

---

WHAT IS THE  
STATE OF MICROSERVICES  
IN 2022?

---



**MICROSERVICES ANTI PATTERNS!**

WHAT WE'VE GOT  
HERE IS FAILURE TO  
**DESIGN**

A dramatic scene from a movie. In the foreground, a man with dark hair and a mustache lies face down on a large, light-colored rock. He is wearing a light-colored shirt and dark trousers. His head is resting on his arms, which are propped up on the rock. In the background, two other men stand on a higher, rocky ledge. They are both wearing dark, long-sleeved shirts and dark trousers. One man has his hands clasped in front of him, while the other stands with his hands at his sides. The background shows a vast, hilly landscape under a clear blue sky.

ON THE CRITERIA TO BE USED  
IN DECOMPOSING SYSTEMS INTO MODULES

D. L. Parnas

Department of Computer Science  
Carnegie-Mellon University  
Pittsburgh, Pa.

August, 1971

# FUNCTIONAL DECOMPOSITION

## Changes

Struggles with knowledge spread across all modules, so often each module has to change in response to a desired functional change.

## Independent Development

Dependent on shared data formats and schema. Must be jointly defined and agreed upon across multiple groups.

## Comprehensibility

You need to know something about how all of the modules work to understand the whole system.

# CAPABILITY DECOMPOSITION

## Changes

Usually isolates a change to a single module.

## Independent Development

Has abstract interfaces that encapsulate the work to be done..

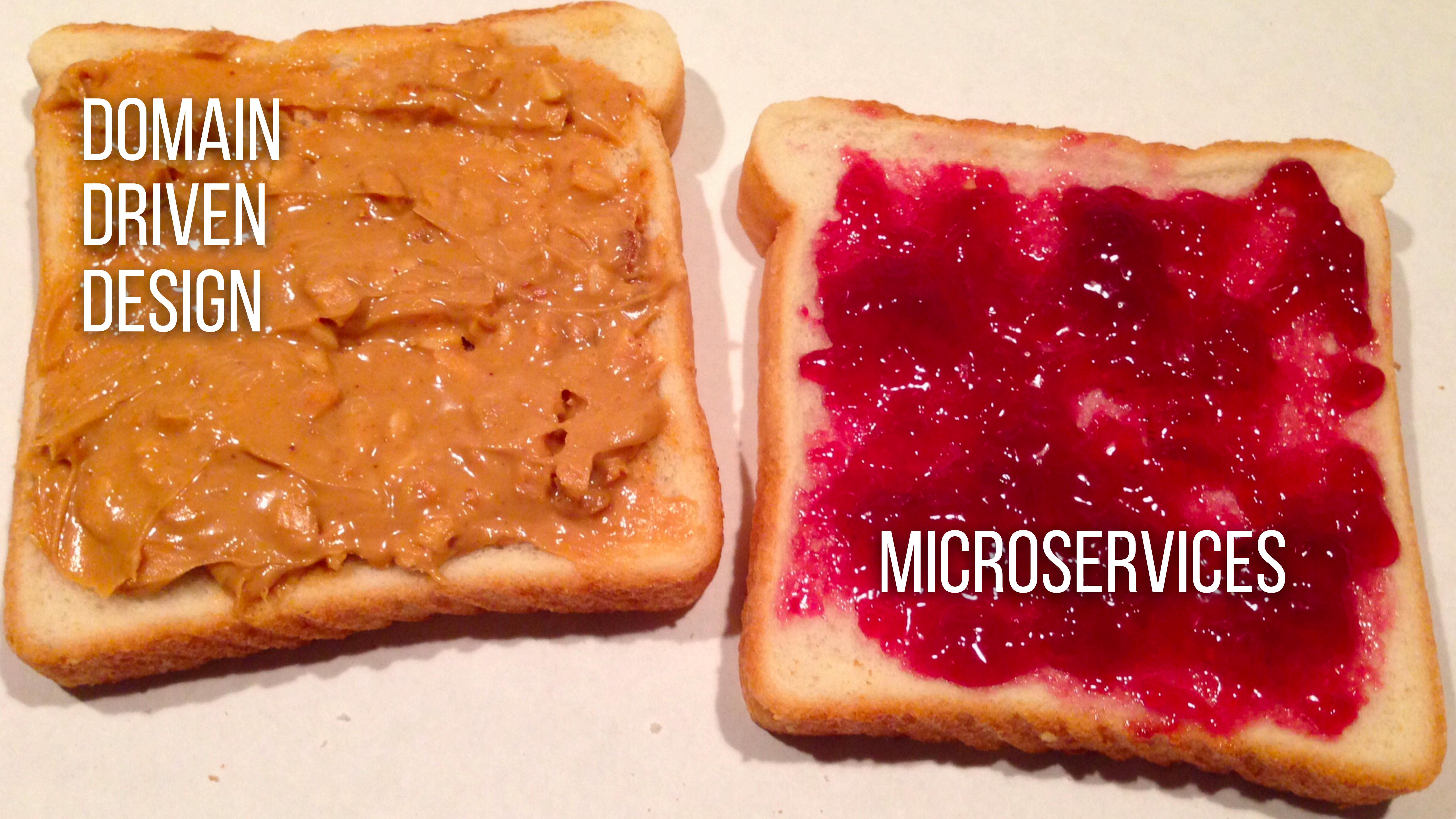
## Comprehensibility

You can understand modules independently.

Decomposition Techniques

---

THE FUNDAMENTAL WAYS IN WHICH WE DESIGN  
MODULES EFFECTIVELY HAS NOT CHANGED IN THE  
LAST 50 YEARS.

A photograph of two slices of white bread. The slice on the left is spread with a thick layer of peanut butter, which has been partially smeared onto the adjacent slice. The slice on the right is spread with a thick layer of red jam. Both slices are resting on a light-colored surface.

DOMAIN  
DRIVEN  
DESIGN

MICROSERVICES

# THE UBIQUITOUS LANGUAGE

---

When discussing the **domain**, and I use a **term**,  
and you use **exactly the same** term, we mean  
**precisely the same** thing.

---

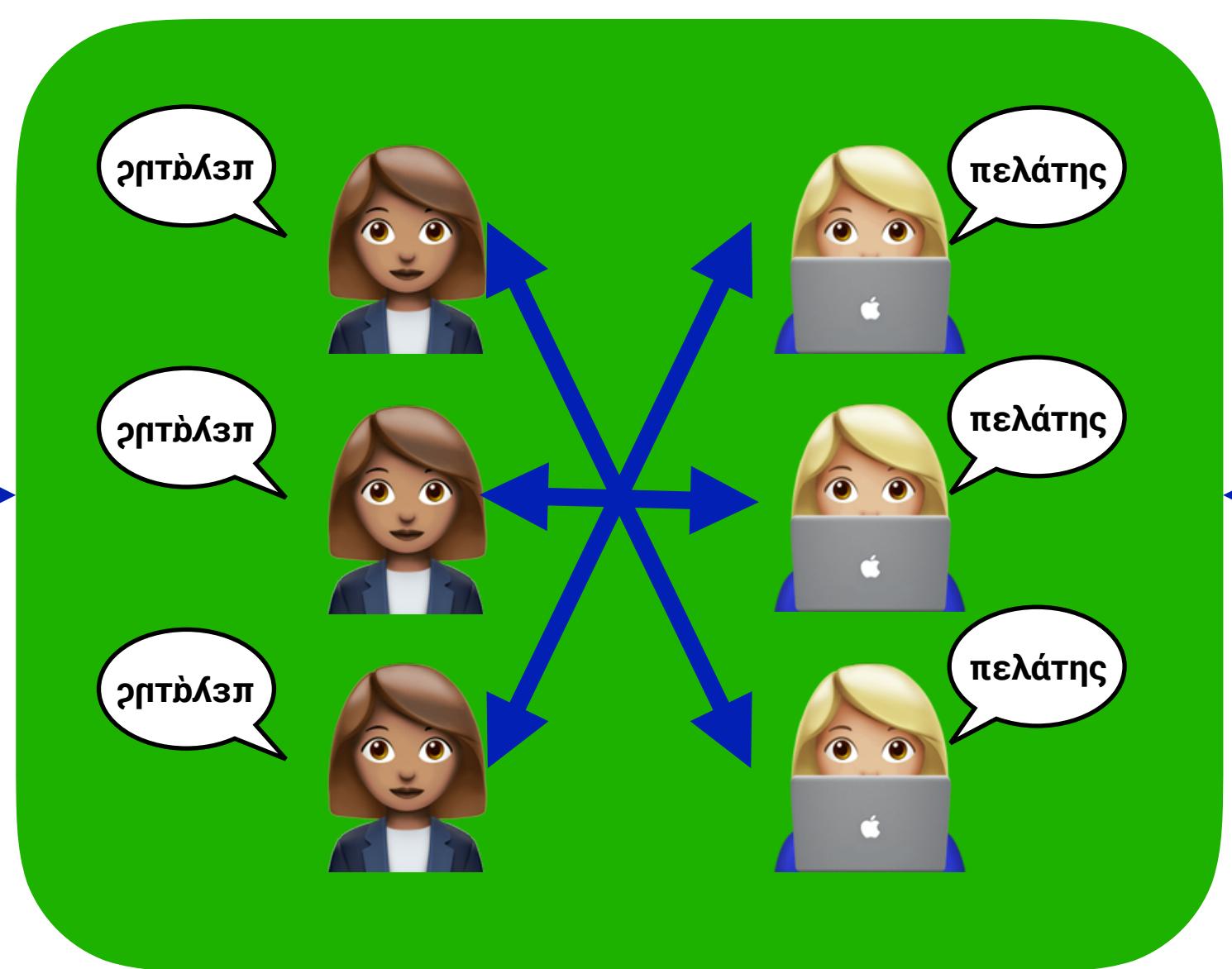
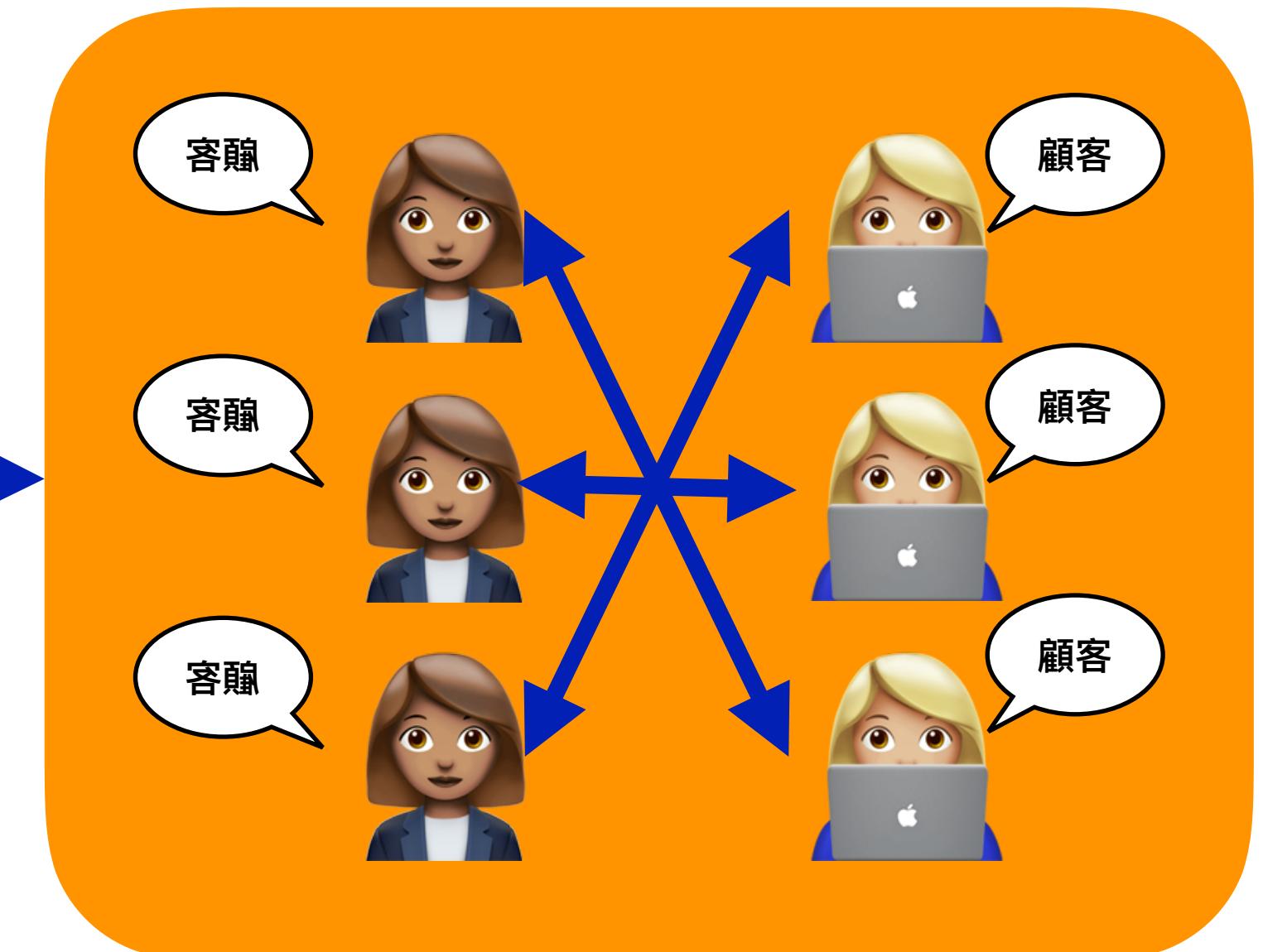
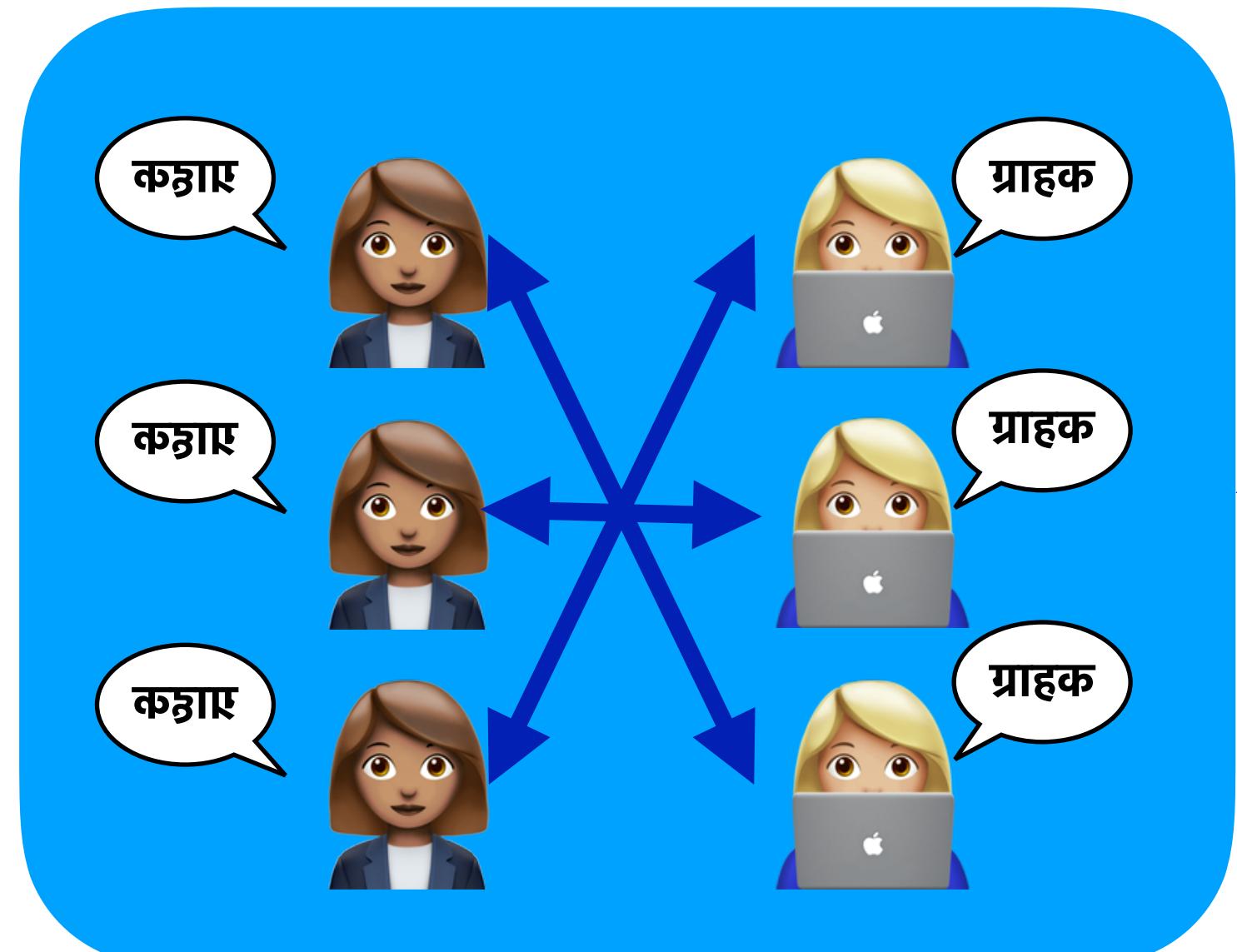


**STRATEGIC DESIGN - THE VIEW FROM 35,000 FEET**

A close-up view of a vibrant patchwork quilt. The quilt features a variety of patterns, including orange and white chevrons, blue and white chevrons, blue stripes, and a dark fabric with blue and orange teardrop shapes. Several panels contain handwritten text: one panel has "dream big little one" with three red asterisks; another panel has "24 July 2014"; and a third panel at the bottom left has "Aunty Kas" and a small red heart. The quilt is composed of many triangles and rectangles.

# FINDING THE SEAMS

# INTEGRATING THE PIECES



# TACTICAL DESIGN?

Now it's time to start speaking the  
Ubiquitous Language within each Bounded Context.



- Business Invariants
- Policies
- Transactions
- State
- Persistence

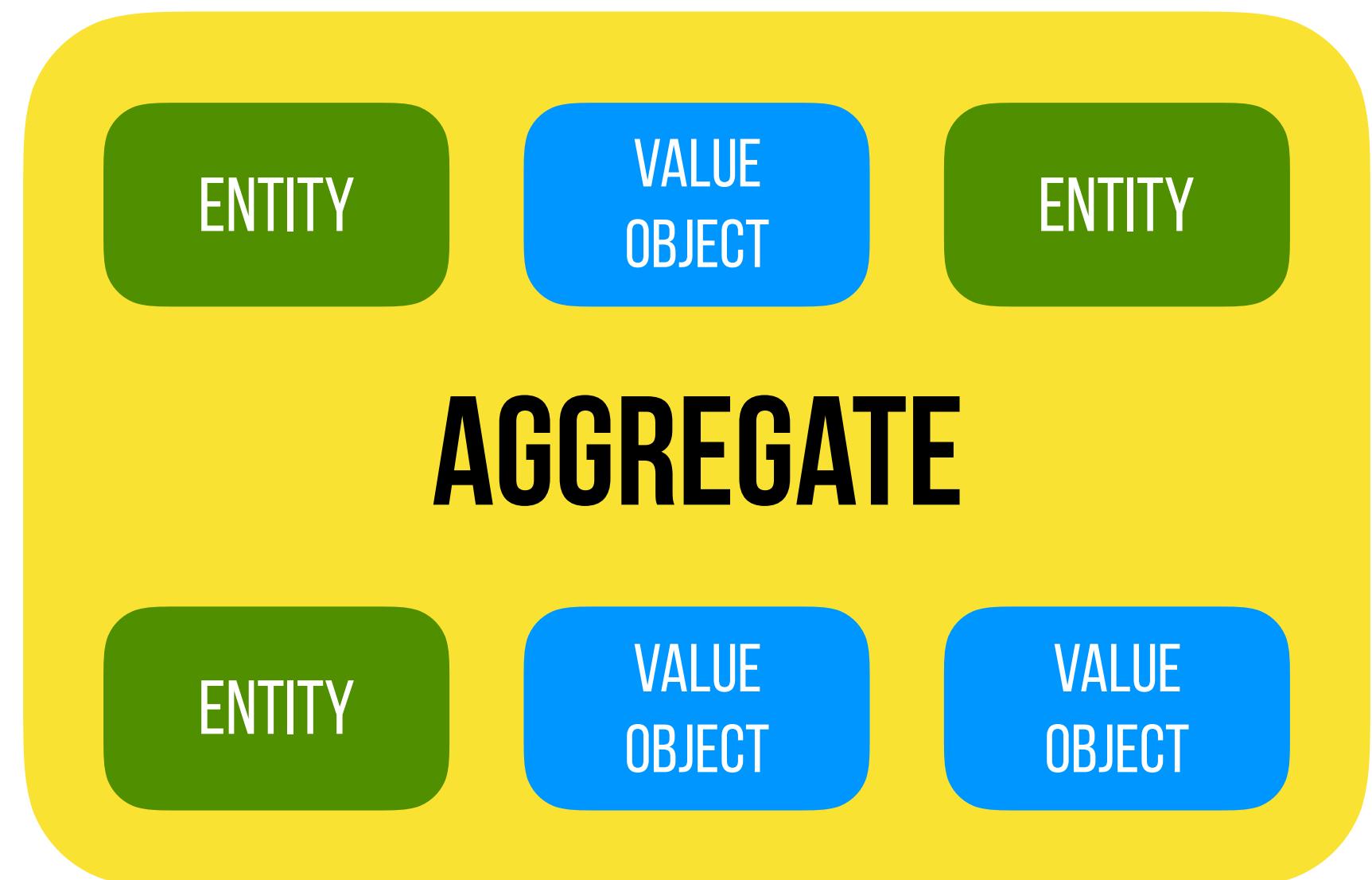
# THE CENTRAL CONCEPT

A cluster of objects  
treated as a single unit.

Only accessed through its  
Root Entity.

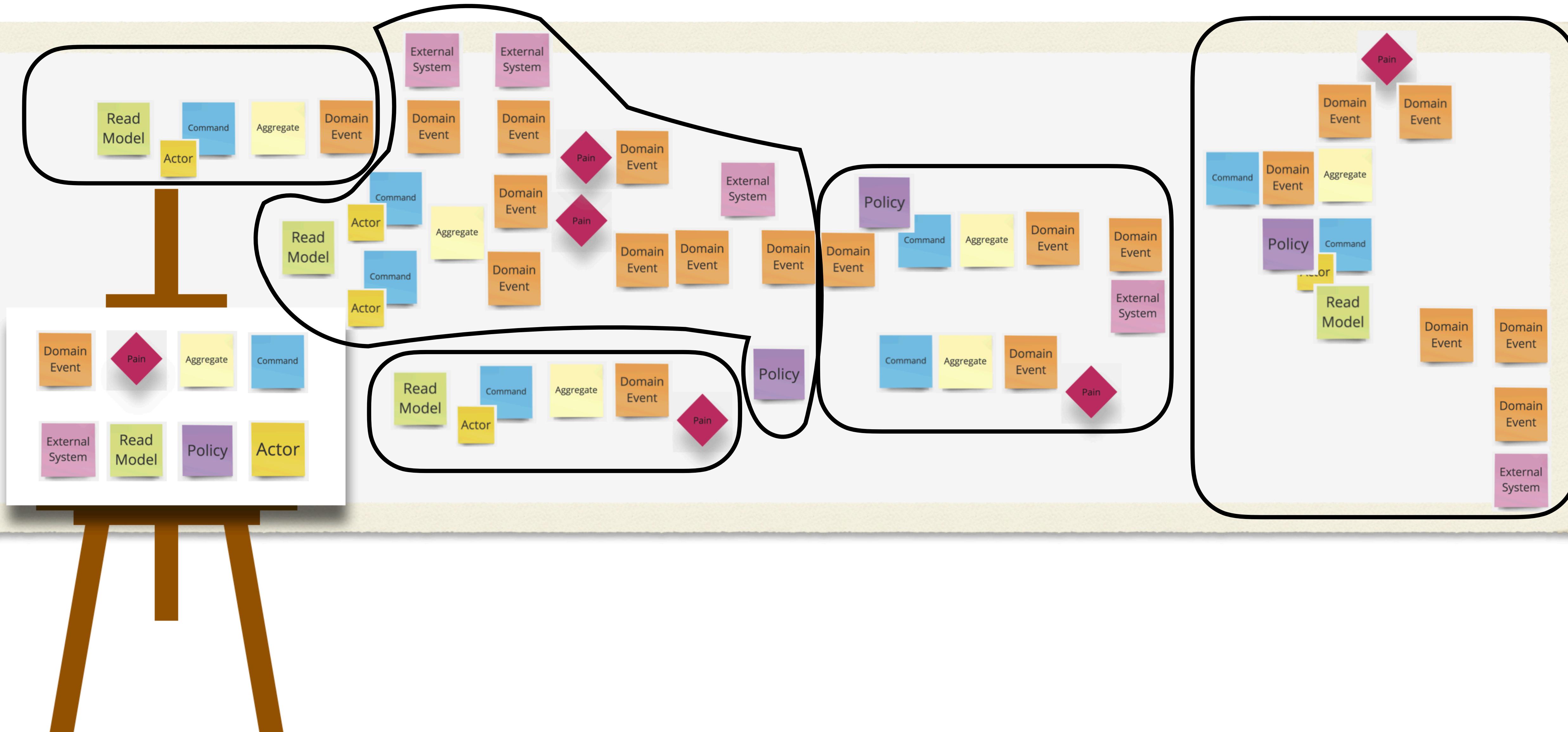
Often modeled as a  
state machine.

The atomic unit for any  
transactional  
behavior.

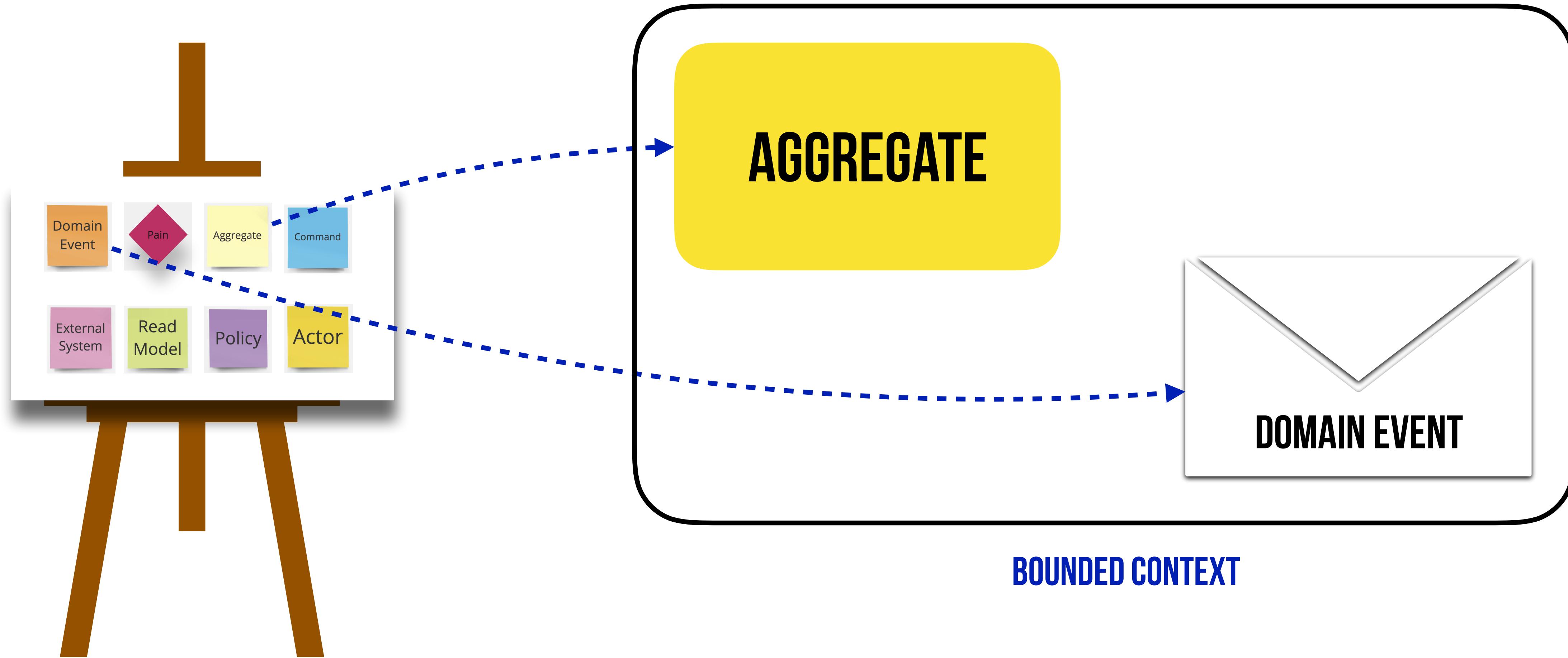


Responsible for maintaining  
any/all business invariants.

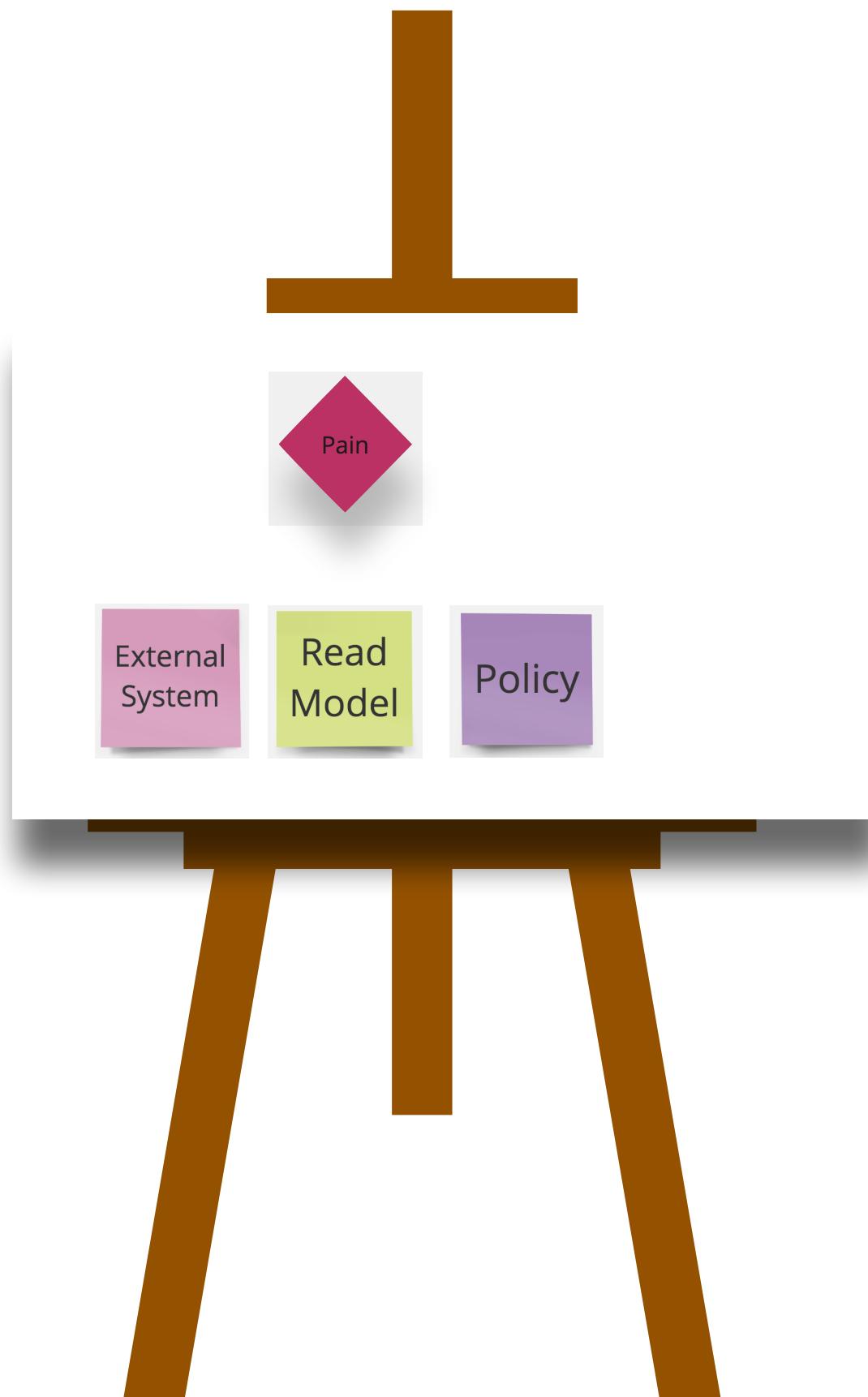
# EVENT STORMING



# FROM EVENT STORM TO DDD

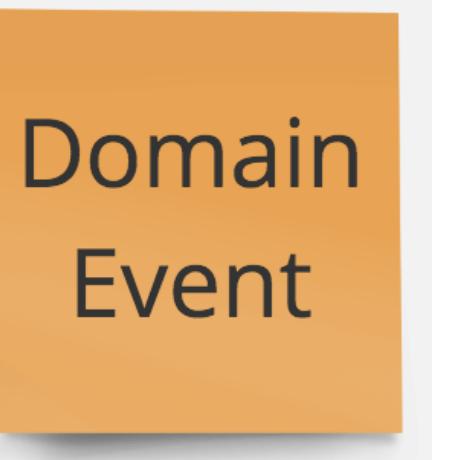


# USER STORIES

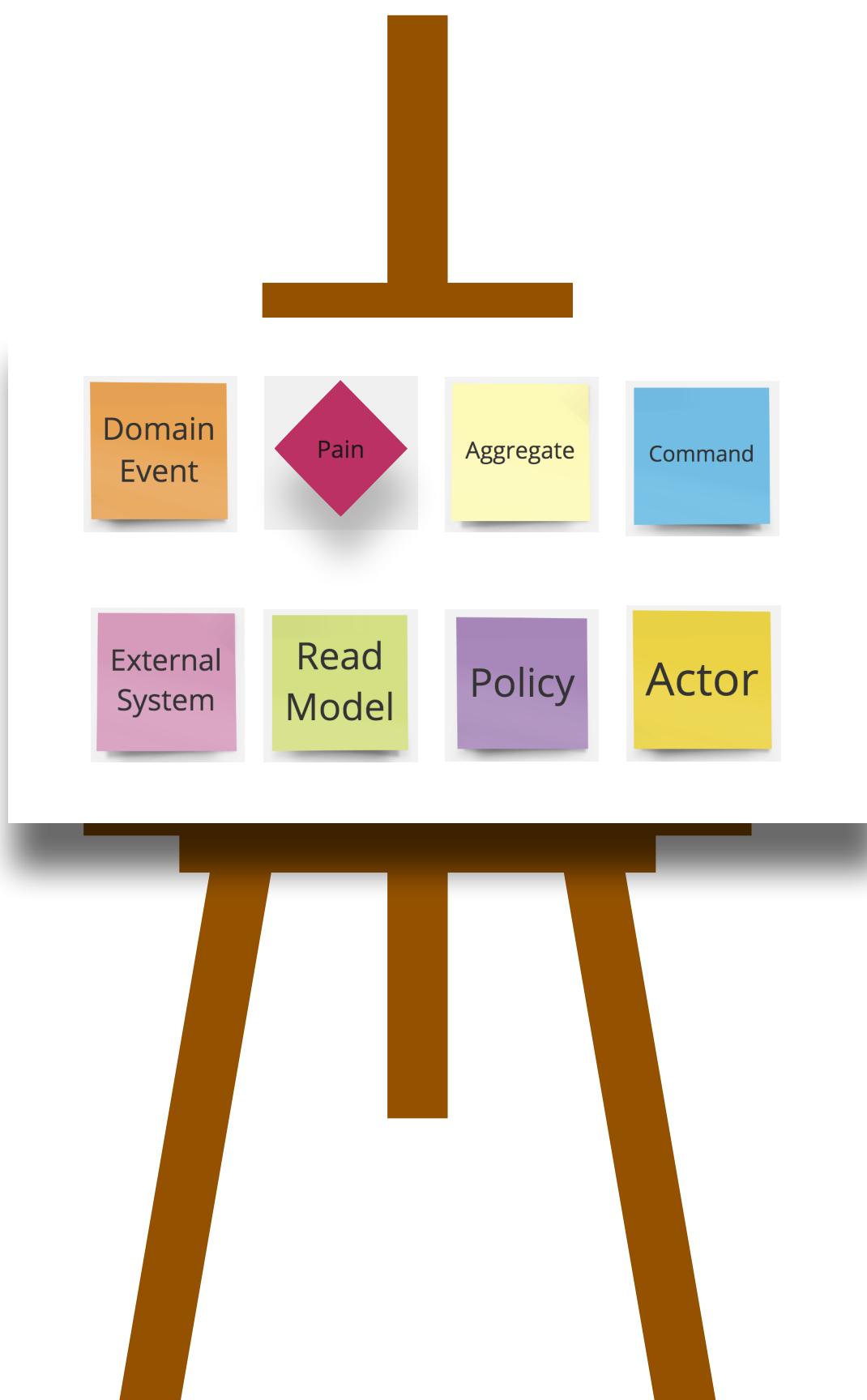


As a **ROLE**  Actor

I want to **ACTION**  Command  Aggregate

So that I can **GOAL**  Domain Event

# WHAT ABOUT TESTS?



- **Aggregate Business Invariants:**  
`should_not_allow_placeOrder_without_a_lineItem()`
- **Aggregate Domain Events:**  
`should_fire_event_when_order_placed()`
- **Policies:**  
`should_check_inventory_when_order_placed()`

REPOSITORY

# WHAT ABOUT CODE?

```
eventLog.subscribe(ProductAddedEvent,  
e -> {});
```

