



Domain Driven Design

Strategic patterns

▲ DDD – Strategic Patterns

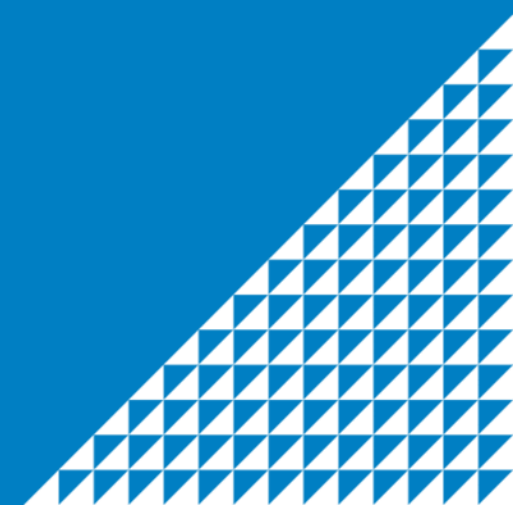
- Ubiquitous language
- Bounded Context
- Event Storming
- Context Map
- Model Integrity





Ubiquitous Language

DDD – Strategic Pattern



▲ Ubiquitous Language

Words mean things



▲ Ubiquitous Language

= **Used everywhere:**

in *user stories, specifications, meetings, emails, technical documentation, source code*

- List of terms and definitions
- Maintained by the development team
- Changes of the language often imply changes to the source code as well.

Eric Evans said: “use the model as the backbone of the language”

Nowadays we say: “use the language to build the model”



▲ Use the language of domain experts

- Delete a booking
 - Submit an order
 - Update a job order
 - Create an invoice
 - Set the state of a game
- Cancel a booking
 - Checkout
 - Extend a job order
 - Register/accept an invoice
 - Start/pause the game
-
- May contain technical terms like login, security, database, cache



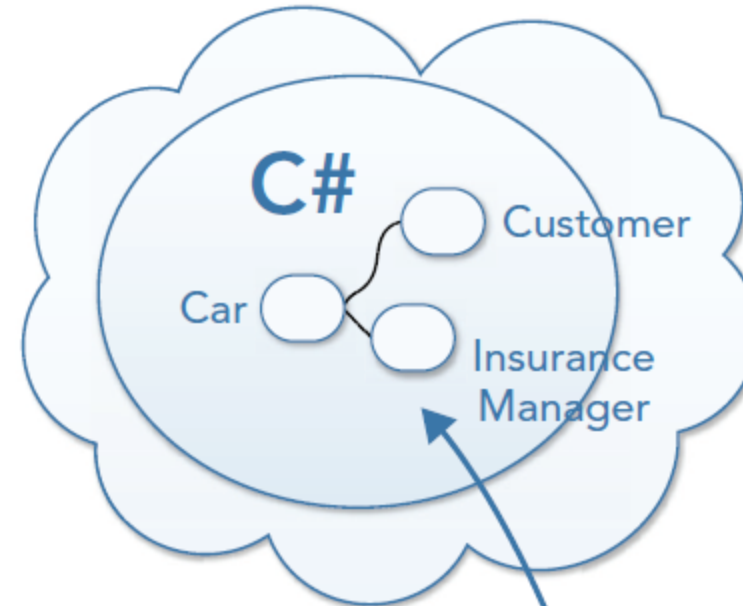
A policy owner can add a named driver. The named driver does not have to already have a policy with us.



Domain Experts

So a car can be associated to two customers? I will have to update the insurance manager and associations to handle this, plus change the database relationship.

The insurance manager looks after the call center and the sales team. He shouldn't need to be involved. A customer is a potential policy owner; he is someone with a quote. Associations are the policy holders over vehicles. Is that what you mean?



Development Team

The development team's interpretation of the concepts that don't match the domain experts

▲ Ubiquitous Language tips from experts:

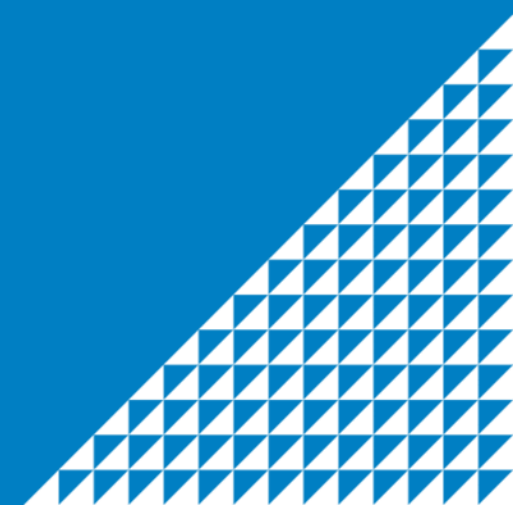
- Keep model and code in sync with language
 - Evans: *continuous integration*
- Use native language of the customer
 - not always English, even for international teams
- Replace acronyms with domain specific alternatives
 - Improves clarity
- Do not allow synonyms
 - Reduces clarity
 - Enforce the domain experts to do so!





Bounded Context

DDD – Strategic Pattern



▲ Ubiquitous \neq Universal

Words mean different things

In different contexts



▲ Bounded Context

- Unit of **Language Consistency**
 - The setting in which a word or statement appears that determines its meaning
- Independent model for a specific purpose
- Make boundaries explicit
 - Team organization (One team per bounded context)
 - Usage in part of application
 - Physical code base
 - Database schemas



▲ Recognizing splinters

- Duplicate concepts
 - Two model elements that actually represent the same concept
- False cognates
 - Two people who are using the same term think they are talking about the same thing, but really are not.



▲ Continuous Integration

“Institute a process of merging all code and other implementation artifacts frequently, with automated tests to flag fragmentation quickly.”

— Eric Evans (2003): Domain Driven Design, page 343

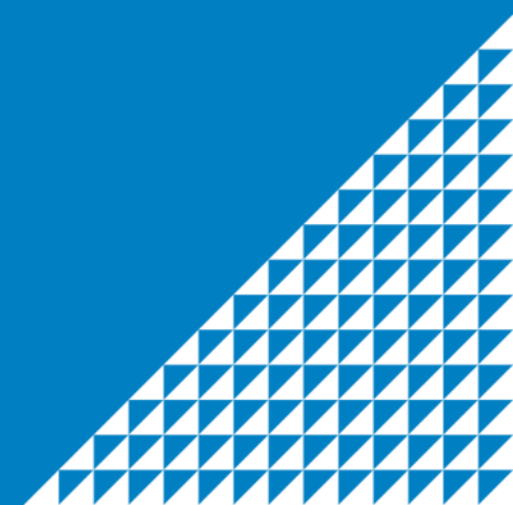
- Reproducible merge/build technique
- Automated test suite
- Rules that set some reasonably small upper limit on the lifetime of unintegrated changes





Event Storming

DDD – workshop-based method



▲ Event Storming

- Invented by Alberto Brandolini, in 2013
 - <http://ziobrando.blogspot.com/2013/11/introducing-event-storming.html>

Very fast way to get a rough model
for a problem

Gather requirements from a
business oriented conversation

▲ Event Storming

- Several forms
 - **Big Picture**
 - Process Modelling
 - Software Design



▲ Event Storming – Big Picture

- Invite the right people:
 - Business people, IT developers, UX people, (other stakeholders)
- Unlimited modelling space
- Model the **whole** business line
 - do not focus on a particular area first!
- Can take a whole day!



▲ Big Picture – steps

1. Explore Domain Events (orange stickies)
2. Enforcing a timeline (breaks the silos, cross-department conversation!)
3. Mark problems and hot spots as stickies (purple stickies)
4. Optionally go deeper by adding, step by step:
 - people, (external) systems, problems, constraints, opportunities, value, etc,
5. Arrow-vote on major problems



▲ Result of Big Picture session

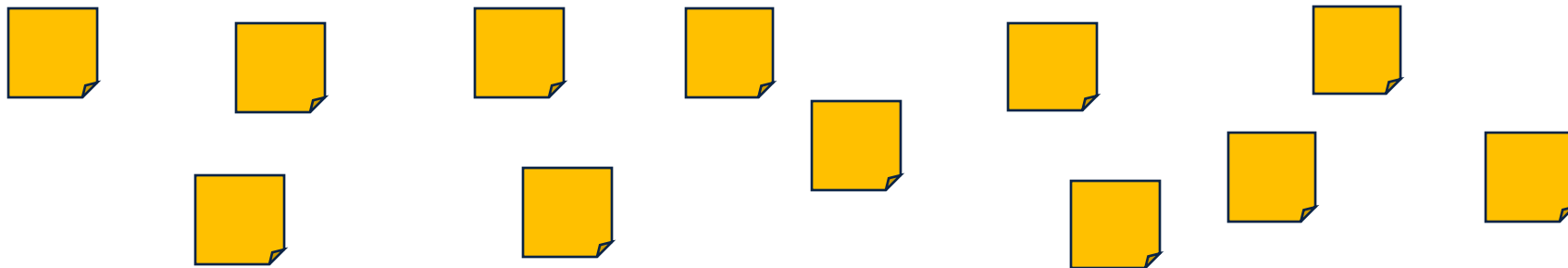
- Clarity
- Core domain visible
- Important bottlenecks visible
- Key blocker
 - no backlog needed (this is it)
 - no estimates (it is hard)
 - DDD approach (experiment with it!)





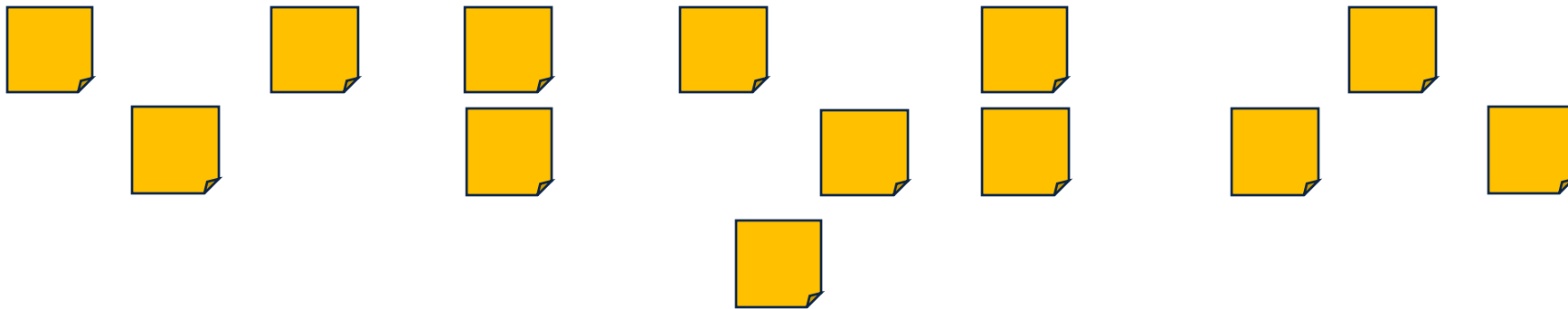
Domain Event

- Past tense
- Events that *domain experts care about*
- Using language domain experts



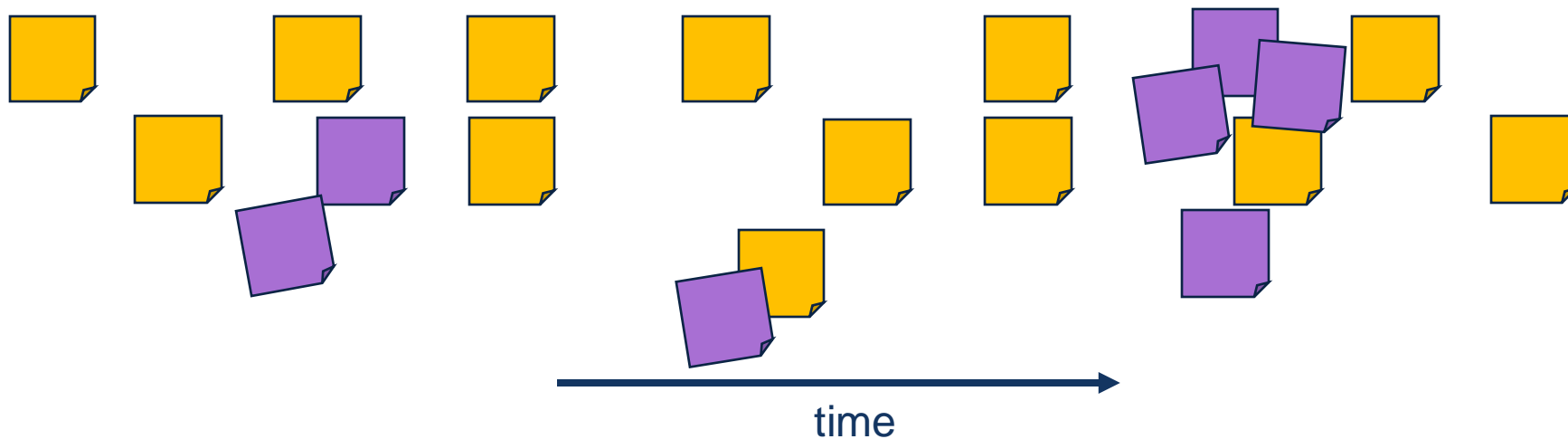


- Add time: from left to right
 - Stack parallel events
- Questions (is this really the first event? Can there be more events?)
 - Aim for everything
 - Ambiguity is fine (for now)



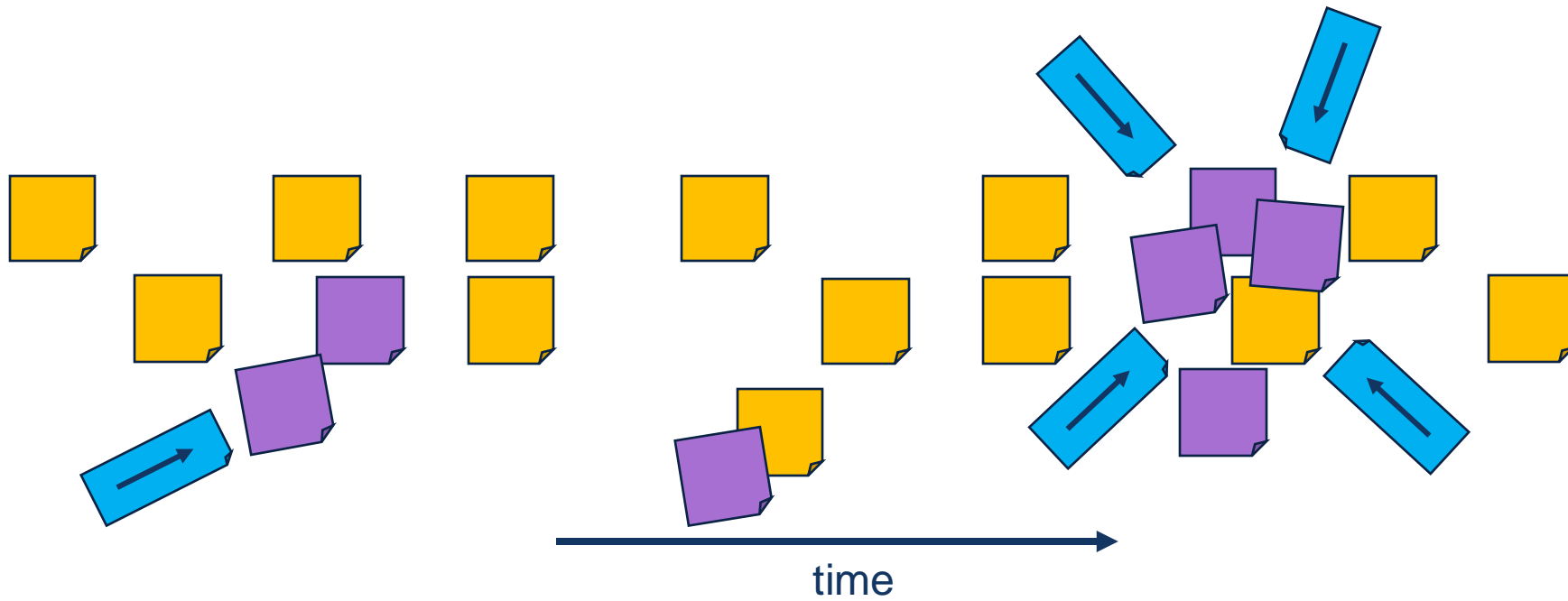
Concern Hot spot

- When adding time, discussions arise
 - especially when crossing boundaries between department silos
- Mark those discussions





- Arrow-vote on major problems



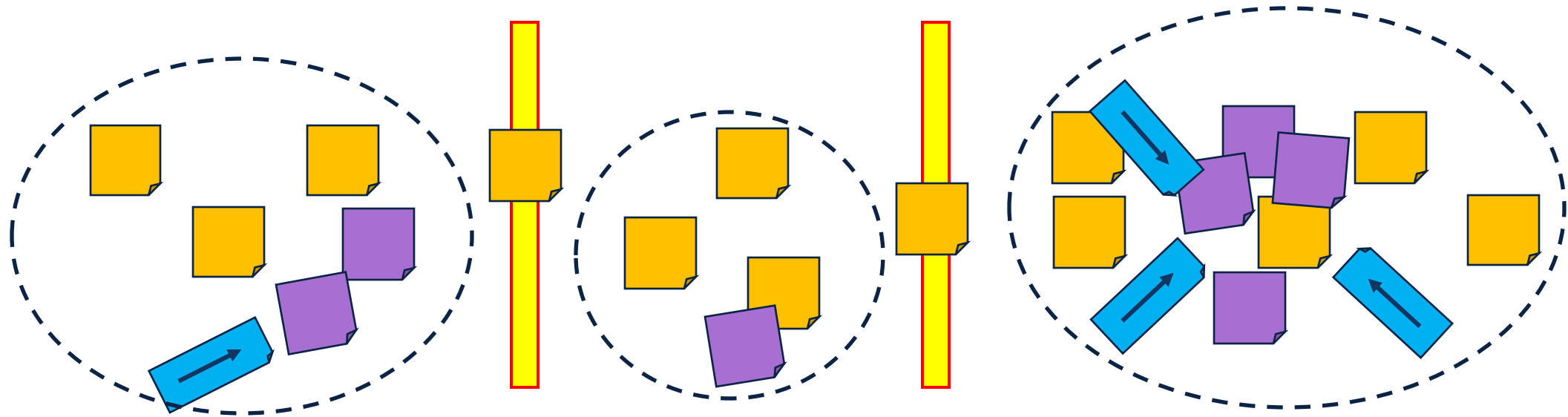
▲ Result of Big Picture session

- Clarity
- Core domain visible
- Important bottlenecks visible
- Key blocker
 - no backlog needed (this is it)
 - no estimates (it is hard)
 - DDD approach (experiment with it!)



Other result of Big Picture

- Bounded contexts
- Boundary events



▲ Event Storming

- Several forms
 - Big Picture
 - **Process Modelling**
 - Software Design



▲ Event Storming – Process Modelling

- Invite the right people:
 - Business people, IT developers, UX people, (other stakeholders)
- For one feature / epic / activity
- Read model + command → Domain Events + Read Model



▲ Event Storming

- Several forms
 - Big Picture
 - Process Modelling
 - **Software Design**



▲ Event Storming

1. Add domain events 

2. Add definitions and concerns  

3. Add Commands (with user roles, if appropriate)   

4. Add External Systems 

5. Add Policies 

6. Add Read Models 

7. Add Aggregates 

8. Group strongly related aggregates in bounded contexts 





Domain Event

- Past tense
- Events that *domain experts care about*
- Using language domain experts




Customer
registered



Product
Added to
Cart

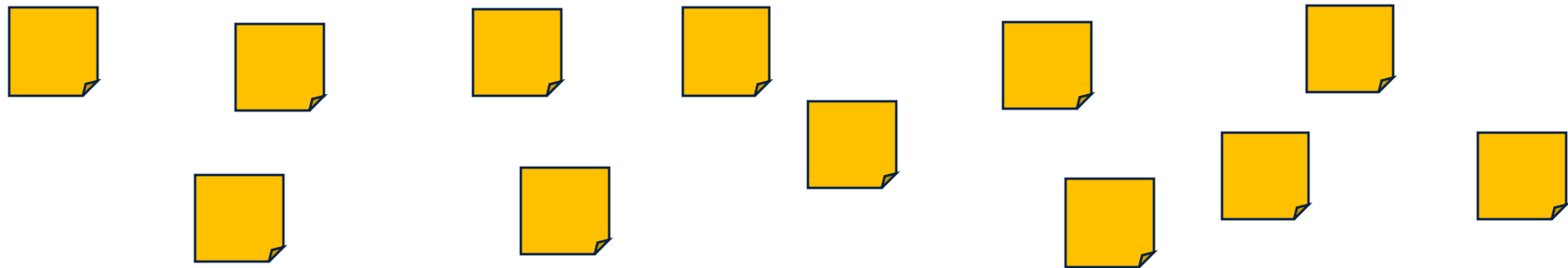


Maintenance
job finished



Day has
passed

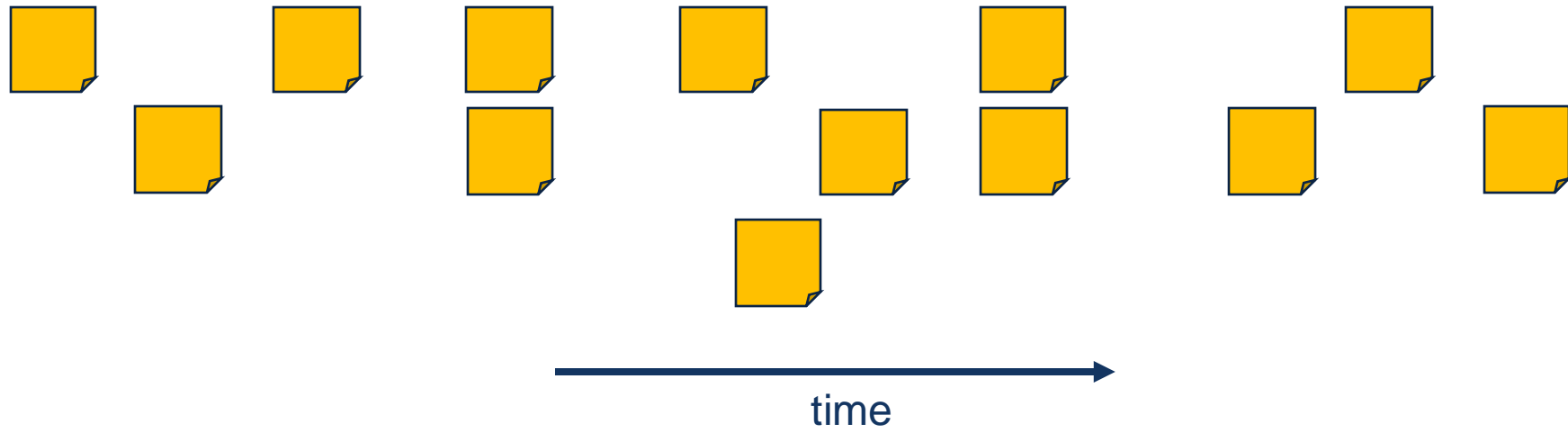
Domain Events





Add time:

- Order events from left to right
- Stack parallel events
- Ask Questions
 - Is this really the first event? Can you think of more events?
- Question the language
 - Force participants to be precise



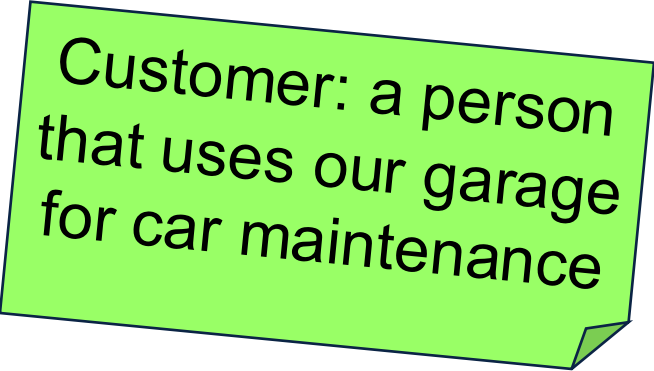


Definition



Concern /
Risk

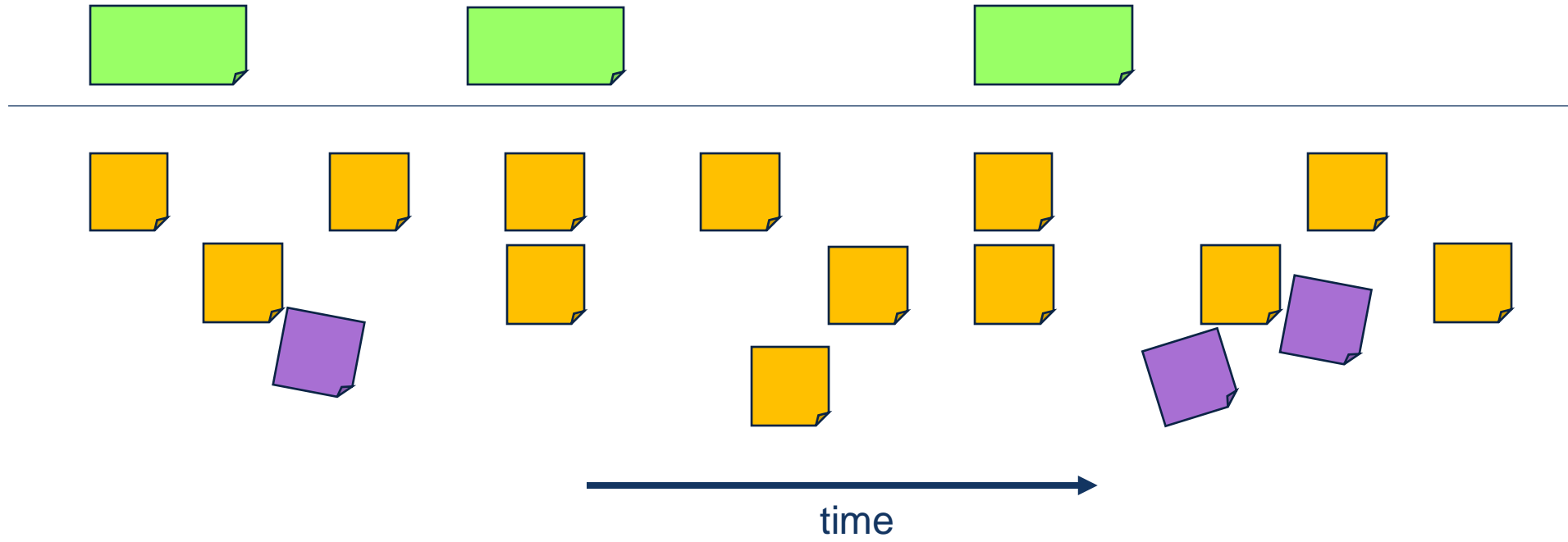
- Definitions help towards a ubiquitous language
- Concerns/Risks are ways visualize problem areas



Customer: a person
that uses our garage
for car maintenance



Printer may
be offline





Command



Command

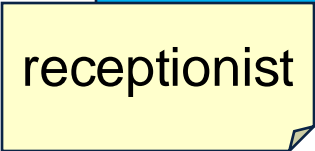


User Role

- Each Domain Event should have Command
 - Except for time events
- Commands should express intent
- Commands are often triggered by a specific user role



Register customer



receptionist



Customer registered



Create workshop planning




Workshop planning created




Reschedule Maintenance job



receptionist



Maintenance job canceled



Maintenance job planned

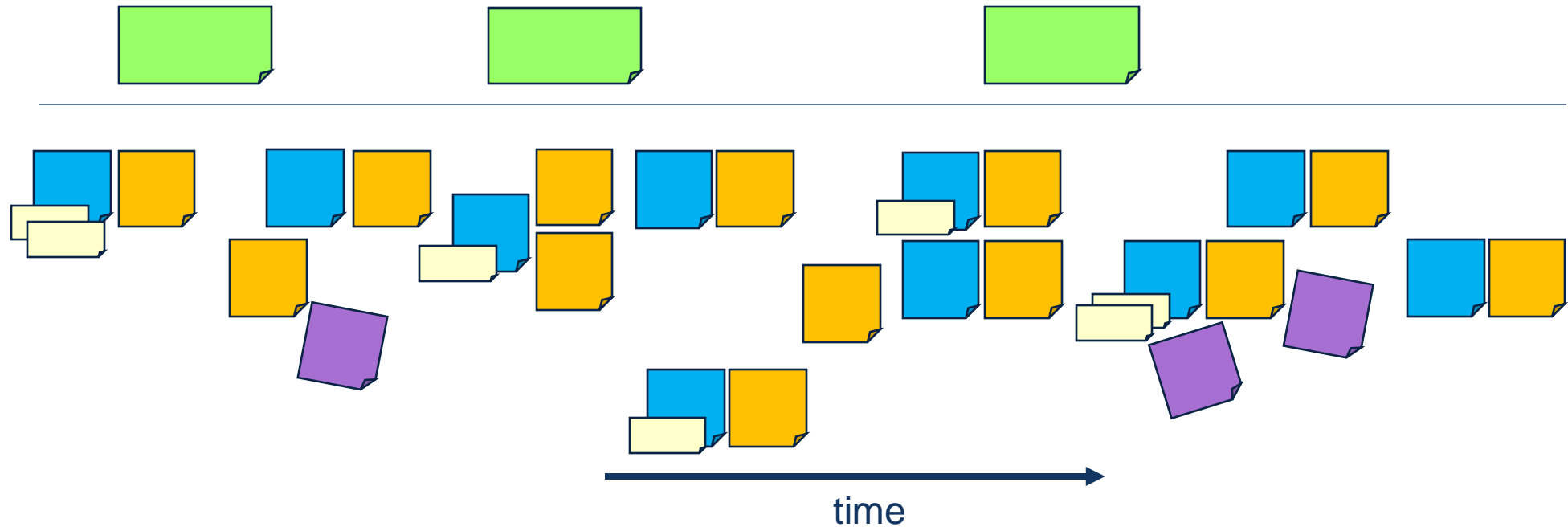
▲ Command vs Event in DDD

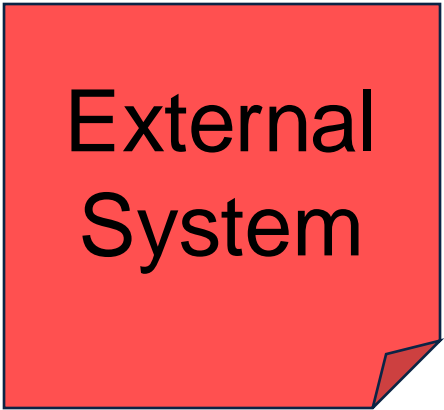
Command

- Imperative (Do it now!!)
- Can fail
- Can have a result
- Has a target
- Tightly coupled

Event

- Past tense (has happened)
- Cannot fail
- Broadcast
- Loosely coupled





External System

- Some commands addressed to external systems ...
- ... that may trigger new events
- Now is also a good moment to draw arrows to visualize the flow



Mail server



Printer

Policy

Policies are important

- Between **command** and **domain event** ...
...AND between **domain event** and **command**
- Draw them out. There are often more than one suspects.
 - Can a maintenance job *always* be planned?
 - Whenever a day has passed, we *always* send an invoice?

Plan
maintenance
job

No more than
three
parallel jobs

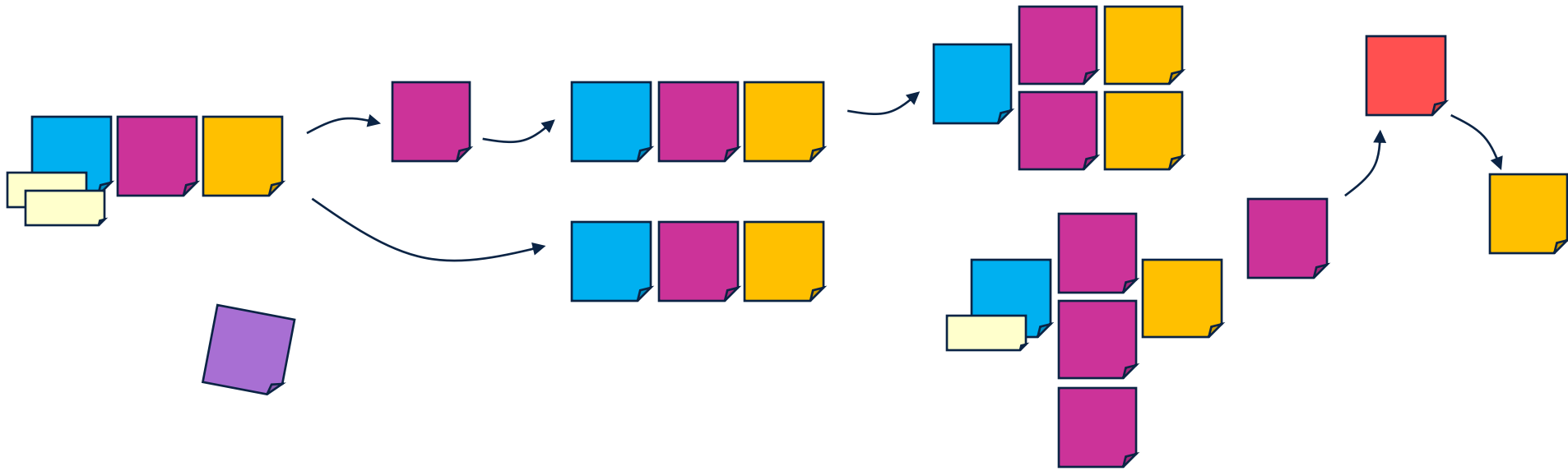
One job per
vehicle

Maintenance
job Planned

A day has
passed

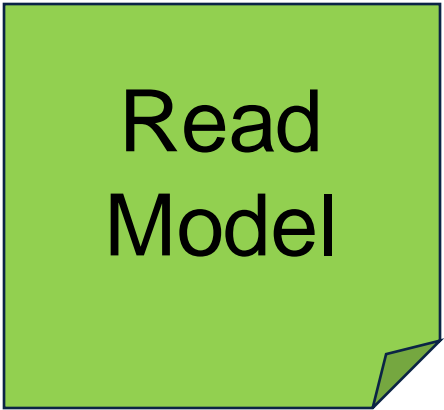
Maintenance
job
completed
yesterday

Send Invoice



time





Read Model

- Represents (readonly) data that is used by users or by the system
- Users make decisions based on data
 - The decision is captured as a Command
 - The data is captured as a Read model
- A read model can also show the result to the user



Customer



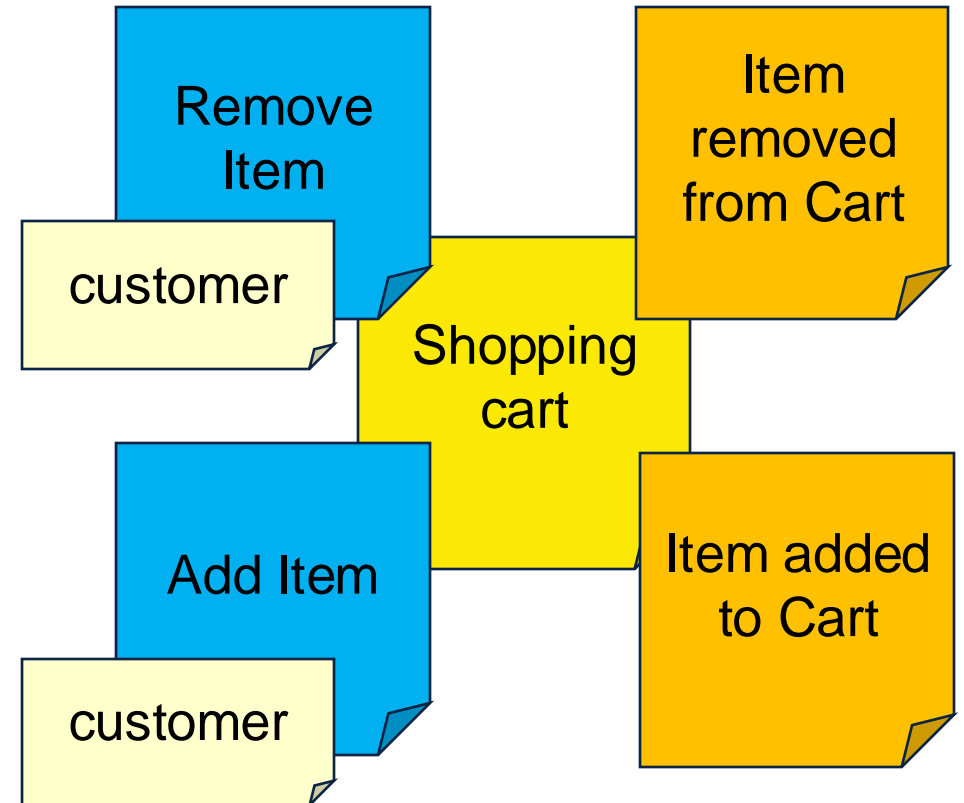
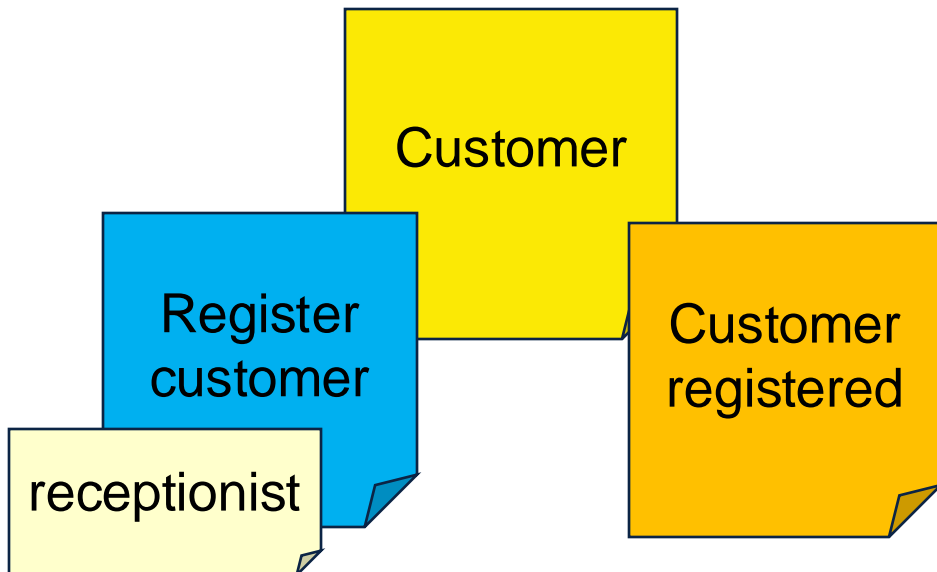
Vehicle



Plan
maintenance
job

Aggregate

- A command changes an aggregate ...
... which causes an event



Summary

1. Domain Event
 2. Definition
 3. Concern / Risk
 4. Command
 5. External System
 6. Policy
 7. Read Model
 8. Aggregate
- User Role
-

▲ Pitstop features

(Info Support Garage Case goes DDD)

- Plan maintenance jobs
 - Register customer
 - Register vehicle
- Send notifications for maintenance jobs that are due today
- Send invoices for completed maintenance jobs

<https://github.com/EdwinVW/pitstop>



Result of Event Storming for Pitstop

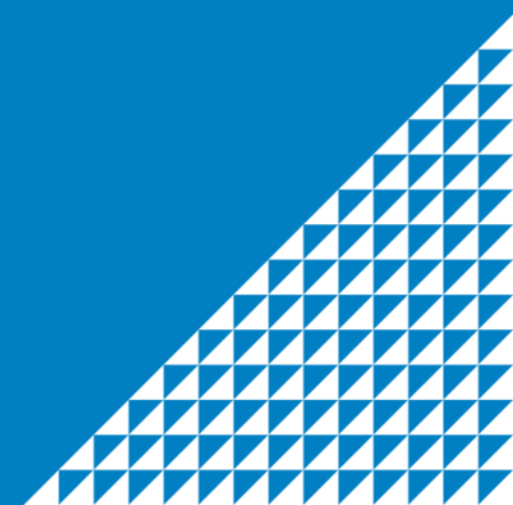
- <https://raw.githubusercontent.com/EdwinVW/pitstop/img/event-storming-result.png>





Context Map

DDD – Strategic Pattern

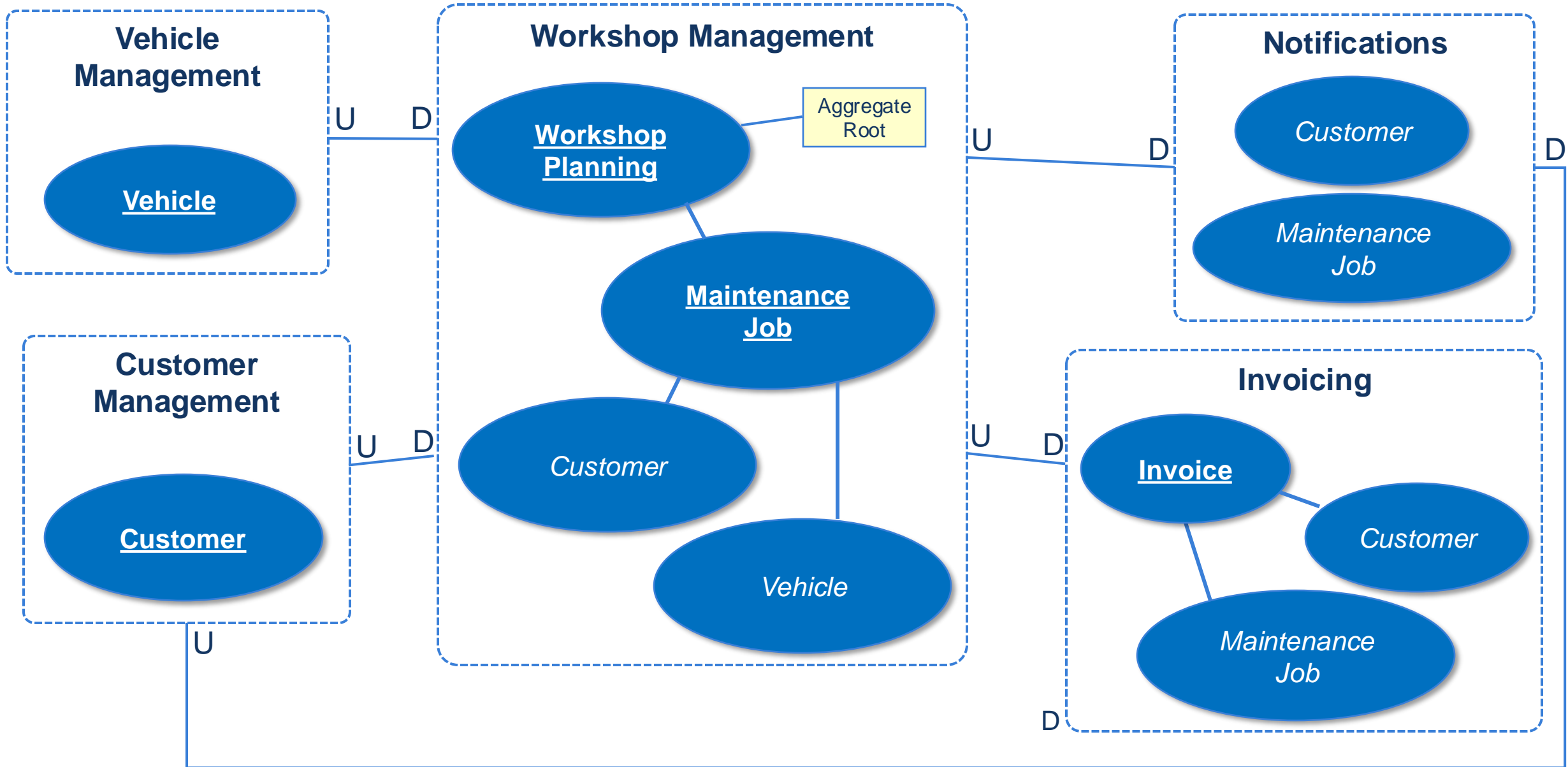


▲ Context Map

- Web of bounded contexts
- About one bounded context for each business context (Conway's Law)
- Upstream / Downstream
 - Upstream always influences Downstream, maybe also the other way around

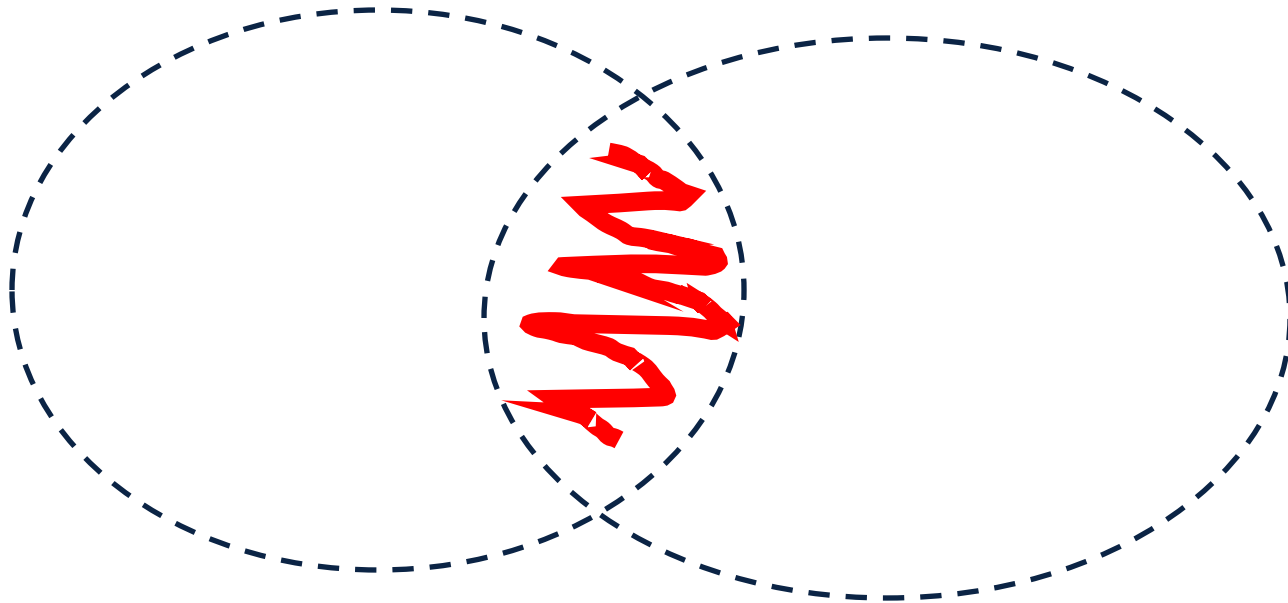


Context map



▲ Shared Kernel

- Two teams working on two closely related models
- Overlapping bounded contexts



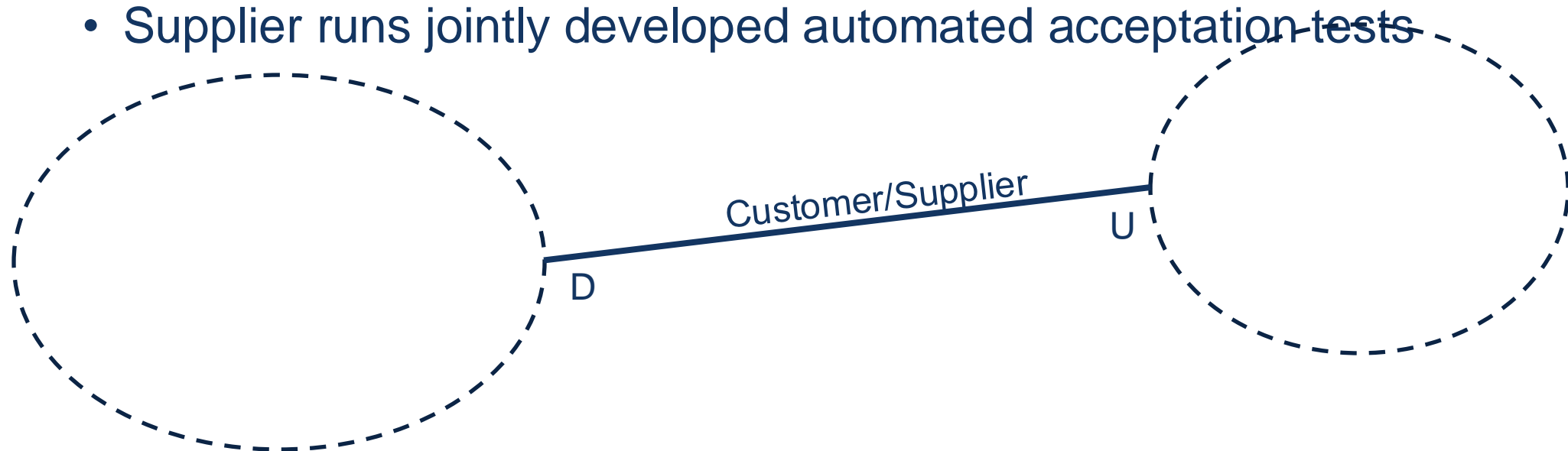
▲ Shared Kernel

- When two teams share responsibility over shared model
- Usually changes more slowly than private model parts
- Frequent communication is paramount



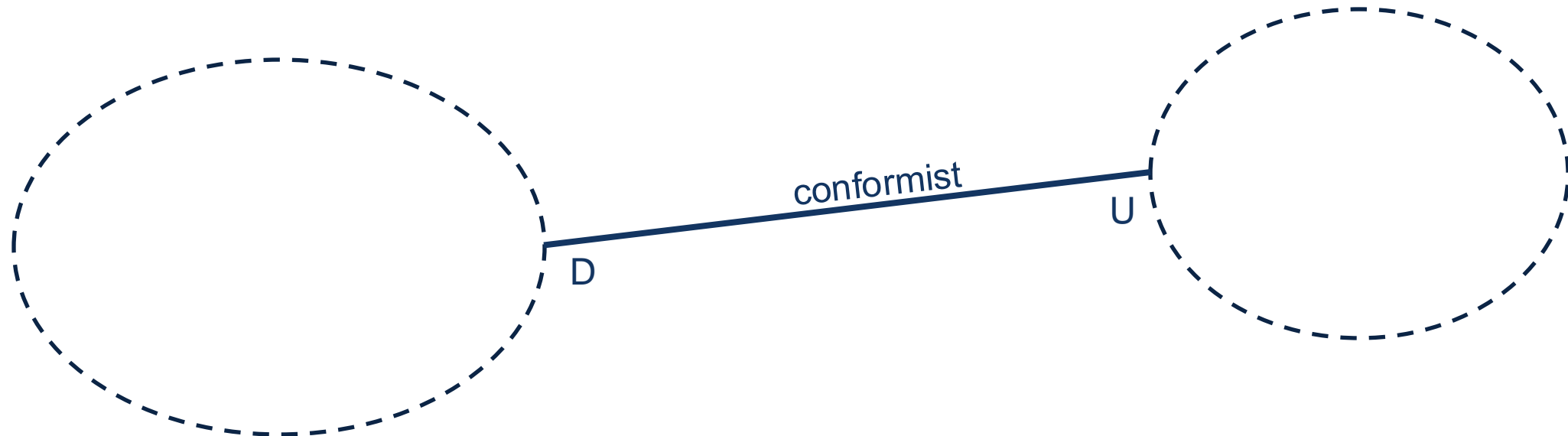
▲ Customer/Supplier

- When one team is dependent on another team ...
... and have a common interest
- Customer is actual client
- Supplier runs jointly developed automated acceptance tests



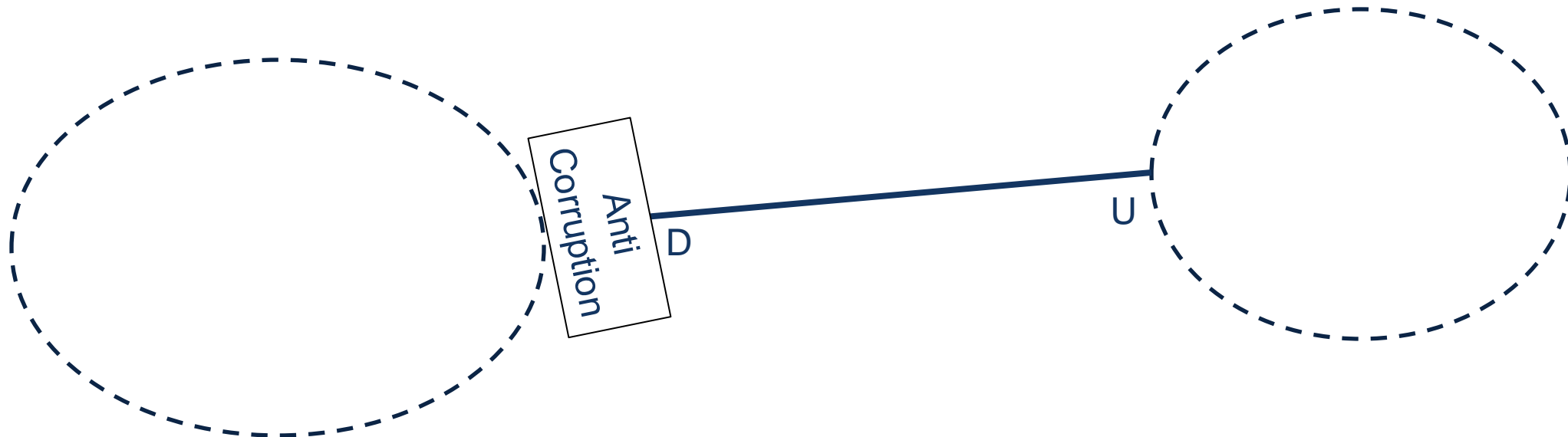
▲ Conformist

- When one team is dependent on another team ...
... but upstream has no motivation to provide for downstream's needs
- Maybe upstream has many customers
- Downstream slavishly adheres to upstreams model



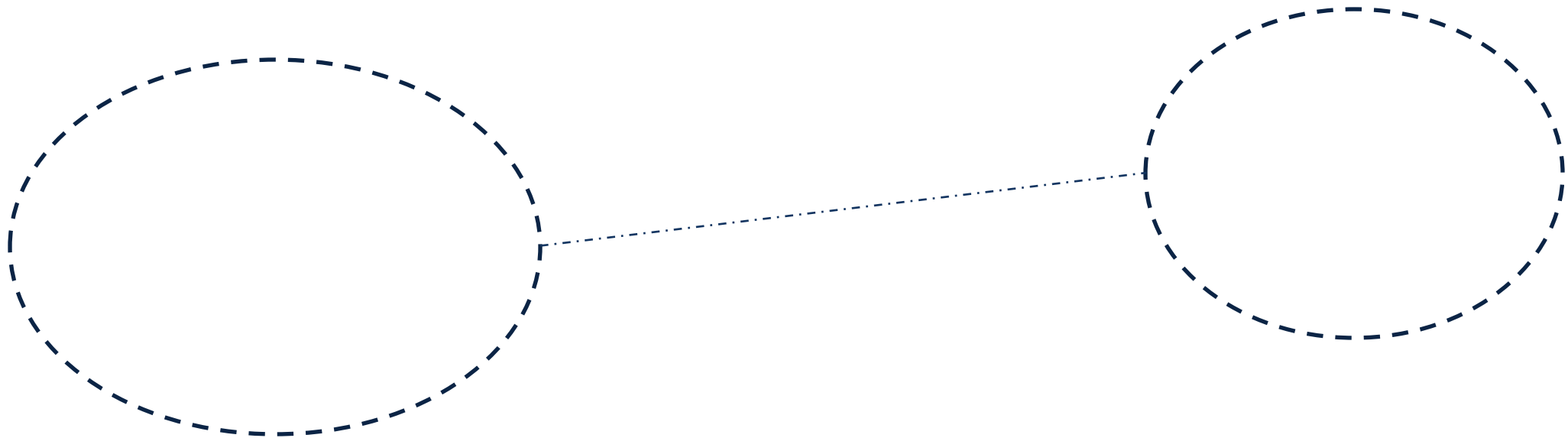
▲ Anti-corruption Layer

- When interfacing with a system that has a 'weak' or 'messy' model
 - Large existing system, legacy system
- Protect Downstream with anti-corruption layer
 - e.g. Service – Adapter with Translators – Façade



▲ Separate ways

- No shared model
- Minimal to no data transfer



▲ Open Host Service

- When being upstream from many other teams/contexts
- Define a protocol
 - Possibly with a published language

