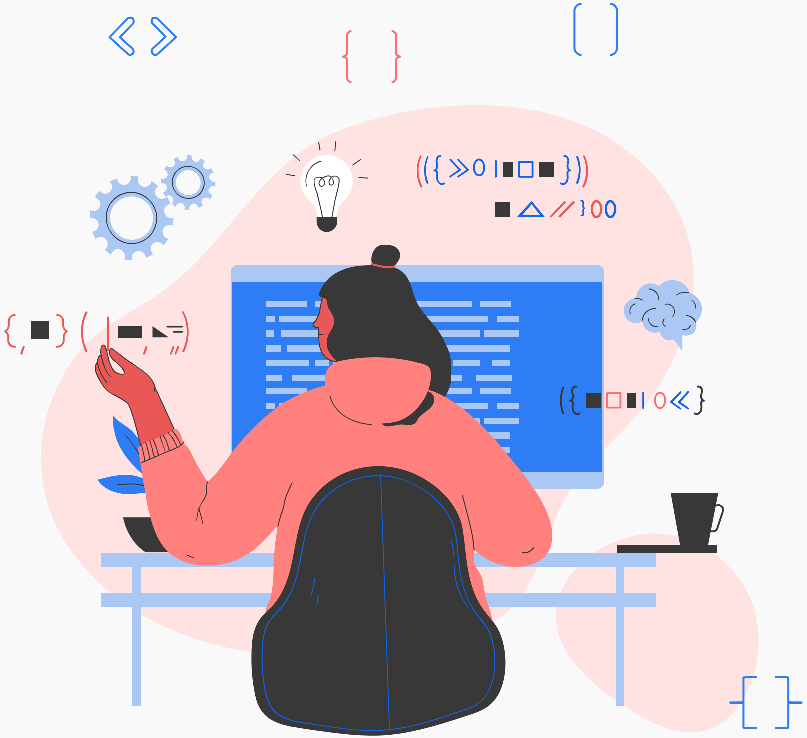
axiateam
RDT company

Microservices

Architecture Basics



Why design matters

The Cost of Poor Design

- Code that’s hard to change
- Teams stepping on each other’s toes
- APIs that confuse consumers
- Data inconsistencies and performance bottlenecks

Good Design Enables:

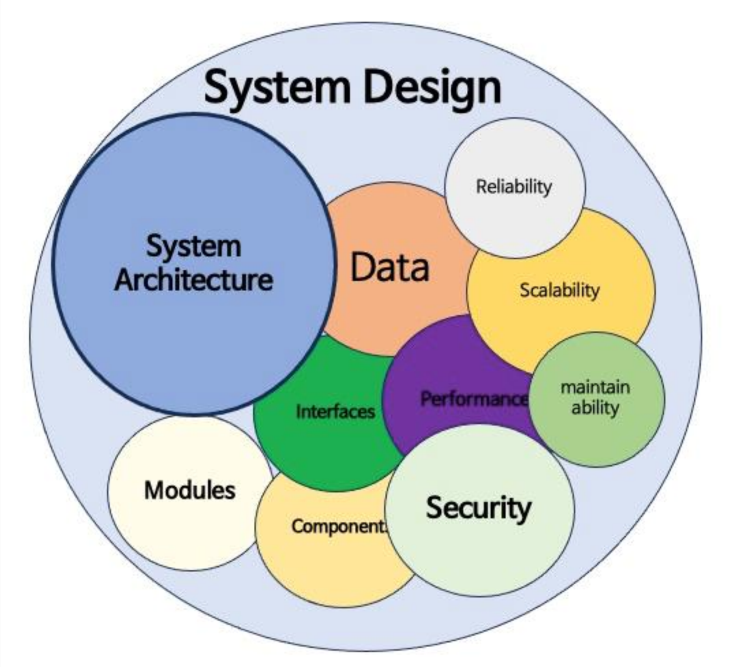
✅ Long-term maintainability

✅ Team autonomy

✅ Scalable evolution

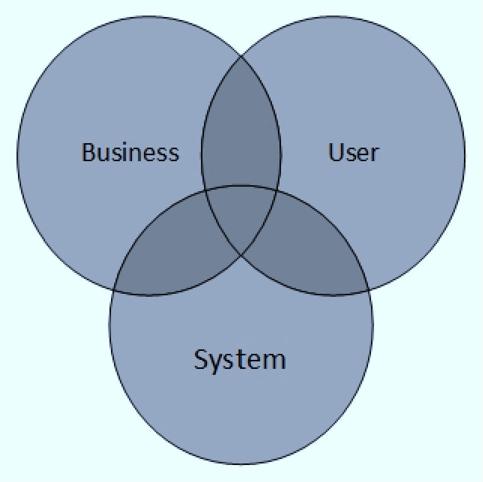
✅ Faster onboarding and safer refactoring

"Complexity is inevitable. Chaos is optional."
– Simon Brown



Domain

The area of expertise, activity, or business for which a software system is built. It encompasses the rules, processes, concepts, and vocabulary that define how things work in that specific context.

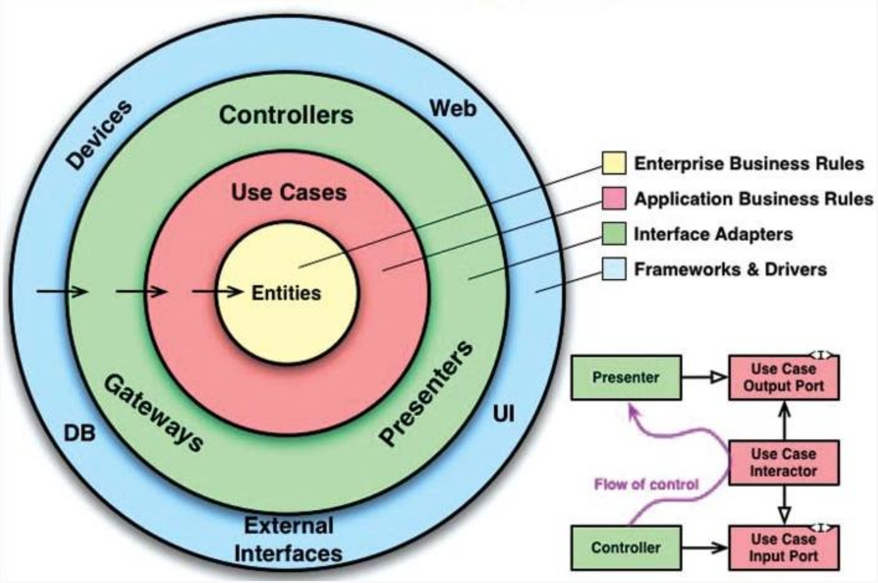


Understanding the domain of your software application is part of what is necessary to discover the appropriate architecture for any solutions you need to develop.

Why Domain Matters

- Ensures software reflects real business needs.
- Reduces miscommunication via ubiquitous language.
- Enables maintainable, testable core logic decoupled from tech.
- Foundation for Domain-Driven Design (DDD) and Clean Architecture.

Clean architecture

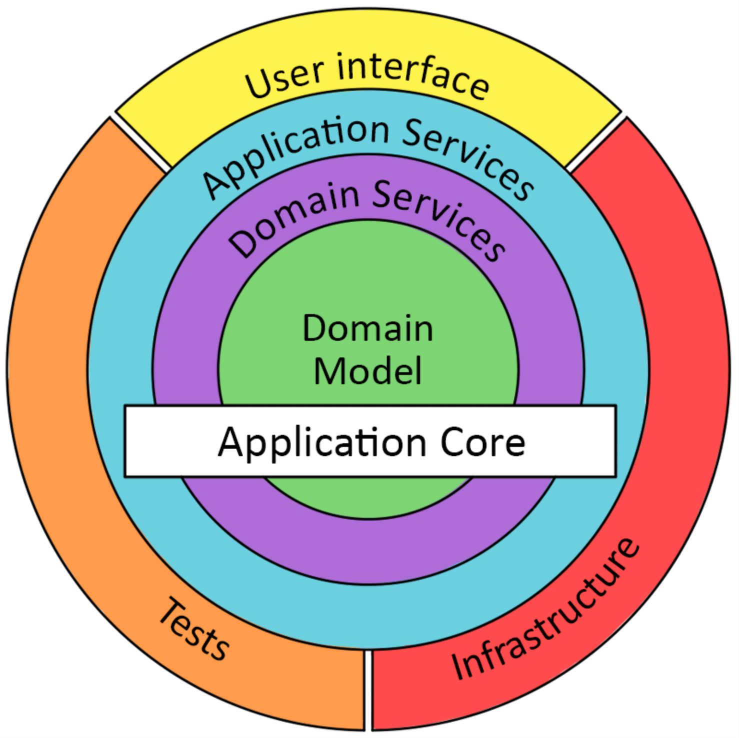


Clean Architecture is a software design philosophy introduced by Robert C. Martin, also known as Uncle Bob.

The primary goal of Clean Architecture is to create systems that are:

- **Layered Structure** – Clear separation of concerns (entities, use cases, interface adapters, frameworks).
- **Framework Independence** – Core logic doesn’t depend on external frameworks.
- **High Testability** – Business rules can be tested without UI, DB, or external services.
- **Dependency Rule** – Inner layers control outer ones; dependencies point inward.
- **UI & DB Agnostic** – Easy to swap UI or database without affecting business logic.
- **Business Logic First** – Core focus is on protecting and organizing business rules.

Domain-Driven Design (DDD)



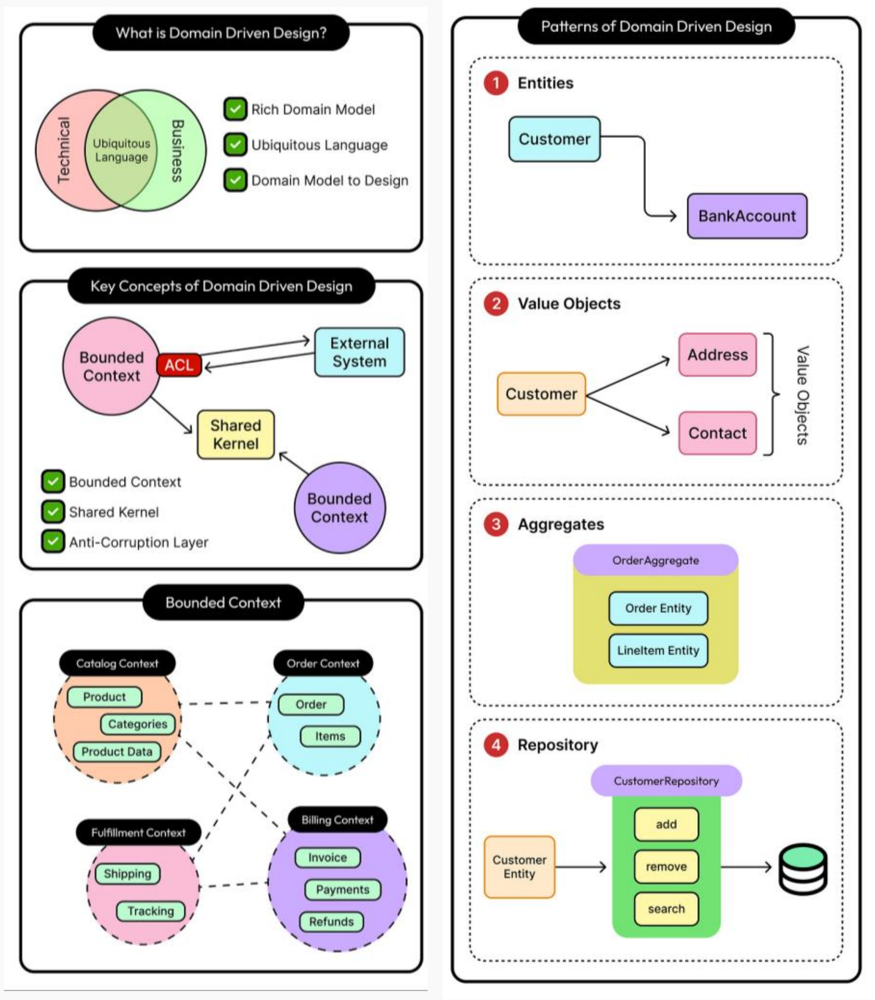
A design approach that focuses on the **core domain** and **domain logic**, aligning software structure with business reality.

Key Concepts:

- **Bounded Context**: A clear boundary where a particular model applies
- **Context Map:** Shows how different bounded contexts interact
- **Aggregate:** A cluster of domain objects treated as a single unit
- **Entity:** An object with identity (e.g., User, Order)
- **Value Object:** An object defined by its attributes (e.g., Address, Money)

> 🎯 DDD shines in complex domains (e.g., finance, logistics, healthcare)

DDD Demystified



What is Domain Driven Design?

Technical

Ubiquitous Language

Business

☑ Rich Domain Model

☑ Ubiquitous Language

☑ Domain Model to Design

Key Concepts of Domain Driven Design

Bounded Context

ACL

External System

Shared Kernel

Bounded Context

☑ Bounded Context

☑ Shared Kernel

☑ Anti-Corruption Layer

Bounded Context

Catalog Context: Product, Categories, Product Data

Order Context: Order, Items

Fulfillment Context: Shipping, Tracking

Billing Context: Invoice, Payments, Refunds

Patterns of Domain Driven Design

1 Entities

Customer

BankAccount

2 Value Objects

Customer

Address

Contact

Value Objects

3 Aggregates

OrderAggregate

Order Entity

LineItem Entity

4 Repository

Customer Entity

CustomerRepository

add

remove

search

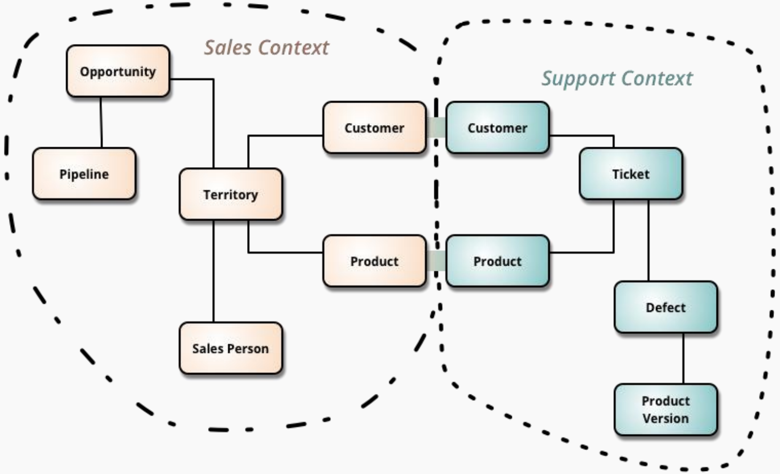
Bounded Contexts in Practice

Problem:

- Multiple teams using the word “Customer” differently:
- Sales: lead status, discounts
- Billing: payment method, invoices
- Support: tickets, SLA

Solution:

- Define separate Bounded Contexts:
- Sales Context → Lead
- Billing Context → Payer
- Support Context → TicketHolder
- Each has its own model, database, and API.



> 💡 **Golden Rule:**
> One term ≠ one meaning across the system.

Aggregates and Consistency

What Is an Aggregate?

A cluster of objects (entities + value objects) treated as a single unit for data changes.

Example: **Order Aggregate**

- Root: Order (entity)
- Children: OrderLineItems, ShippingAddress
- Invariants enforced: total amount, item limits

Rules:

Only the aggregate root is referenced externally
All changes go through the root
Ensures transactional consistency within the boundary

> ❌ Never reference an *OrderLineItem* directly from outside

When to Use DDD

Use DDD when:

- Domain is complex and central to business
- Multiple subdomains exist (core, supporting, generic)
- Teams need autonomy (e.g., microservices)
- Ubiquitous language improves communication

Avoid DDD when:

- System is simple (CRUD apps)
- Domain logic is minimal
- Team lacks modeling experience

> 🎯 **Rule of Thumb:**
> If your main challenge is not in the domain, DDD may be overkill.

Strategies to Reduce Coupling

Common Coupling Smells:

- Direct database sharing between components
- Tight API dependencies (e.g., breaking changes)
- Shared libraries with global state

Solutions:

✅ Publish-Subscribe Events – Decouple via messages

✅ API Gateways / Facades – Hide internal complexity

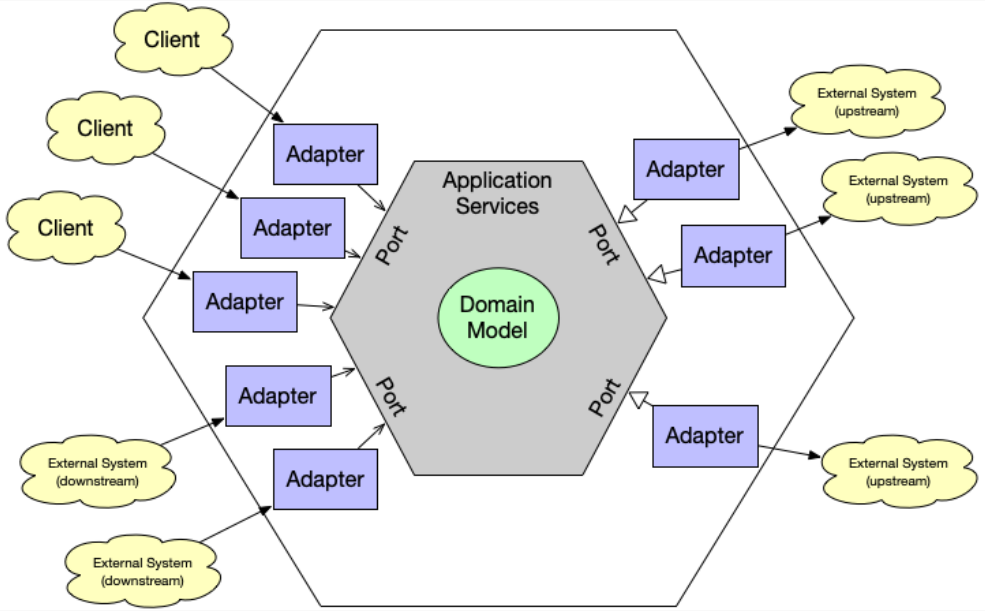
✅ Service Mesh – Externalize communication logic

✅ Hexagonal Architecture – Depend on ports, not implementations

✅ Semantic Versioning – Communicate change impact

> 🎯 **Golden Rule:**
> Components should communicate through contracts, not internals.

Hexagonal Architecture


Client
Client
Client
Adapter
Adapter
Adapter
Adapter
Adapter
Application Services
Port
Port
Port
Port
Domain Model
External System (upstream)
External System (upstream)
External System (upstream)
Adapter
Adapter
Adapter
External System (downstream)
External System (downstream)

Also known as Ports & Adapters, it's a design pattern that:

- Places the core application logic at the center
- Surrounds it with ports (interfaces) and adapters (implementations)

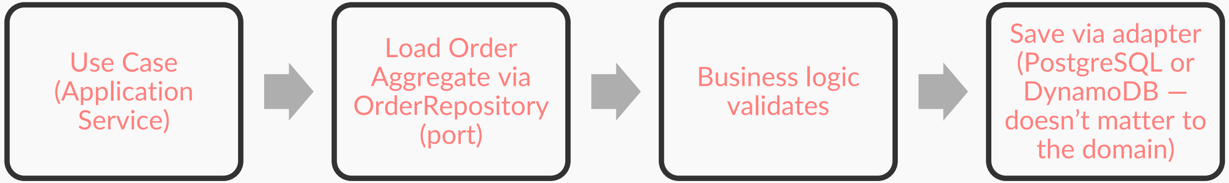
Key Concepts:

- Ports: Abstract interfaces (e.g., UserRepository, NotificationService)
- Adapters: Concrete implementations (e.g., DatabaseUserRepository, EmailNotificationAdapter)
- Inside Out: Domain logic drives interactions; external tools are pluggable

How DDD and Hexagonal Work Together

DDD Concept	Fits Into Hexagonal As
Aggregate	Core domain logic (inside the hexagon)
Repository Interface	Port (abstraction)
Database Implementation	Adapter (e.g., JPA, MongoDB)
Domain Service	Internal logic, uses ports
Application Service	Orchestrator, sits at boundary

> **Example -** When placing an order:



Use Case (Application Service)

Load Order Aggregate via OrderRepository (port)

Business logic validates

Save via adapter (PostgreSQL or DynamoDB — doesn’t matter to the domain)

Benefits of the Combination

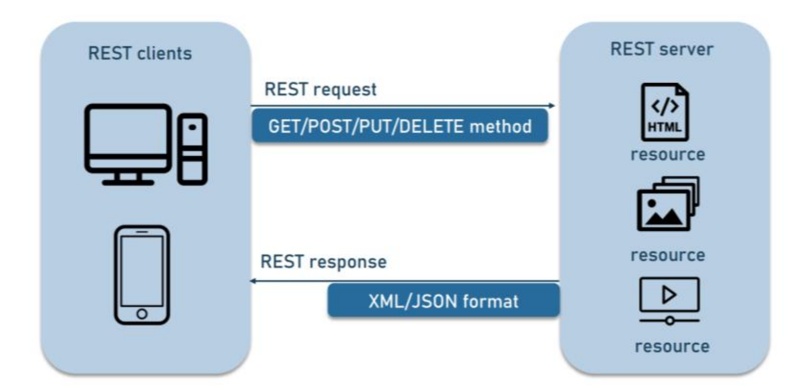
✅ **Testability:** Swap real DB with in-memory for tests

✅ **Flexibility:** Change ORM, framework, or database without touching domain

✅ **Clarity:** Clear boundaries between business rules and tech details

✅ **Future-Proof:** Replace web framework or cloud provider with minimal impact

API Design – RESTful Principles



REST (Representational State Transfer)

- Architectural style for networked applications (Roy Fielding, 2000)

Key Concepts:

- **Stateless:** Each request contains all needed info
- **Resource-Based:** Use nouns (e.g., /users/123)
- **HTTP Methods:** GET, POST, PUT, DELETE
- **HATEOAS:** Responses include links to next actions (hypermedia)
- **Versioning:** Use URL path (/v1/users) or headers