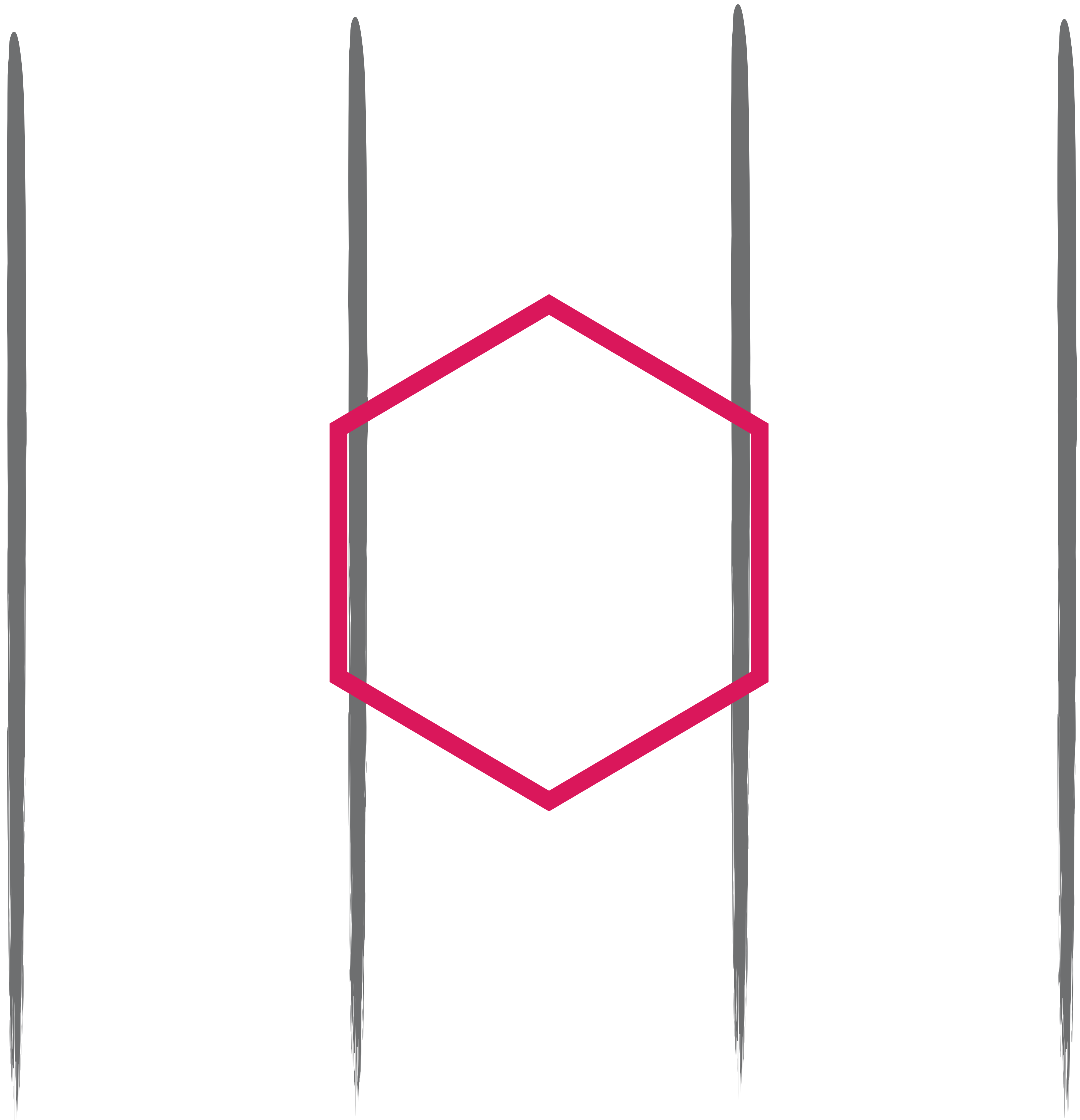


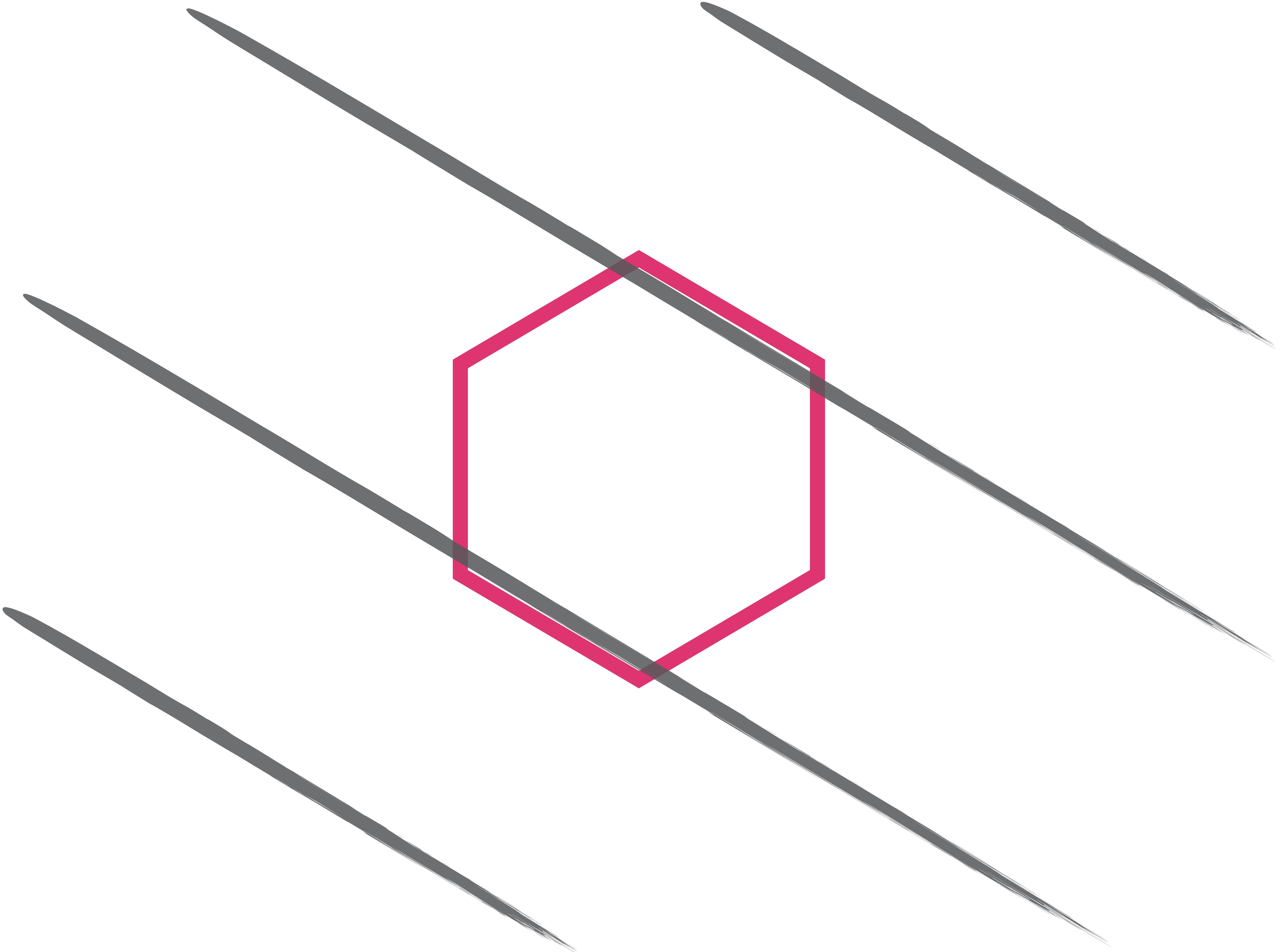
Problem Domain
(***Essential Complexity***)

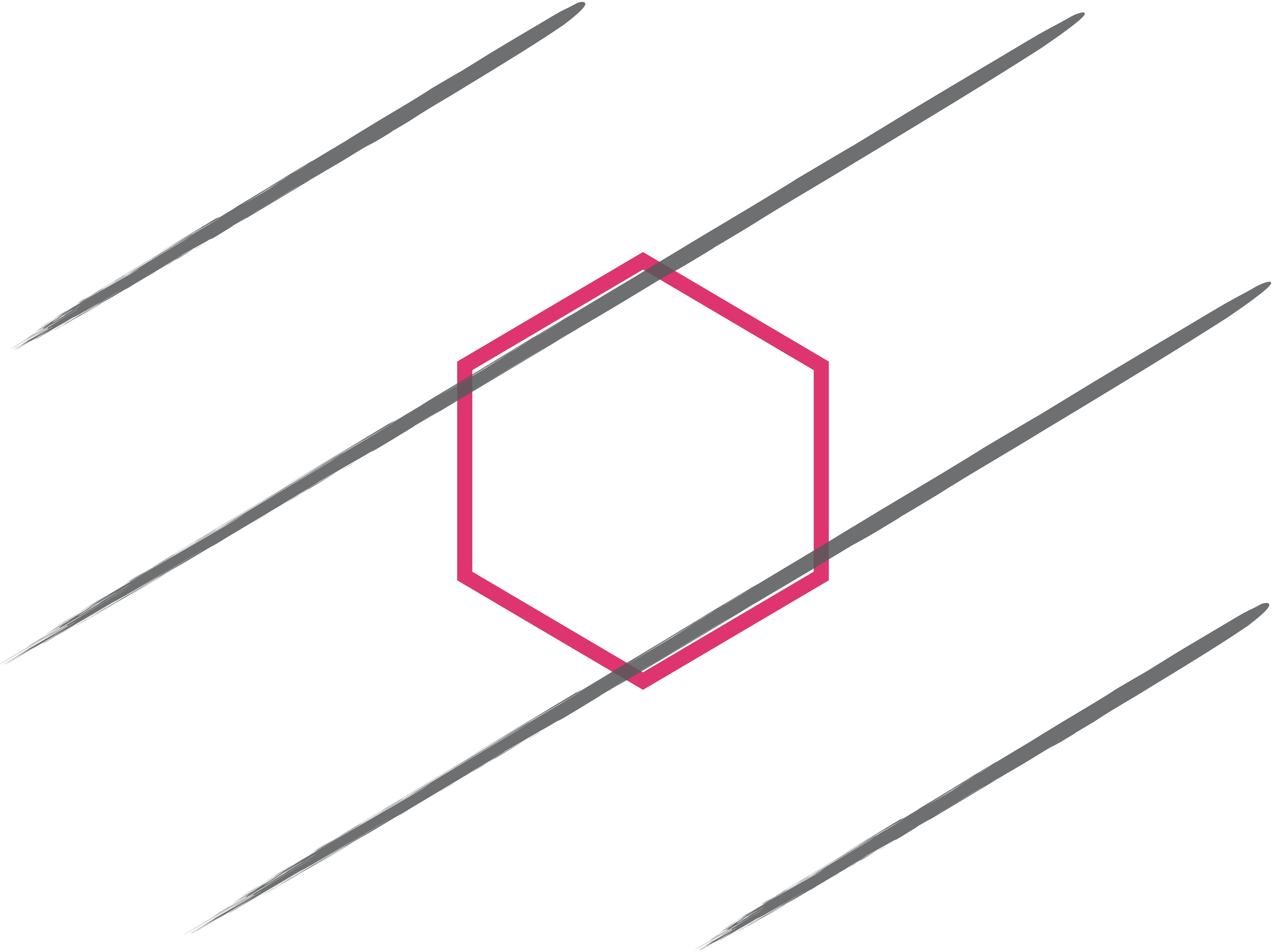
Possible Behaviours
(*Accidental Complexity*)


Simplistic

Optimal







(2)

Reducing Accidental State

Immutability

✔ De-couples state from behaviour

✘ Control is still a problem

✘ Doesn't reduce the amount of state

Functions

- ✔ Referential transparency
- ✔ No hidden state, everything visible in parameters

- ✘ Control is still a problem
- ✘ How much state can we stuff into parameters?

Transactions

✓ All the benefits of functions

✓ Atomicity allows us to ignore intermediate states

? All state pushed into a parameter (the database)

✗ Control is still a problem

Derived State (1)

```
{(x :mail/read-at 2019-01-01)
 (x :mail/read? true)
 (y :mail/read-at 2019-02-02)
 (y :mail/read? true)
 (user :unread-mails 2)}
```

Derived State (2)

```
(x :mail/read-at 2019-01-01) -> (x :mail/read? true)
```

$\underbrace{\texttt{(x :mail/read-at 2019-01-01)}}_{\text{essential}}$ $\underbrace{\texttt{(x :mail/read? true)}}_{\text{derived}}$

Derived State (2)

```
{(x :mail/read-at 2019-01-01)
 (y :mail/read-at 2019-02-02)
 (user :unread-mails 2)}


(defn is-read? [state id]
  (contains? state (id :mail/read-at? _)))
```

Derived State (3)

```
{x | (x :mail/read? true)} -> (user :unread-mails <count>)
```

derived derived

Derived State (4)

```
{(x :mail/read-at 2019-01-01)
 (y :mail/read-at 2019-02-02)}
```

```
(defn unread-count [state id]
  (->> (user-mails state id)
       (remove is-read?)
       (count)))
```

Reify Underlying Records (1)

```
{(player :position/x 5)
 (player :position/y 4)}
```

Reify Underlying Records (2)

```
[(player :move/up)
 (player :move/up)
 (player :move/right)
 (player :move/up)]
```

Reification can expose new functional dependencies.

Object-Orientation?

- squinting a bit, methods are per-object transactions
- problem 1: cross-object transactions
- problem 2: object identity

Out Of The <u>State</u> Pit

(1) Use immutable state wherever possible.

(2) Model essential state in relations.

(3) From those, derive all other state as views.

Note: Some more distinctions in the paper.

Essential derived vs *accidental derived,* mutable derived vs immutable derived.

(3)

Reducing Accidental Control

Overspecified Control

Most languages force us to deal with control (e.g. sequencing and branching).

- ✘ Impedance mismatches throughout the stack
- ✘ Concurrency and distribution. Please no.

Logic Programming

- ✓ Declarative: *what?* not *how?*
- ✓ Ordering follows entirely from data dependencies
- ✓ Programs fully reversible out-of-the-box

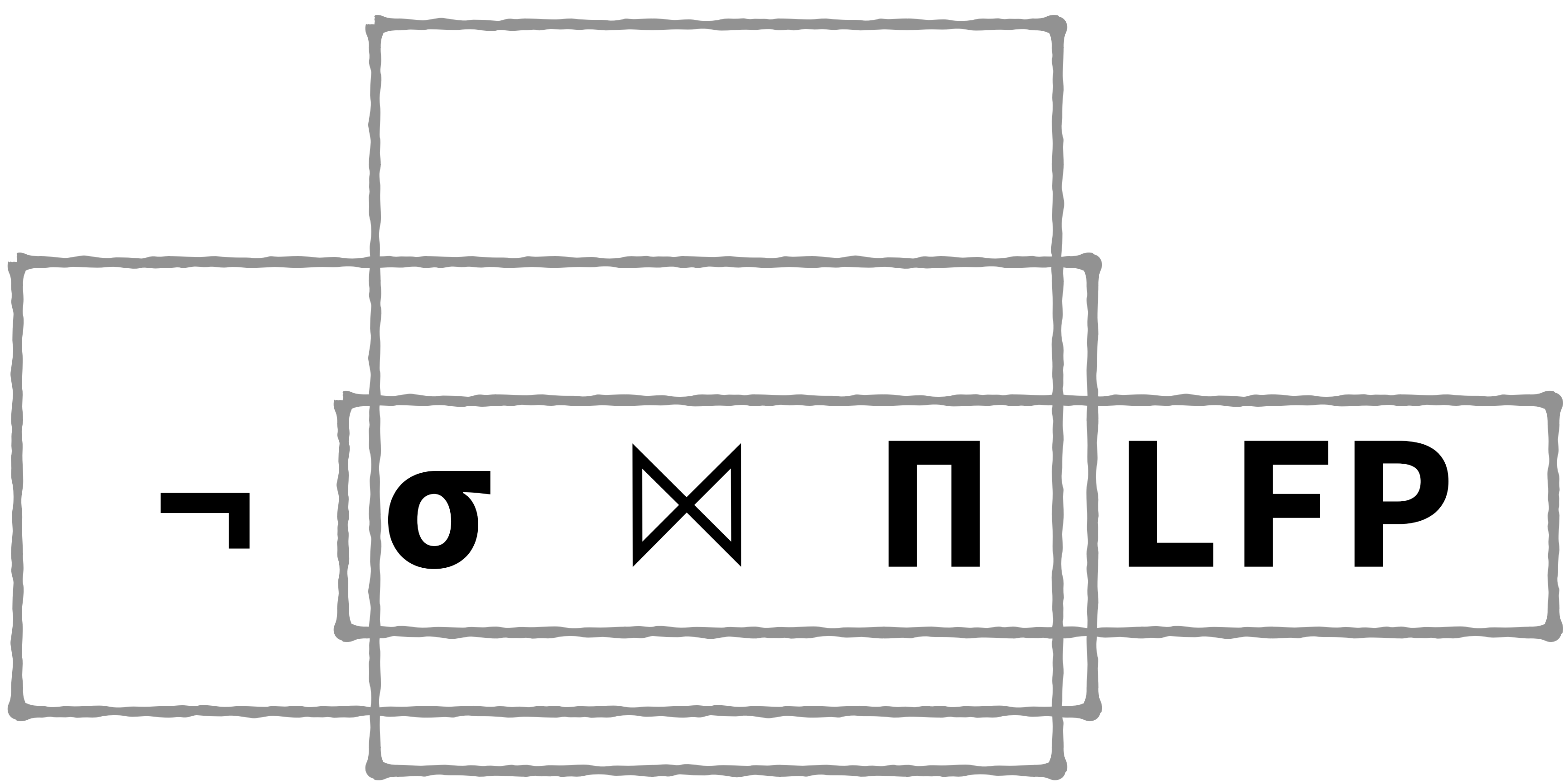
- ✗ Not really declarative in practice (e.g. Prolog)
- ✗ No efficient implementations

Relational Model

- ✓ Declarative views: join, select, project, …
- ✓ Data logically independent of physical layout
- ✓ Efficient and extensible

- ✗ Programs only reversible w.r.t references

Language Fragments



from Peter Alvaro, “I See What You Mean”

Language Power

We are used to classifying time and space complexity.

Classifying descriptive complexity and picking an appropriate language fragment can get us…

- ✔ Control-free, declarative programs
- ✔ Efficient, general-purpose implementations
- ✔ Fine-grained concurrency and distribution for free

Derived State – Revisited

```
{(x :mail/read-at 2019-01-01)
 (y :mail/read-at 2019-02-02)}


[[(mail/read? ?mail)
  [?mail :mail/read-at _]]

 [(unread-count ?user ?count)
  [?user :user/mail ?mail]
  (not (mail/read? ?mail))
  [(count ?mail) ?count]]]
```

Out Of The Control Pit

(1) Express derivations in appropriate language fragment.

(2) Prefer dataflow to control flow.

The relational model and it’s extensions provide a sweet-spot!

(4)

Feasibility

Out Of The Tar Pit

(1) Use immutable state wherever possible.

(2) Model essential state in relations.

(3) From those, derive all other state as views.

(4) Express derivations in appropriate language fragment.

(5) Prefer dataflow to control flow.

(1) Use immutable state wherever possible.

- first-class language support: Clojure, Elm
- DataScript, ImmutableJS, Mori
- append-only logs, Kafka

✓ Immutable data structures are efficient and proven in practice.

✓ Immutability as a concept is used on all levels of the stack.

(2) Model essential state in relations.

(3) From those, derive all other state as views.

(4) Express derivations in appropriate language fragment.

(5) Prefer dataflow to control flow.

(1) Use immutable state wherever possible.

(2) Model essential state in relations.

(3) From those, derive all other state as views.

- problem: consistent, incremental view maintenance

✓ works today, in the small: React + Redux

? goal: query power of DBs + performance of stream-processors

(4) Express derivations in appropriate language fragment.

(5) Prefer dataflow to control flow.

Differential Dataflow

An expressive programming framework that quickly updates computations when its inputs change.

- describe functional-relational dataflow computations
  (`map`, `reduce`, `join`, <u>`iterate`</u>, `filter`, …)

- efficient, incrementalized execution as inputs change

- distributable out-of-the-box (laptop -> cluster)

frankmcsherry.org

github.com/TimelyDataflow/differential-dataflow

3DF (Declarative Differential Dataflow)

- describe your views in Datalog
- views are incrementally maintained
- feeds from durable data sources

github.com/comnik/clj-3df github.com/comnik/declarative-dataflow

(1) Use immutable state wherever possible.

(2) Model essential state in relations.

(3) From those, derive all other state as views.

(4) Express derivations in appropriate language fragment.

(5) Prefer dataflow to control flow.

- previous attempts: tuple stores, coordination languages
- lots of R&D dollars to build on: databases, query optimization, …
- CALM work by Hellerstein and Alvaro
- relational extensions: probabilistic, temporal, constraint solving, …

✔ It's all there for the taking!

2nd Order Benefits

- ✓ History and Analytics are first-class
- ✓ Concurrency and distribution out-of-the-box
- ✓ Tracing and Provenance, Meta-Queries
- ✓ Efficient runtimes, coordination free distribution

Adoption?

- Log-based architectures, event-sourcing
- Stream Processing
- Web Frontends, real-time sync front<->back
- Distributed correctness, formal verification

Into The Tar Pit

- State
- Conditionals
- Objects and Methods
- Classes and Inheritance
- Pattern matching
- Actors, Futures, Async/Await, Promises, …
- ORMs
- …

Out Of The Tar Pit

(1) Use immutable state wherever possible.

(2) Model essential state in relations.

(3) From those, derive all other state as views.

(4) Express derivations in appropriate language fragment.

(5) Prefer dataflow to control flow.

Pointers

- “Simple Made Easy”, by Rich Hickey
- “I See What You Mean”, by Peter Alvaro
- github.com/TimelyDataflow
- github.com/comnik/declarative-dataflow
- github.com/tonsky/datascript
- “The Declarative Imperative”, by Joseph Hellerstein