Event Sourced Story

CQRS

ES

DDD

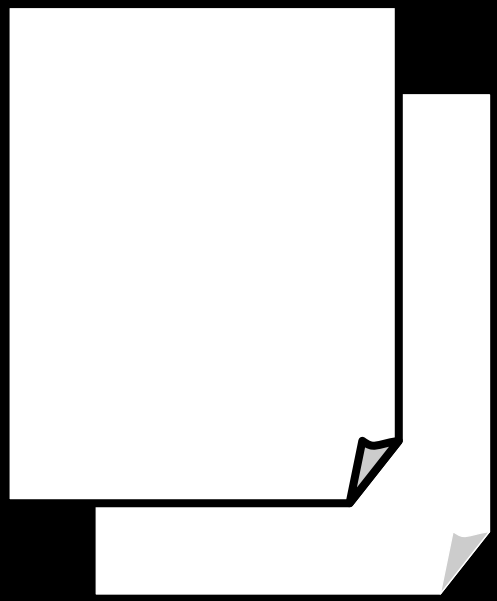
EDA

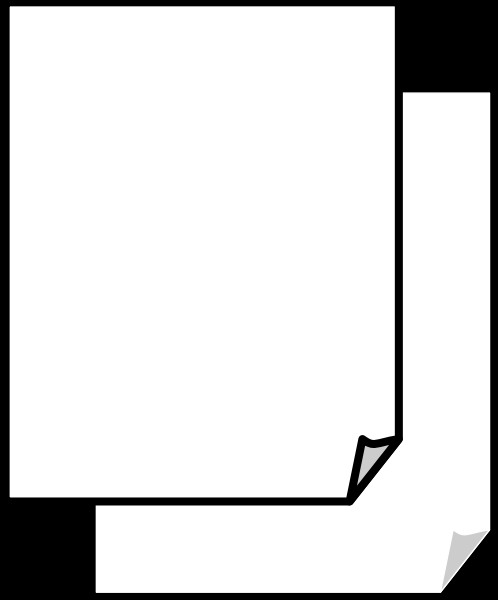
CQRS

ES

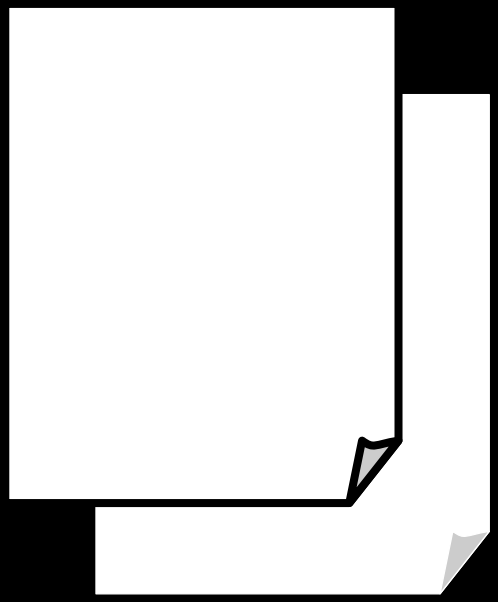
DDD

EDA



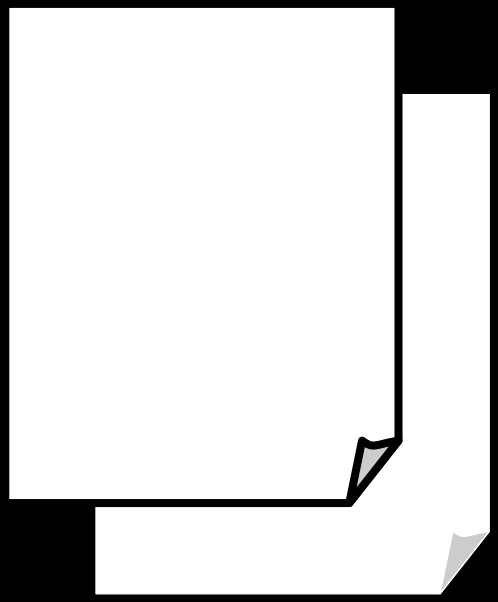


id		

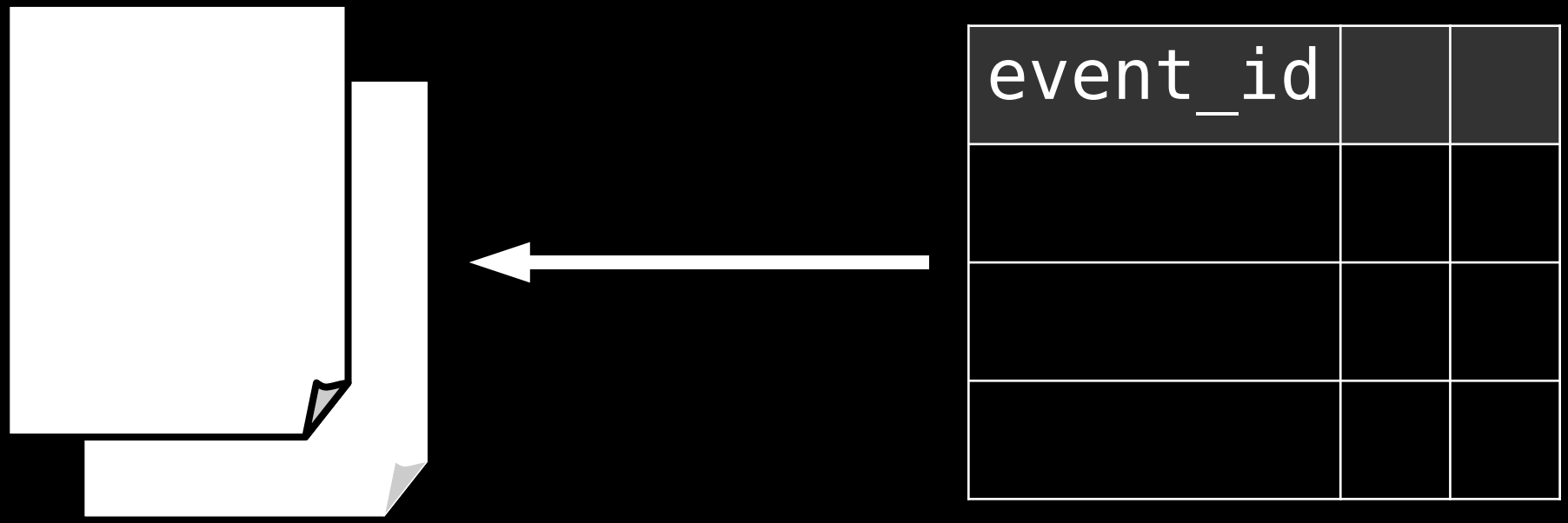


id		

event_id		



event_id		


event_id

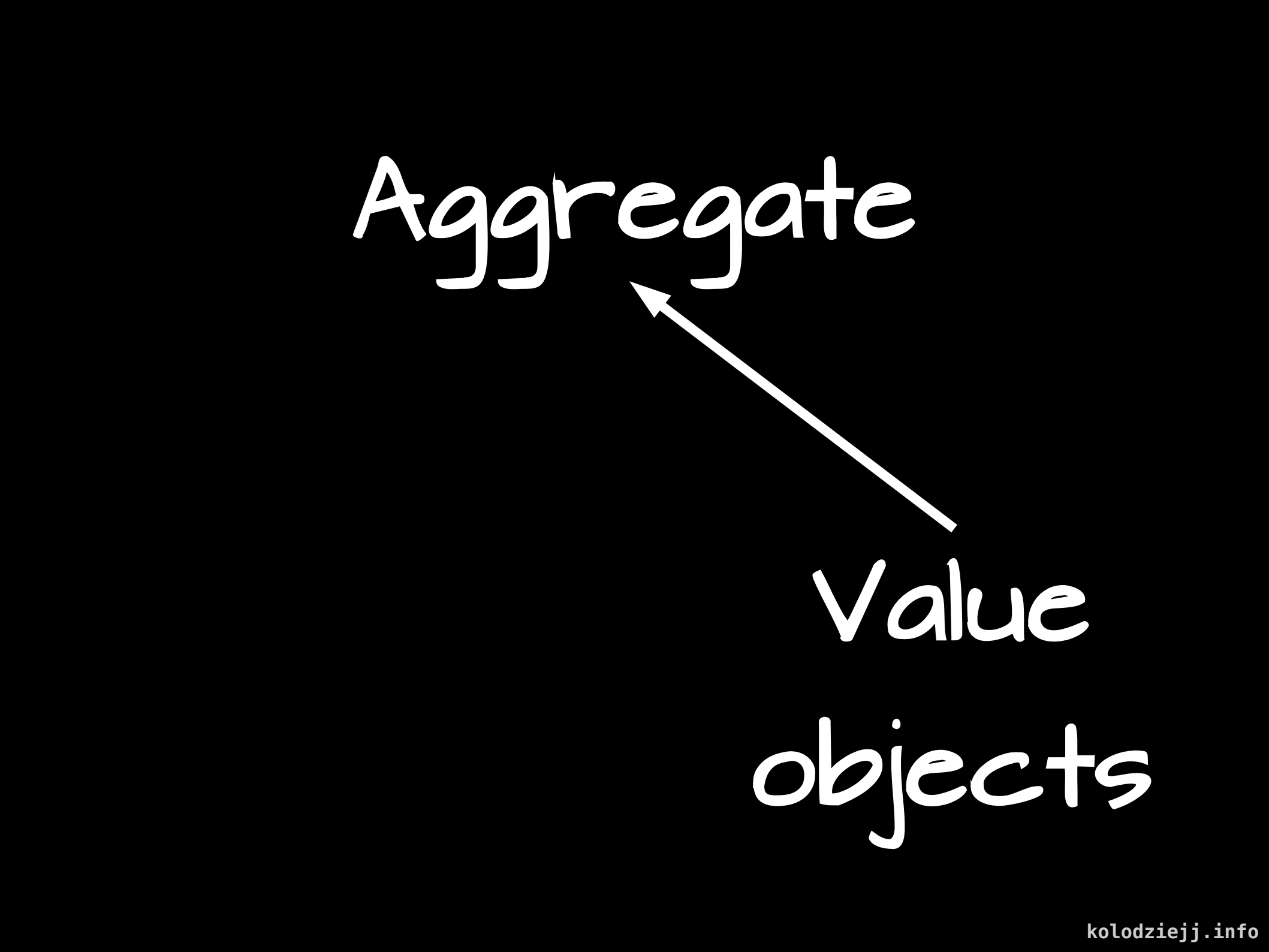
Aggregate

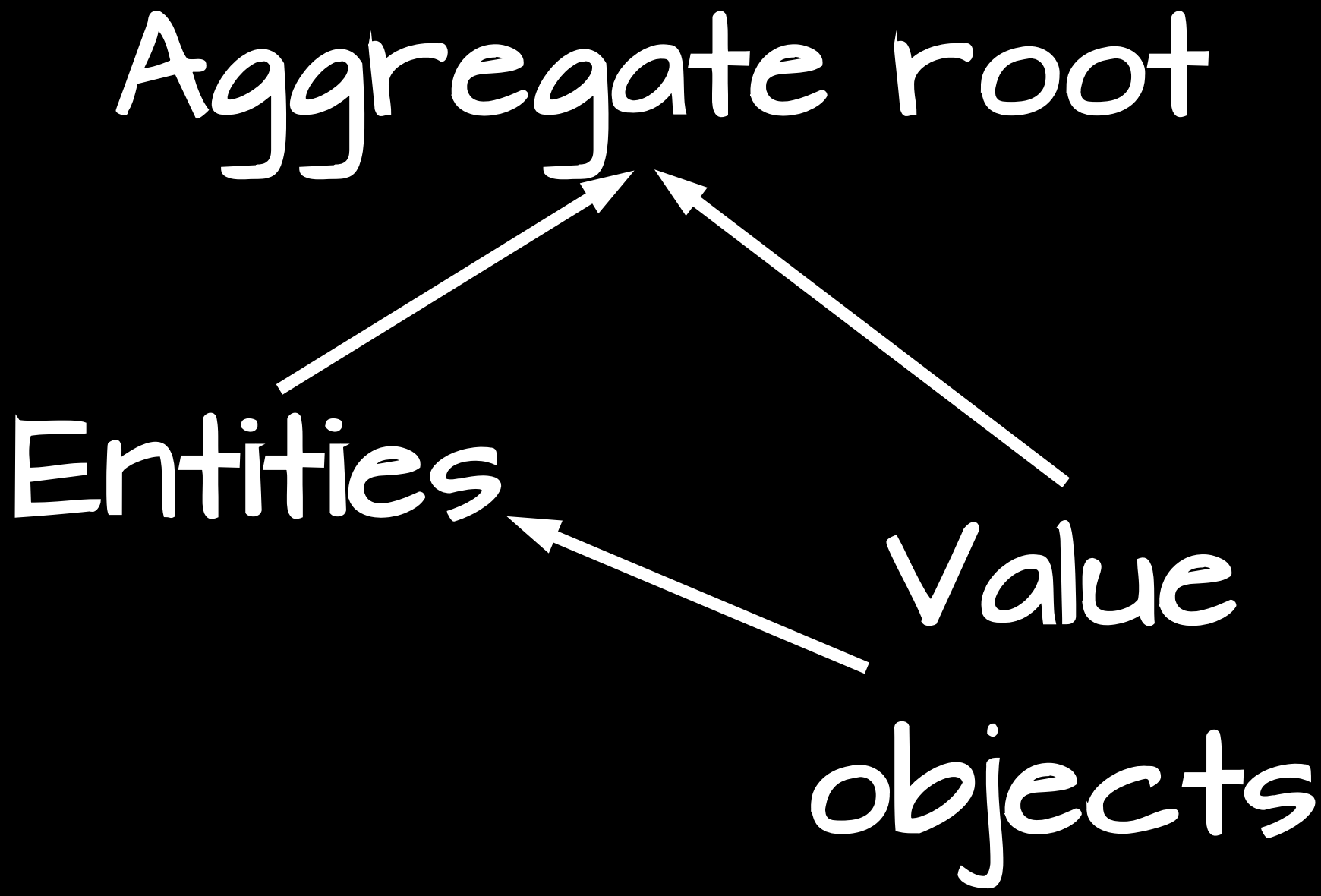
Aggregate

- replaying current state
  - event handlers

Aggregate

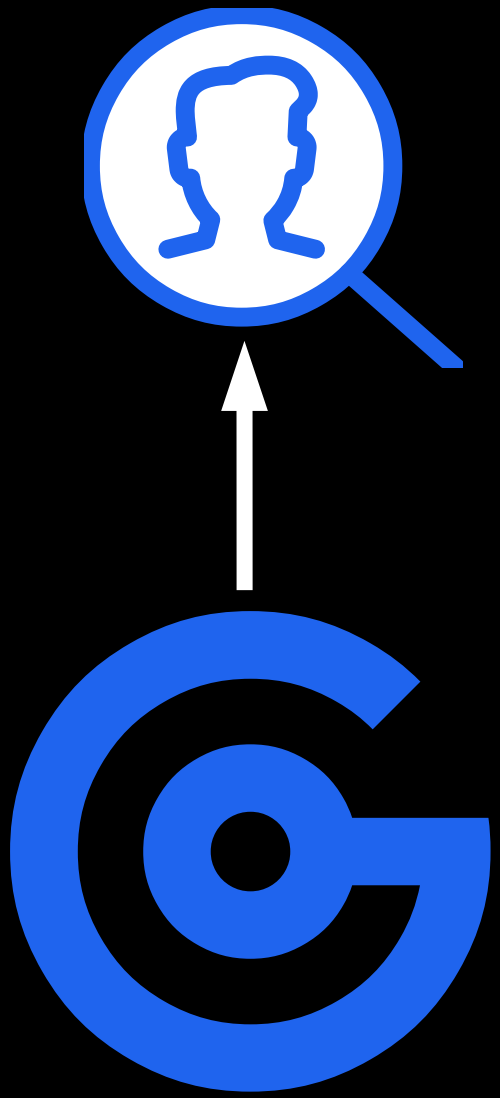
- replaying current state
  - event handlers
- executing commands

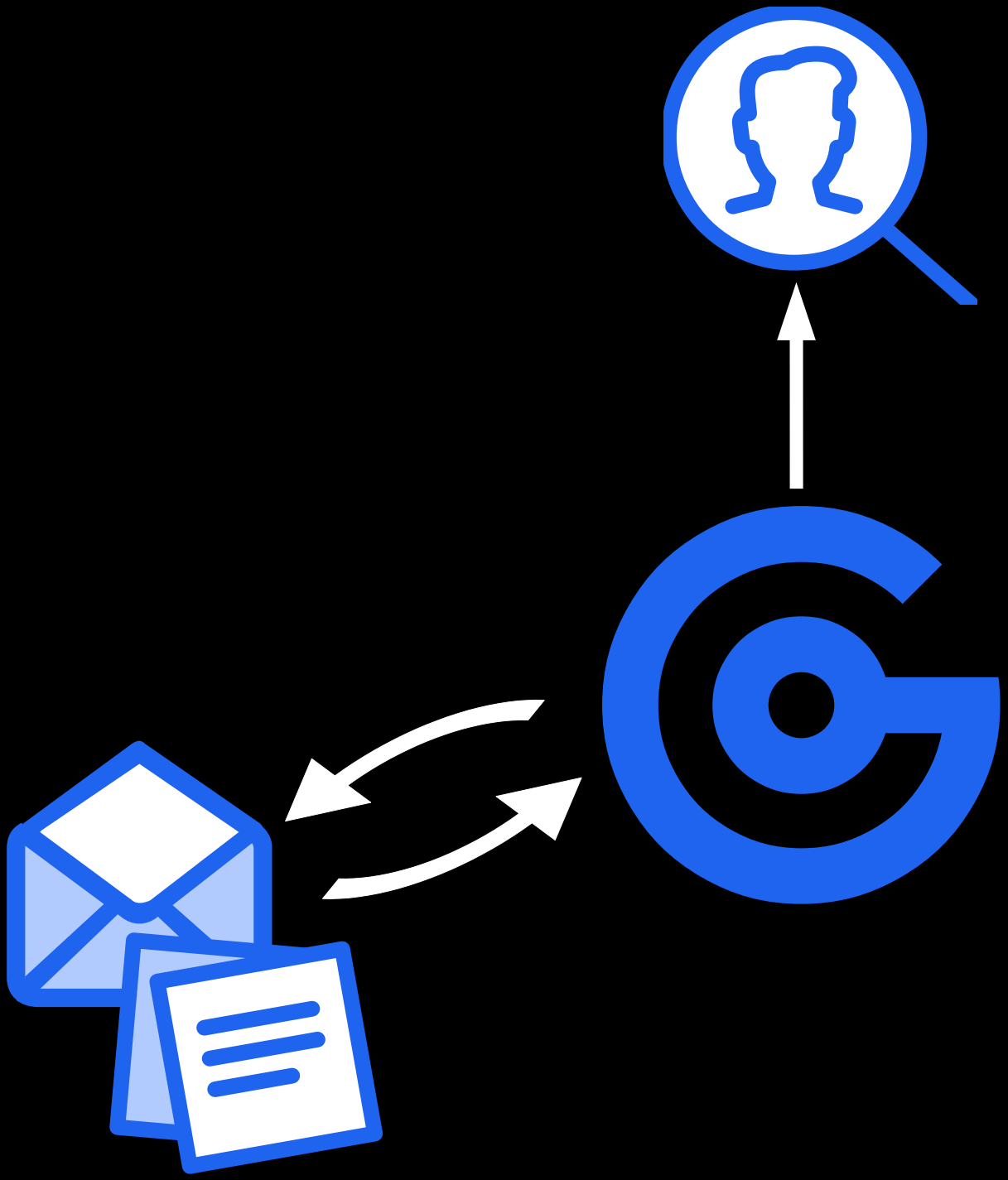

Aggregate
Value
objects

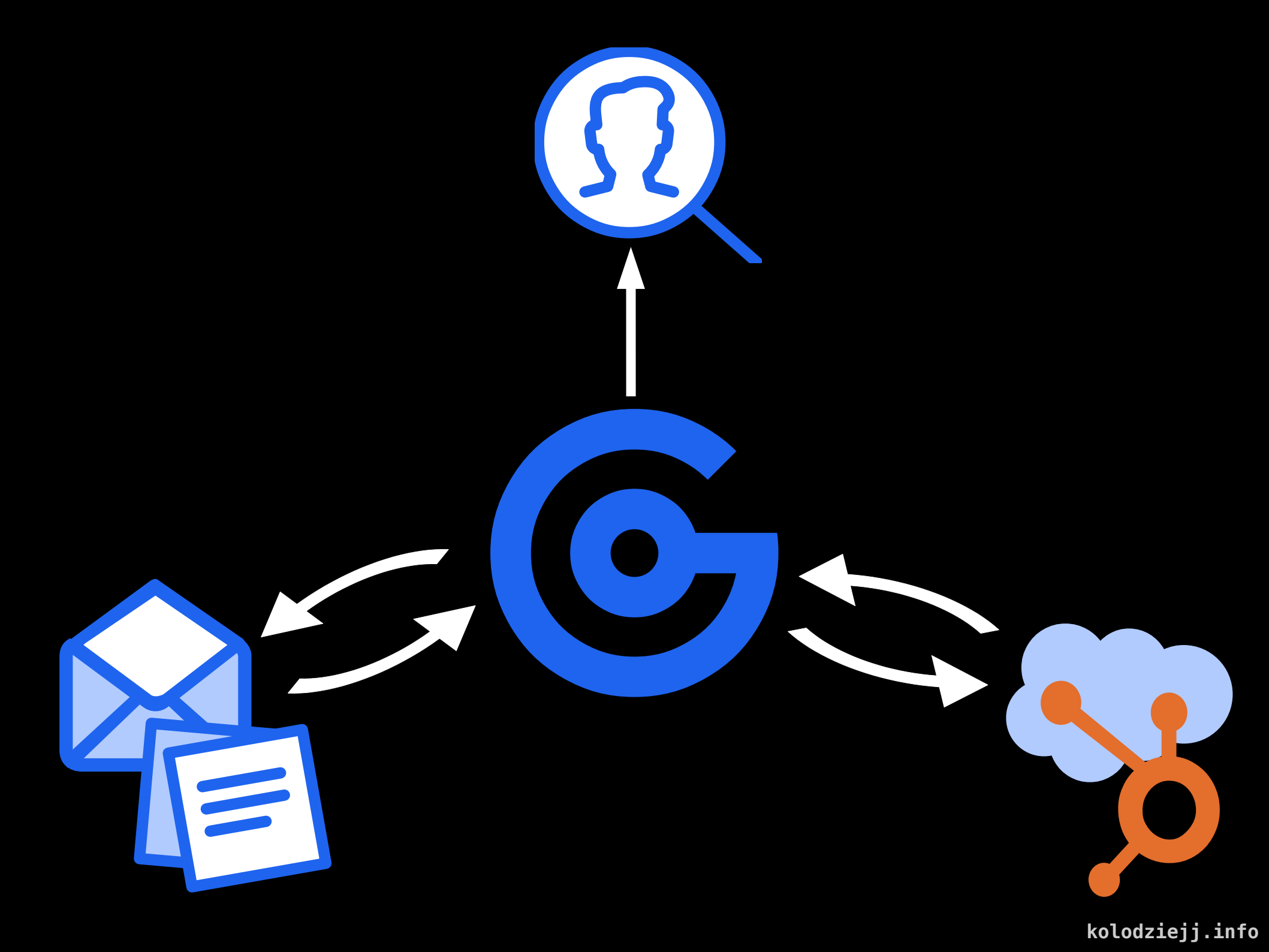

Aggregate root
Entities
Value objects

Why, tho?

Growbots






kolodziejj.info

Relevant?

- Web application

Relevant?

- ~~Web application~~
- Web services

Relevant?

- ~~Web application~~
- ~~Web services~~
- Using (mostly) RDBMSs as storage

Relevant?

- ~~Web application~~
- ~~Web services~~
- ~~Using (mostly) RDBMSs as storage~~

Problems

- Complicated lifecycle of objects

Problems

- Complicated lifecycle of objects
- Business logic based on events

Problems

- Complicated lifecycle of objects
- Business logic based on events
- Issues hard/impossible to trace

Problems

- Complicated lifecycle of objects
- Business logic based on events
- Issues hard/impossible to trace
- Need to expose history to users

Learn

Learn

think

Learn

think

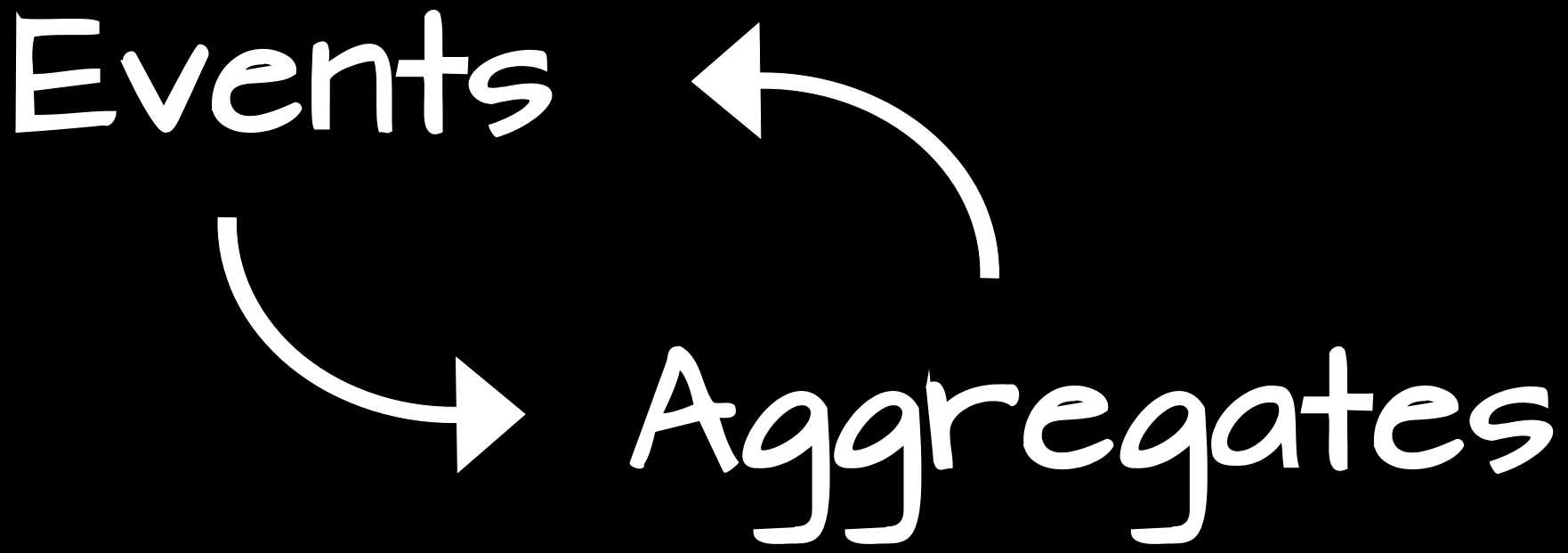
talk

Ubiquitous language

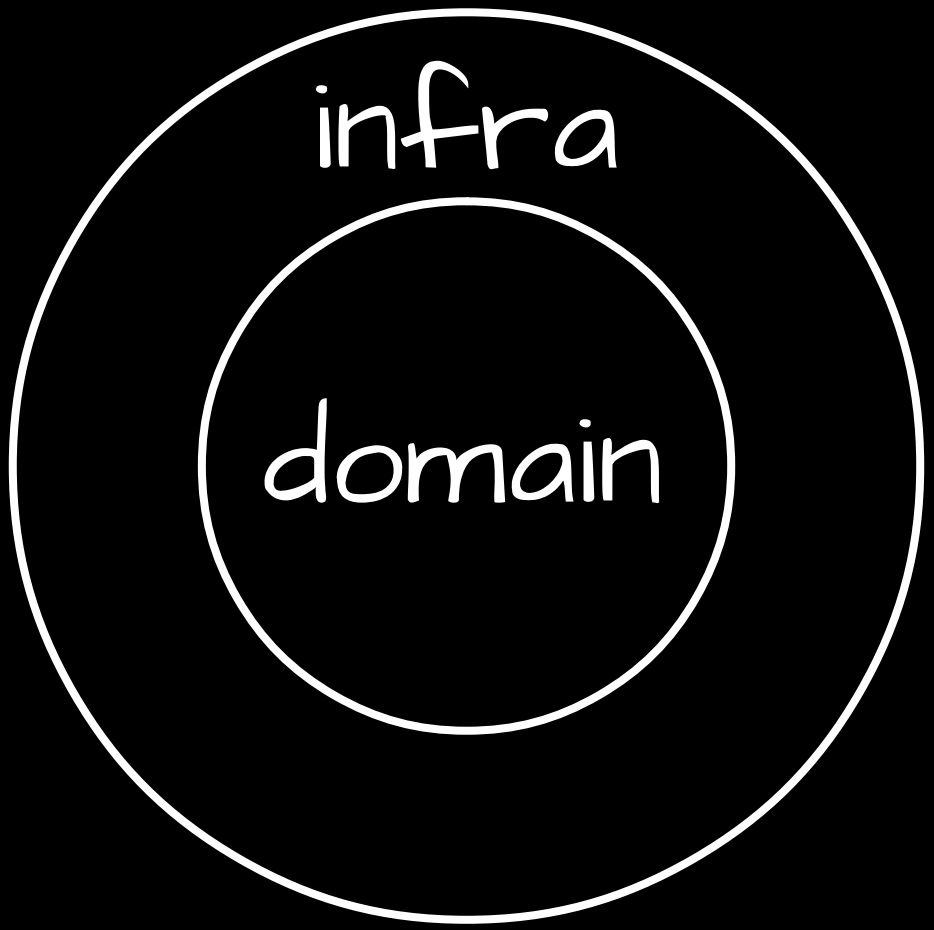
Domain

Events

Aggregates


Events
Aggregates

Code


infra
domain

```
class MyEvent:
    def __init__(self, ...):

    @classmethod
    def unserialize(
        cls, payload: dict, …
    ) -> “MyEvent”:

    def serialize(self) -> dict:
```

```
class MyEvent:
    def __init__(self, ...):

    @classmethod
    def unserialize(
        cls, payload: dict, …
    ) -> “MyEvent”:

    def serialize(self) -> dict:
```

```
class MyEvent:
    def __init__(self, ...):

    @classmethod
    def unserialize(
        cls, payload: dict, …
    ) -> “MyEvent”:

    def serialize(self) -> dict:
```

```python
class MyEvent:
    def __init__(self, ...):

    @classmethod
    def unserialize(
        cls, payload: dict, …
    ) -> “MyEvent”:

    def serialize(self) -> dict:
```

```
class MyEvent:
    def __init__(self, ...):

    @classmethod
    def unserialize(
        cls, payload: dict, …
    ) -> “MyEvent”:

    def serialize(self) -> dict:
```

```
class MyAggregate:
    @classmethod
    def create(cls, ...) ->
“MyAggregate”:

    def do_stuff(self) -> None:

    def apply_event(self, event):
```

```
class MyAggregate:
    @classmethod
    def create(cls, ...) ->
“MyAggregate”:

    def do_stuff(self) -> None:

    def apply_event(self, event):
```

```
class MyAggregate:
    @classmethod
    def create(cls, ...) ->
“MyAggregate”:

    def do_stuff(self) -> None:

    def apply_event(self, event):
```

```
class MyAggregate:
    @classmethod
    def create(cls, ...) ->
“MyAggregate”:

    def do_stuff(self) -> None:

    def apply_event(self, event):
```

```python
class MyRepository:
    def persist(
        self, aggregate,
    ) -> None:
        self._insert_new_events(
            aggregate
            .pop_pending_events()
        )
```

```
class MyRepository:
    def persist(
        self, aggregate,
    ) -> None:
        self._insert_new_events(
            aggregate
            .pop_pending_events()
        )
```

```python
class MyRepository:
    def persist(
        self, aggregate,
    ) -> None:
        self._insert_new_events(
            aggregate
            .pop_pending_events()
        )
```

```
class Aggregate:
    def pop_pending_events(self)
-> List[Event]:
        return (
            self._pending_events
        )
```

Testing

```
def test_my_event_serialize():
    event = ...

    assert event.serialize() == …

def test_my_event_unserialize():
    assert MyEvent.unserialize()
== …
```

```
def test_my_agg_do_stuff():
    aggregate = ...

    aggregate.do_stuff()

    assert (
        aggregate._pending_events
        == ...
    )
```

```
def test_my_agg_stuff_done():
    aggregate = ...
    event = ...

    aggregate.apply_event(event)

    assert agg....
```

Event Store

Event store

- RDBMS (PostgreSQL, MySQL...)

Event store

- RDBMS (PostgreSQL, MySQL...)
- specialized store (Event Store)

Event store

- RDBMS (PostgreSQL, MySQL...)
- specialized store (Event Store)
- ...whatever works for you

Event store

- ID (integer, auto increment)

Event store

- ID (integer, auto increment)
- aggregate ID

Event store

- ID (integer, auto increment)
- aggregate ID
- version (integer, from app)

Event store

- ID (integer, auto increment)
- aggregate ID
- version (integer, from app)
  - UNIQUE(aggregate ID, version)

Event store

- ID (integer, auto increment)
- aggregate ID
- version (integer, from app)
  - UNIQUE(aggregate ID, version)
- payload (JSON/string)

Event store, optional

- event type (string)
- tenant ID
- correlation ID
- timestamp

Ordering vs. concurrency

Concurrency

- pessimistic (explicit locks)

Concurrency

- pessimistic (explicit locks)
- or optimistic
  - UNIQUE(aggregate ID, version)

Projections

Projections

Events

id	event

Read model

Projections

- built synchronously

Projections

- built synchronously
- or asynchronously

Projections

- built synchronously
- or asynchronously
  - eventual consistency

Events evolution

Events evolution

- payload as JSON/string helps

Events evolution

- payload as JSON/string helps
- versioning: new event type, e.g. stuff_done_v2

Events evolution

- payload as JSON/string helps
- versioning: new event type, e.g. stuff_done_v2
- tech debt factory

Data inspection

Immutability

Immutability

Yes - in regular operation :)

Introducing ES in an existing app

Some numbers

Some numbers

- 300M rows
- 640 GB
- 50-100 ms 95th perc.
  aggregate fetching
- <= 200 new events/s

Stuff we don't even

- Compensating events

Stuff we don't even

- Compensating events
- Snapshots

Stuff we don't even

- Compensating events
- Snapshots
- Es library/framework

Where not to ES

Related

- DDD (Domain-Driven Design)
- Repository pattern
- Layered architecture
- CQRS (Command-Query Responsibility Segregation)
- EDA (Event-Driven Architecture)

Cons

Cons

- complexity

Cons

- complexity
  - eventual consistency

Cons

- complexity
  - eventual consistency
  - "unremovable" event handlers

Cons

- complexity
  - eventual consistency
  - „unremovable" event handlers
  - architectural overhead

Pros

- History as first-class citizen

Pros

- History as first-class citizen
- Better structure for complicated business logic

Pros

- History as first-class citizen
- Better structure for complicated business logic
- Temporal thinking

Pros

- History as first-class citizen
- Better structure for complicated business logic
- Temporal thinking
- Interactions awareness

Pros

- History as first-class citizen
- Better structure for complicated business logic
- Temporal thinking
- Interactions awareness
- Domain understanding

Thank you! :)

kolodziejj.info/talks/es

@unit03

Growbots