bns it

DDD Concepts

Good, bad and effective design

- Software development is considered a cost center rather than a profit center.
- Developers are too wrapped up with technology.
- The database is given too much priority.

Good, bad and effective design

- Developers don’t give proper emphasis to naming objects.
- Often the business stakeholders spend too much time in isolated work.
- Project estimates are in high demand.

Good, bad and effective design

- There are broken, slow, and locking database queries that block users from performing time-sensitive business operations.
- Developers attempt to address all current and imagined future needs.
- There are strongly coupled services, direct calls to another.

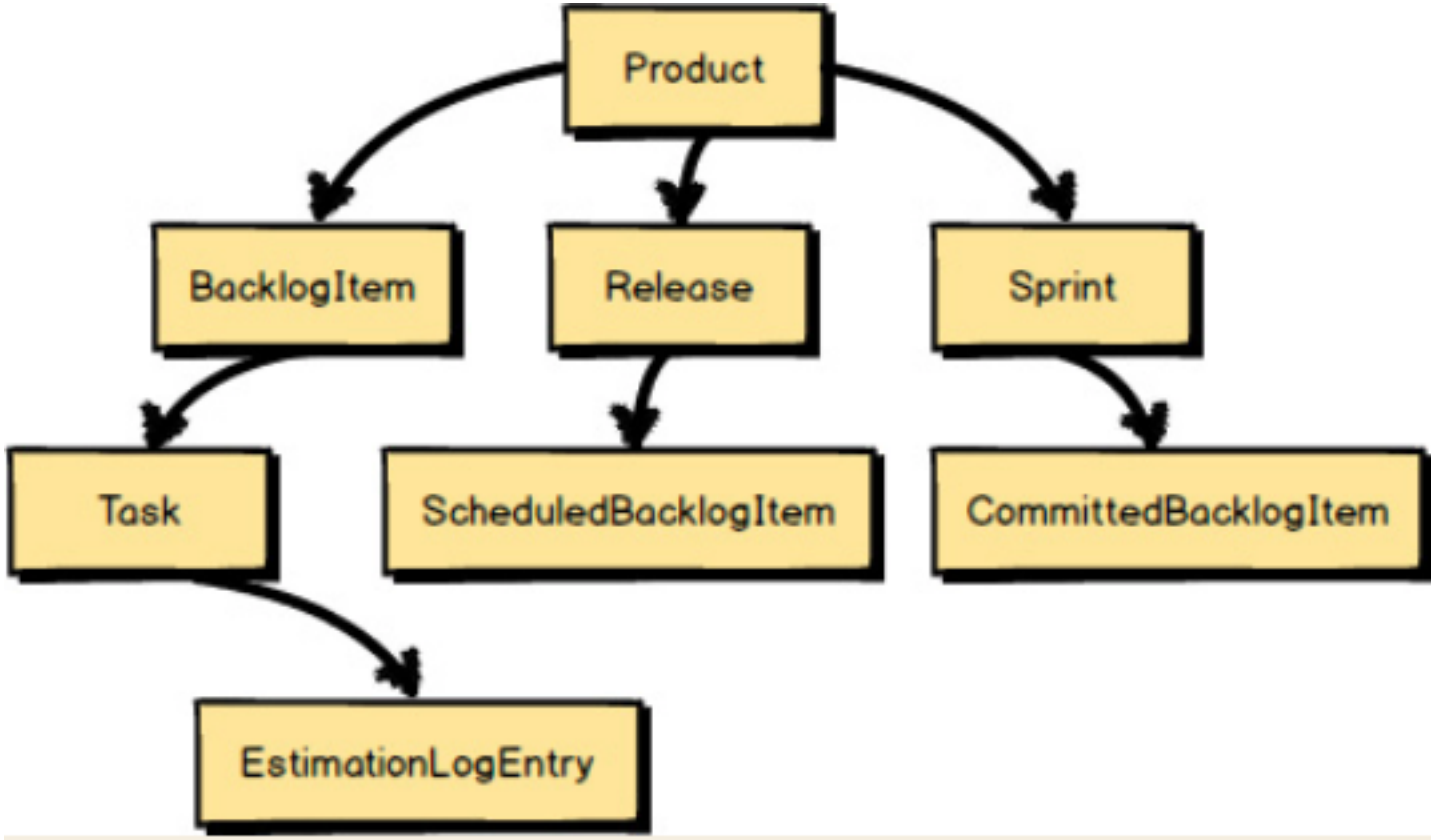
bns it

Scrum Management Tool - Example

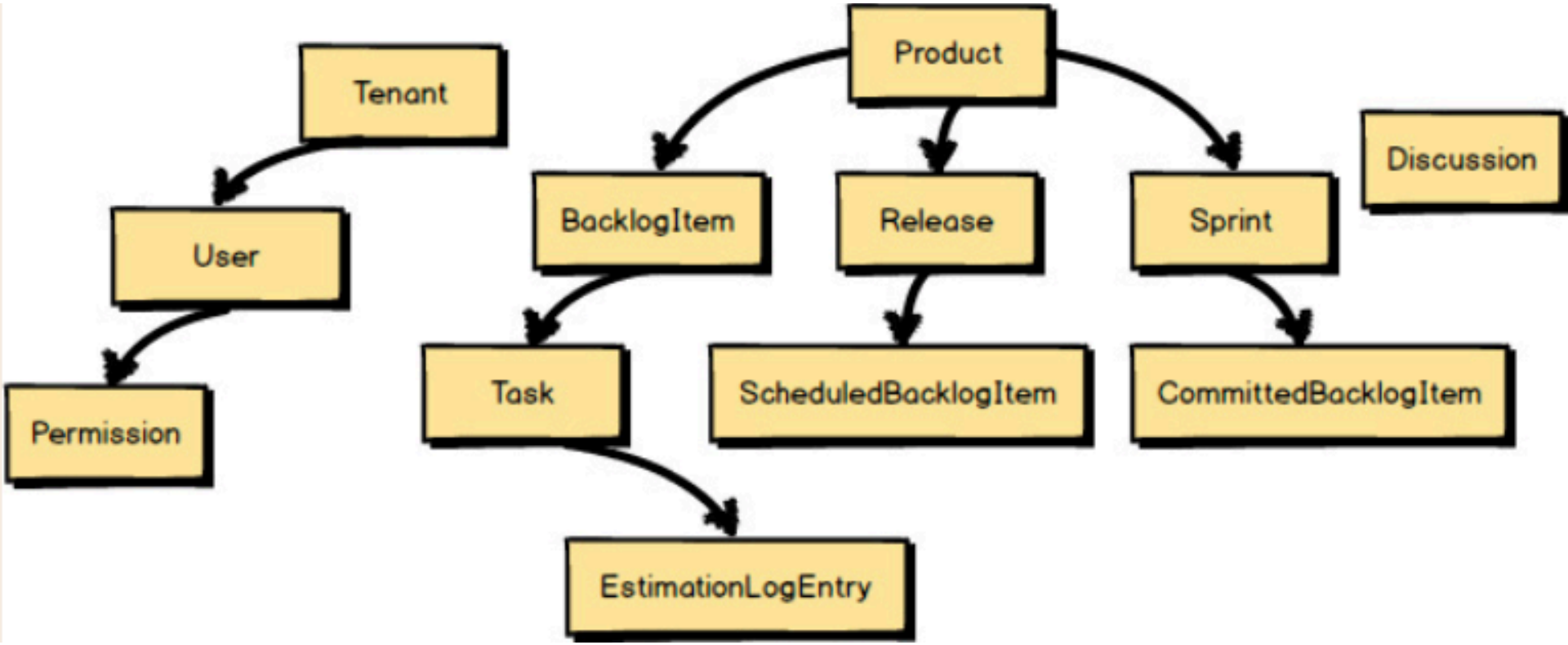
Different meanings

- You can plan product development – features, releases, sprints
- Available to many users and customers (SaaS)
- Different Support Plans and Incident Management
- Disussions through forums
- Development event in calendar
- Team management and resource planning

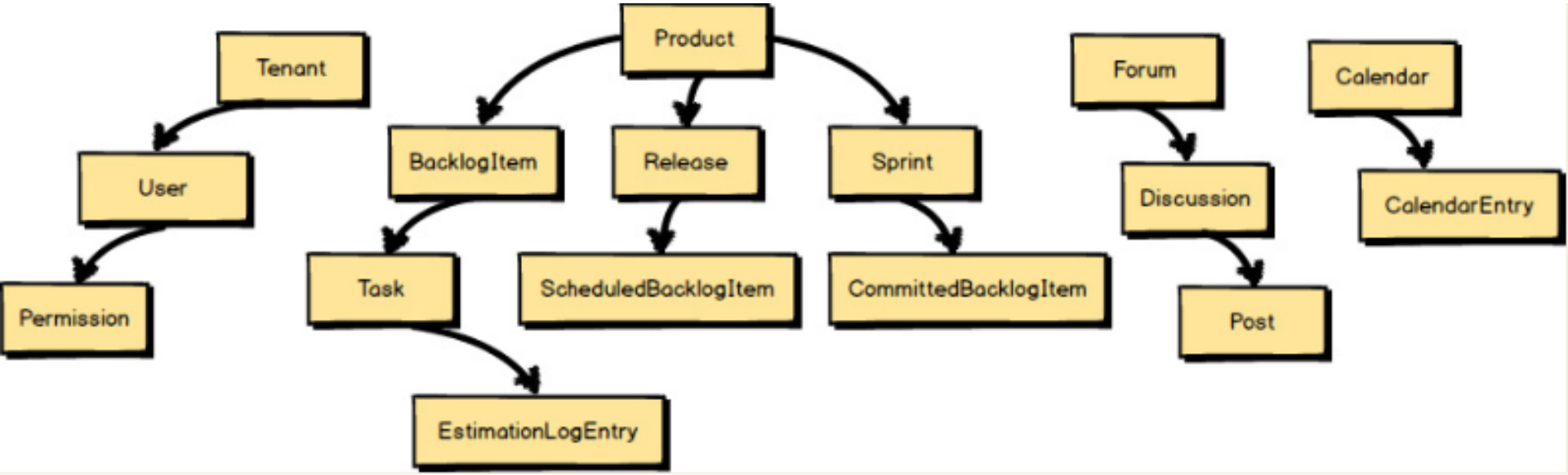
Releases, Sprints, Tasks



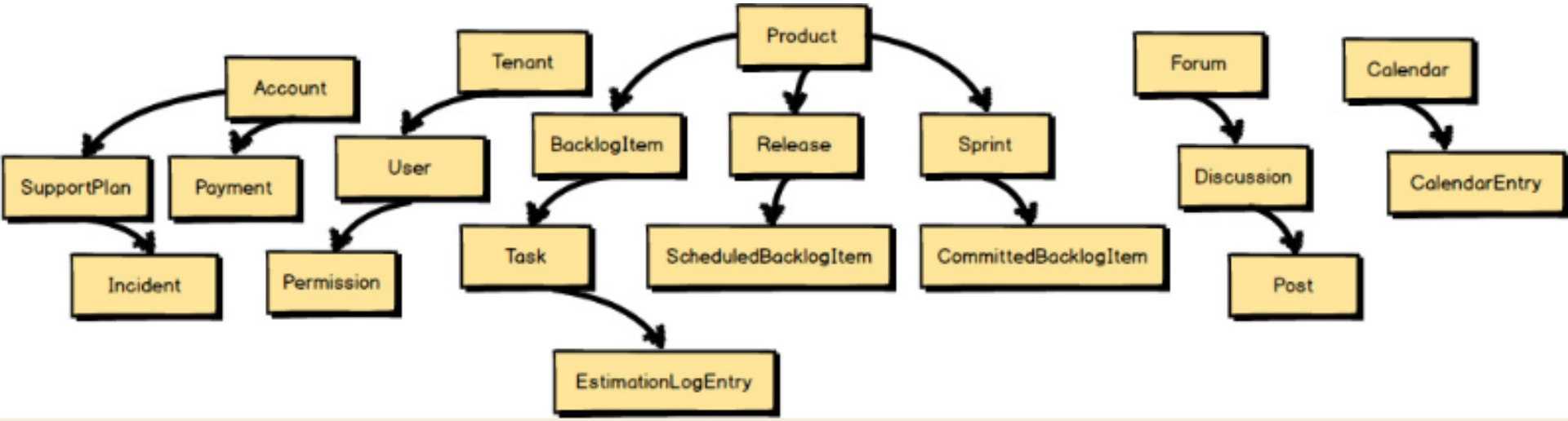
Many users


Tenant
Product
Discussion
User
BacklogItem
Release
Sprint
Permission
Task
ScheduledBacklogItem
CommittedBacklogItem
EstimationLogEntry

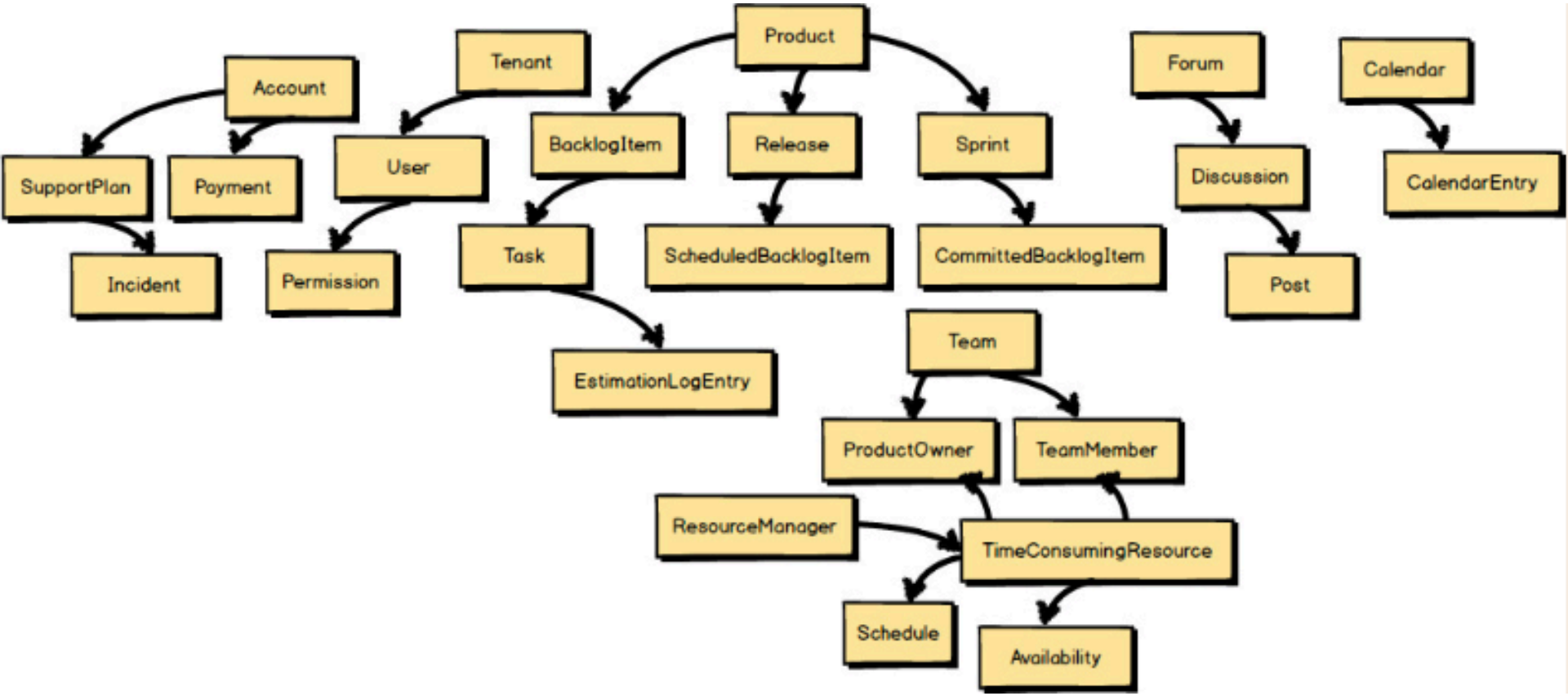
Forum and calendar



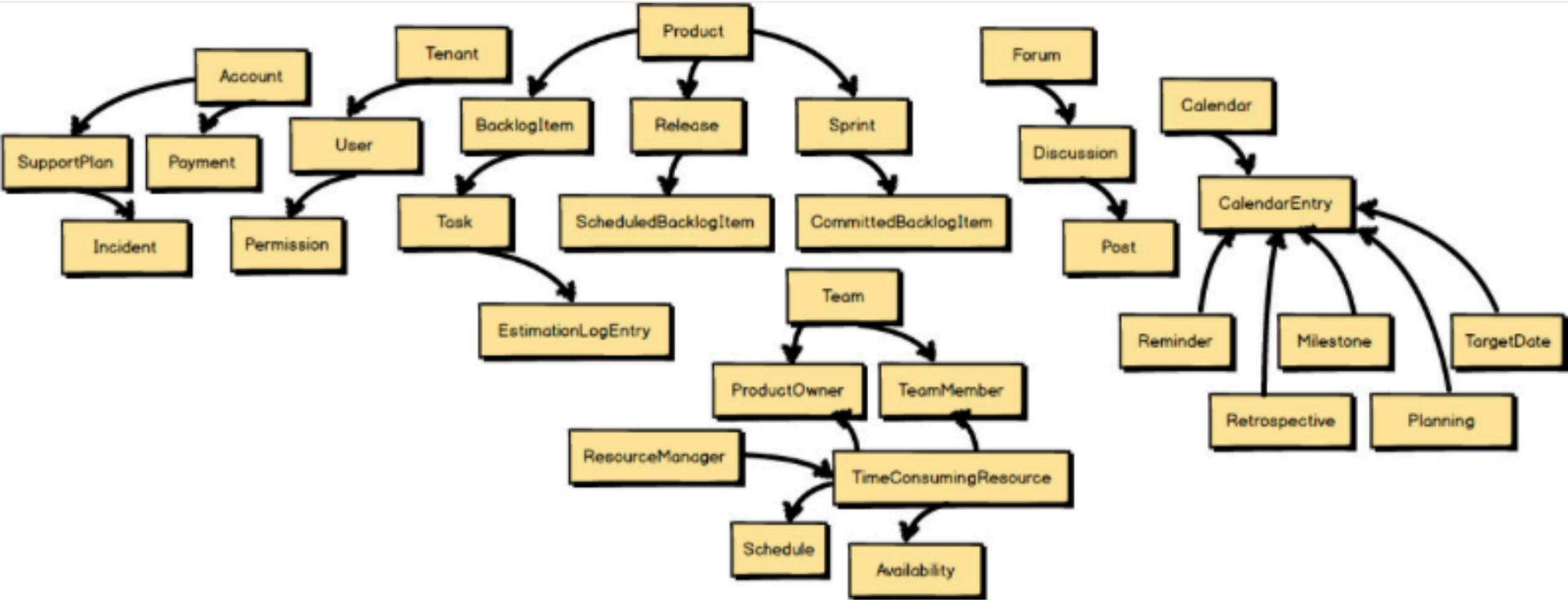
Customer accounts



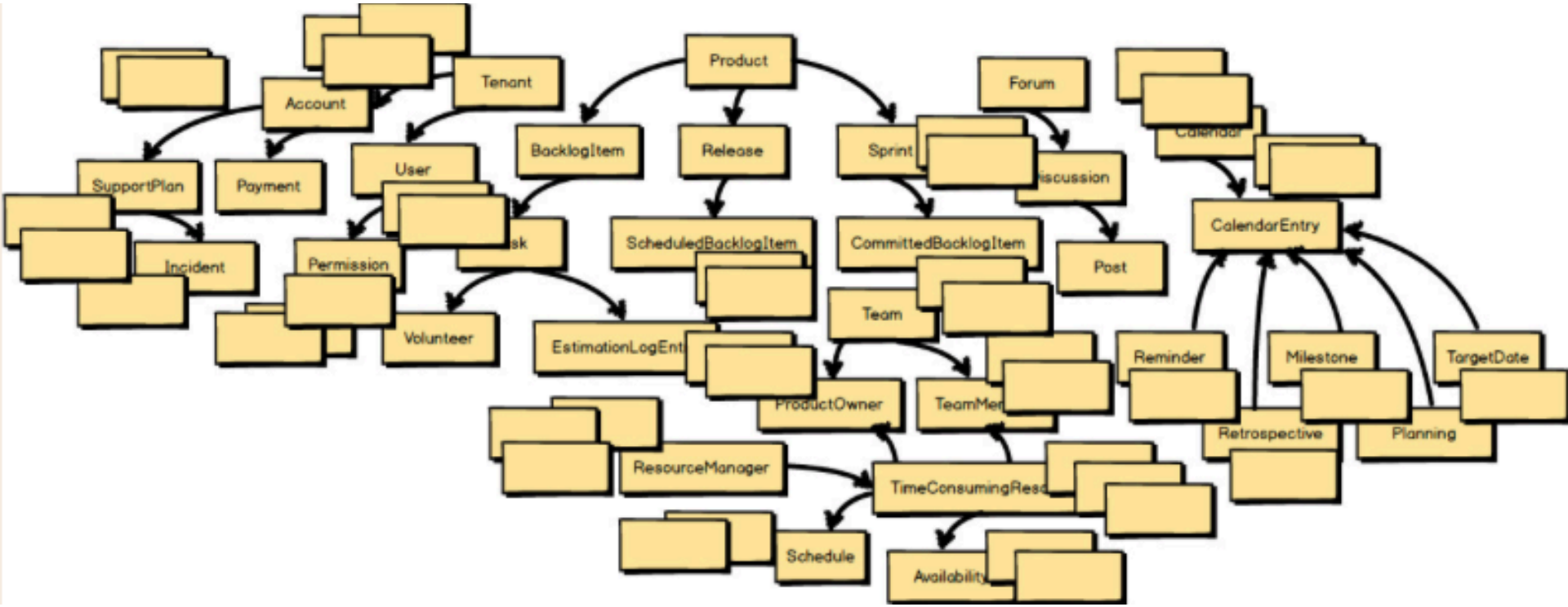
Teams


Product
Tenant
Account
Forum
Calendar
BacklogItem
Release
Sprint
User
Discussion
CalendarEntry
SupportPlan
Payment
Task
ScheduledBacklogItem
CommittedBacklogItem
Incident
Permission
Post
EstimationLogEntry
Team
ProductOwner
TeamMember
ResourceManager
TimeConsumingResource
Schedule
Availability

Reminders, Restrospectives… in calendar

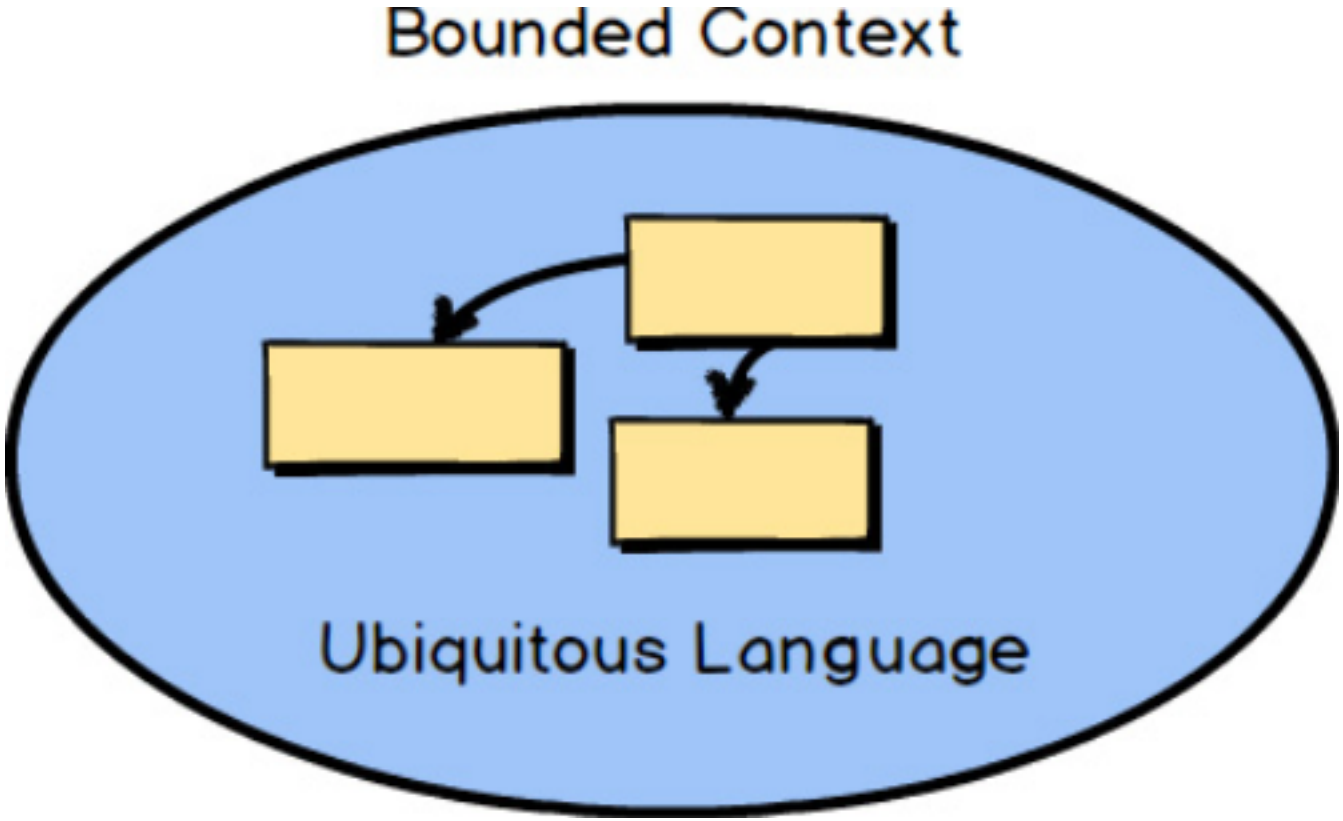


Big Ball of Mud

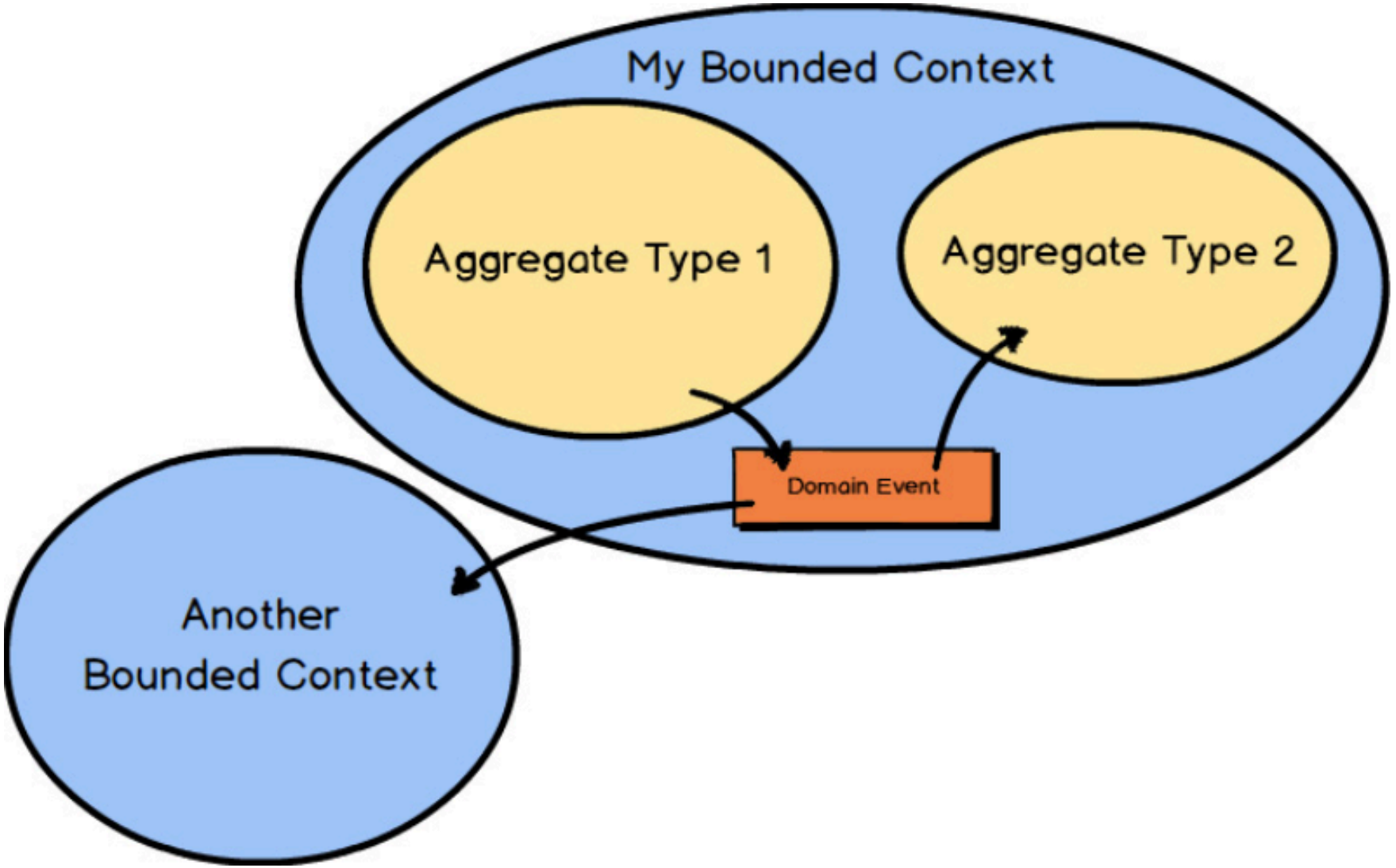


Bounded context and ubiquitous langauge

Strategic design



Strategic design - Context Maps



Tactical design – knowledge crunching

- Learning process is a matter of discovery through group conversation and experimentation.

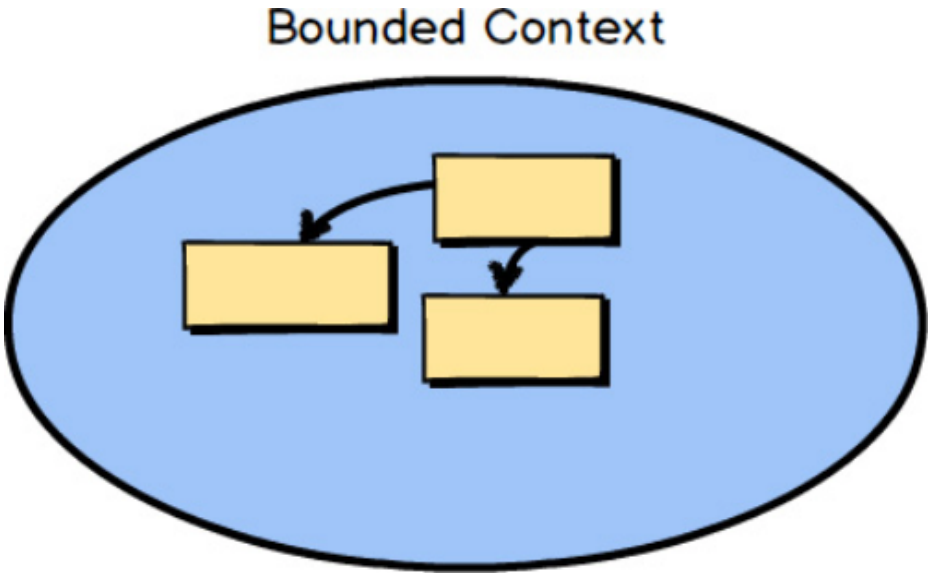
Tactical design – Building blocks

- # Aggregates
- # Entities
- # Factories
- # Policies
- # Value Objects
- # Services
- # Specifications

Bounded Contexts and Ubiquitous Language

Ubiquitous language

- Language developed by the team
- Working in the Bounded Context
- Spoken by every member of the team
- Rigorous—strict, exact, stringent, and tight.



Core domain

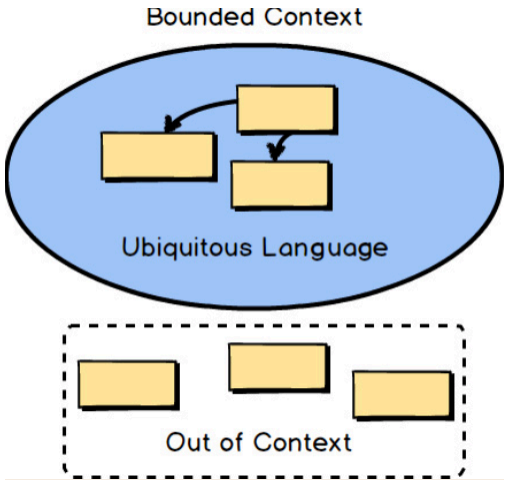
- Core Domain is developed to distinguish your organization competitively from all others

- Commit your best resources to a Core Domain.

Bounded context – practical hints

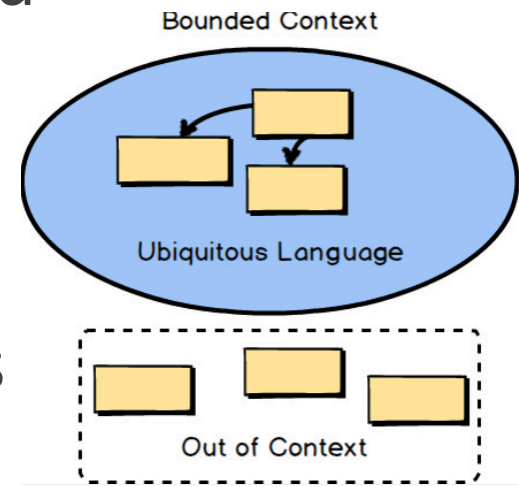
- There should be one team assigned to work on one Bounded Context.

- There should be a separate source code repository for each Bounded Context.

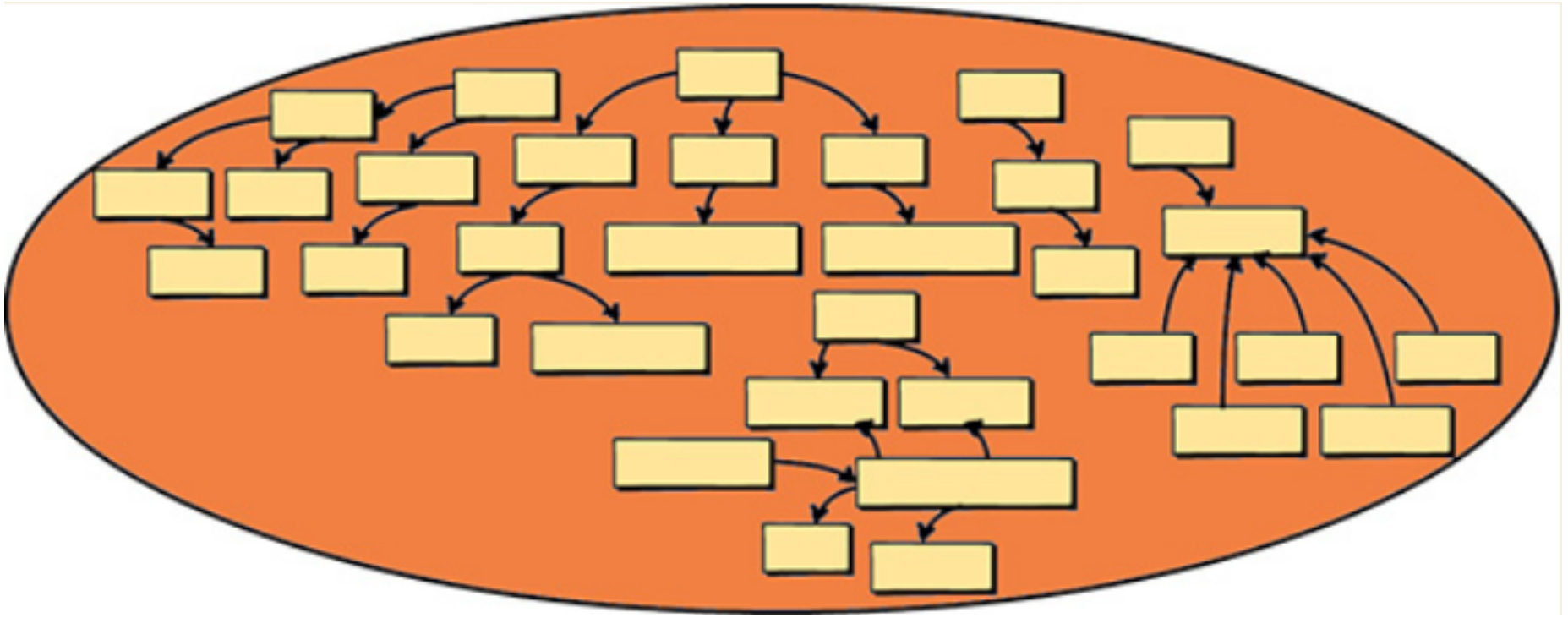


Bounded context – practical hints

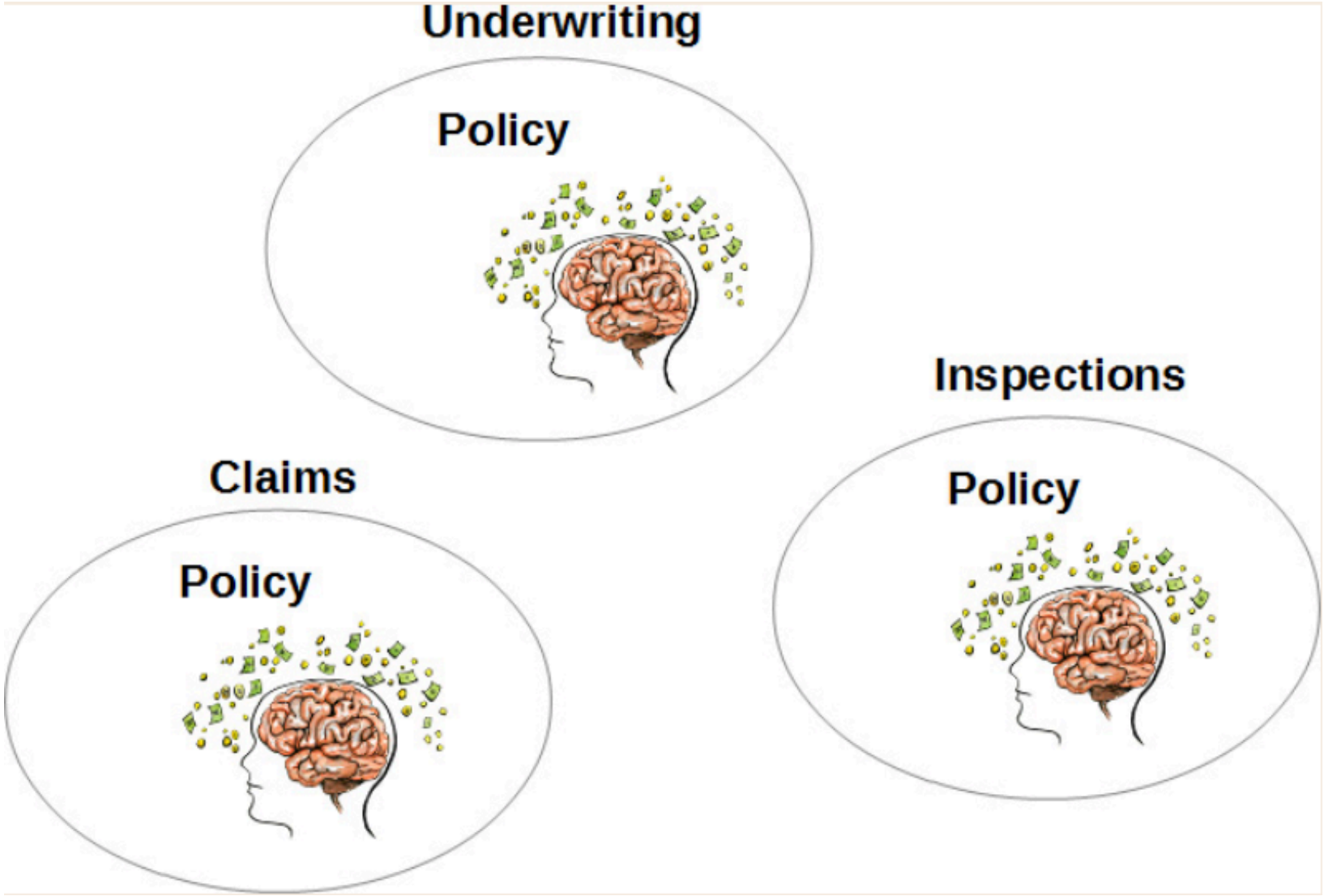
- It is possible that one team could work on multiple Bounded Contexts, but multiple teams should not work on a single Bounded Context.
- Cleanly separate the source code and database schema for each Bounded Context.
- Keep acceptance tests and unit tests together with the main source code.



Big Ball of Mud



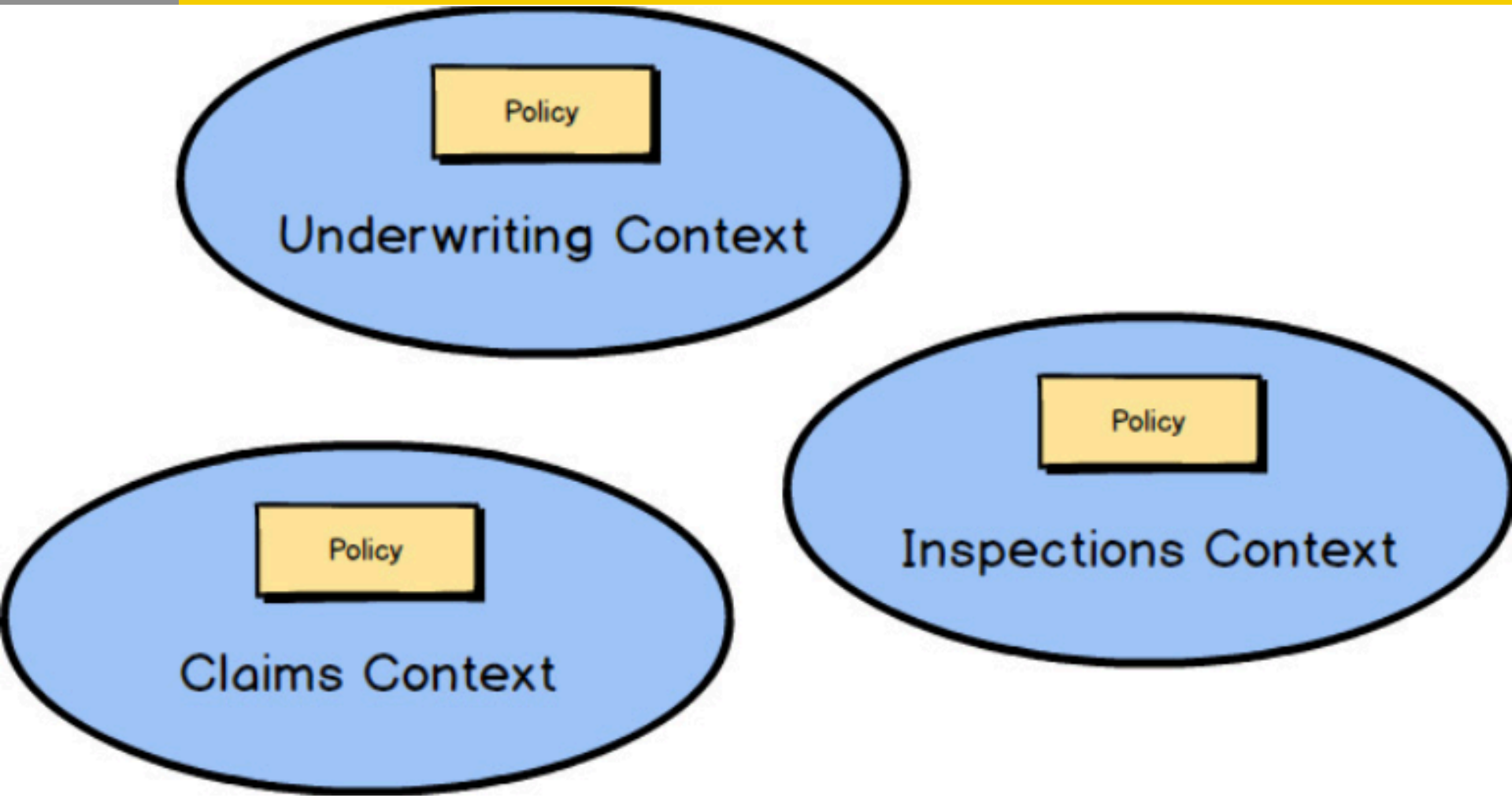
The same word – different meanings



Different meanings

- Policy in Underwriting
  - based on the evaluation of the risks of the insured entity
- Policy in Inspections
  - based on inspecting a property that is to be insured, information found during inspections - photos and notes
  - can be referenced by underwriting to negotiate the final cost
- Policy in Claims
  - tracks the request for payment
  - focused on, for example, damages to the insured property and reviews performed by the claims personnel

Different meanings



Different meanings

Flight

- a single takeoff and landing, where the aircraft is flown from one airport to another
- flight that is defined in terms of aircraft maintenance
- flight that is defined in terms of passenger ticketing, either nonstop or one-stop

Different Bounded Contexts

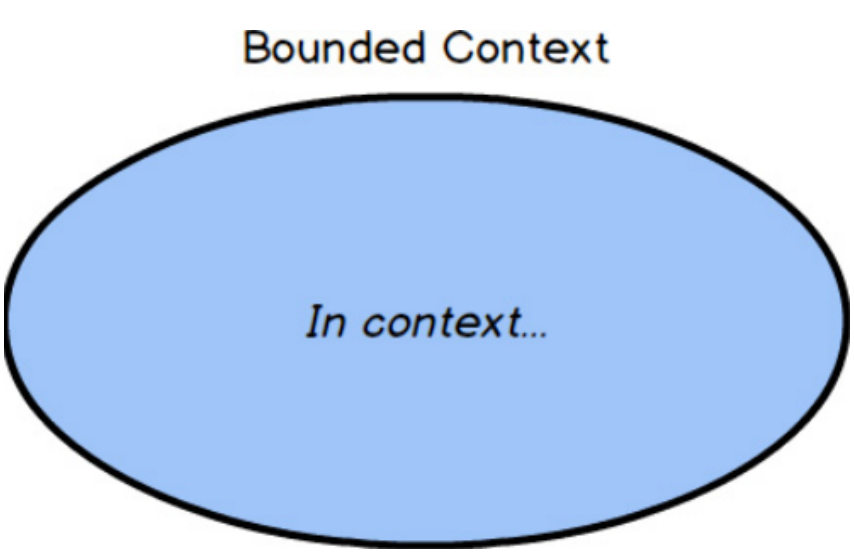
Different meanings

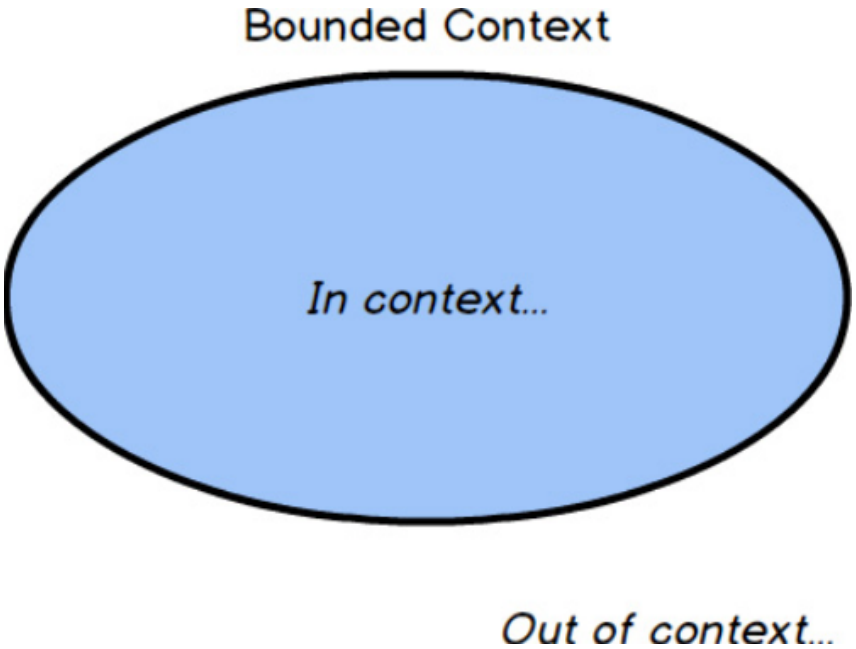
```
public class Lead
{
    public IEnumerable<Opportunity> Opportunities { get; }
    public Person Contact { get; }
}

public class Client
{
    public IEnumerable<Invoice> GetOutstandingInvoices();
    public Address BillingAddress { get; }
    public IEnumerable<Order> PurchaseHistory { get; }
}

public class Customer
{
    public IEnumerable<Ticket> Tickets { get; }
}
```

Where is the core?

What are the Bounded Contexts?

Bounded contexts definition

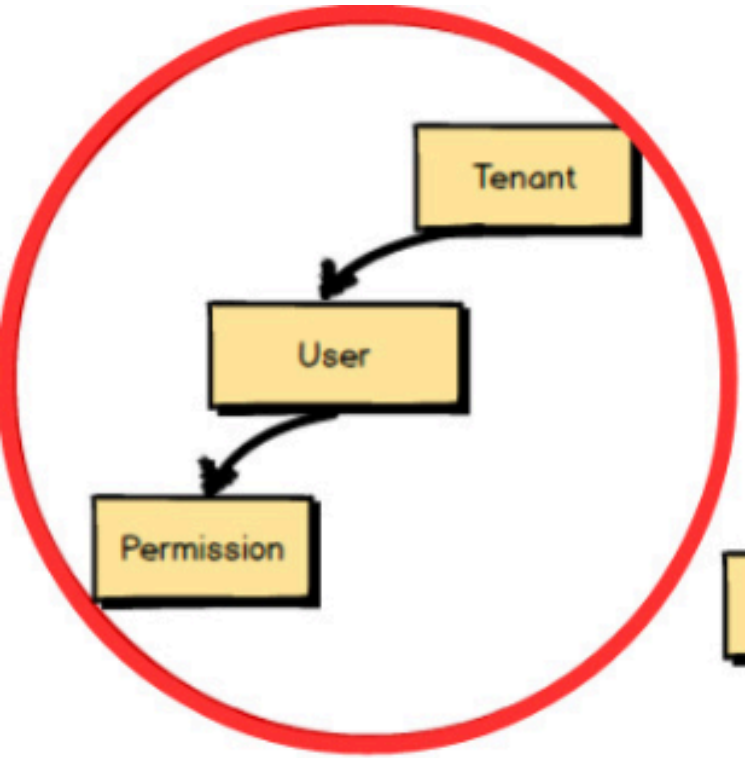


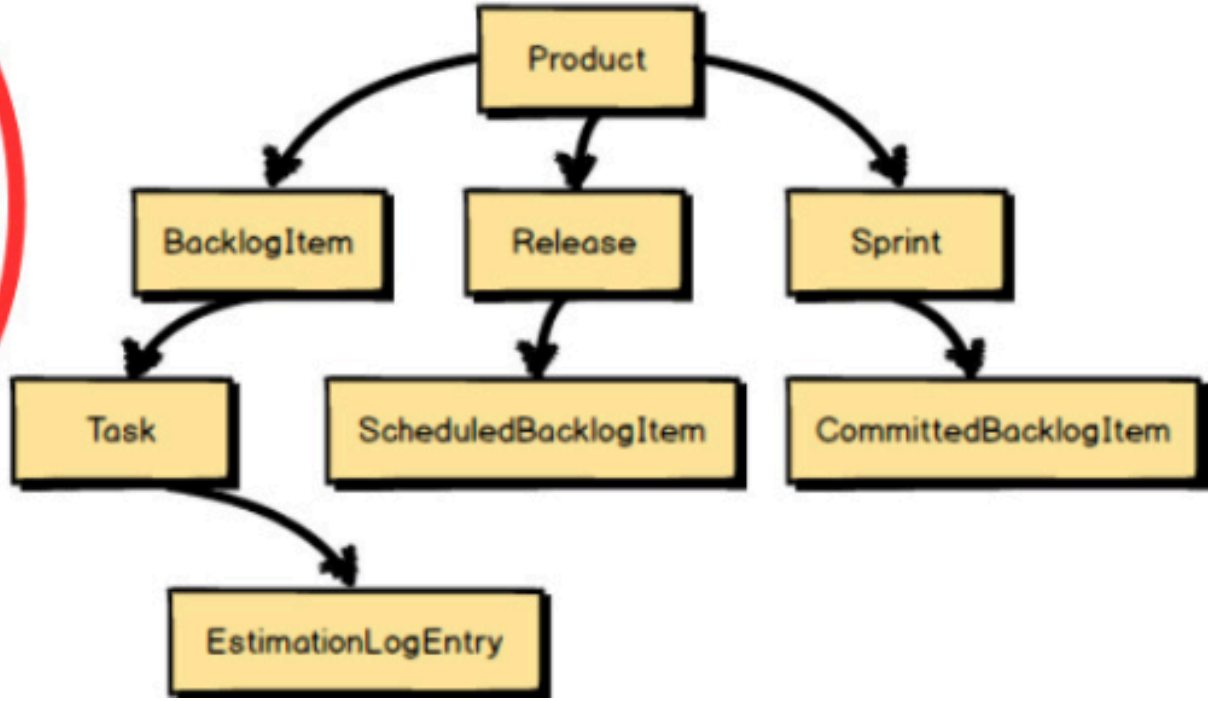


Product Owner or Domain Expert?

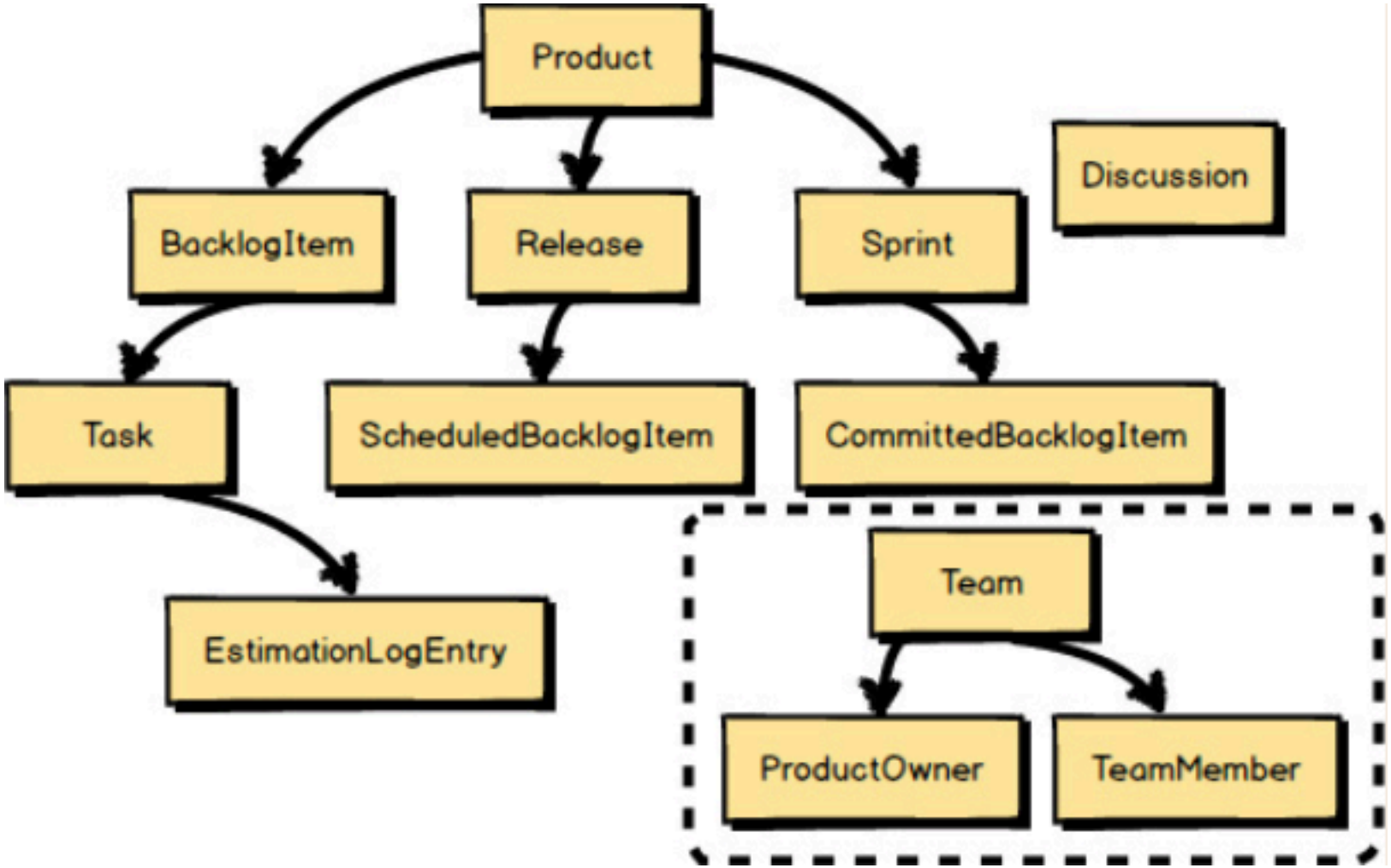
Focus on Business Complexity, Not Technical Complexity

Where is the core?

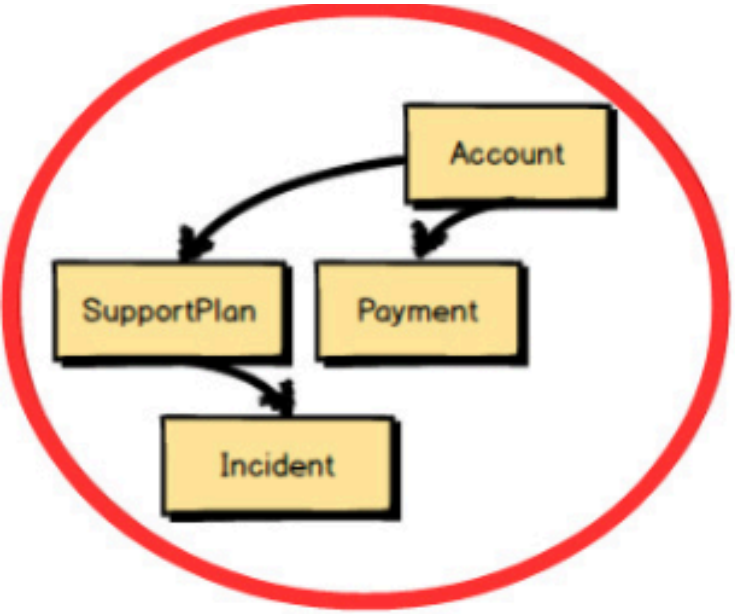


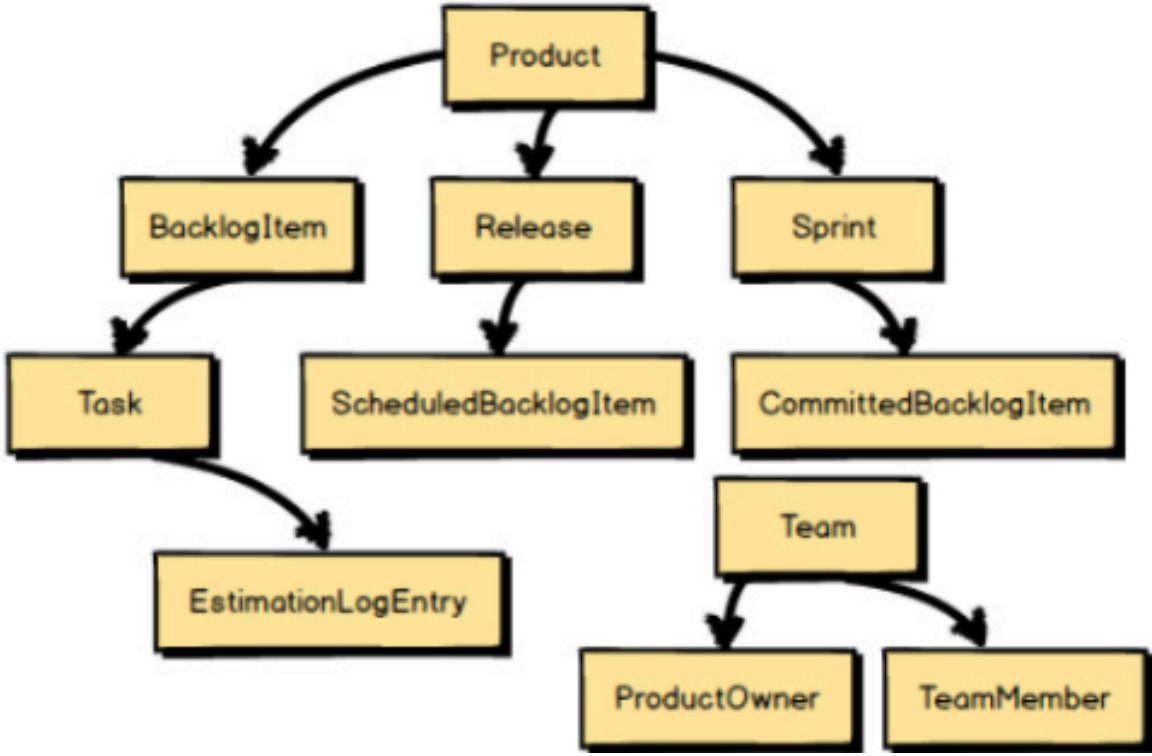


Where is the core?

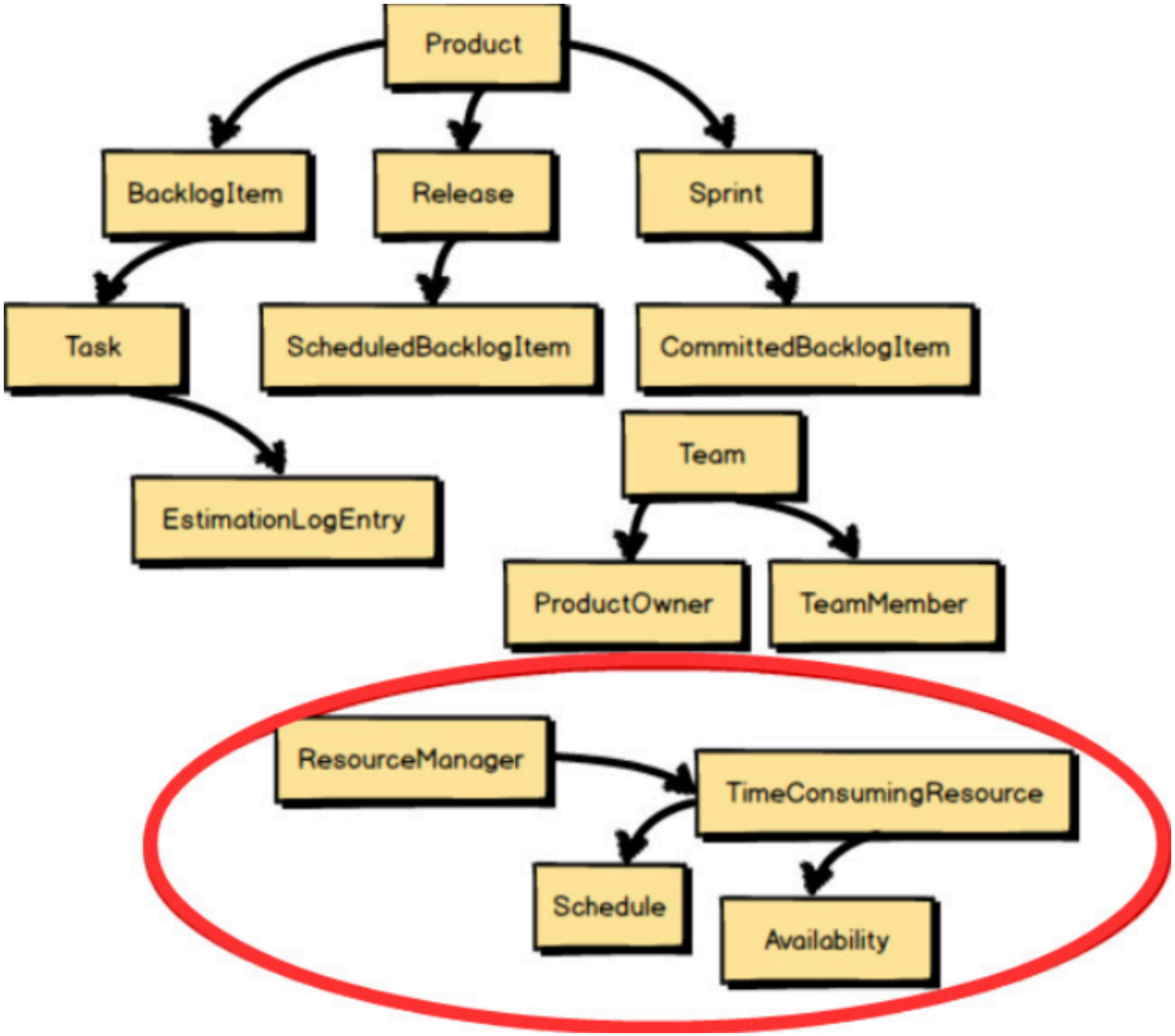


Where is the core?

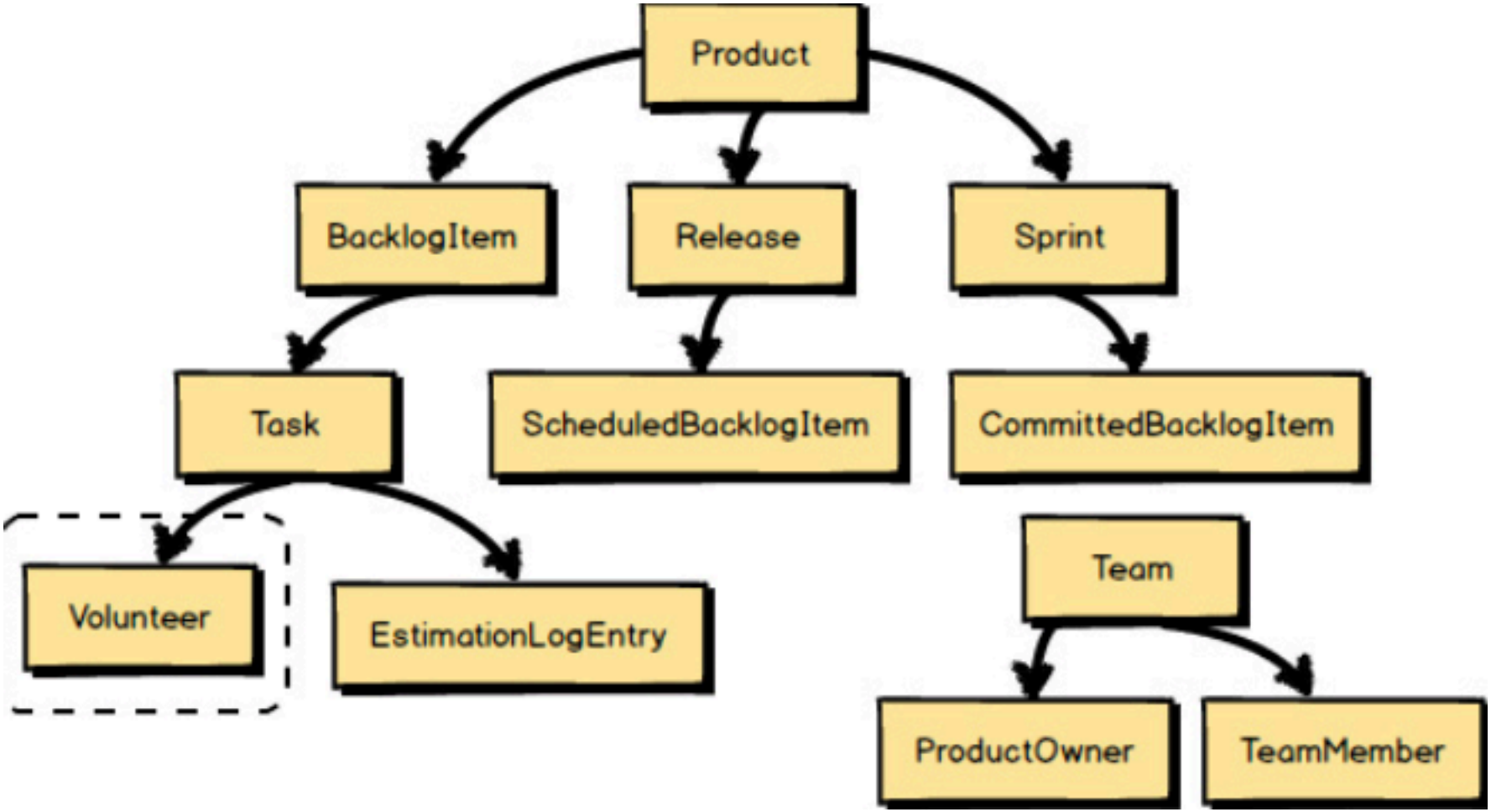




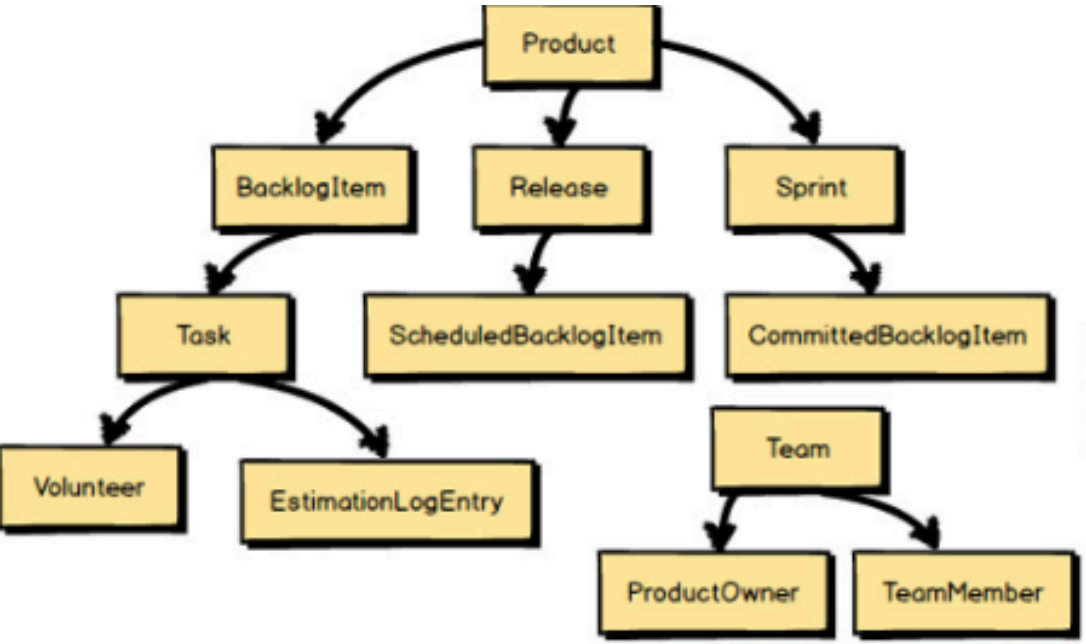
Where is the core?

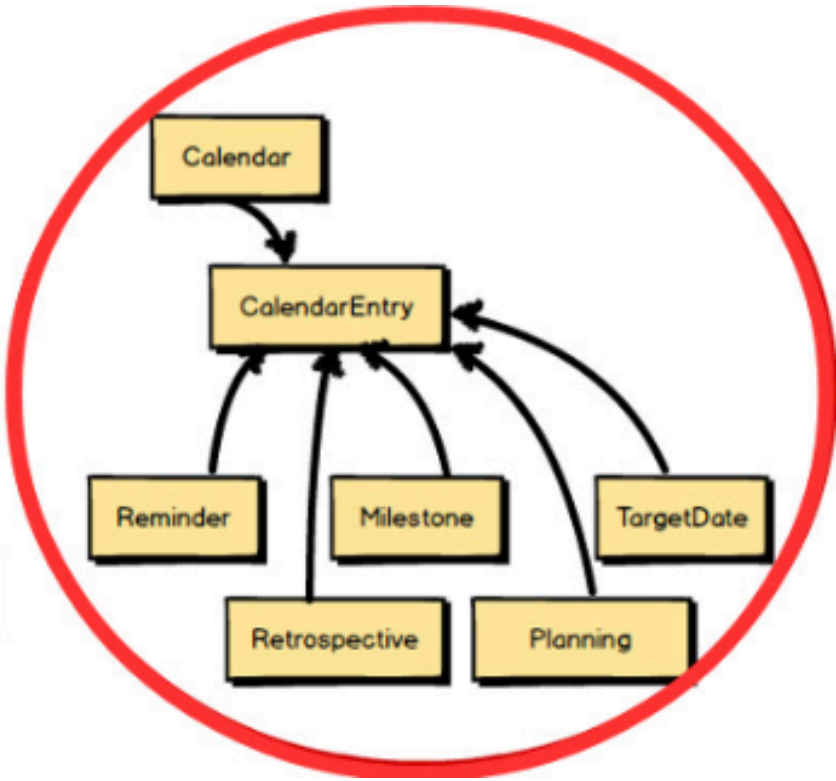


Where is the core?

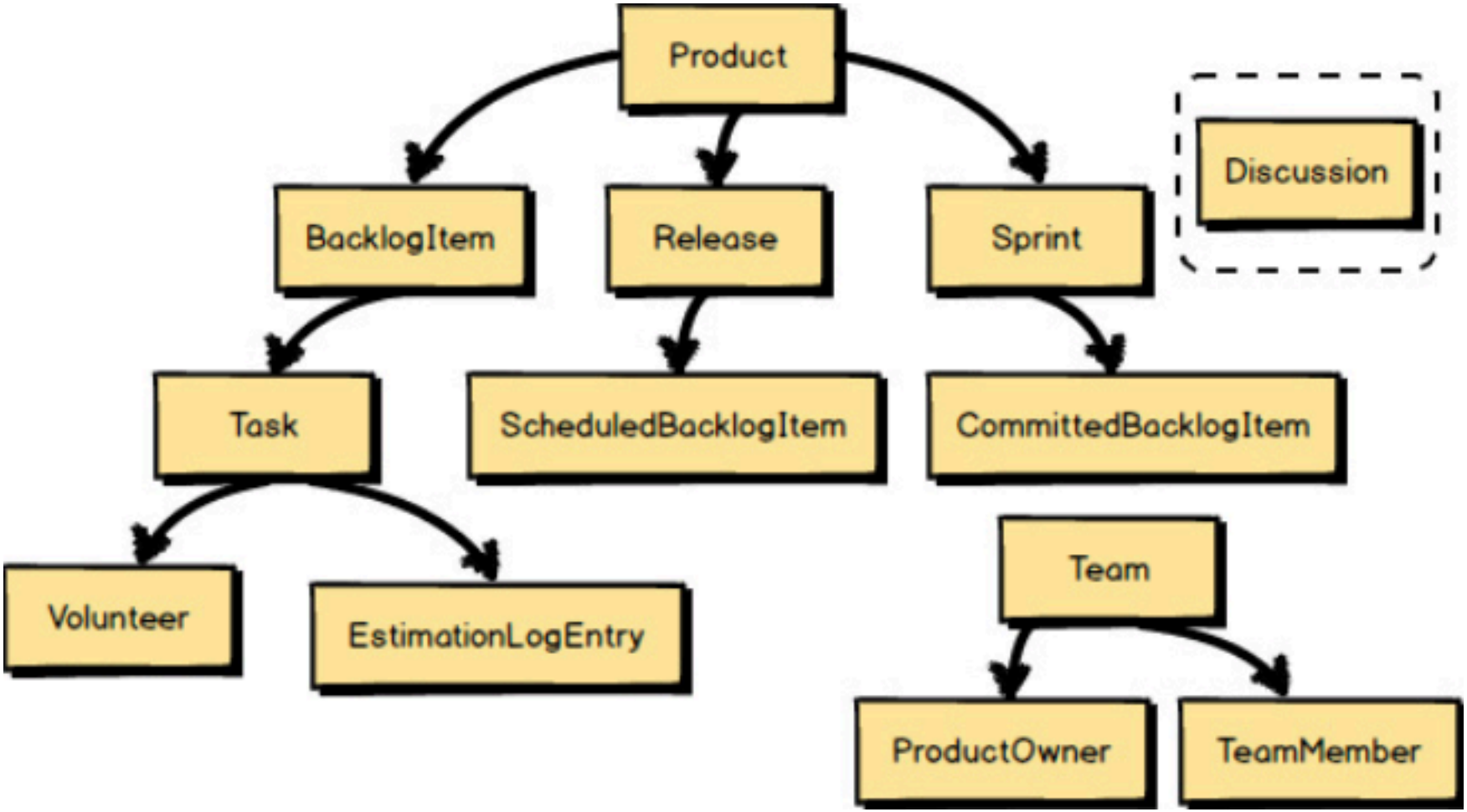


Where is the core?

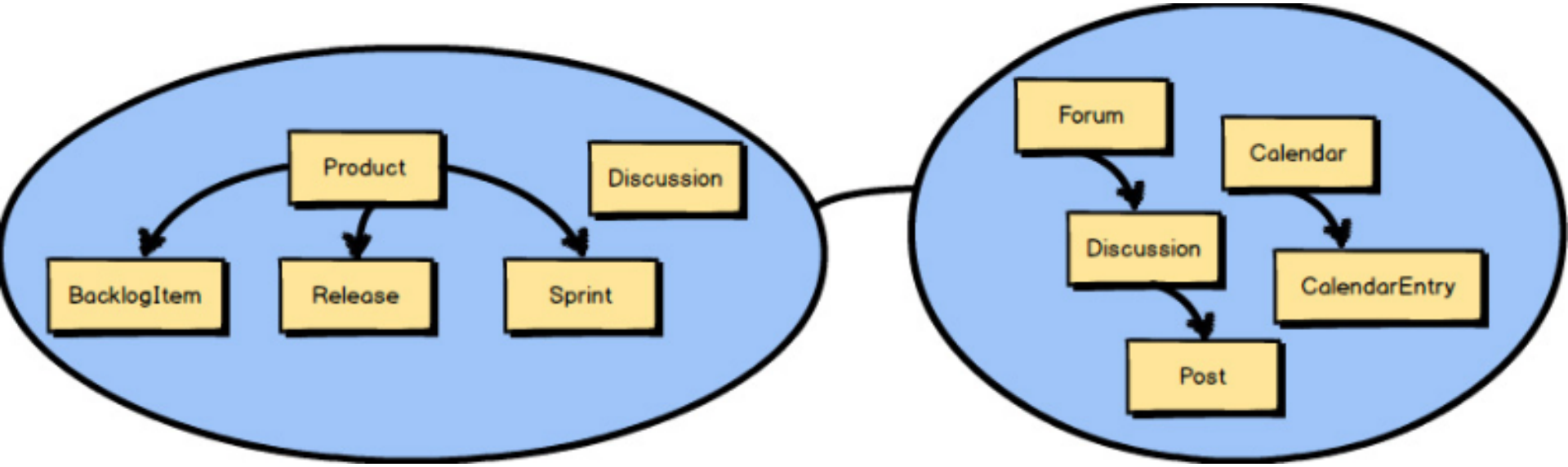




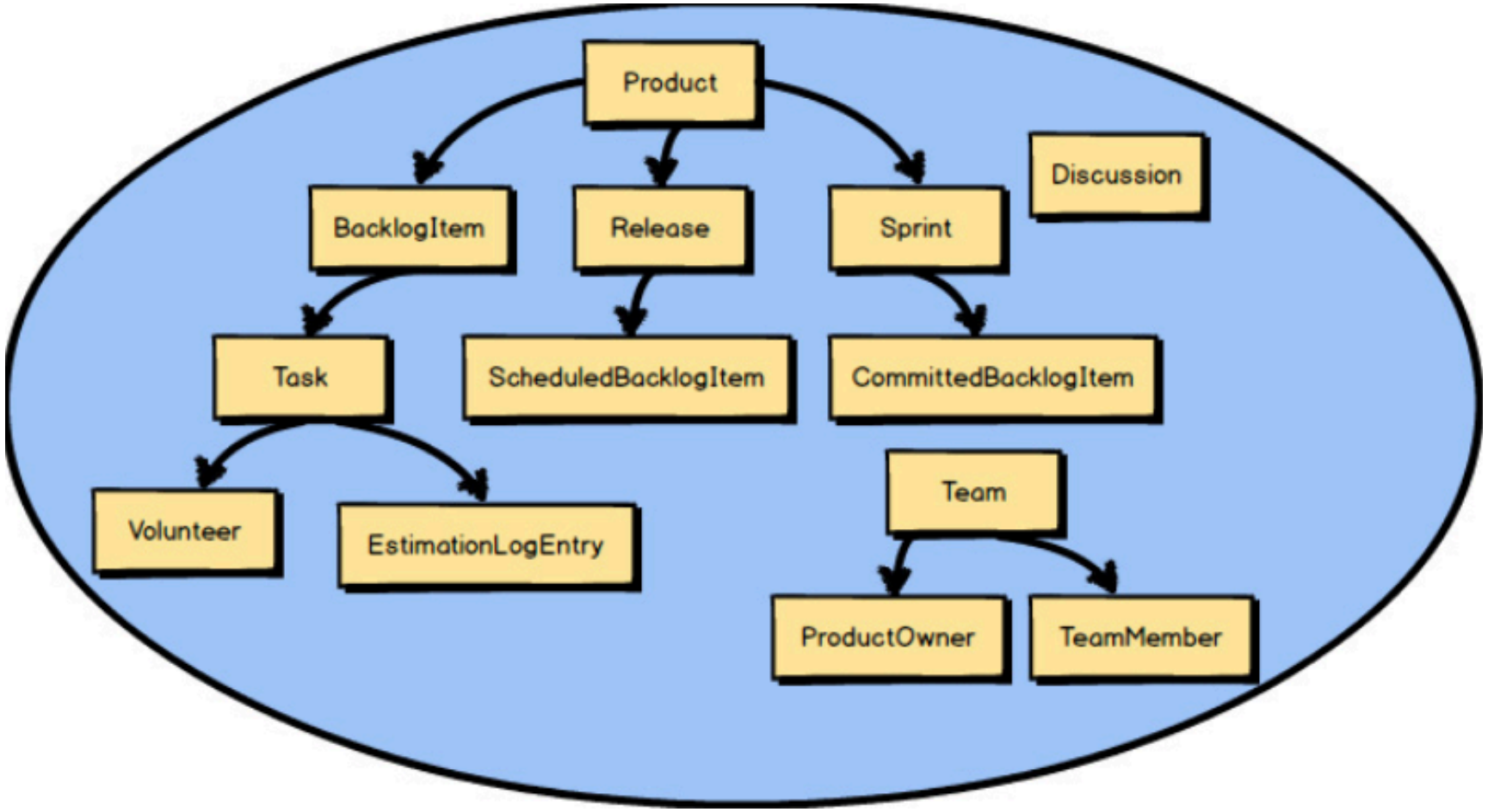
Where is the core?



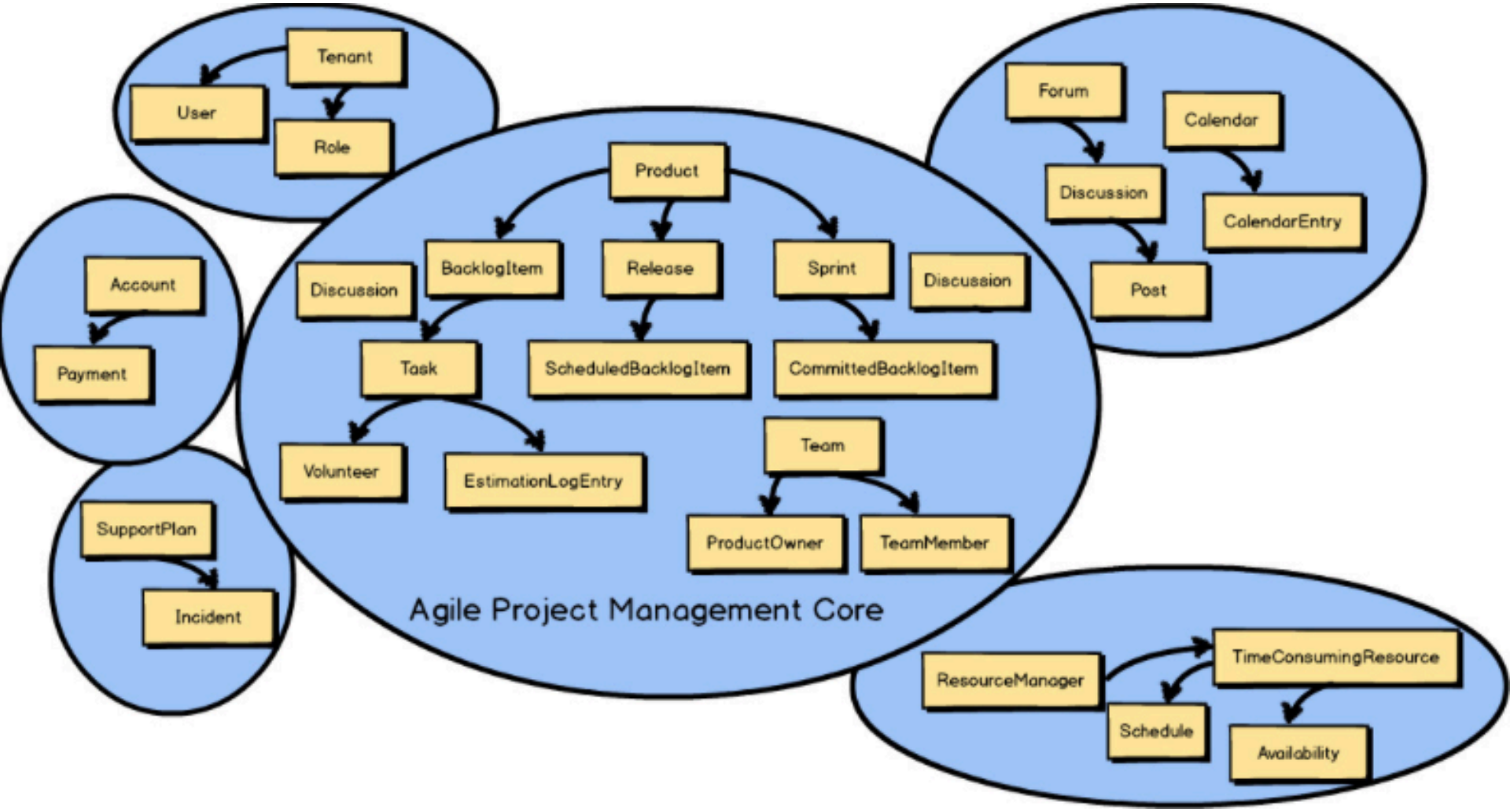
Where is the core?



Where is the core?



Where is the core?

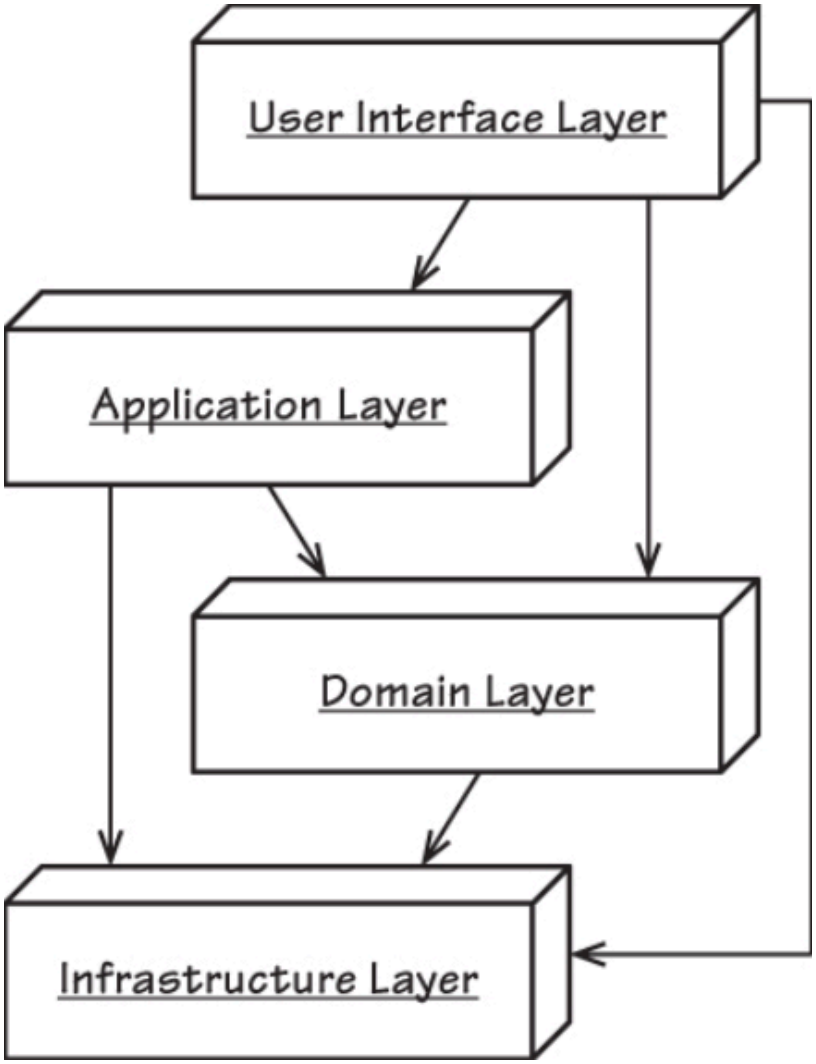


bns it

Architecture

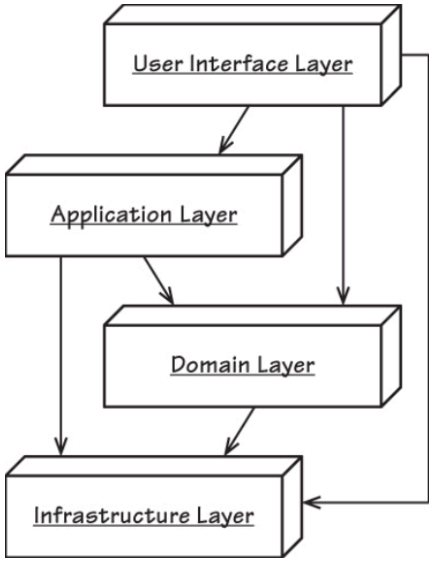
Layered architecture

- Strict Layers Architecture
- Relaxed Layers Architecture
- Communication backwards with Observer or Mediator pattern



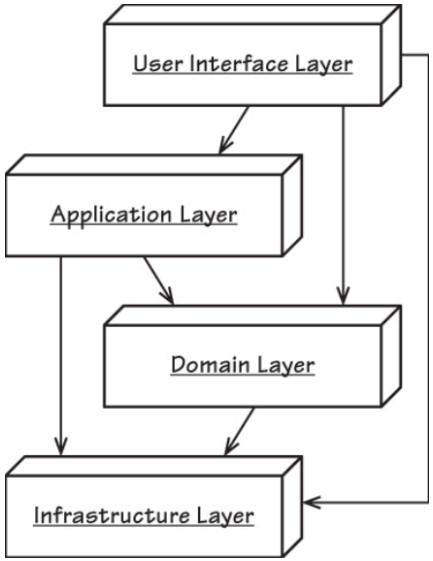
UI Layer

- Out of the scope od DDD
- Any valid model like MVC, Presentation Model, MVP
- Operations attributed to User Experience
- UI validations, no business logic



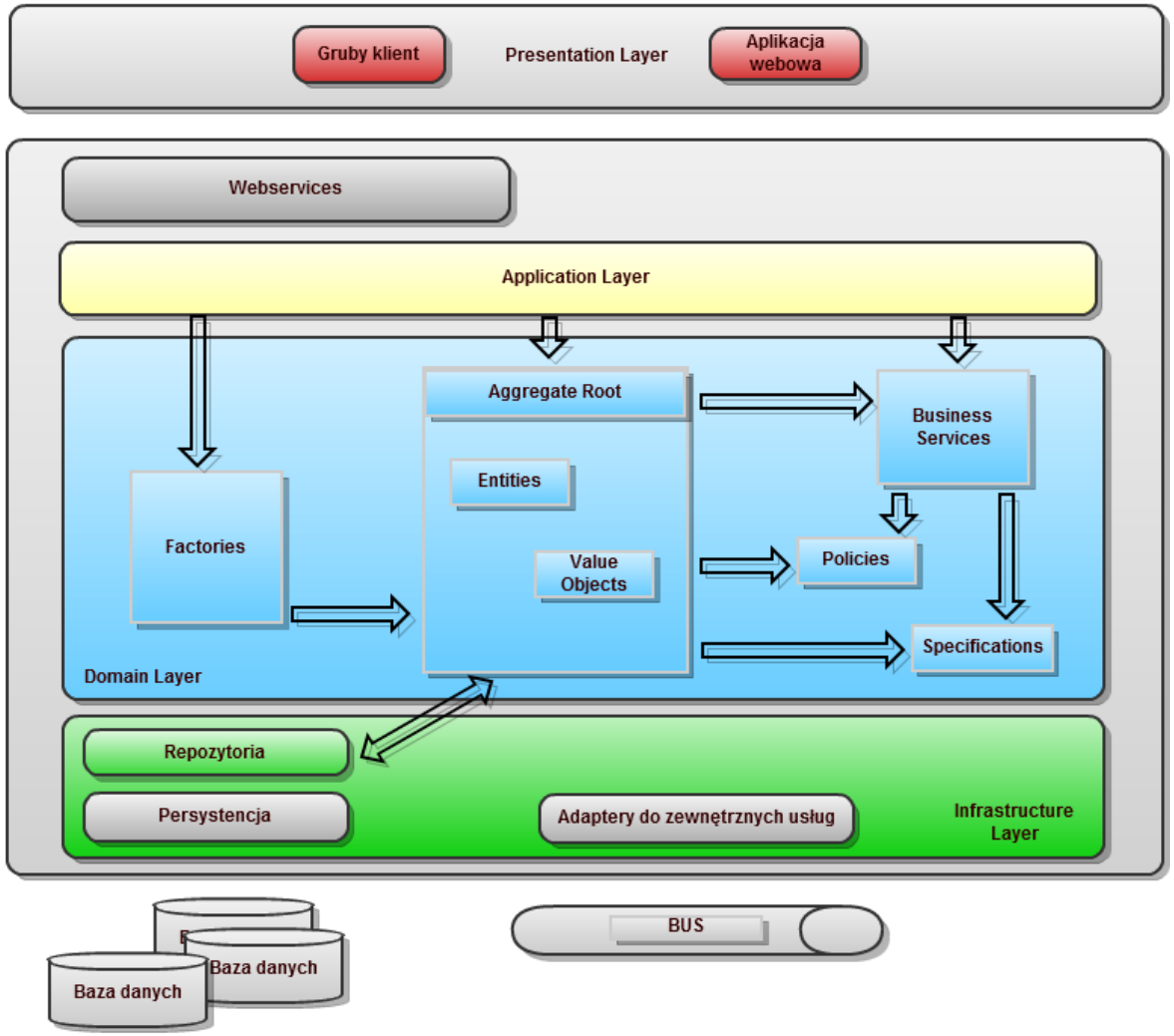
Application Layer

- Application Services (not Domain Services)
- Transactions and security
- Sending event notifications or exposing services to the world
- Use cases execution - orchestration
- No domain logic



Domain layer

- Events
- Entities
- Value Objects
- Factory
- Policy
- Specification
- Services



Application service

```
@Transactional

public void commitBacklogItemToSprint(
    String aTenantId, String aBacklogItemId, String aSprintId) {

    TenantId tenantId = new TenantId(aTenantId);

    BacklogItem backlogItem =
        backlogItemRepository.backlogItemOfId(
                tenantId, new BacklogItemId(aBacklogItemId));

    Sprint sprint = sprintRepository.sprintOfId(
                tenantId, new SprintId(aSprintId));

    backlogItem.commitTo(sprint);

}
```

Entity

```
public class Product extends Entity {

    private Set<ProductBacklogItem> backlogItems;
    private String description;
    private ProductDiscussion discussion;
    private String discussionInitiationId;
    private String name;
    private ProductId productId;
    private ProductOwnerId productOwnerId;
    private TenantId tenantId;
```

```
    public void initiateDiscussion(DiscussionDescriptor aDescriptor) {
```

```
    public BacklogItem planBacklogItem(
            BacklogItemId aNewBacklogItemId,
            String aSummary,
            String aCategory,
            BacklogItemType aType,
            StoryPoints aStoryPoints) {
```

Entity

```
public class Sprint extends Entity {

    private Set<CommittedBacklogItem> backlogItems;
    private Date begins;
    private Date ends;
    private String goals;
    private String name;
    private ProductId productId;
    private String retrospective;
    private SprintId sprintId;
    private TenantId tenantId;

    public void commit(BacklogItem aBacklogItem) {
    public void uncommit(BacklogItem aBacklogItem) {
    public void reorderFrom(BacklogItemId anId, int anOrderOfPriority)
    public void nowBeginsOn(Date aBegins) {
    public Date ends() {
    public String goals() {
```

Repository

```
public interface ProductRepository {

    public Collection<Product> allProductsOfTenant(TenantId aTenantId);

    public ProductId nextIdentity();

    public Product productOfDiscussionInitiationId(TenantId aTenantId,
                                                   String aDiscussionInitiationId);

    public Product productOfId(TenantId aTenantId, ProductId aProductId);

    public void remove(Product aProduct);

    public void removeAll(Collection<Product> aProductCollection);

    public void save(Product aProduct);

    public void saveAll(Collection<Product> aProductCollection);
}
```

Value Object

```
public class Colour
{
    public int Red { get; private set; }
    public int Green { get; private set; }
    public int Blue { get; private set; }

    public Colour(int red, int green, int blue)
    {
        this.Red = red;
        this.Green = green;
        this.Blue = blue;
    }

    public Colour MixInTo(Colour other)
    {
        return new Colour(
            Math.Avg(this.Red, other.Red),
            Math.Avg(this.Green, other.Green),
            Math.Avg(this.Blue, other.Blue));
    }
}
```

Value Object

```
public class BusinessPriorityRatings extends ValueObject {

    private int benefit;
    private int cost;
    private int penalty;
    private int risk;
```

```
public final class Repetition extends AssertionConcern {

    private Date ends;
    private RepeatType repeats;
```

```
public final class TimeSpan extends AssertionConcern {

    private Date begins;
    private Date ends;
```

Business Service

```
public interface TripService
{
    float GetDrivingDistanceBetween(Location a, Location b);
}
```

Specification

```
class GoldCustomerSpecification : ISpecification<Customer>
{
    public bool IsSatisfiedBy(Customer candidate)
    {
        return candidate.TotalPurchases > 1000.0m;
    }
}

if (new GoldCustomerSpecification().IsSatisfiedBy(employee))
    // apply special discount
```

Policy aka GoF Strategy

```
@DomainPolicy
public interface TaxPolicy {
/**
* calculates tax per product type based on net value
*/
    public Tax calculateTax(ProductType type, Money net);
```

bns it

Command Query Responsibility Segregation

Command/Query Separation

```
while((row = rowSet.next()) != null) {

   //...

}
```

```
while( rowSet.hasNext() ) {
   rowSet.moveNext();
   row = rowSet.get();
   //...
}
```

- Kind of side effect
- A method should do one of the following:
  - Reading model state (*query*)
  - Change model state (*command*)

CQRS Introduction

- From the architectural point of view we can differentiate two types of operations:
  - Command – modifying system state.
  - Query – reading fragments of system state.

CQRS Introduction

Consistency

- Commands – need some kind of transactional processing in context of aggregate/BC.

- Queries – may be eventually consistent.

CQRS Introduction

Storing data

- Commands – this side usually has models in 3rd normal form, for easy transactional consistence
- Queries – data usually denormalized, optimized for queries (like in datawarehouses)

CQRS Introduction

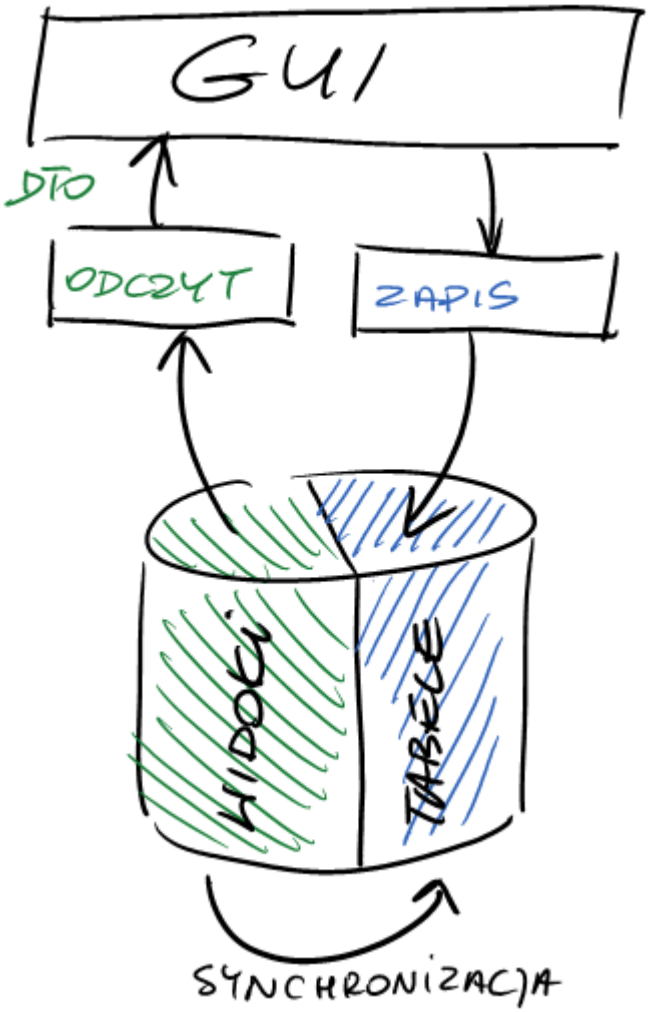
Scalability

- Commands – in most systems there are much less requests for writing than for reading, so we can favour consistency (transactional processing) over availability

- Queries – usually generate more processing, so we favour availability over consistency (eventual consistency).

Architecture point of viwe

- *Command* Part
  - eg. transactional database, ORM
- *Query* Part
  - Database views
  - Transfer objects
  - Denormalized models
- Synchronization
  - In one database you can use triggers
  - External scheduler
  - Synchronizing events



CQRS detailed