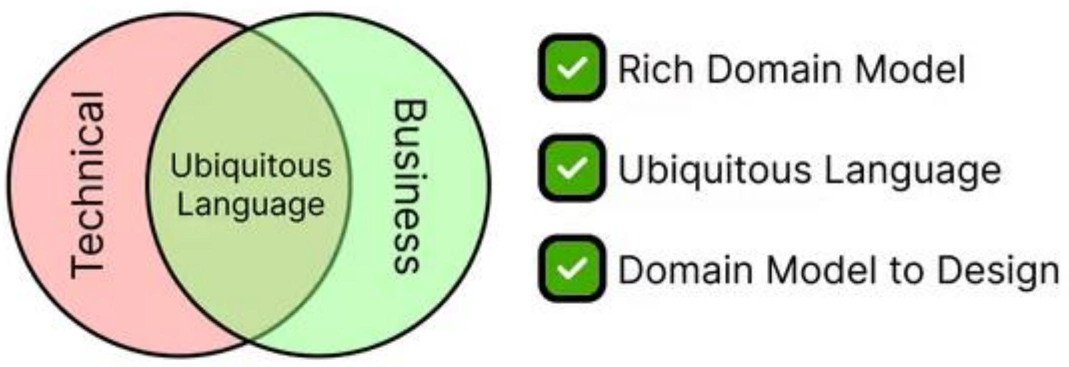
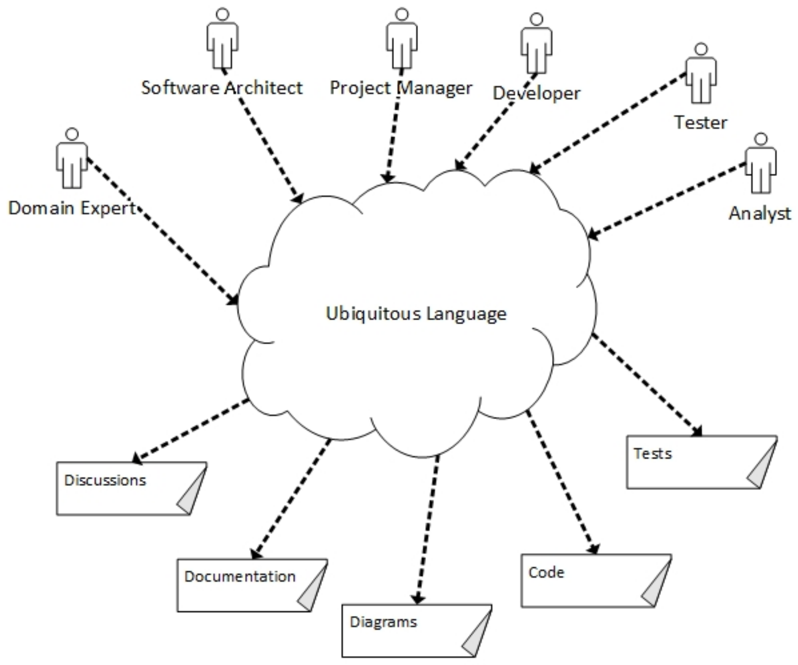
Domain-Driven Design Demystified

Domain-Driven Design (DDD) is an architectural and design approach that focuses on aligning software systems with the business domain they serve. It emphasizes collaboration between technical teams and domain experts to create models that accurately reflect real-world business processes.

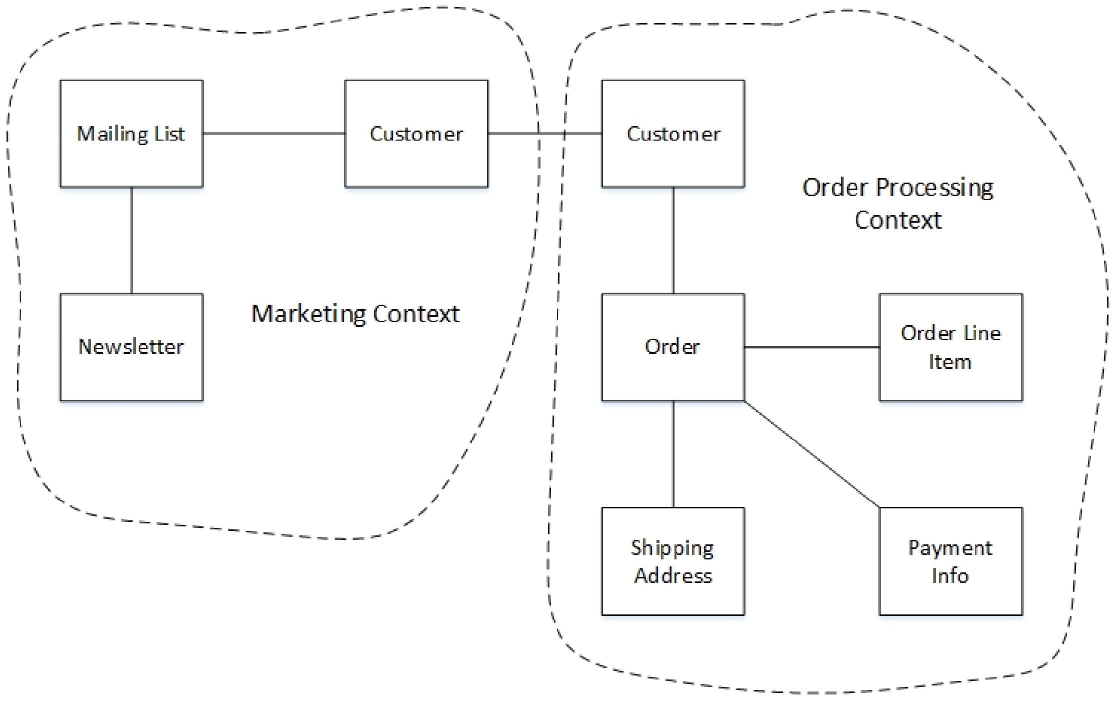


Core Concepts

1. **Ubiquitous Language**:
    - A shared language used by developers, domain experts, and stakeholders to ensure clear communication.
    - Reduces misunderstandings and ensures consistency across the team.

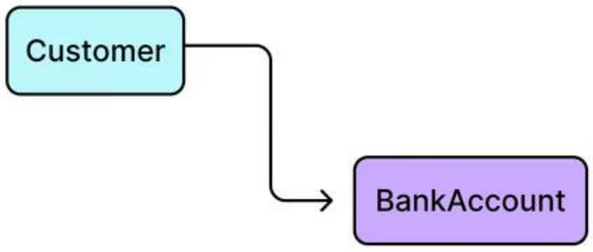


2. **Bounded Context**:
    - Defines the boundaries within which a particular model applies.
    - Prevents ambiguity by ensuring terms and concepts are scoped to specific contexts.

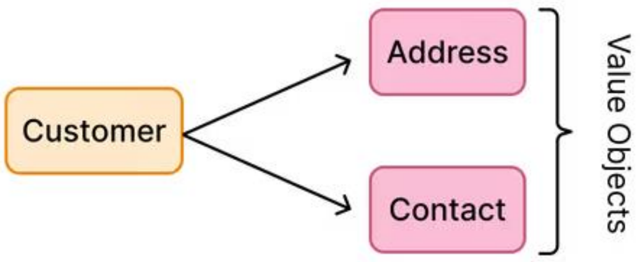


3. **Entities**:

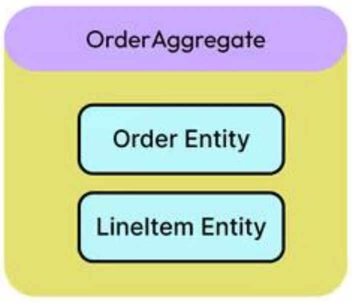
- Objects with a distinct identity that persists over time (e.g., Customer, Order).
- Focuses on behavior rather than just data.



4. **Value Objects**:
   - Immutable objects that represent descriptive aspects without identity (e.g., Address, Money).
   - Enhance modularity and reduce side effects.

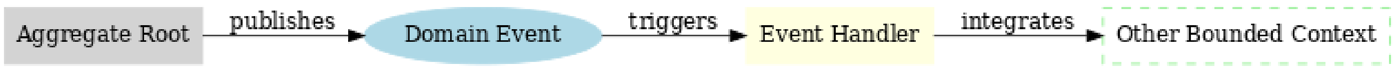


5. **Aggregates**:
   - Clusters of related objects treated as a single unit for data changes.
   - Enforces consistency and transactional integrity.

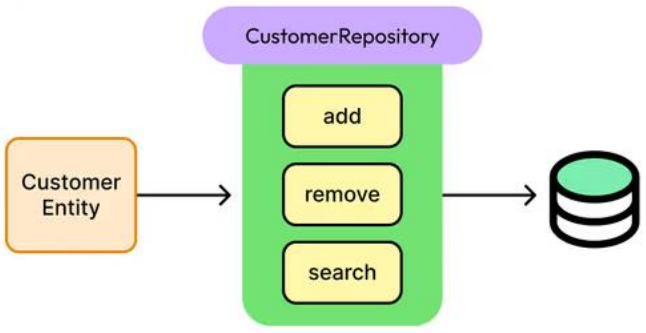


6. **Domain Events**:
   - Represent significant occurrences in the domain (e.g., OrderPlaced, PaymentProcessed).

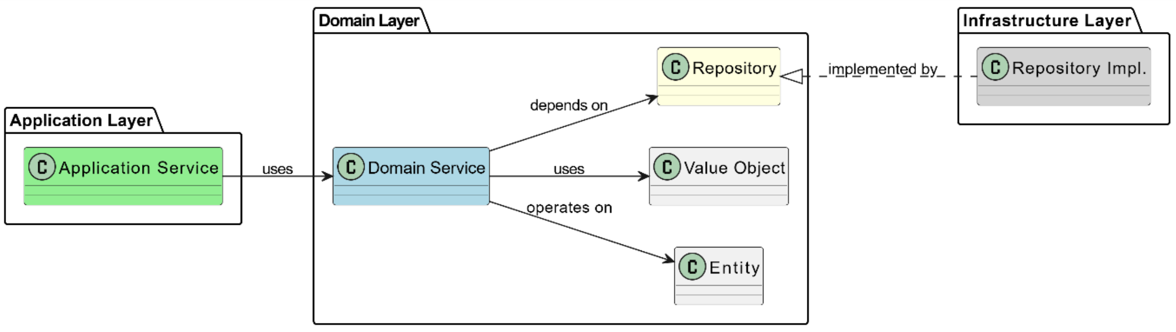
- Enable decoupled, event-driven architectures.



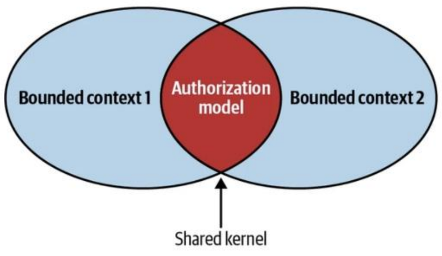
7. **Repositories**:
    - Provide an abstraction for accessing aggregates from a data store.
    - Simplifies persistence logic and hides database implementation details.



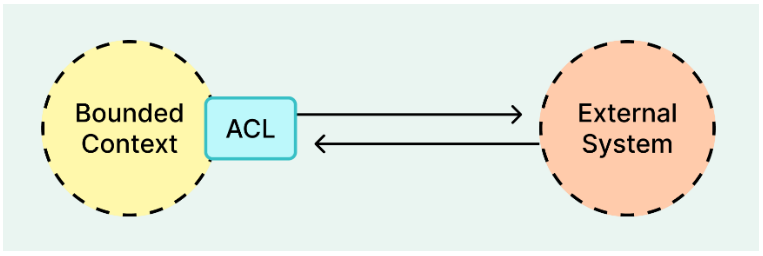
8. **Services**:
    - Encapsulate operations that don't naturally fit within an entity or value object.
    - Often used for cross-aggregate or external system interactions.



9. **Shared Kernel**:
    - A small, carefully managed subset of the domain model (e.g., common entities, value objects, interfaces) that is shared by agreement between two or more Bounded Contexts.



10. **Anti-Corruption Layer (ACL)**:
    - A design pattern that isolates one Bounded Context from the model or complexity of another, especially an external or legacy system.
    - It acts as a translator, converting data and concepts from the "foreign" model into the ubiquitous language of the local context.



Full conceptual Map (to review)

Category	Key Concepts
Strategic Design	Bounded Context, Ubiquitous Language, Context Mapping (ACL, SK, OHS, etc.), Core Domain, Subdomains
Tactical Patterns	Entities, Value Objects, Aggregates, Repositories, Services, Domain Events, Factories, Modules
Integration & Flow	Domain Events, Sagas, CQRS, Event Sourcing, Published Language
Modeling Tools	Event Storming, Specification by Example, Domain Storytelling

Key Practices

1. **Collaboration with Domain Experts**:
    - Work closely with business stakeholders to understand the domain deeply.

    - Use workshops or interviews to refine the ubiquitous language.
2. **Modeling with Bounded Contexts**:
    - Identify distinct areas of the domain and define their boundaries.
    - Use context maps to visualize relationships between bounded contexts.
3. **Event Storming**:
    - A collaborative technique to discover domain events and workflows.
    - Helps identify key processes and pain points in the system.
4. **Focus on Core Domain**:
    - Prioritize the most critical parts of the domain that provide competitive advantage.
    - Delegate less critical functionality to supporting or generic subdomains.
5. **Layered Architecture**:
    - Separate concerns into layers: Presentation, Application, Domain, and Infrastructure.
    - Keep domain logic isolated from technical details.

Architectural Decisions

1. **Monolith vs Microservices**:
    - Use **monolithic architecture** for simpler domains or early-stage projects.
    - Adopt **microservices** when bounded contexts naturally align with independent services.
2. **Event-Driven Design**:
    - Leverage domain events to enable loose coupling between components.
    - Use message brokers (e.g., Kafka, RabbitMQ) for asynchronous communication.
3. **Persistence Strategy**:
    - Choose between relational databases, NoSQL, or event sourcing based on domain needs.
    - Ensure repositories abstract persistence details from the domain layer.
4. **Integration Patterns**:
    - Use APIs, messaging, or anti-corruption layers to integrate bounded contexts.
    - Avoid tight coupling between contexts to maintain flexibility.

When to Apply DDD

- **Complex Domains**: When the business logic is intricate and requires deep understanding.
- **Collaborative Teams**: When domain experts and developers need to work closely together.
- **Event-Driven Systems**: When modeling workflows or systems with significant state changes.
- **Microservices Architectures**: When designing services around bounded contexts.