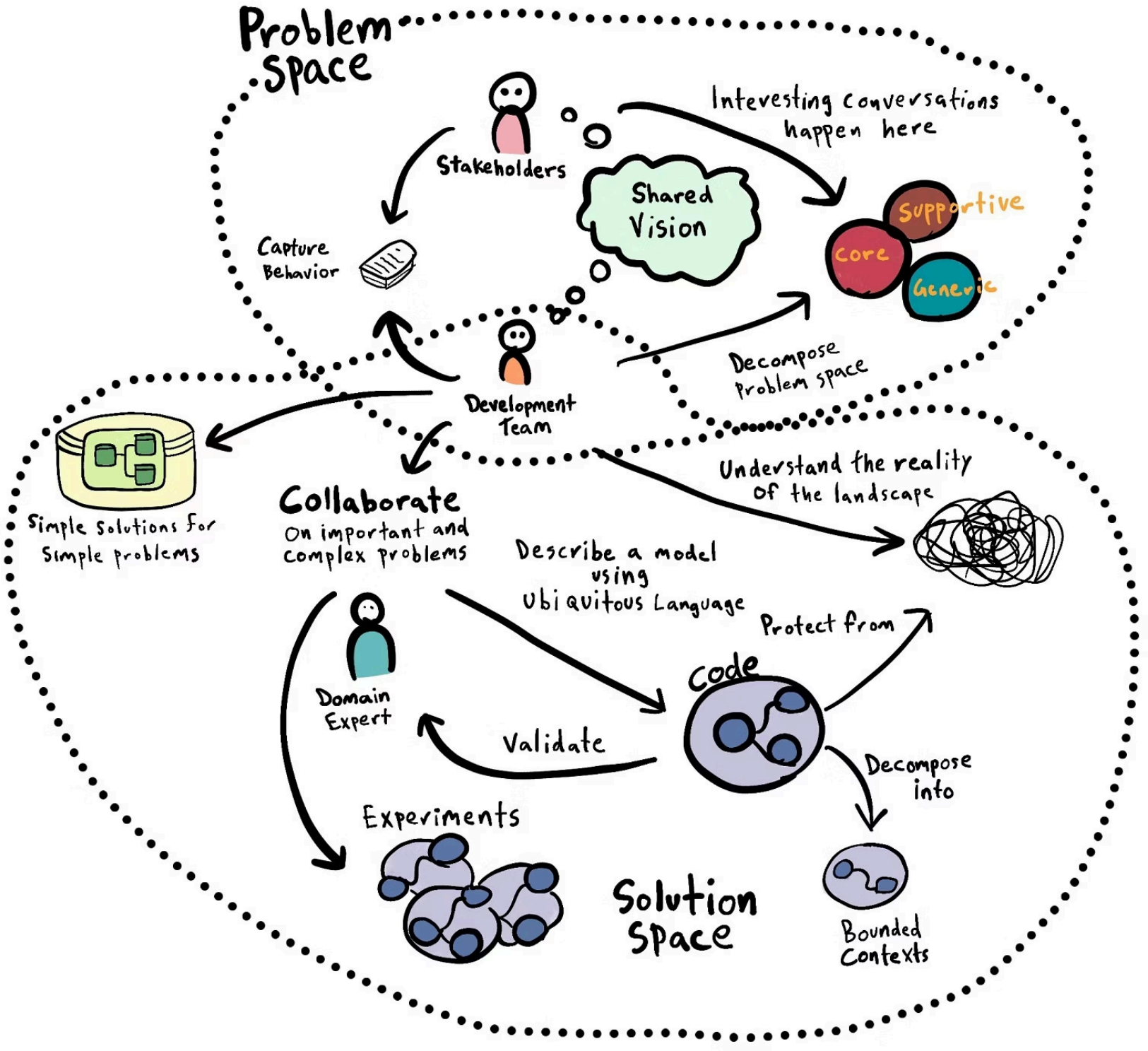
Domain Driver Design and Related Patterns



Bounded Contexts

A bounded context defines explicit boundaries where particular domain terms, rules, and definitions apply consistently.

Implementation Details

- Identify bounded contexts: Conduct collaborative domain modeling (event storming or domain storytelling) involving domain experts and developers to clarify contexts clearly.
- Draw context boundaries explicitly: Clearly document context boundaries with context maps, showing interactions and dependencies.
- Align to business capabilities: Each bounded context should match a well-defined, stable business capability or subdomain.

Common Pitfalls

- Overly large bounded contexts lead to monolithic complexity.
- Too granular contexts create communication overhead and complexity.

Context Maps

Context maps illustrate interactions and integration patterns between bounded contexts clearly.

Implementation Details

- Define explicit relationships: Clearly indicate relationships such as customer-supplier, shared kernel, conformist, anti-corruption layer, and open-host service.
- Use visual tools: Lucidchart, Miro, or PlantUML clearly depict context maps for clarity and ease of understanding.
- Continuous updates: Regularly revisit context maps during architecture reviews to ensure alignment with evolving business and technical landscapes.

Common Pitfalls

- Neglecting continuous maintenance, causing the context map to become outdated.
- Ambiguous relationships creating confusion among developers.

useful link: **https://github.com/ddd-crew/context-mapping**

Organizational Patterns

1. Anti-Corruption Layer (ACL)

A boundary service protecting internal domain models from external domain complexities.

Dedicated Translation Services

Implement a dedicated service layer that translates external models (e.g., legacy systems or third-party APIs) into internal domain models.

Translation Logic

Clearly define mapping and validation logic explicitly within the ACL.

Isolation Enforcement

ACL must enforce strict isolation between external and internal domains, clearly protecting internal domain integrity.

Common pitfalls:

- Weak or incomplete isolation resulting in external complexities leaking into the internal domain.
- Performance bottlenecks due to overly complex translation logic.

2. Shared Kernel

A clearly defined subset of the domain shared explicitly across multiple bounded contexts.

Extract Core Domain Logic

Identify shared logic or models beneficial to multiple contexts

Version Management

Manage shared kernels with versioning and clear change policies

Collaborative Governance

Involve stakeholders from all consuming contexts

Common pitfalls:

- Uncontrolled growth or poor governance causing tight coupling and versioning challenges.
- Overusing shared kernels, causing loss of autonomy in bounded contexts.

3. Open-Host Service

A clearly defined service exposing standardized APIs, ensuring stable, explicit interactions between bounded contexts.

Key Aspects:

API Design

Clearly define and document public-facing APIs using standards (OpenAPI) with explicit, stable contracts.

API Stability

Maintain explicit stability through careful versioning and backward compatibility policies.

Monitoring and Compliance

Explicitly monitor API usage and enforce compliance with established API standards.

Common pitfalls:

- Inadequate API governance leading to inconsistent API usage.
- Poorly documented or ambiguous APIs creating confusion and integration issues.

4. Published Language

An explicitly defined, standardized communication model used across multiple bounded contexts.

Formal Schemas

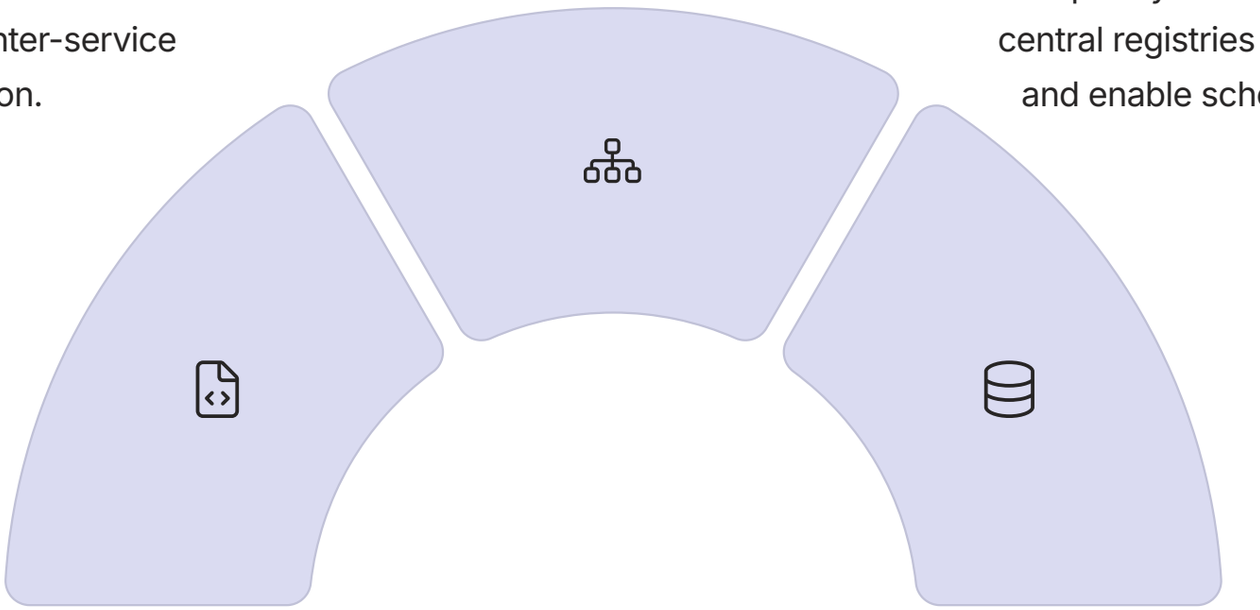
Use structured schema languages (JSON Schema, Avro, Protobuf) to clearly define shared data formats explicitly.

Define Canonical Models

Explicitly create and document shared data models used in inter-service communication.

Schema Registry

Explicitly manage schemas through central registries to enforce consistency and enable schema evolution clearly.



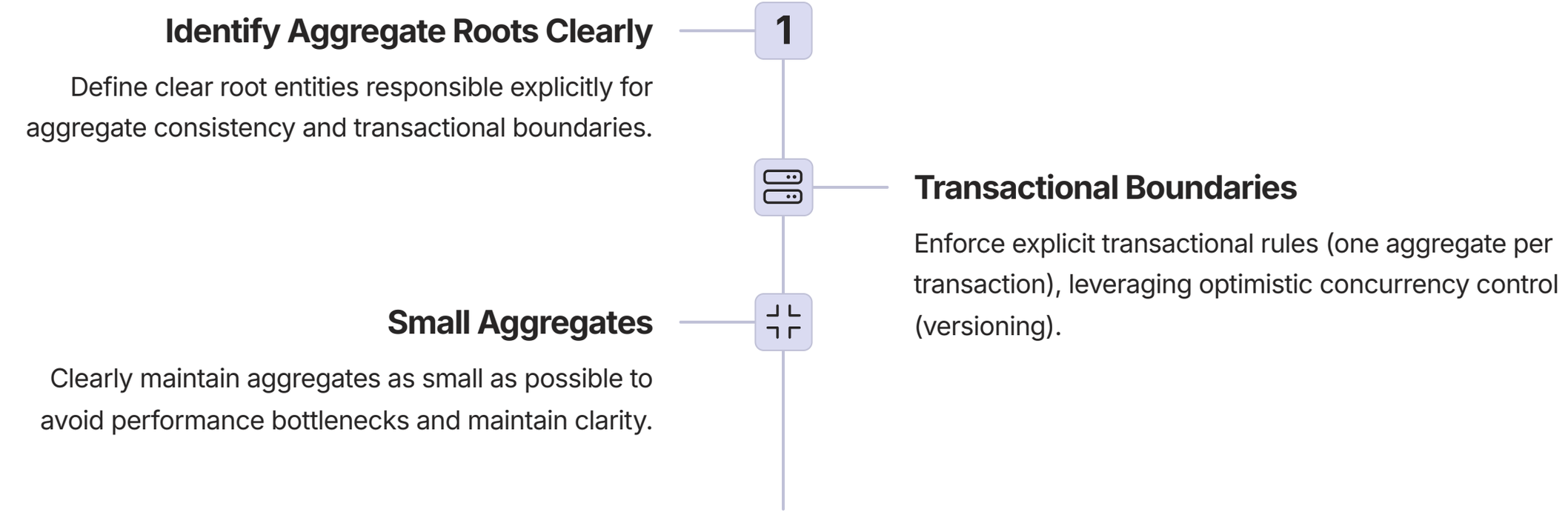
Common pitfalls:

- Lack of central management causing divergent implementations.
- Ambiguous or overly complex schemas making integration difficult.

Tactical DDD Patterns

1. Aggregates

Explicitly designed clusters of entities and value objects that enforce transactional consistency boundaries.



Common pitfalls:

- Excessively large aggregates leading to performance degradation and transactional complexity.
- Misidentifying aggregate roots, causing inconsistency and complexity.

2. Entities and Value Objects

Explicitly differentiate domain objects by identity (Entities) and immutability (Value Objects).

Entities

- Assign explicit identifiers clearly defined within your domain (UUIDs, composite IDs).
- Ensure entity lifecycle management and identity stability explicitly.

Value Objects

- Ensure explicit immutability, clearly defined equality semantics (e.g., override equals and hashCode methods).
- Model complex domain concepts (Address, Price) explicitly as value objects.

Common pitfalls:

- Treating Value Objects as mutable or identifiable, breaking immutability guarantees.
- Entities lacking stable, clearly defined identity, creating ambiguity and confusion.

3. Domain Services

Stateless domain operations clearly defined that don't naturally fit within Entities or Aggregates.

Explicit Business Logic

Clearly define stateless operations encapsulating domain logic.

Service Isolation

Keep domain services explicitly pure (no external dependencies), clearly focused on domain logic.

Discoverability

Document domain services explicitly and clearly for easy discovery and reuse.

Common pitfalls:

- Domain services that inadvertently maintain state or external dependencies, creating tight coupling.
- Overuse leading to anemic domain models with excessive procedural logic.

4. Repositories

Clearly abstract data persistence complexity from the domain layer.

Explicit Persistence Interfaces

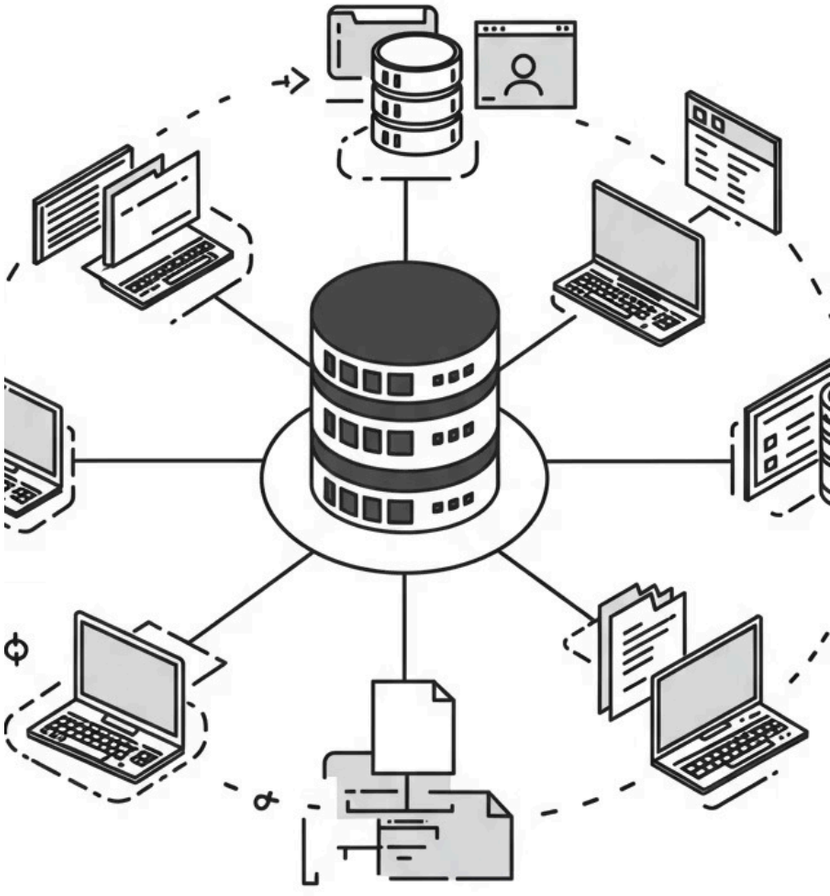
Clearly define interfaces aligned to domain terminology

Abstraction of Storage

Explicitly hide data store implementation details

Separation of Concerns

Ensure domain logic does not leak into repositories



Common pitfalls:

- Repository methods reflecting database concepts rather than domain concepts, causing tight coupling.
- Poor abstraction leading to inflexible, overly complex domain implementations.

Advanced Techniques for DDD Implementation:

- Event Storming: Collaborative visual modeling explicitly capturing domain events, aggregates, commands, and bounded contexts.
- Domain Storytelling: Narrative-driven modeling sessions explicitly uncovering domain knowledge and context boundaries clearly.
- Fitness Functions: Explicit automated tests or metrics ensuring domain correctness, consistency, and alignment over time.

Common Pitfalls in Enterprise DDD

Lack of Explicit Domain Expert Involvement

Leads to ambiguous models and misunderstanding of domain concepts.

Inconsistent Ubiquitous Language

Causes confusion, ineffective communication, and integration challenges.

Poor Governance of Bounded Contexts

Results in unclear boundaries and complex integration scenarios.

Excessive Tactical Complexity

Overengineering with too many entities, aggregates, or unnecessary patterns, creating maintenance overhead.

Ignoring Organizational Alignment

Architectural misalignment with organizational structures causing inefficiencies.