Introduction to DDD

Go Meetup Brno
28 February 2024

About me

- Pocket Gopher at ModernTV
- FI MUNI
- https://standa.dev (https://standa.dev)

Why DDD?

Goal

- *"introduction"*
- get a basic sense of DDD
- cover key components
- won't go in-depth
- you won't become DDD experts
- starting point

Outline

- origins of DDD
- DDD key components
  - ubiquitous language
  - strategic design
  - tactical design
- event storming
- architecture patterns
- sample app

Quick Survey

Domain-Driven Design

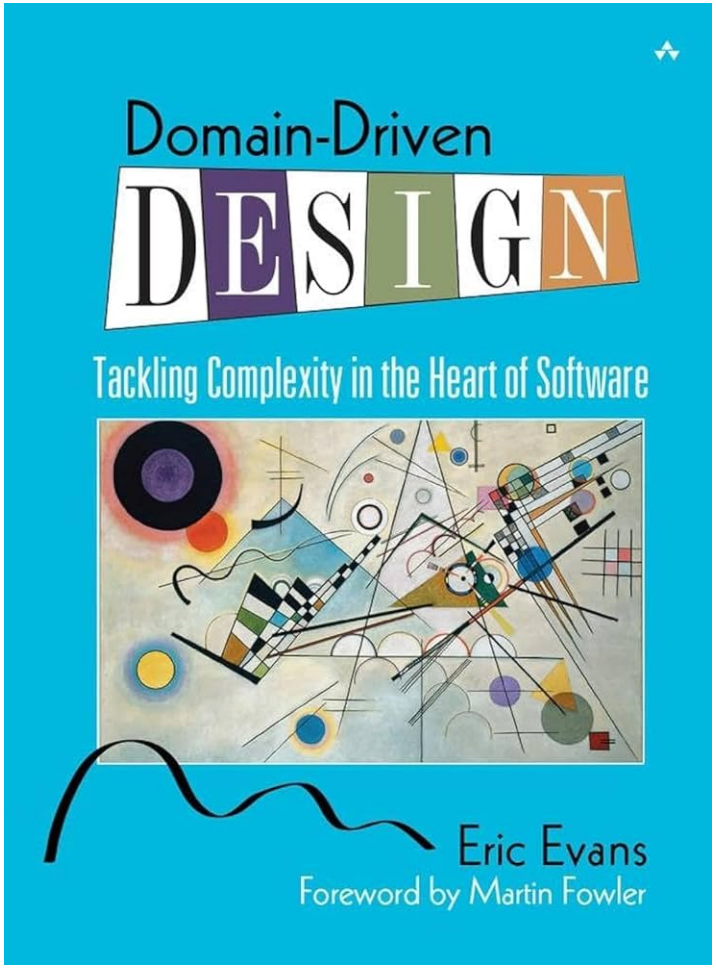
- a software design approach
- maintainable and scalable software
- set of practices, patterns, and philosophy
- focus on communication
    - the common root cause of project failures

Origins

- Martin Fowler: Pattern of Enterprise Application Architecture (https://martinfowler.com/books/eaa.html) [2002]
- Eric Evans: Domain-Driven Design: Tackling Complexity in the Heart of Software [2003]

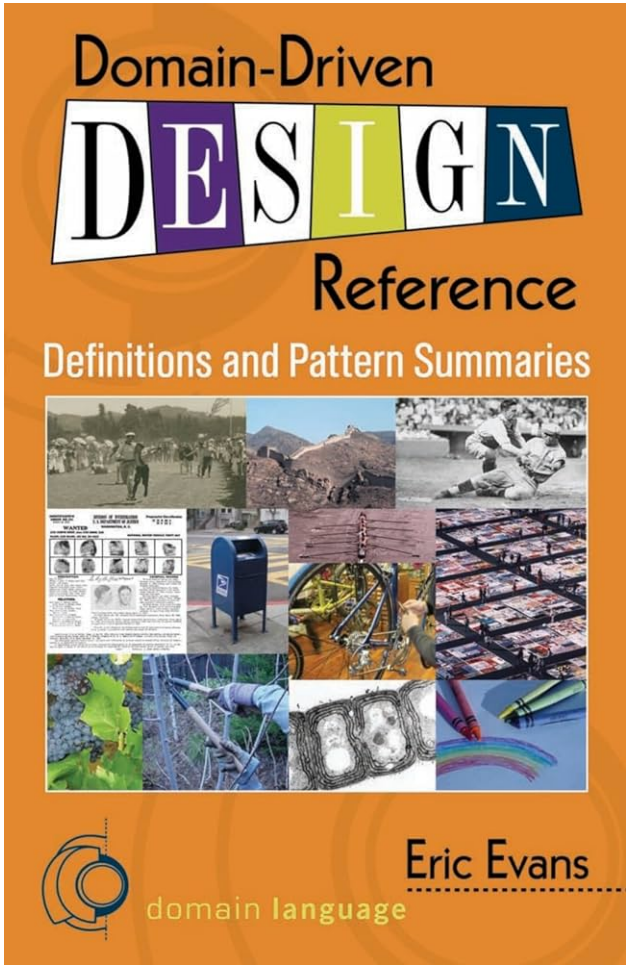
Origins

- "The Blue Book" (https://www.oreilly.com/library/view/domain-driven-design-tackling/0321125215/)



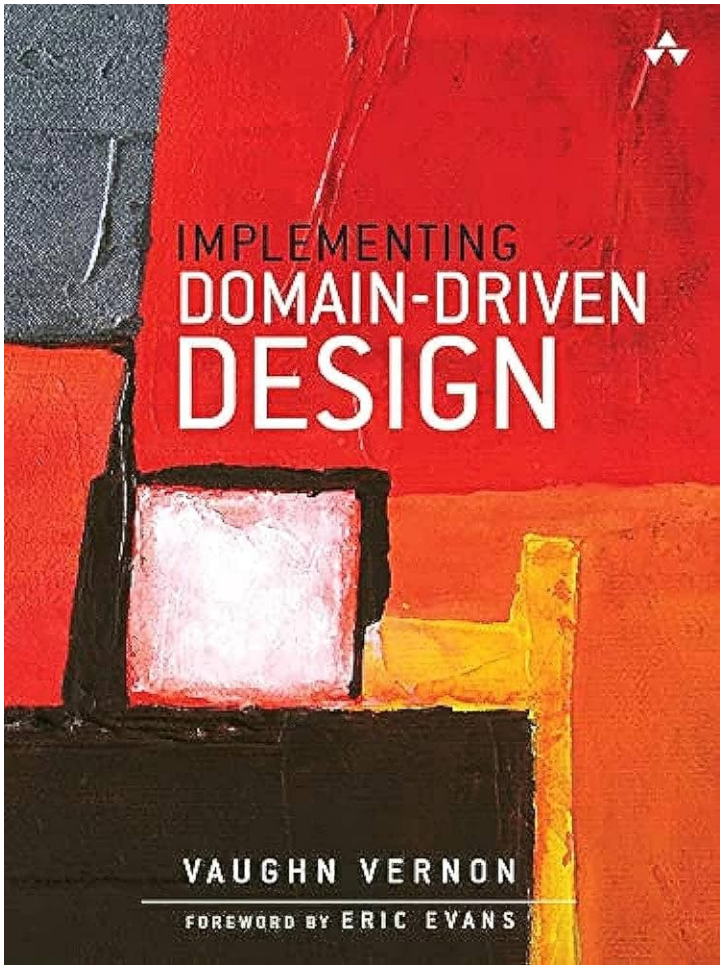
Origins

- "Reference" (https://www.domainlanguage.com/wp-content/uploads/2016/05/DDD_Reference_2015-03.pdf)



Origins

- Vaughn Vernon [2013]
- **"The Red Book"** (https://www.amazon.com/Implementing-Domain-Driven-Design-Vaughn-Vernon/dp/0321834577)



The Three Pillars of DDD

- ubiquitous language
- strategic design
- tactical design


push state change with
DOMAIN EVENTS
access with
REPOSITORIES
access with
SERVICES
express model with
express change with
ENTITIES
encapsulate with
act as root of
express identity with
encapsulate with
AGGREGATES
express state & computation with
VALUE OBJECTS
encapsulate with
MODEL-DRIVEN DESIGN
model gives structure to
isolate domain expressions with
encapsulate with
LAYERED ARCHITECTURE
define model within
encapsulate with
FACTORIES
UBIQUITOUS LANGUAGE
cultivate rich model with
keep model unified by
CONTINUOUS INTEGRATION
names enter
BOUNDED CONTEXT
SHARED KERNEL
SHARED KERNEL
interdependent contexts form
overlap allied contexts through
CORE DOMAIN
relate allied contexts as
CUSTOMER /SUPPLIER
work in autonomous, clean
assess/overview relationships with
CONTEXT MAP
minimize translation
CONFORMIST
avoid overinvesting in
support multiple clients through
segregate the conceptual messes
free teams to go
OPEN HOST SERVICE
GENERIC SUBDOMAINS
translate and insulate unilaterally with
loosely couple contexts through
formalize
BIG BALL OF MUD
ANTI-CORRUPTION LAYER
SEPARATE WAYS
PUBLISHED LANGUAGE

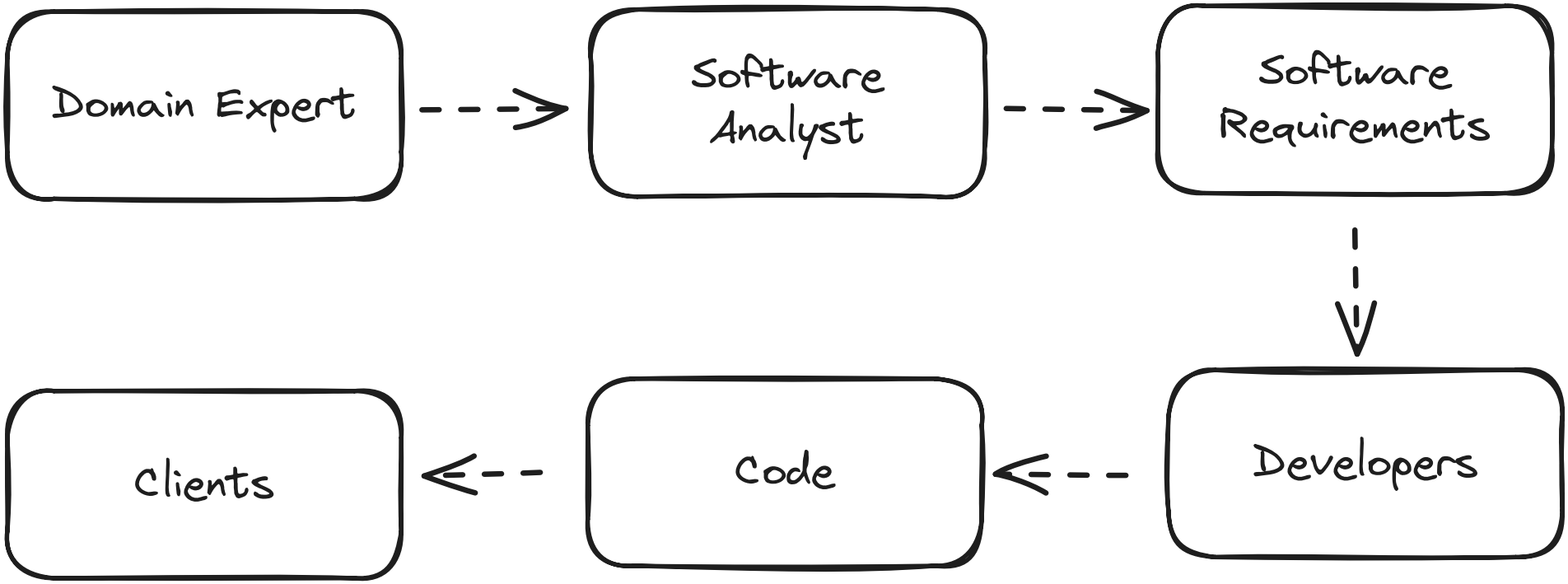
Ubiquitous Language

Ubiquitous Language

- common language shared throughout the business
- use the same terminology
- must be precise and consistent
  - synonymous and ambiguous terms
  - glossary
  - a shared effort
- domain experts & domain knowledge
- sharing across all stakeholders
- evolves

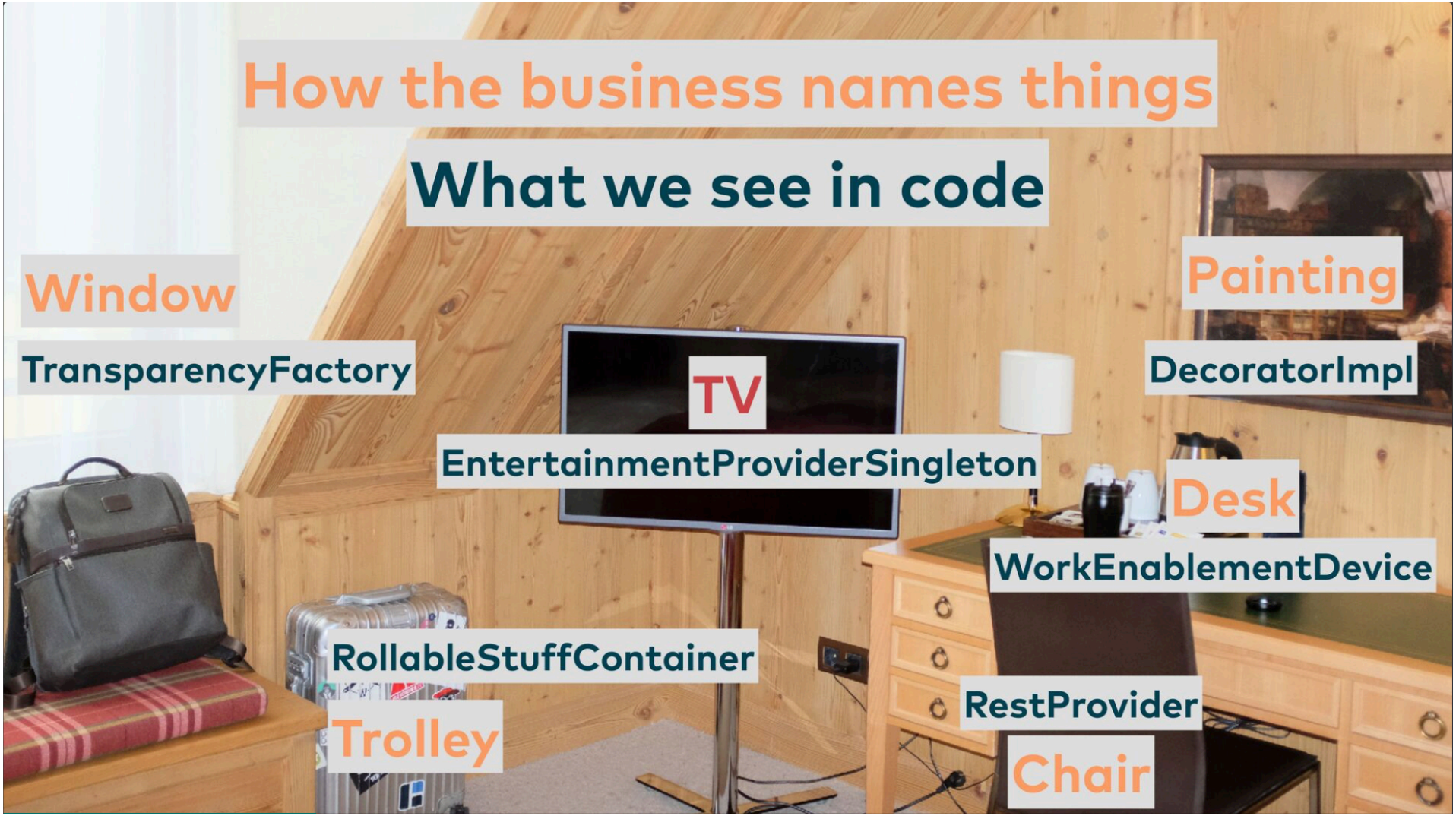
Traditional Knowledge Flow

Telephone



Michael Plod: Getting modules right with Domain-driven Design (https://www.youtube.com/watch?v=Q_0XW46llHY)

[Spring I/O, 2022]



Michael Plöd: Getting modules right with Domain-driven Design (https://www.youtube.com/watch?v=Q_0XW46llHY)

[Spring I/O, 2022]

Strategic Design

Strategic Design

- analysis of business domains and strategy
- *"what"* & *"why"*
- patterns:
  - domain
  - subdomain
  - bounded context

Domain

- focus of business

Subdomain

- fine-grained area of business activity
- a company can operate in multiple domains

Domain Types

- can change as domains evolve
- help make design decisions
- types:
  - core domain
  - supporting domain
  - generic domain

Core Subdomain

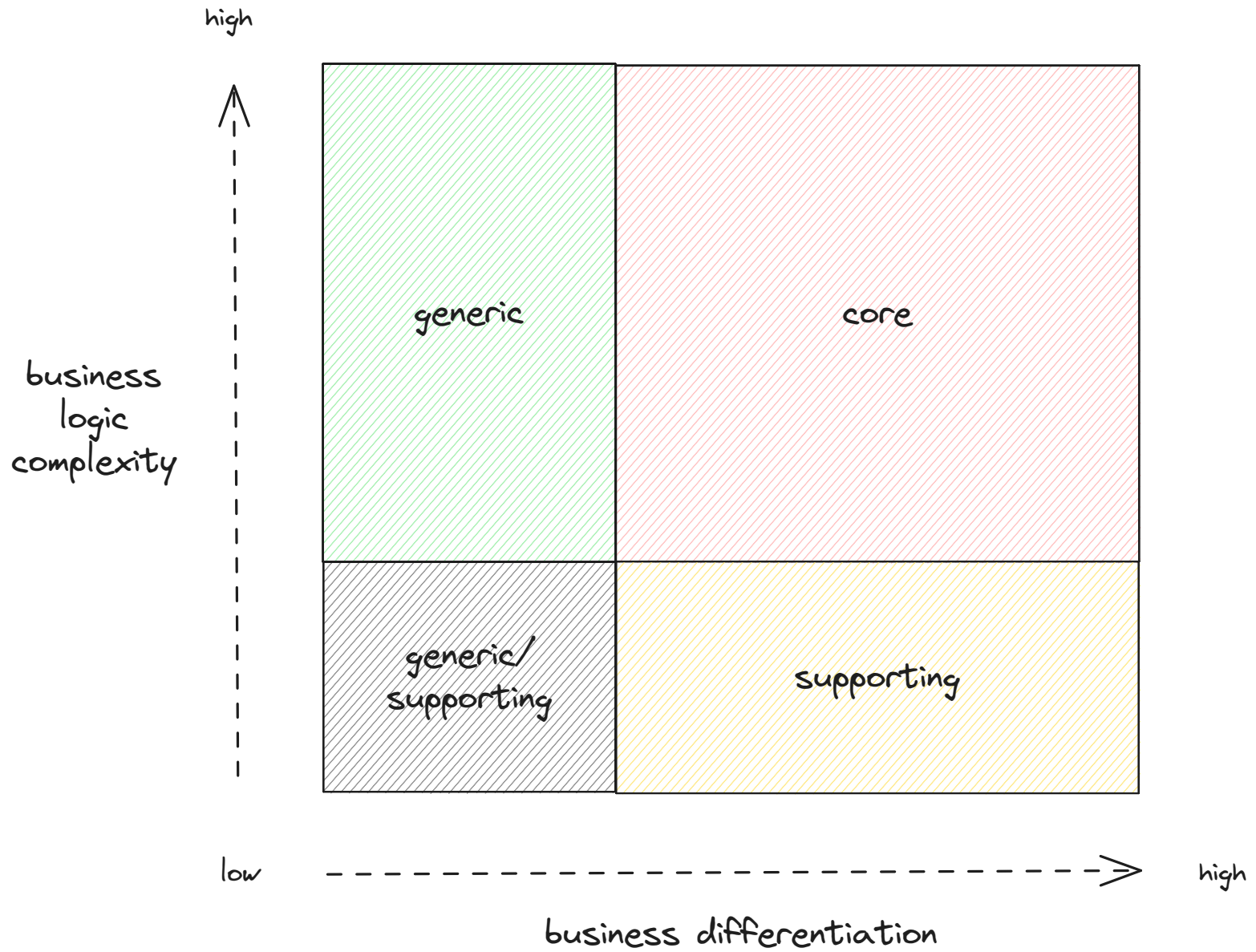
- the center of business
- brings in money
- provides a competitive advantage
- complex invariants and rules
- built in-house

Supporting Subdomain

- support core subdomains
- usually CRUD apps
- built in-house or out-sourced

Generic Subdomain

- generic systems
- no need for innovation
- buy existing or integrate OS solution
- CMS, CRM systems ...

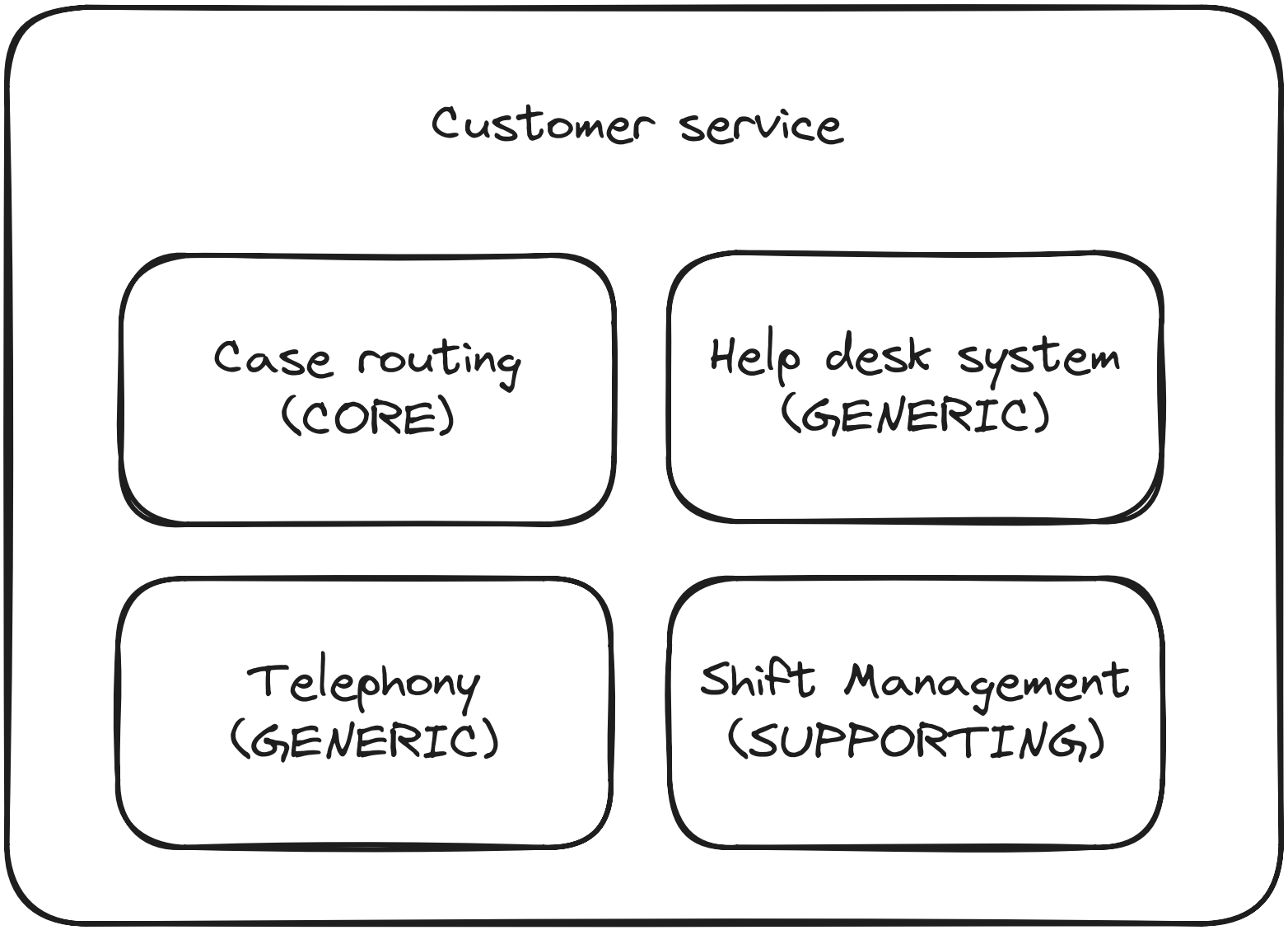

high
business
logic
complexity
generic
core
generic/
supporting
supporting
low
high
business differentiation

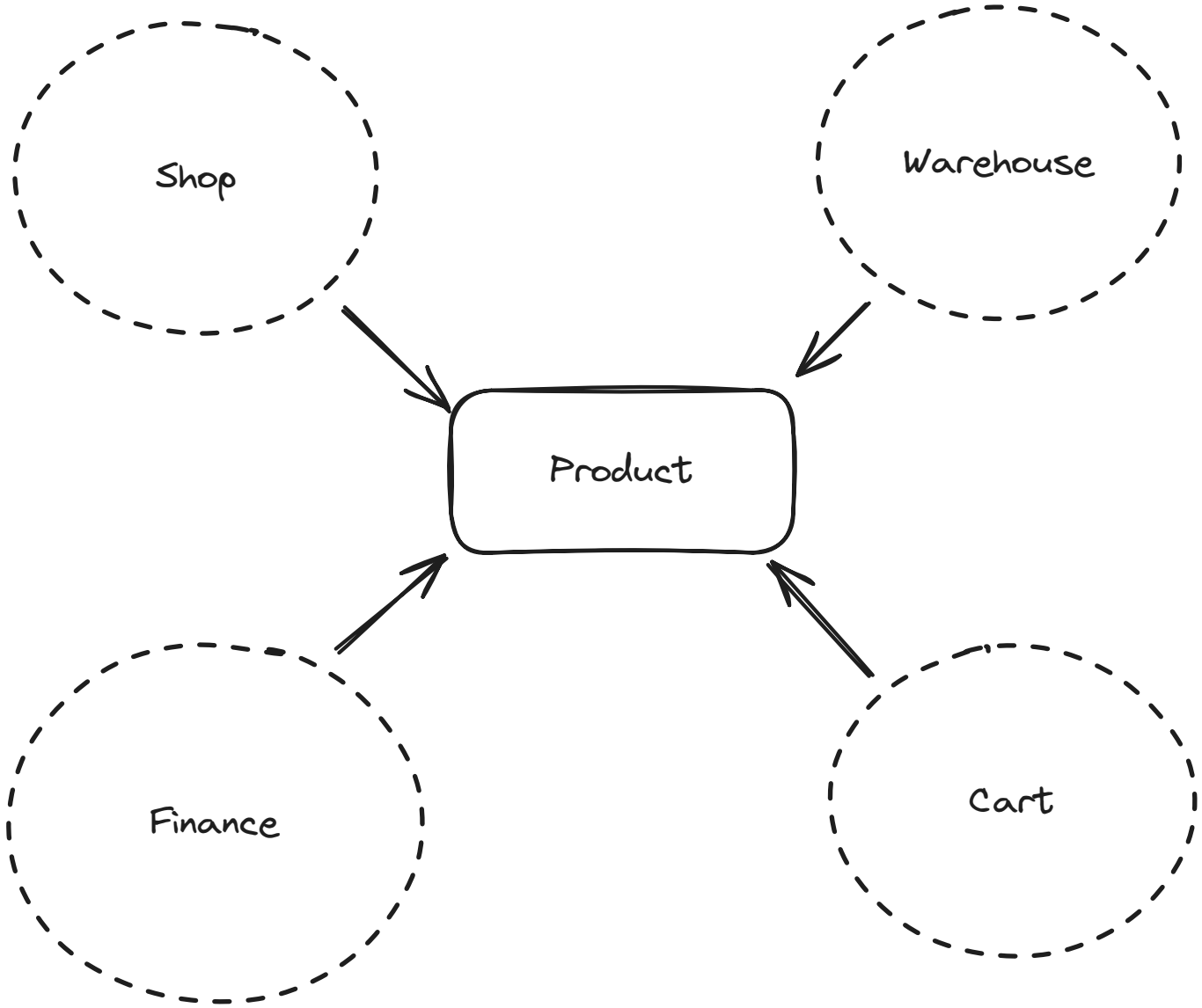
Identifying domains

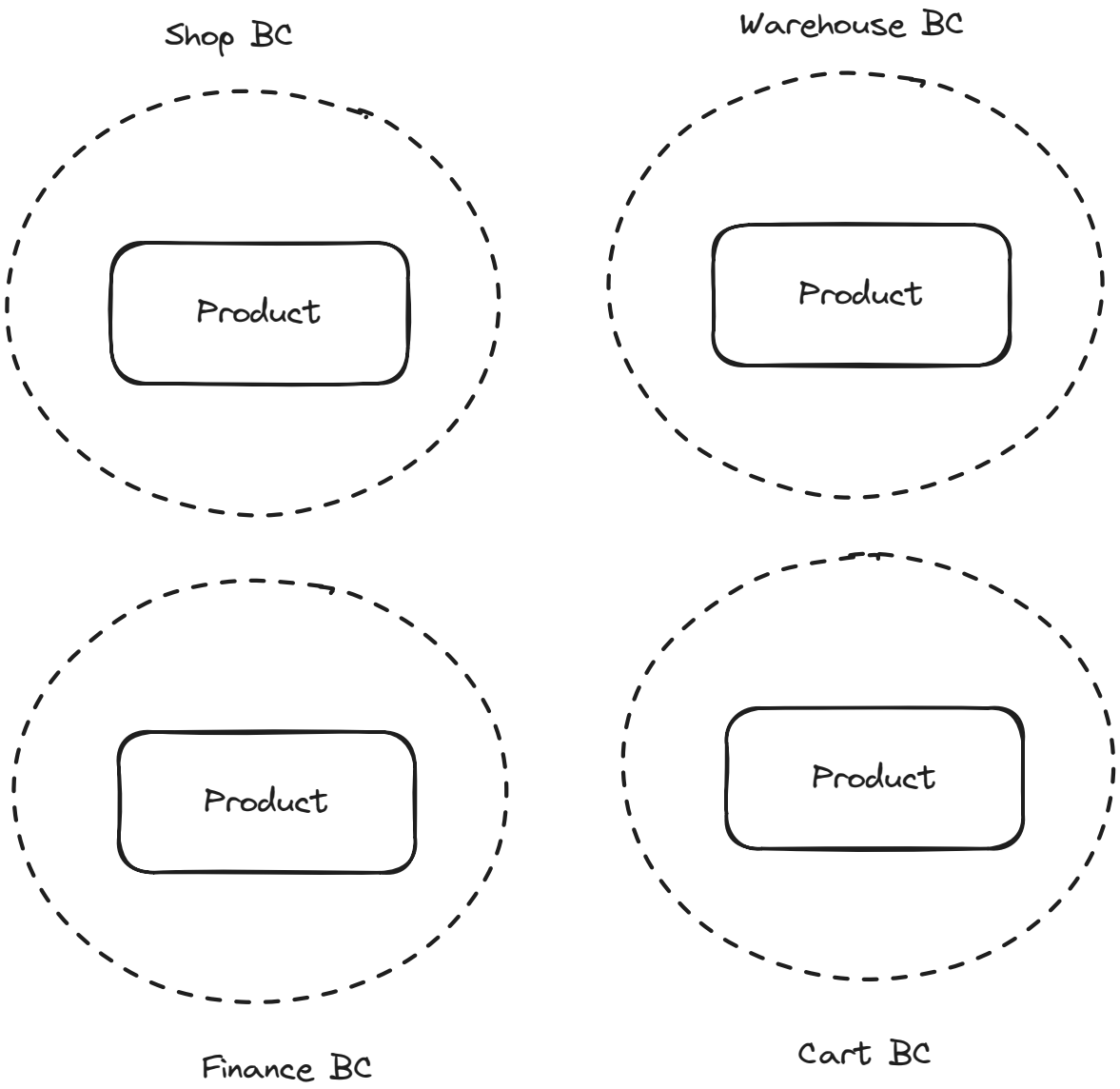
- company departments
- a retail shop:
  - marketing
  - finance
  - **customer service**
  - shipping
  - warehouse
  - ...

Identifying domains

- customer service:
  - help desk system
  - shift management
  - telephone system
  - ...
- more can be discovered later


Customer service
Case routing
(CORE)
Help desk system
(GENERIC)
Telephony
(GENERIC)
Shift Management
(SUPPORTING)


Shop
Warehouse
Product
Finance
Cart


Shop BC
Product
Warehouse BC
Product
Product
Finance BC
Product
Cart BC

Bounded Context

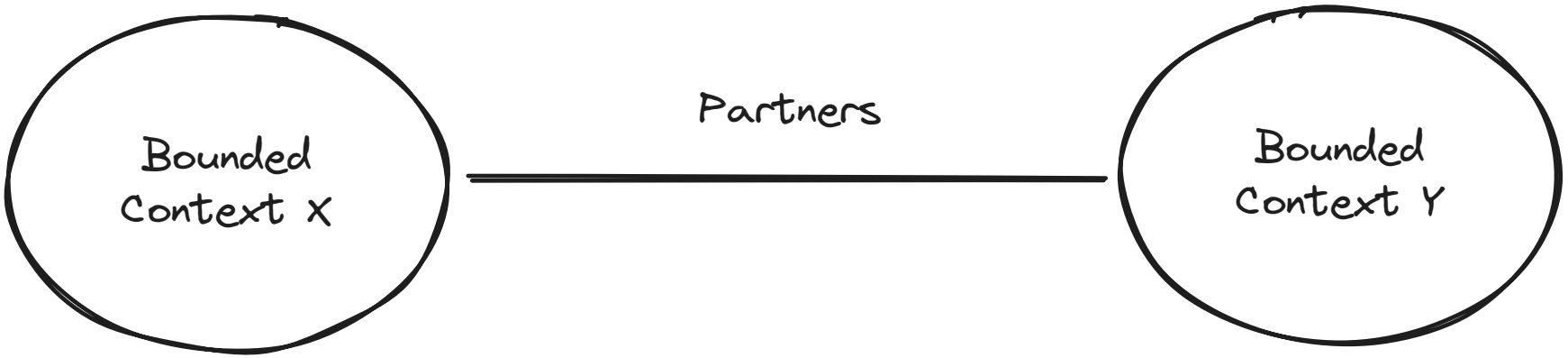
- divides ubiquitous language
- relates to business in a specific area
- designed
- protects models
- specifies physical boundaries
  - service/project
  - single team

Relationships

- cooperation (symmetric)
  - partnership
  - shared kernel
- customer-supplier (asymmetric)
  - customer-supplier
  - conformist
  - open-host service, published language
  - anticorruption layer

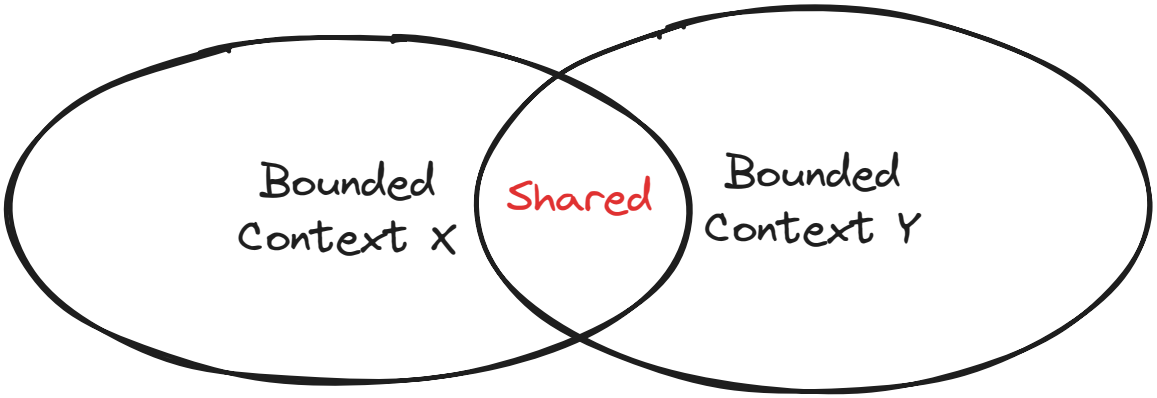
Partnership

- changes are in an ad-hoc manner
- dependent sets of goals
- frequent synchronizations and communication
- no team dictates the contracts



Shared Kernel

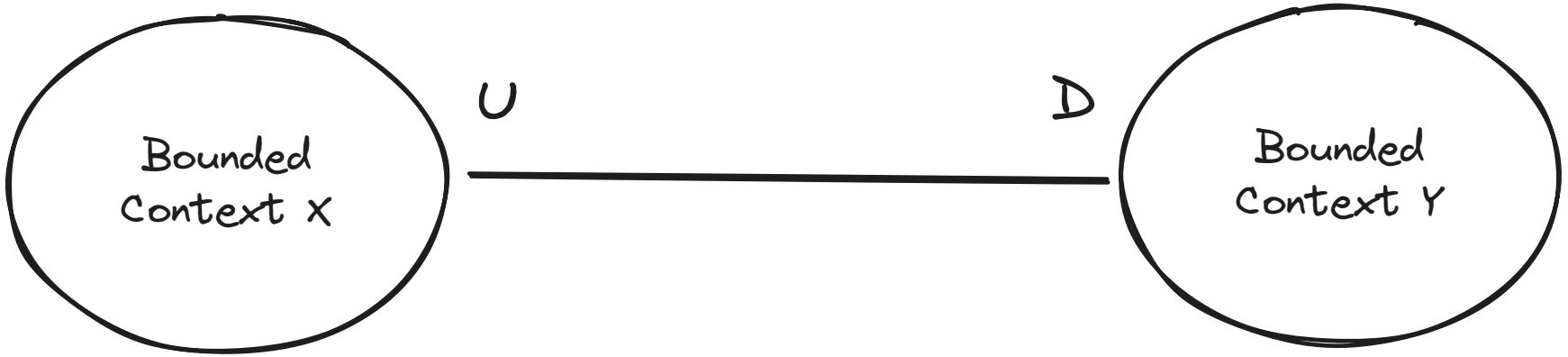
- shared code between teams
  - monorepo
  - library
- sharing should be minimal



...Separate Ways

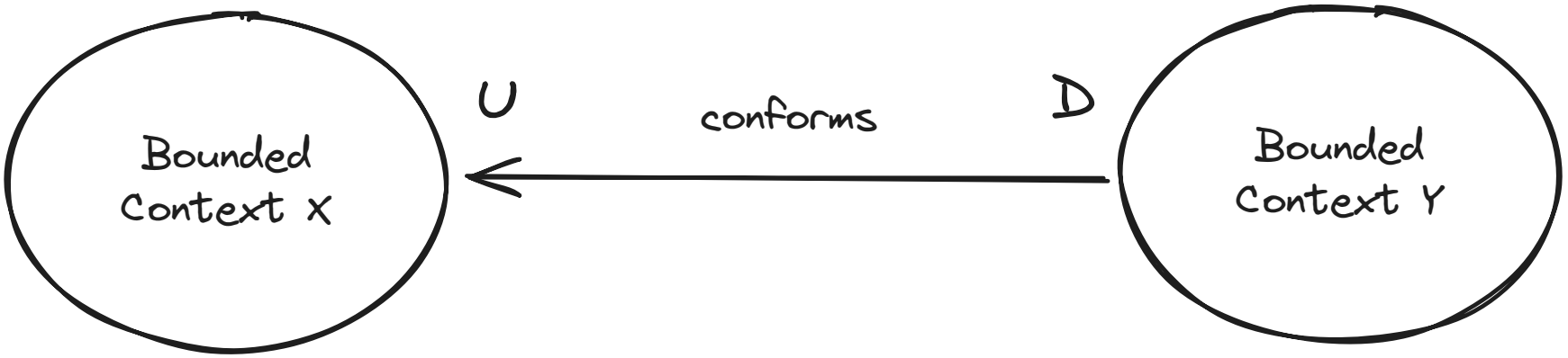
Customer-Supplier

- cooperation between teams
- the upstream team reflects the requirements of the downstream team



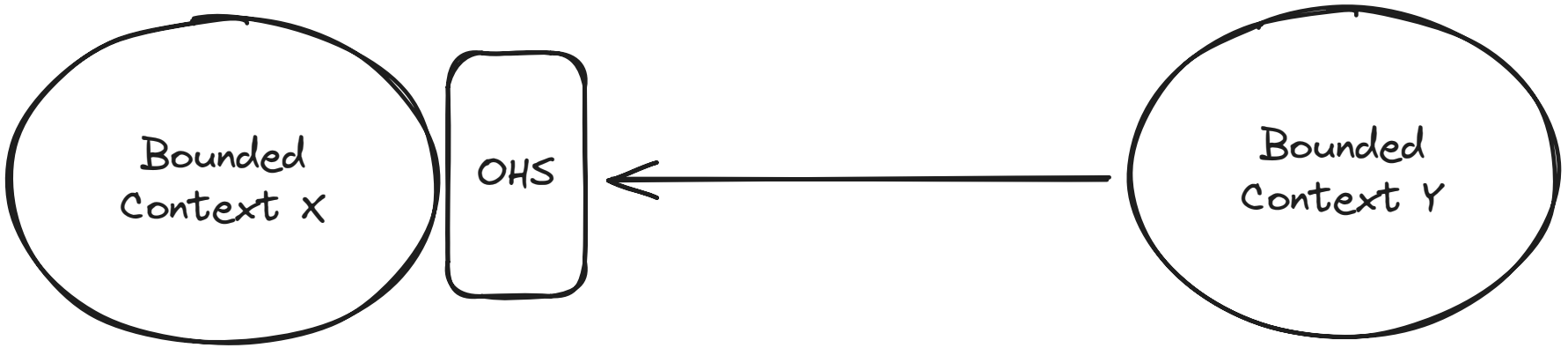
Conformist

- upstream is no longer capable of supporting the downstream team
- the downstream team conforms to the upstream



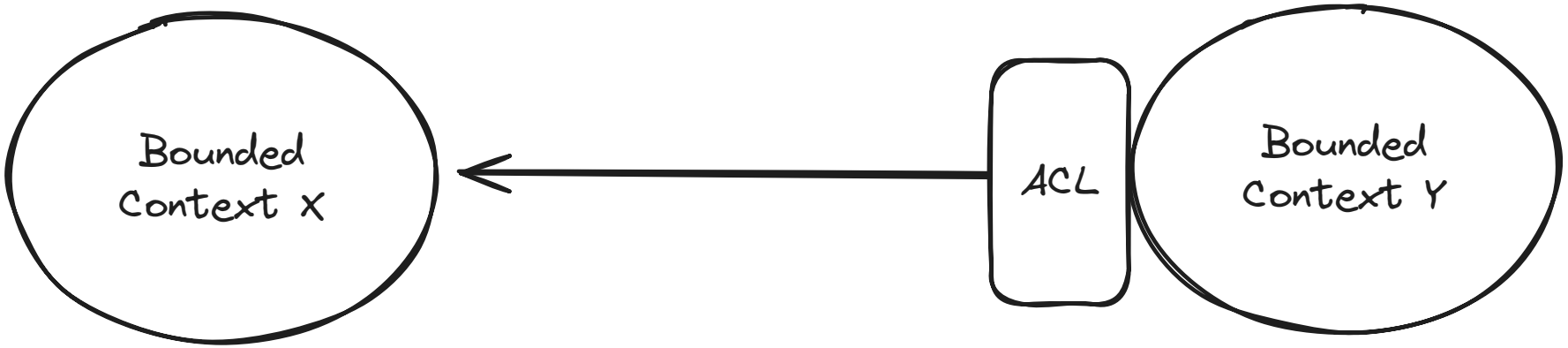
Open Host Service

- service defines a protocol or interface
- contract:
  - XML schema
  - OpenAPI spec
  - GraphQL schema
  - Protobuf
- often goes along with **Published Language**



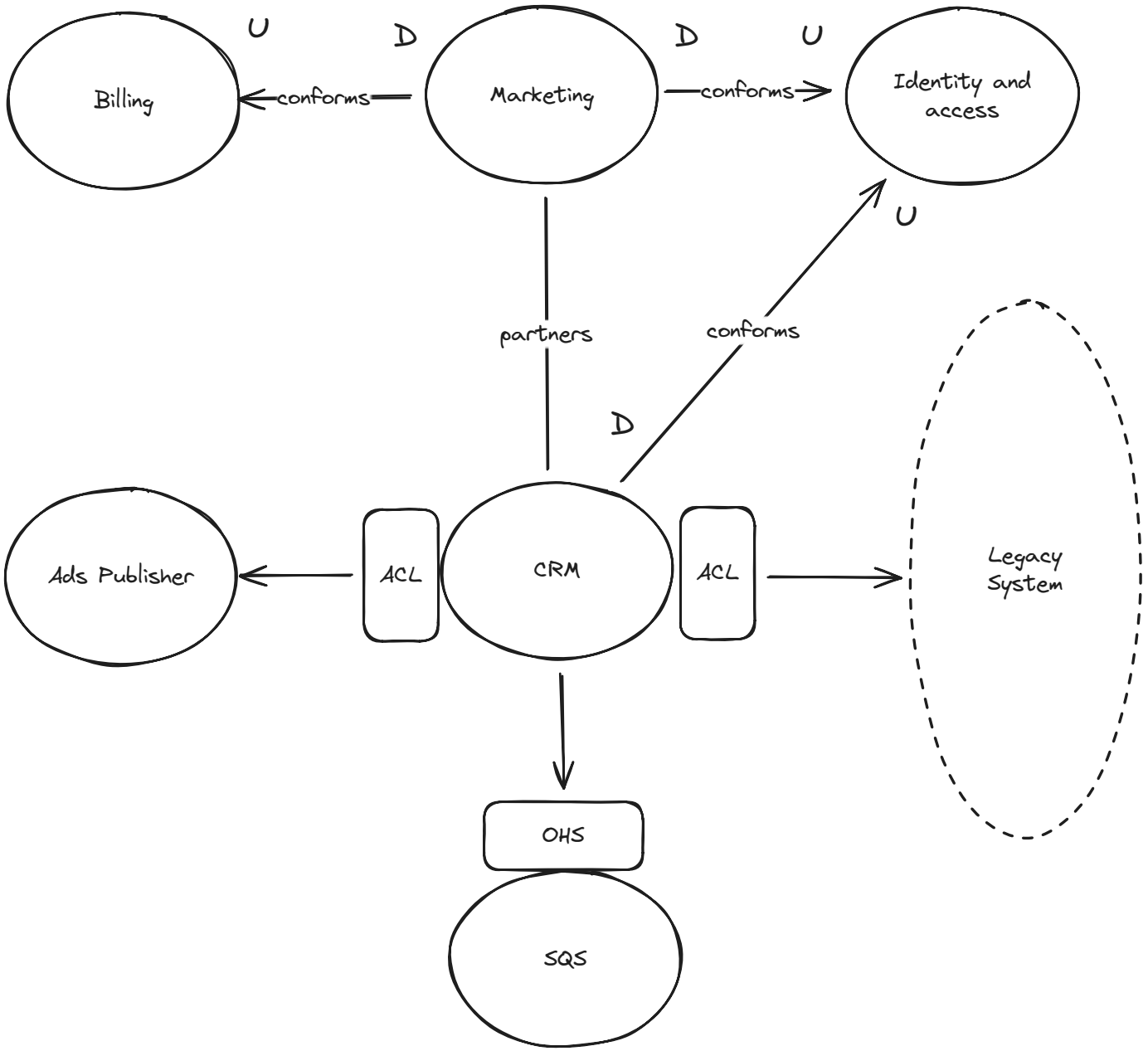
Anticorruption layer

- defensive pattern
- protects bounded context
- keeps it isolated from foreign models
- can be implemented as a proxy



Context Map

- specifies relationships between bounded contexts


Billing
U
conforms
D
Marketing
D
conforms
U
Identity and access
U
partners
conforms
D
Ads Publisher
ACL
CRM
ACL
Legacy System
OHS
SQS

Tactical Design

Tactical Design

- code focused
- helps us write software that reflects the business domain
  - tightly relate code to business domain models
- *"how"*
- patterns:
  - value object
  - entity
  - aggregate
  - event
  - service

Domain Model

- object model
- both behavior and data
- focus on implementing business logic
- building blocks:
  - value objects
  - entities
  - aggregates
  - event
  - service

Help desk system

Value Object

- immutable objects
- identified by it's values
- used to describe, quantify, or measure entities

```
type Colour struct {
    Red uint8
    Green uint8
    Buil uint8
}
```

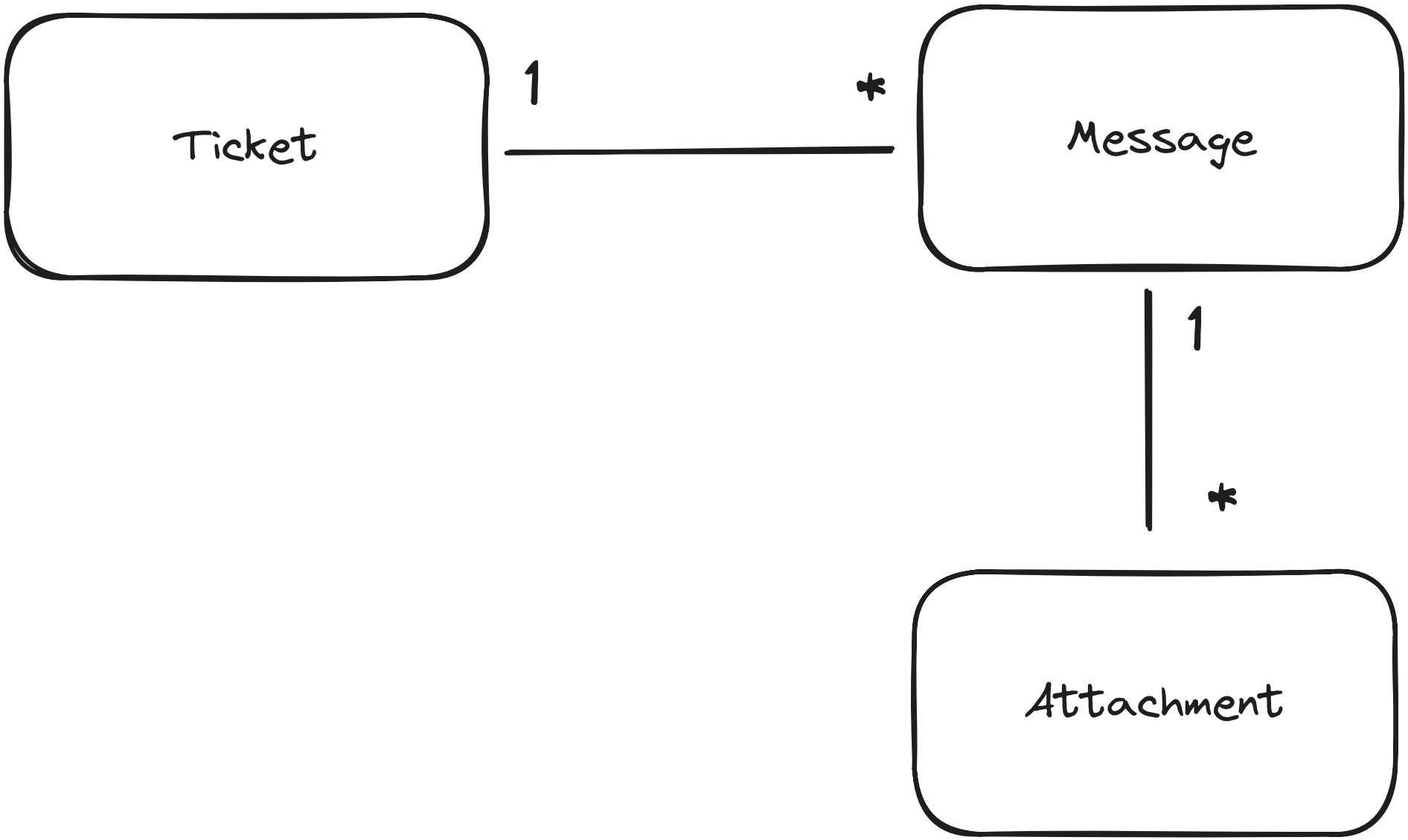
Entity

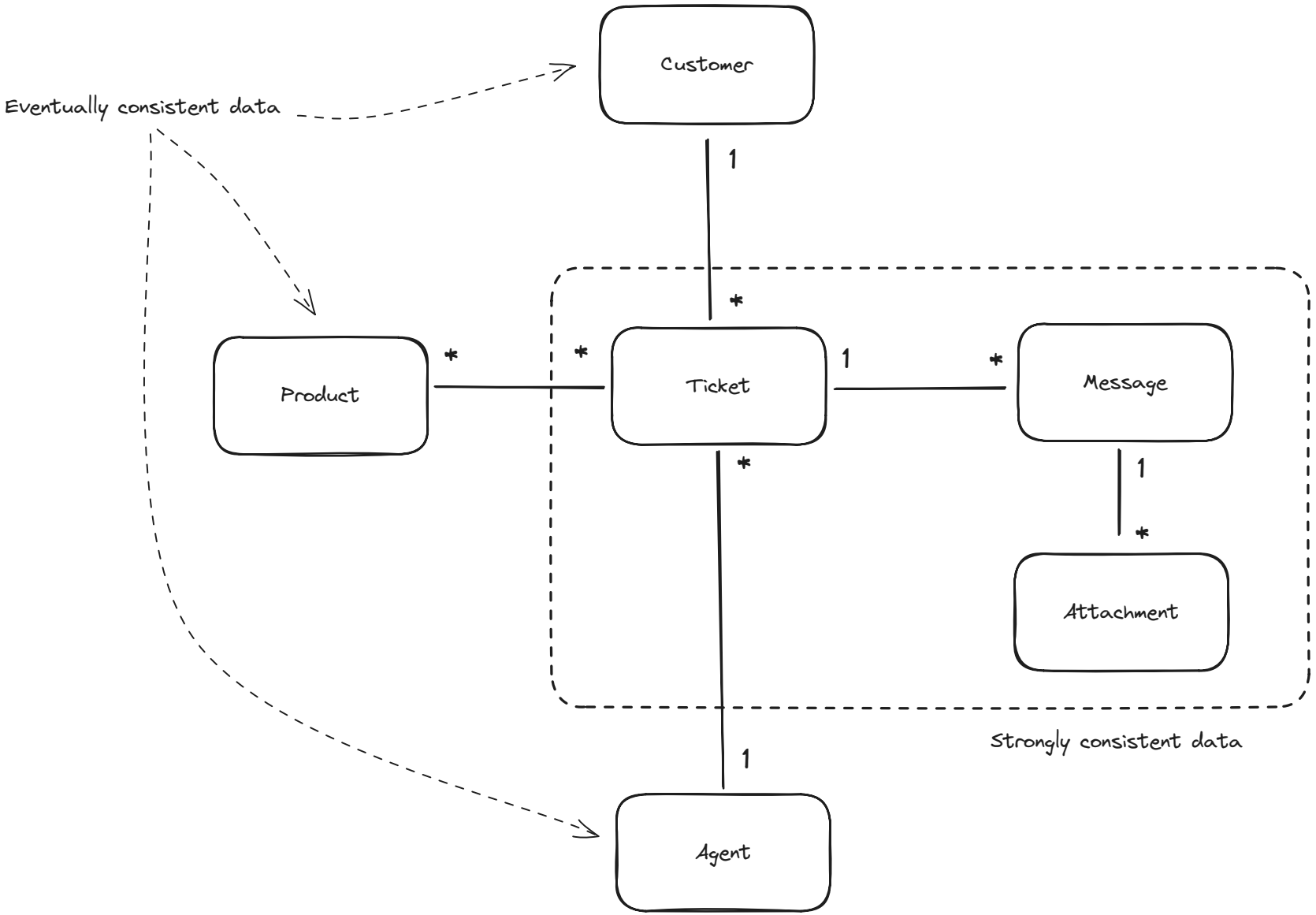
- identity
- identified by its identity
- mutable

```
type Customer struct {
    Id uuid.UUID
    Name string
    Surname string
    Email string
    Address Address
}
```

Aggregate

- a couple of interconnected objects
- single aggregate root
- enforces consistency of its data
- only the aggregate itself can change its state
- transaction boundary
- does not cross bounded context boundary
- other aggregates are referenced by their identity
- aggregate public interface methods are referred to as commands
- updates across aggregates are done using eventual consistency


Ticket
1
*
Message
1
*
Attachment

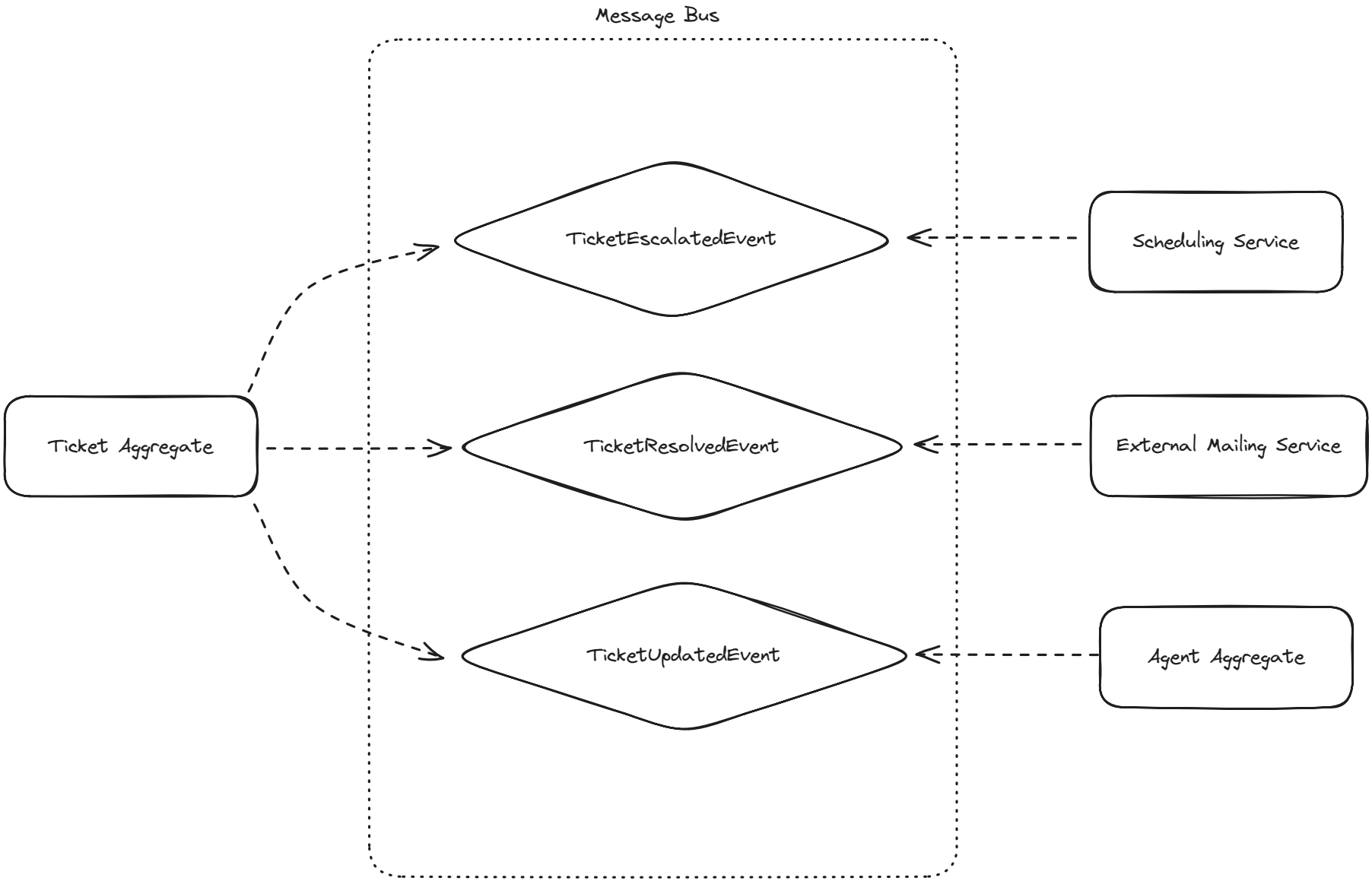

Customer
Eventually consistent data
1
*
Product
*
*
Ticket
1
*
Message
*
1
*
Attachment
Strongly consistent data
1
Agent

Domain Event

- a significant event that occurred
- part of aggregates public API
- past tense

```
type TicketResolvedEvent {
    TicketID uuid.UUID
    EventID uuid.UUID
    OpenedAt time.Time
    ResolvedAt time.Time
}
```

- **event sourcing**


Message Bus
TicketEscalatedEvent
Scheduling Service
Ticket Aggregate
TicketResolvedEvent
External Mailing Service
TicketUpdatedEvent
Agent Aggregate

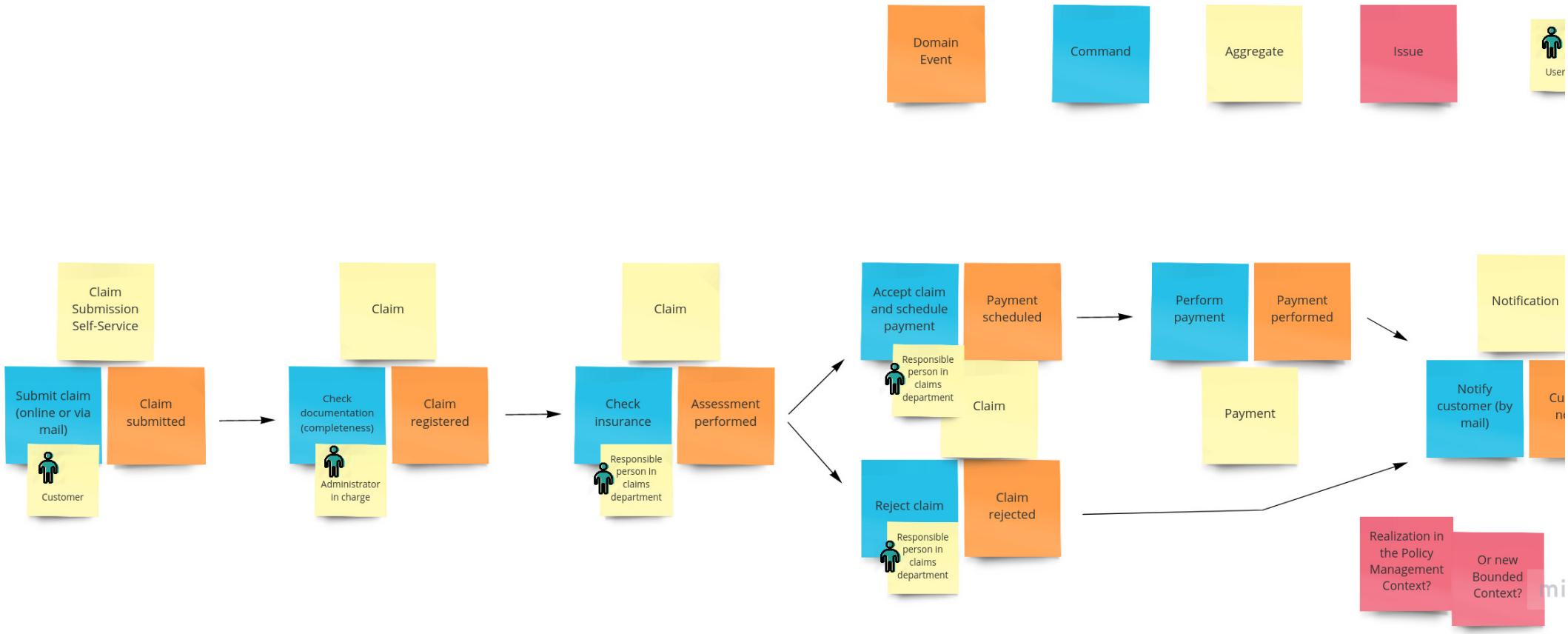
Domain Service

- stateless object
- abstracts common logic
- coordinates logic relevant to multiple aggregates

Event Storming

Event Storming Workshop

- Alberto Brandolini [2013]
- exploring business domains

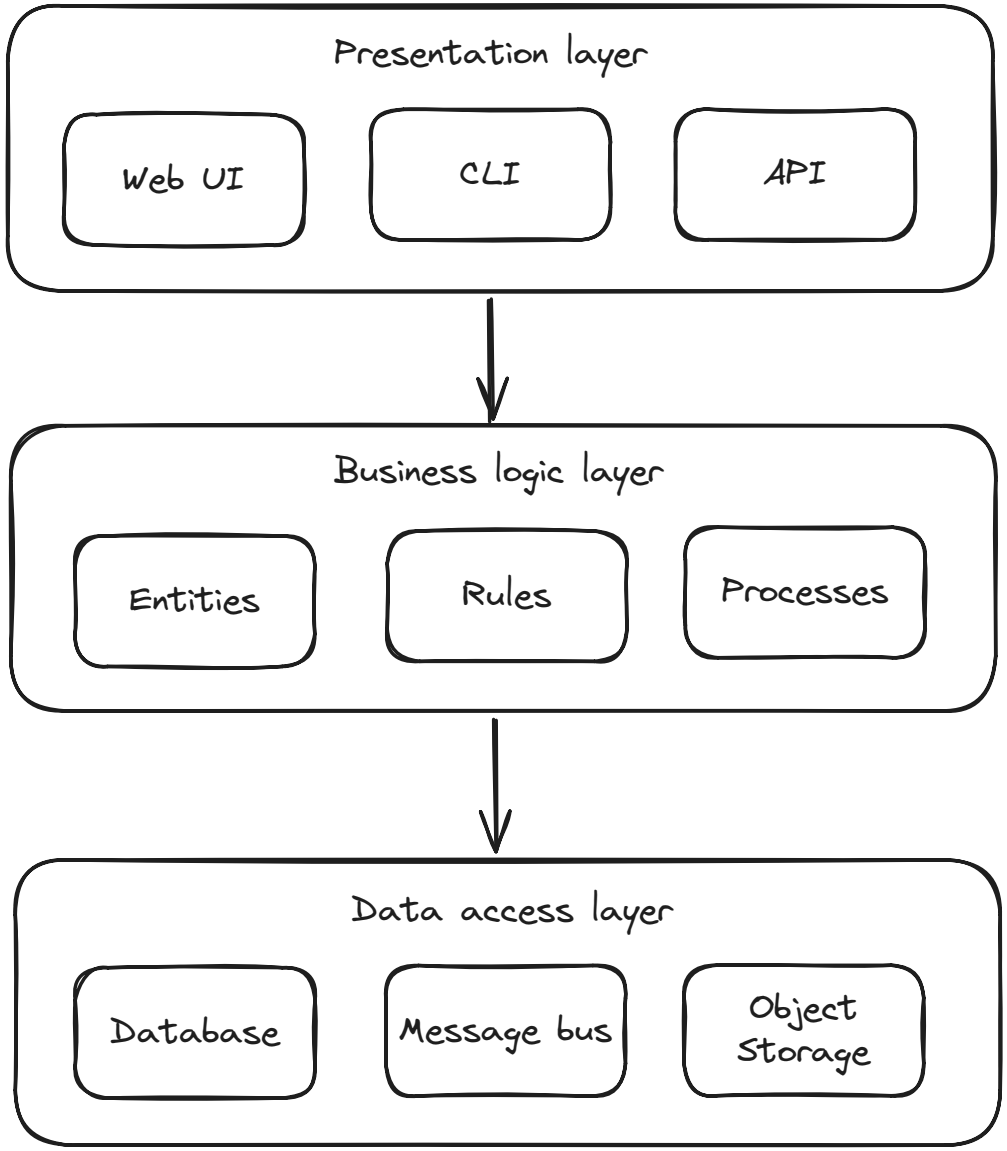


Context Mapper: Model Event Storming Results (https://contextmapper.org/docs/event-storming/)

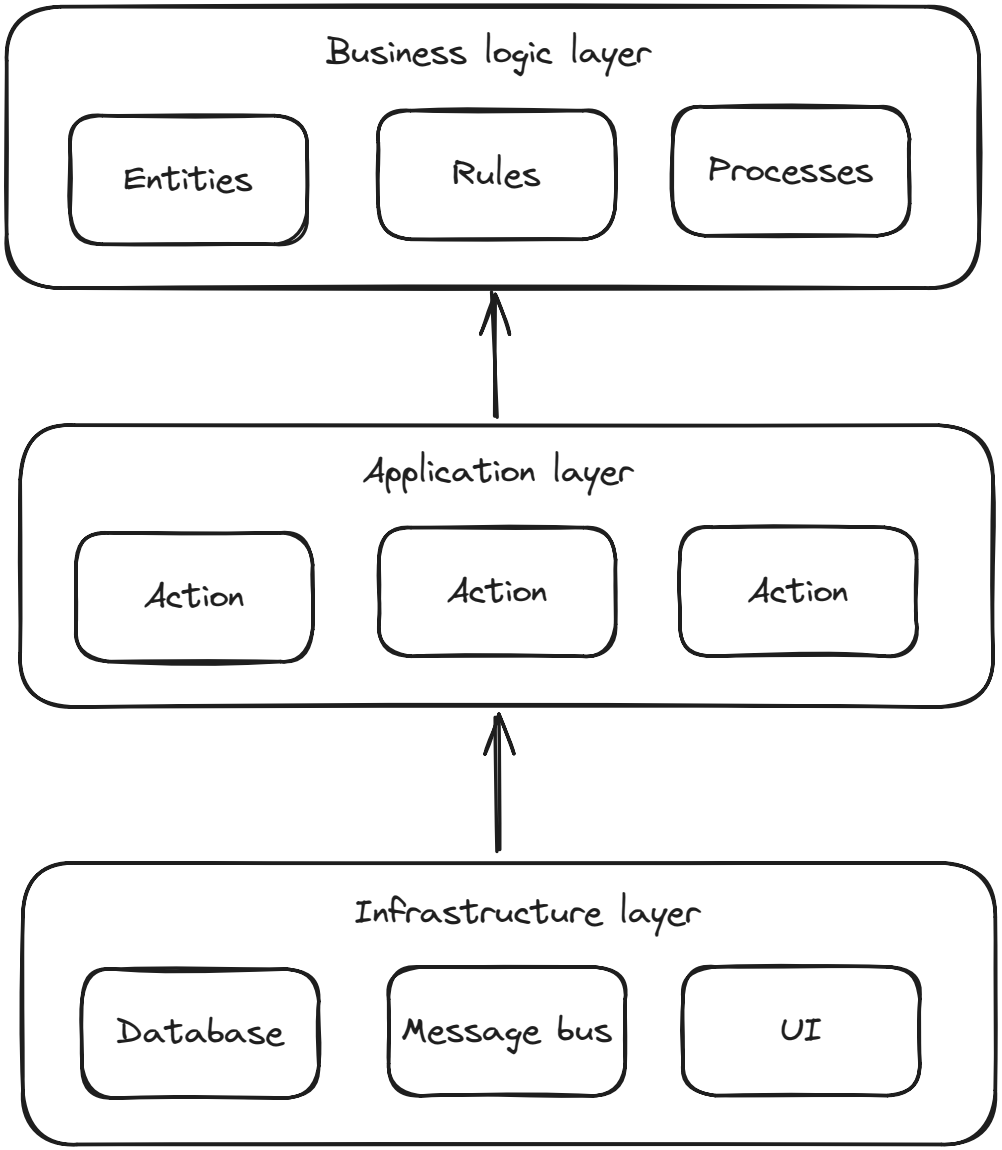

TIME

Architectural Patterns

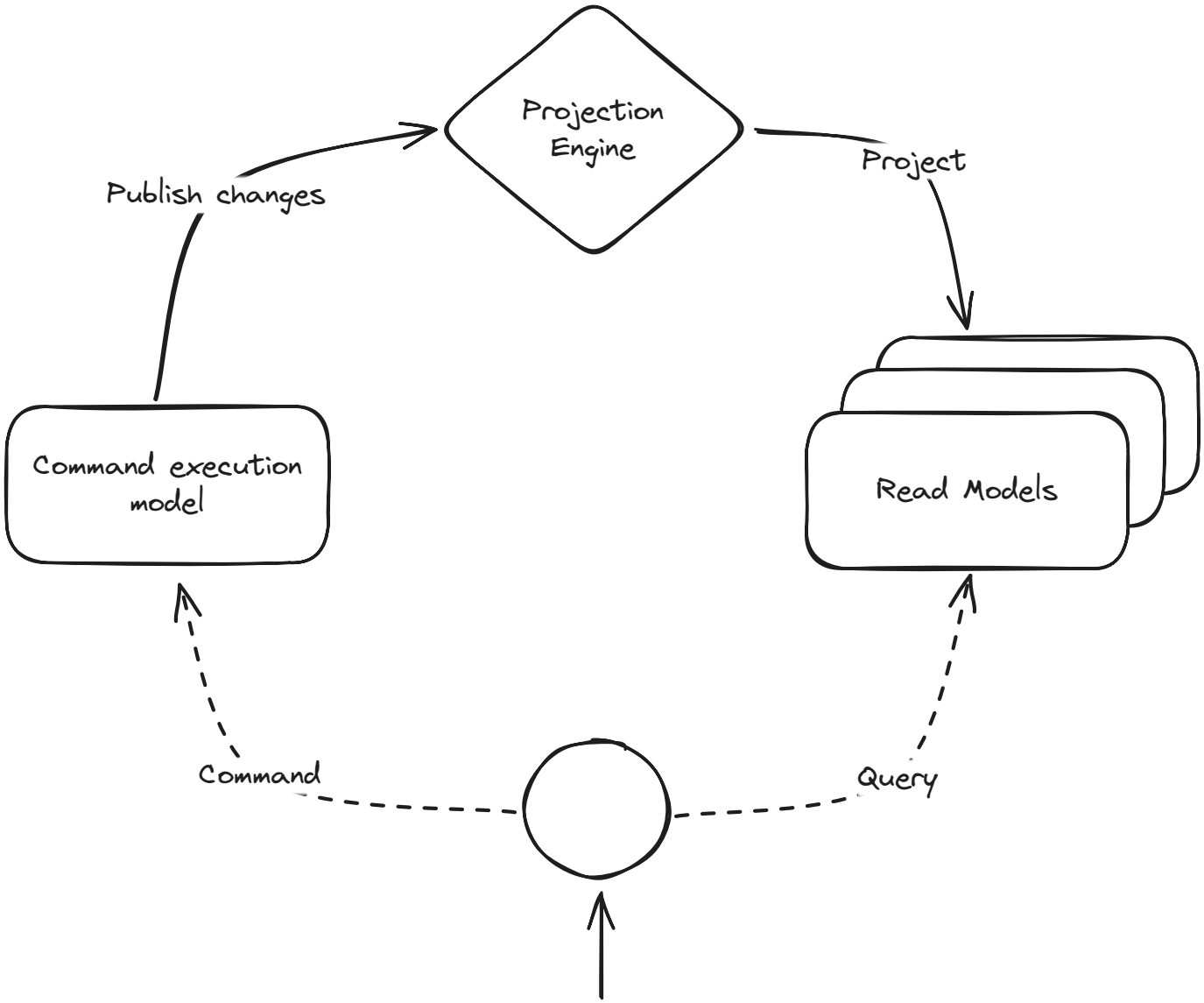
Layered Architecture



Ports & Adapters



CQRS


Projection Engine
Publish changes
Project
Command execution model
Read Models
Command
Query

Sample

ThreeDotsLab: Wild Workouts Go DDD example

- https://github.com/ThreeDotsLabs/wild-workouts-go-ddd-example

  (https://github.com/ThreeDotsLabs/wild-workouts-go-ddd-example)

- continuous refactorings with articles
- DDD, Clean Architecture & CQRS
- Domain: Workout information system. Users can schedule trainings with their trainers. Trainers can set their availability via a calendar.
- Bounded Context: Reservation System
- 3 microservices:
  - trainers
  - **trainings**
  - users

Structure

- **/internal**
  - directory for each service
  - shared **common** directory
    - decorators, middleware
    - logs, genproto, errors
- each service
  - **adapters** (infrastructure)
  - **app** (app)
  - **domain** (domain)
  - **ports**
  - **service**

Domain: Training Domain Model

```
type Training struct {
    uuid string
    userUUID string
    userName string
    time  time.Time
    notes string
    proposedNewTime time.Time
    moveProposedBy  UserType
    canceled bool
}
```

```
func (t *Training) RescheduleTraining(newTime time.Time) error {
    if !t.CanBeCanceledForFree() {
        err := CantRescheduleBeforeTimeError{
            TrainingTime: t.Time(),
        }
        return errors.WithStack(err)
    }

    t.time = newTime

    return nil
}
```

Domain: Repository

```
type NotFoundError struct {
    TrainingUUID string
}

func (e NotFoundError) Error() string {
    return fmt.Sprintf("training '%s' not found", e.TrainingUUID)
}

type Repository interface {
    AddTraining(ctx context.Context, tr *Training) error

    GetTraining(ctx context.Context, trainingUUID string, user User) (*Training, error)

    UpdateTraining(
        ctx context.Context,
        trainingUUID string,
        user User,
        updateFn func(ctx context.Context, tr *Training) (*Training, error),
    ) error
}
```

App: Application

- **query** and **comand** directories

```
type Application struct {
    Commands Commands
    Queries  Queries
}

type Commands struct {
    ApproveTrainingReschedule command.ApproveTrainingRescheduleHandler
    CancelTraining            command.CancelTrainingHandler
    RejectTrainingReschedule  command.RejectTrainingRescheduleHandler
    RescheduleTraining        command.RescheduleTrainingHandler
    RequestTrainingReschedule command.RequestTrainingRescheduleHandler
    ScheduleTraining          command.ScheduleTrainingHandler
}

type Queries struct {
    AllTrainings     query.AllTrainingsHandler
    TrainingsForUser query.TrainingsForUserHandler
}
```

App: Reschedule Training Command

```
type RescheduleTraining struct {
    TrainingUUID string
    NewTime      time.Time

    User training.User

    NewNotes string
}

type RescheduleTrainingHandler decorator.CommandHandler[RescheduleTraining]

type rescheduleTrainingHandler struct {
    repo           training.Repository
    userService    UserService
    trainerService TrainerService
}
```

App: Reschedule Training Command Handler - Constructor

```
func NewRescheduleTrainingHandler(
    repo training.Repository,
    userService UserService,
    trainerService TrainerService,
    logger *logrus.Entry,
    metricsClient decorator.MetricsClient,
) RescheduleTrainingHandler {
    if repo == nil {
        panic("nil repo")
    }
    if userService == nil {
        panic("nil userService")
    }
    if trainerService == nil {
        panic("nil trainerService")
    }

    return decorator.ApplyCommandDecorators[RescheduleTraining](
        rescheduleTrainingHandler{repo: repo, userService: userService, trainerService: trainerServi
        logger,
        metricsClient,
    )
}
```

App: Reschedule Training Command - Handle

```
func (h rescheduleTrainingHandler) Handle(ctx context.Context, cmd RescheduleTraining) (err error) {
    defer func() {
        logs.LogCommandExecution("RescheduleTraining", cmd, err)
    }()
    return h.repo.UpdateTraining(
        ctx,
        cmd.TrainingUUID,
        cmd.User,
        func(ctx context.Context, tr *training.Training) (*training.Training, error) {
            originalTrainingTime := tr.Time()
            if err := tr.UpdateNotes(cmd.NewNotes); err != nil {
                return nil, err
            }
            if err := tr.RescheduleTraining(cmd.NewTime); err != nil {
                return nil, err
            }
            err := h.trainerService.MoveTraining(ctx, cmd.NewTime, originalTrainingTime)
            if err != nil {
                return nil, err
            }
            return tr, nil
        },
    )
}
```

Service: Application Service

```
type HttpServer struct {
    app app.Application
}

func (h HttpServer) RescheduleTraining(w http.ResponseWriter, r *http.Request, trainingUUID string)
    rescheduleTraining := PostTraining{}
    if err := render.Decode(r, &rescheduleTraining); err != nil {
        httperr.BadRequest("invalid-request", err, w, r)
        return
    }
    user, err := newDomainUserFromAuthUser(r.Context())
    if err != nil {
        httperr.RespondWithSlugError(err, w, r)
        return
    }
    err = h.app.Commands.RescheduleTraining.Handle(r.Context(), command.RescheduleTraining{
        User:         user,
        TrainingUUID: trainingUUID,
        NewTime:      rescheduleTraining.Time,
        NewNotes:     rescheduleTraining.Notes,
    })
    // error handling...
}
```

Infrastructure: Training Firebase Repository

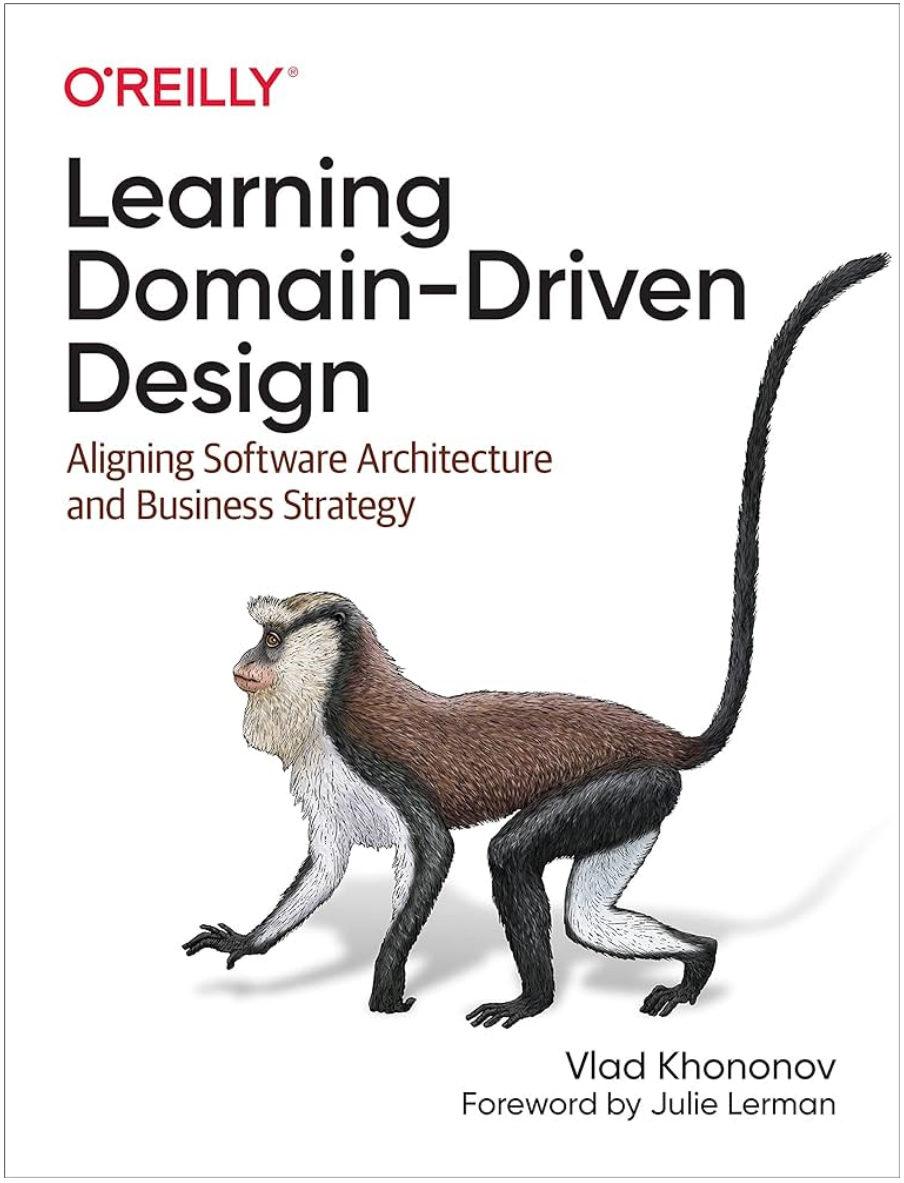
```
type TrainingModel struct {
    UUID     string `firestore:"Uuid"`
    UserUUID string `firestore:"UserUuid"`
    User     string `firestore:"User"`

    Time  time.Time `firestore:"Time"`
    Notes string    `firestore:"Notes"`

    ProposedTime   *time.Time `firestore:"ProposedTime"`
    MoveProposedBy *string    `firestore:"MoveProposedBy"`

    Canceled bool `firestore:"Canceled"`
}

type TrainingsFirestoreRepository struct {
    firestoreClient *firestore.Client
}
```

Infrastructure: Training Firebase Repository

```
func (r TrainingsFirestoreRepository) UpdateTraining(ctx context.Context, trainingUUID string,
    user training.User, updateFn func(ctx context.Context, tr *training.Training) (*training.Trainin
) error {
    trainingsCollection := r.trainingsCollection()
    return r.firestoreClient.RunTransaction(ctx, func(ctx context.Context, tx *firestore.Transaction
        documentRef := trainingsCollection.Doc(trainingUUID)
        firestoreTraining, err := tx.Get(documentRef)
        if err != nil {
            return errors.Wrap(err, "unable to get actual docs")
        }
        tr, err := r.unmarshalTraining(firestoreTraining)
        if err != nil {
            return err
        }
        if err := training.CanUserSeeTraining(user, *tr); err != nil {
            return err
        }
        updatedTraining, err := updateFn(ctx, tr)
        if err != nil {
            return err
        }
        return tx.Set(documentRef, r.marshalTraining(updatedTraining))
    })
}
```

Conclusion

- pros:
  - better communication
  - flexibility
  - maintainability (single place for domain logic)
- cons:
  - domain experts
  - more communication
  - time & cost
  - abstractions
- makes sense for large projects


O'REILLY®
Learning Domain-Driven Design
Aligning Software Architecture and Business Strategy
Vlad Khononov
Foreword by Julie Lerman

References

- Vlad Khonovov: Learning Domain-Driven Design (https://www.oreilly.com/library/view/learning-domain-driven-design/9781098100124/) [O'Reilly, 2021]
- Eric Evans: Domain-Driven Design: Tacking software complexity (https://www.oreilly.com/library/view/domain-driven-design-tackling/0321125215/) [Addison-Wesley Professional, 2003]
- Eric Evans: Domain-Driven Design Reference: Definitions and Pattern Summaries (https://www.domainlanguage.com/wp-content/uploads/2016/05/DDD_Reference_2015-03.pdf) [Addison-Wesley Professional, 2015]
- Matthew Boyle: Domain-Driven Desing with Golang (https://www.packtpub.com/product/domain-driven-design-with-golang/9781804613450) [Packt, 2022]
- Lukáš Grolig: Software Architectures (https://is.muni.cz/predmet/fi/podzim2023/PV293) [FI MUNI, 2023]
- Awesome Go Educations (https://mehdihadeli.github.io/awesome-go-education/ddd/)
- ThreeDotsLab: Wild Workouts Go DDD example (https://github.com/ThreeDotsLabs/wild-workouts-go-ddd-example/tree/master)
- ThreeDotsLab: Go With The Domain (https://threedots.tech/go-with-the-domain/)

- Domain-Driven Design Crew (https://github.com/ddd-crew)

Q & A

Thank you

Stanislav Zeman <zeman@standa.dev> (mailto:Stanislav%20Zeman%20%3czeman@standa.dev%3e)

Backend Developer, ModernTV s.r.o.

https://standa.dev (https://standa.dev)

https://github.com/stanislav-zeman/talks (https://github.com/stanislav-zeman/talks)