Domain-Driven DESIGN

Tackling Complexity in the Heart of Software

Eric Evans

Foreword by Martin Fowler

What is domain-driven design?

- Placing the project's primary focus on the core domain and domain logic
- Basing complex designs on a model of the domain
- Initiating a creative collaboration between technical and domain experts to iteratively refine a conceptual model that addresses particular domain problems

Problem space		Solution space	
Domain	Business problem to be addressed	**Domain model**	Abstraction of a business problem
Sub-domain	Smaller part of the doma	**Bounded context**	Delimits the domain model

The goal of a domain-driven design is an alignment between the domain and the software.

How do we identify subdomains?

- Business capability:
  - Insurance — underwriting, claims, sales & marketing
- Organisational structure:
  - Insurance products — home, motor, life, travel
  - Hospital departments — GP, A&E, paediatrics, social care
- Organisational communication structures (Conway's Law)

Types of subdomains

1. **Core domain**
2. Supporting subdomain
3. Generic subdomain

Core domain

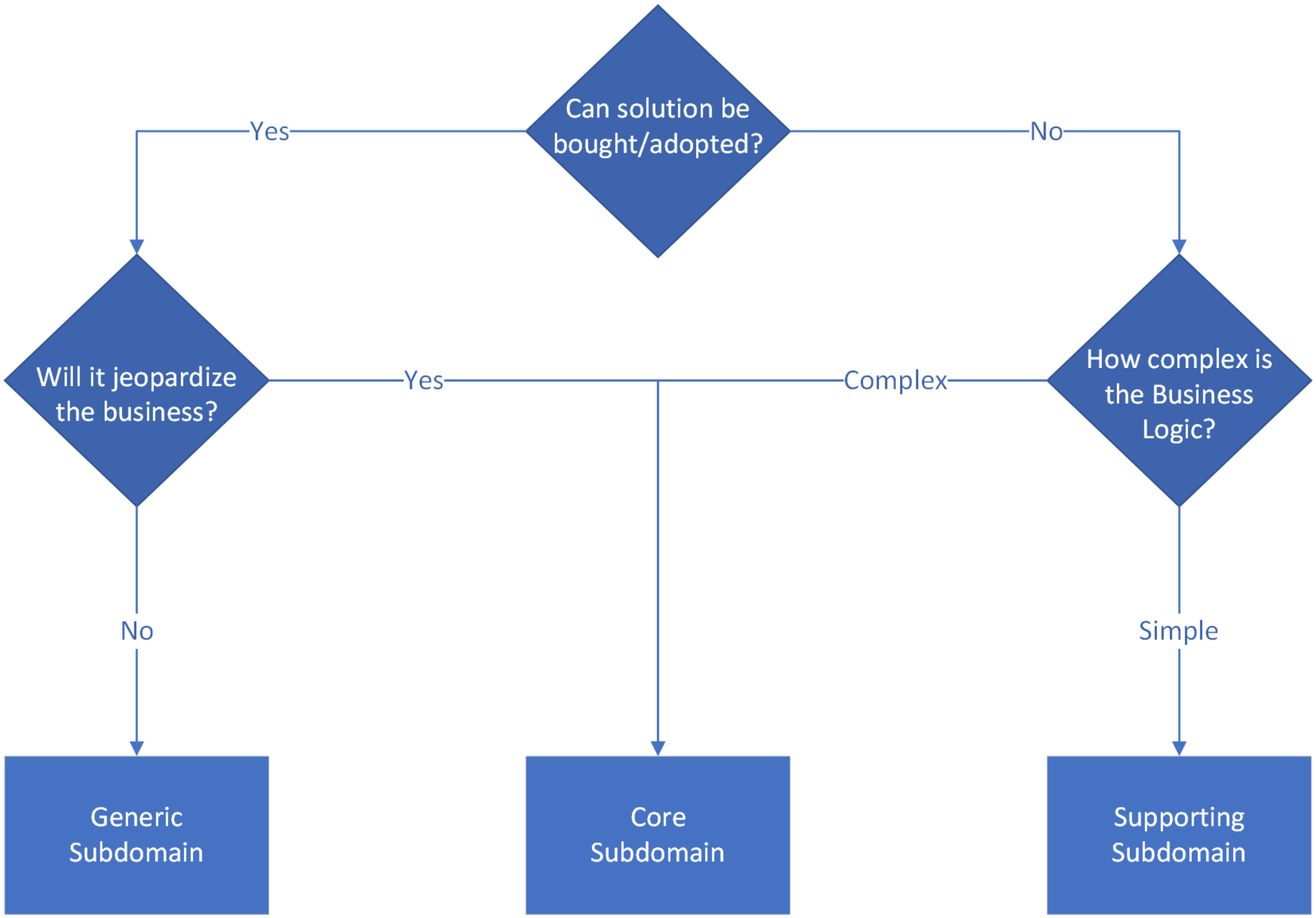
- Strategic investment in a single, well-defined domain model
- High value and priority
- The company's *secret sauce* to distinguish it from competitors

Supporting subdomain

- Custom development — no off-the-shelf solution
- Consider outsourcing development

Generic subdomain

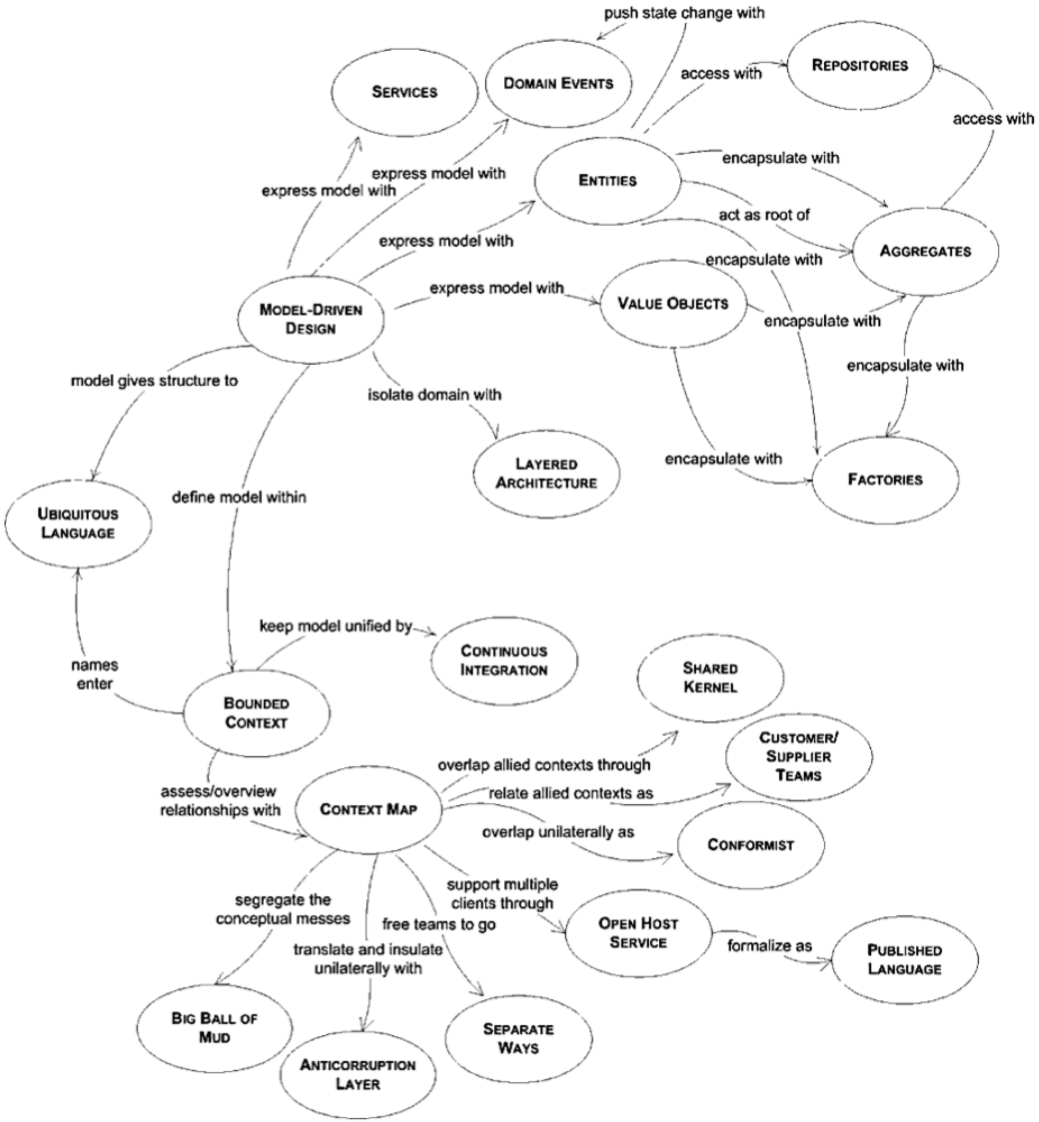
- Purchase off-the-shelf solution
- Outsource development
- Examples:
    - Accounting
    - CRM
    - Identity / authentication


Can solution be bought/adopted?
Yes
No
Will it jeopardize the business?
Yes
Complex
How complex is the Business Logic?
No
Simple
Generic Subdomain
Core Subdomain
Supporting Subdomain

Domain-driven comprises …

1. Strategic design:
   - Ubiquitous language
   - Bounded context
   - Context map
   - Continuous integration

2. Tactical design:
   - Entity
   - Value object
   - Aggregate
   - Domain event
   - Service
   - Repository
   - Factory


push state change with
SERVICES
DOMAIN EVENTS
access with
REPOSITORIES
access with
express model with
express model with
ENTITIES
encapsulate with
act as root of
AGGREGATES
express model with
encapsulate with
MODEL-DRIVEN DESIGN
express model with
VALUE OBJECTS
encapsulate with
encapsulate with
model gives structure to
isolate domain with
LAYERED ARCHITECTURE
encapsulate with
FACTORIES
UBIQUITOUS LANGUAGE
define model within
keep model unified by
CONTINUOUS INTEGRATION
names enter
BOUNDED CONTEXT
SHARED KERNEL
overlap allied contexts through
CUSTOMER/ SUPPLIER TEAMS
relate allied contexts as
assess/overview relationships with
CONTEXT MAP
overlap unilaterally as
CONFORMIST
support multiple clients through
segregate the conceptual messes
free teams to go
OPEN HOST SERVICE
formalize as
PUBLISHED LANGUAGE
translate and insulate unilaterally with
BIG BALL OF MUD
ANTICORRUPTION LAYER
SEPARATE WAYS

Strategic design

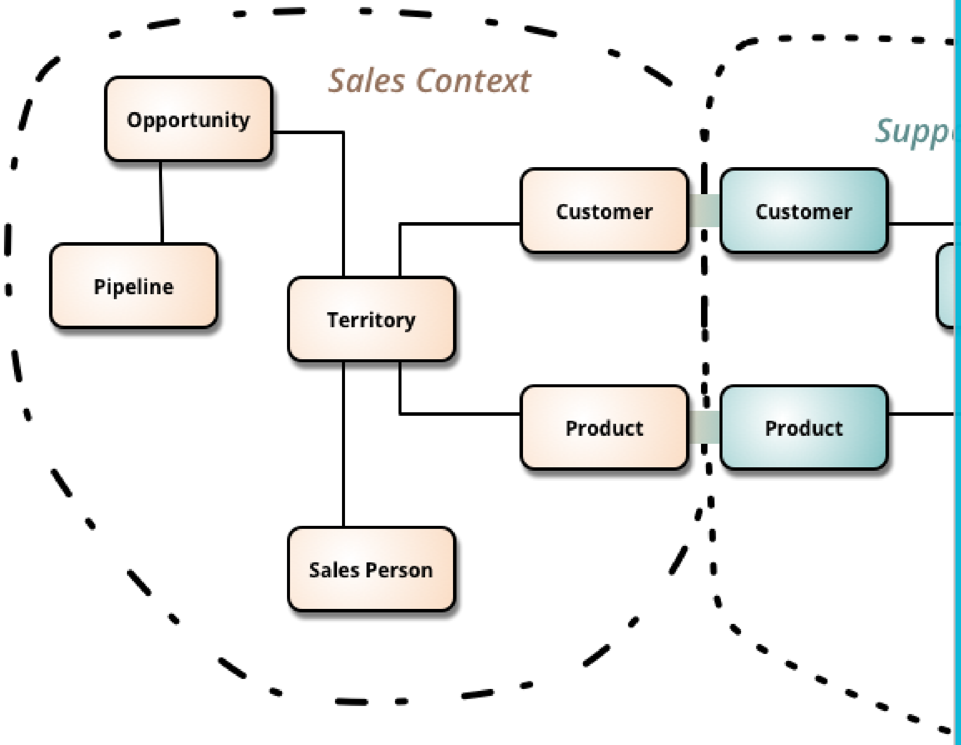
Ubiquitous language

- A language used by all team members
- What is a **Policy**?
    - Underwriting context
    - Claims context
    - Marketing & sales context

A domain specific term can have multiple meanings.

Understanding the meaning of the word is dependent upon the context.

Bounded context

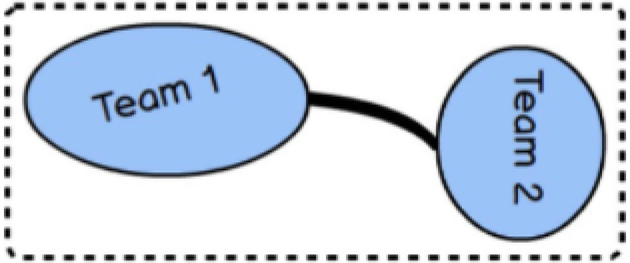


- Semantic contextual boundary for a model
- Ubiquitous language is consistent *within* a bounded context
- Separate software artifacts for each bounded context
- Keep the model strictly consistent within these bounds

Context map

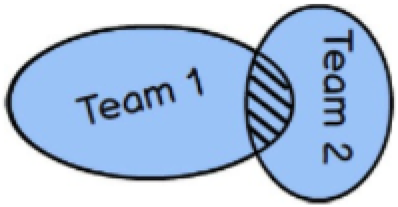
- Define relationship and translation between bounded contexts (and ubiquitous languages)
- Kinds of mappings:
    - Partnership
    - Customer-supplier
    - Anticorruption layer
    - Published language
    - Shared kernel
    - Conformist
    - Open host service
    - Separate ways

Partnership



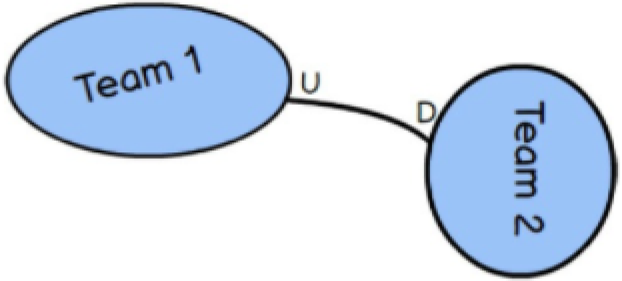
- Each team responsible for one bounded context
- Aligned with a dependent set of goals
- Two teams will succeed or fail together
- Challenging relationship to maintain due to high synchronisation & committment

Shared kernel



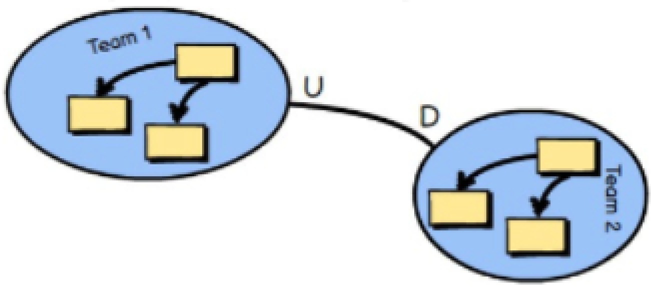
- Teams share a small but common model
- Difficult to conceive and maintain due to aggreement on what is required

Customer-supplier



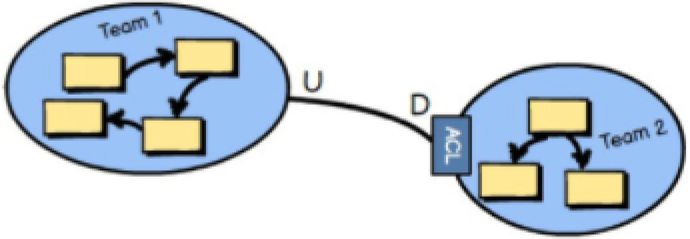
- *Supplier* is upstream and *customer* is downstream
- Supplier provides what the customer needs (but determines what & when)
- Typical relationship between teams witin an organisation

Conformist



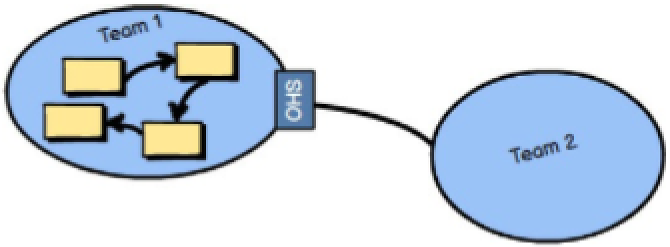
- As customer-supplier, except upstream team has no motivation to support the downstream team
- Downstream team cannot aford to translate the ubiquitous language, so conforms to upstream model as is

Anticorruption layer



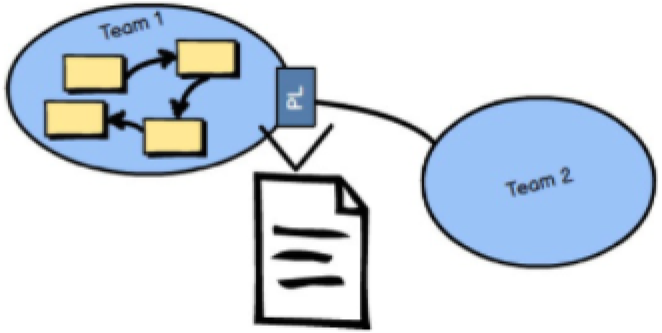
- Most defensive mapping relationship
- Donstream team creates a translation layer between the upstream's model and its own
- Provides isolation between contexts, but translation costs may be too high

Open host service



- Define an interface or protocol that gives access to your bounded context
- "Open" protocol to allow anyone to integrate with relative ease
- Well documented service API
- Translation often not required by consumers

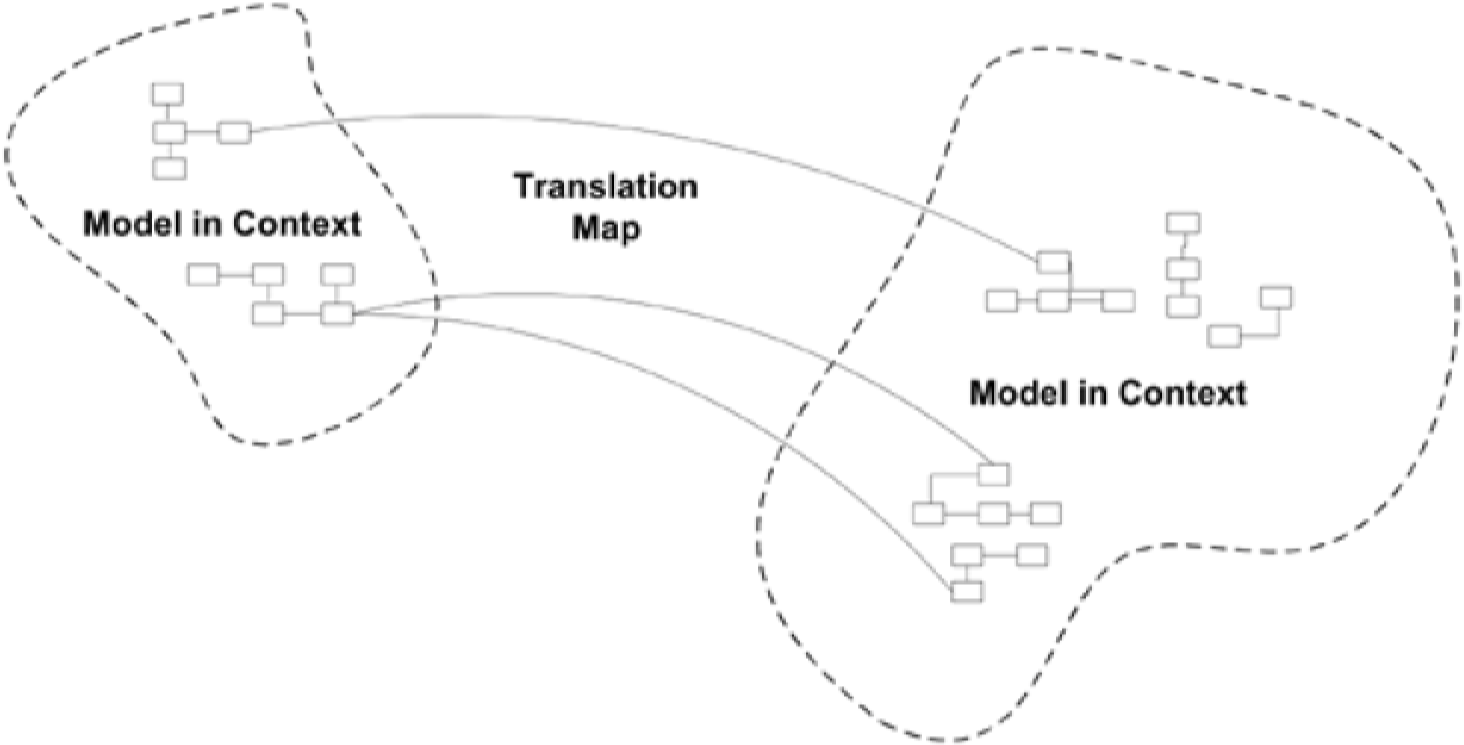
Published language



- Well-documented information exchange language
- Enables simple consumption and translation by any number of consumers
- Published language defined by a schema (e.g. XML Schema, JSON Schema) or wire format (e.g. Protobuf)

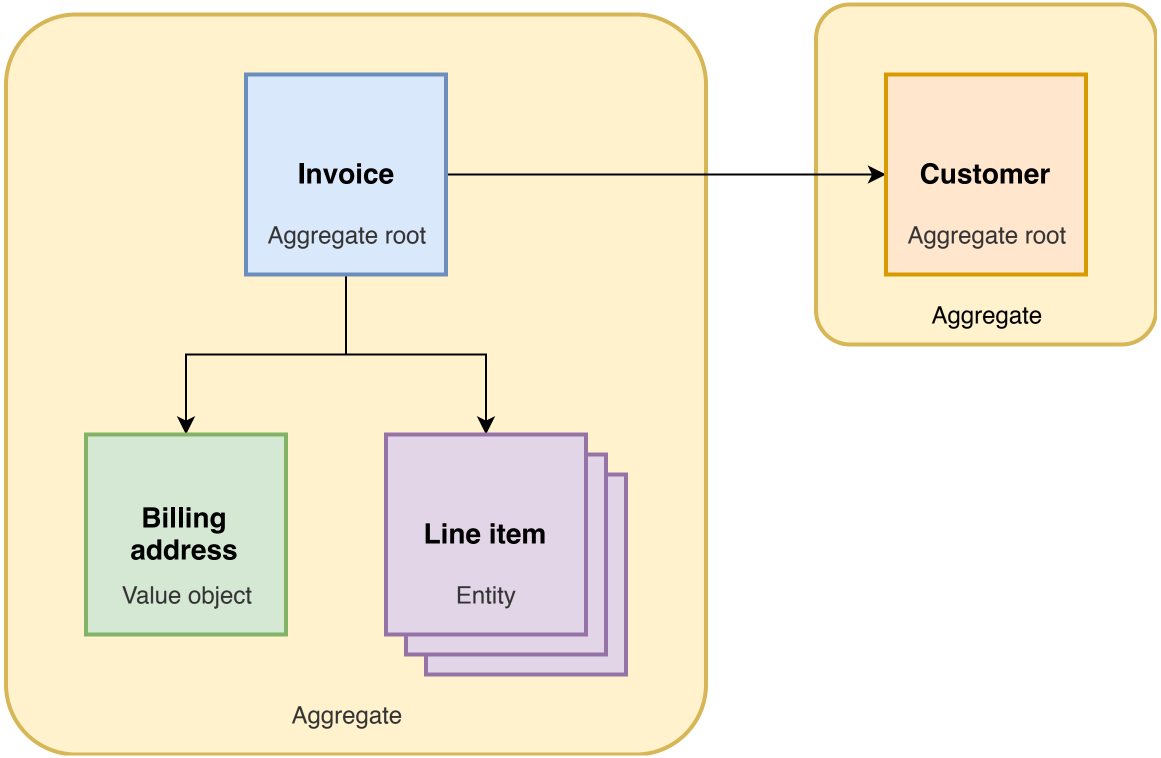
Separate ways

- Integration between contexts not worth the effort
- Implement your own specialised solution internally — don't attempt to integrate


Translation
Map
Model in Context
Model in Context

Tactical design

Aggregate, root, & entity



Entity

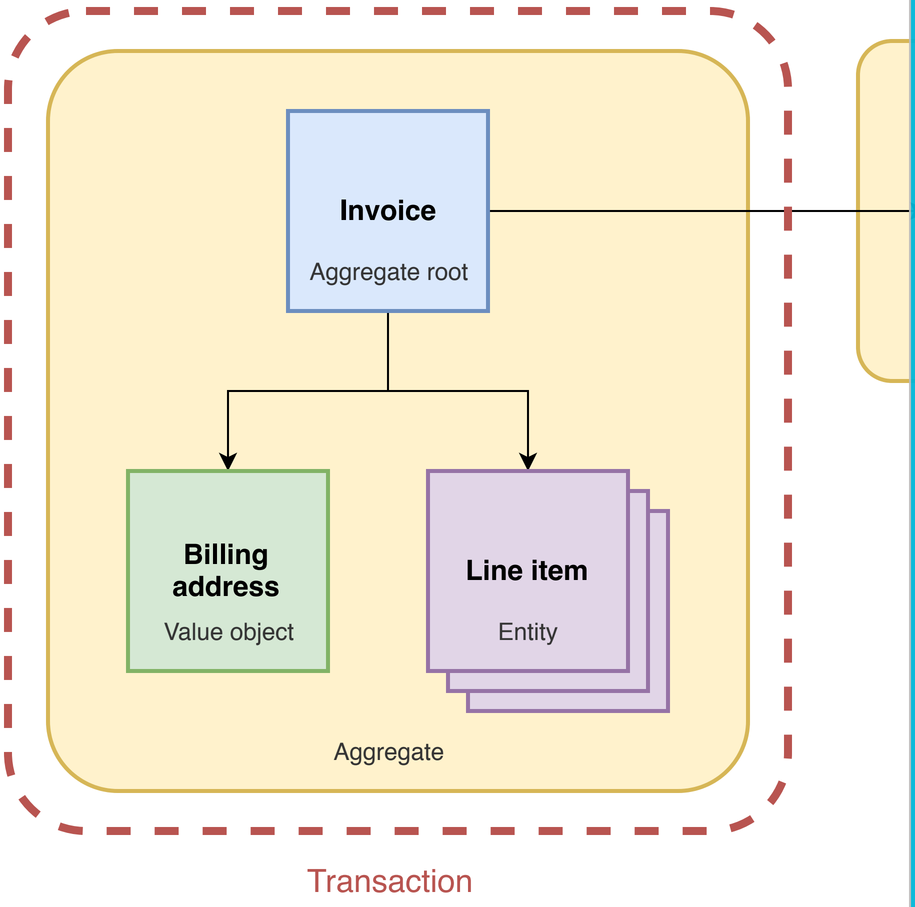
- Models an individual thing
- Has a unique identity
- Is mutable — its state changes over time
- Examples:
  - Invoice
  - Line item
  - Customer

Value object

- Models just a value
- Doesn't have a unique identity
- Is immutable
- Equivalence is determined by its attributes
- Examples:
    - Address
    - Money

Aggregate

- Composed of one or more entities and value objects
- Forms a transactional consistency boundary
- One entity is called the **aggregate root**:
    - Owns all other elements clustered inside it
    - Access to the aggregate *must* go through the root entity
- Examples:
    - Invoice
    - Customer



- Aggregate enforces transactional consistency
- Business invariants must be protected within the boundary
- Must be stored in a whole and valid state
- Allows concurrent transactions for different aggregate instances

Four rules of aggregate design

1. Protect business invariants inside aggregate boundaries
2. Design small aggregates
3. Reference other aggregates by identity only
4. Update referenced aggregate using eventual consistency

Domain event

- Record of some business-significant occurrence in a bounded context
- Immutable *facts*
- Named in the past tense using the ubiquitous language
- Can be used for inter-service messaging
- Examples:
    - CustomerBilled
    - InvoicePaid

Service

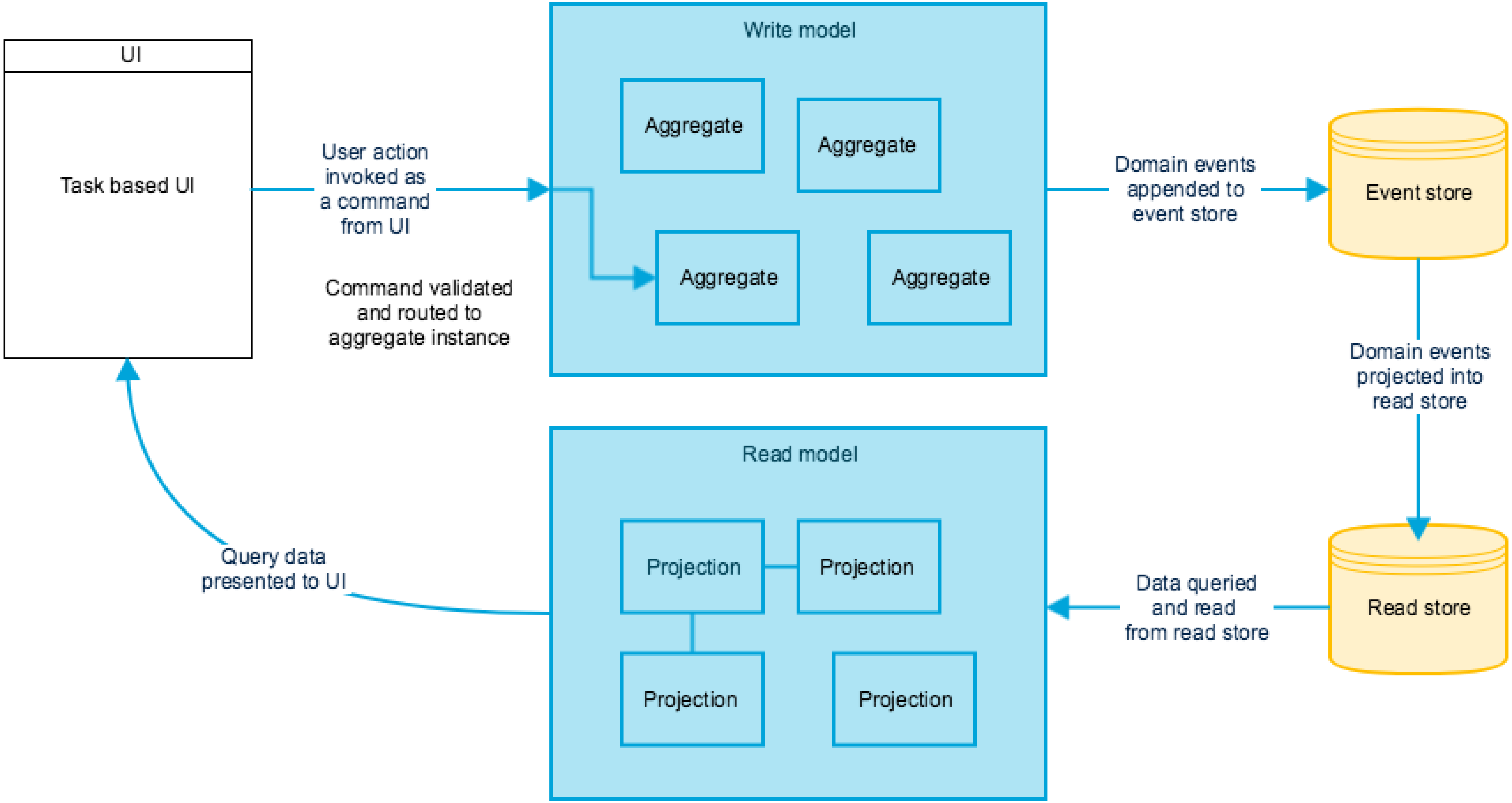
- Contains domain operations that don't belong to an entity or value object
- Is stateless
- Examples:
    - Price calculation
    - Currency conversion

Repository

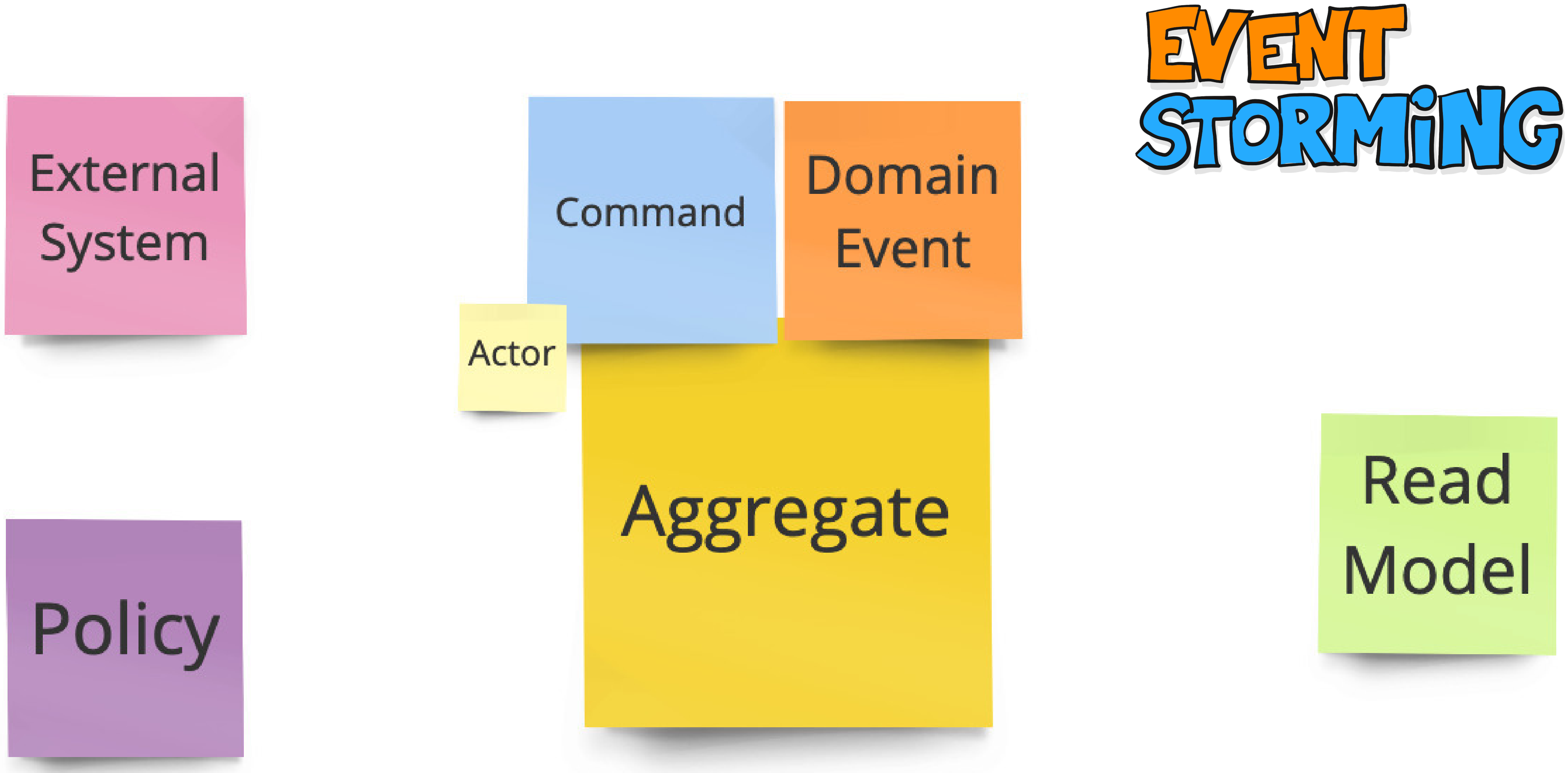
- Retrieve domain objects (aggregates) from storage

Factory

- Create domain objects


UI
Task based UI
User action invoked as a command from UI
Command validated and routed to aggregate instance
Write model
Aggregate
Aggregate
Aggregate
Aggregate
Domain events appended to event store
Event store
Domain events projected into read store
Read store
Data queried and read from read store
Read model
Projection
Projection
Projection
Projection
Query data presented to UI

EVENT STORMING



External System

Command

Domain Event

Actor

Aggregate

Policy

Read Model

Further reading

- Domain-Driven Design: Tackling Complexity in the Heart of Software by Eric Evans
- Domain-Driven Design Distilled by Vaughn Vernon
- Implementing Domain-Driven Design by Vaughn Vernon
- Domain Driven Design Quickly (free download)