Domain-Driven Design (DDD) - User Guide

Introduction to DDD

Domain-Driven Design (DDD) is an approach to software development that focuses on building software based on the core business domain. It emphasizes collaboration between developers and domain experts to create models that reflect real-world processes.

Core Concepts of DDD

1. **Domain** – The problem space or business context (e.g., Banking, Healthcare).
2. **Model** – A representation of the domain using code.
3. **Ubiquitous Language** – Common language used by developers and domain experts.
4. **Bounded Context** – A boundary within which a model is defined and consistent.
5. **Entities** – Objects with identity (e.g., Customer, Account).
6. **Value Objects** – Objects without identity (e.g., Address, Money).
7. **Aggregates** – A cluster of entities and value objects treated as one unit.
8. **Repositories** – Mechanisms for storing/retrieving aggregates.
9. **Services** – Domain logic that doesn’t fit in entities/value objects.
10. **Factories** – Used to create complex objects/aggregates.

Strategic Design

Strategic design deals with how different parts of a large system interact.
- **Bounded Contexts** – Define clear boundaries.
- **Context Maps** – Show relationships between contexts (e.g., Customer Context, Payments Context).

Tactical Design

Tactical design patterns help in implementing domain models:
- **Aggregates** – Ensure consistency of business rules.
- **Domain Events** – Events representing something that happened in the domain.
- **Repositories** – Abstract persistence mechanisms.

Advanced DDD Concepts

- **CQRS (Command Query Responsibility Segregation)** – Separate read and write models for scalability.

- **Event Sourcing** – Persist state as a sequence of domain events.
- **Saga/Process Manager** – Handle long-running processes across bounded contexts.

Example: Citibank Account Domain

In a banking application:
- **Entity**: Account (has identity via Account Number).
- **Value Object**: Money (amount + currency).
- **Aggregate**: Account with associated transactions.
- **Repository**: AccountRepository for storing/retrieving accounts.
- **Service**: TransferService for handling money transfers.
- **Bounded Contexts**: Payments, Customer Management, Risk Assessment.

Benefits of DDD

■ Aligns software with business needs.
■ Encourages collaboration between developers and domain experts.
■ Improves scalability and maintainability.
■ Handles complex business rules effectively.

Challenges of DDD

■■ Steep learning curve.
■■ Requires strong collaboration with domain experts.
■■ Can be overkill for simple applications.

When to Use DDD?

■ When business logic is complex.
■ When domain experts are available for collaboration.
■ Avoid DDD for simple CRUD applications where complexity is low.