



Day 4: The Model in Application Architecture

For Senior Engineers and Managers

Inbound (Driving) Actors



Outbound (Driven) Actors



Business / Domain Logic



Interfaces



Day 4 Overview & Learning Goals

Learning Goals



Design Port-Adapter Architecture

Implement Hexagonal Architecture with ports and adapters



Distinguish DDD from BDD

Understand differences and when to apply each approach



Identify DDD Patterns

Recognize and apply tactical DDD patterns in applications



Apply Architecture Patterns

Implement patterns appropriately in different contexts

Modules Covered



Hexagonal Architecture

CQRS Pattern



DDD vs BDD

Tactical Patterns



Architecture Comparisons

Dependency Injection



Industry Case Studies

Mini-Project



Complete Resource Pack

Tools, exercises, case studies, and hands-on manual

What is Hexagonal Architecture?

Also known as **Ports and Adapters**

Separates domain logic from external concerns, making the core independent of UI, database, and infrastructure

Core Principles

1 Domain at the Center

Pure business logic without external dependencies

2 Ports as Interfaces

Define interactions with external systems

3 Adapters as Implementations

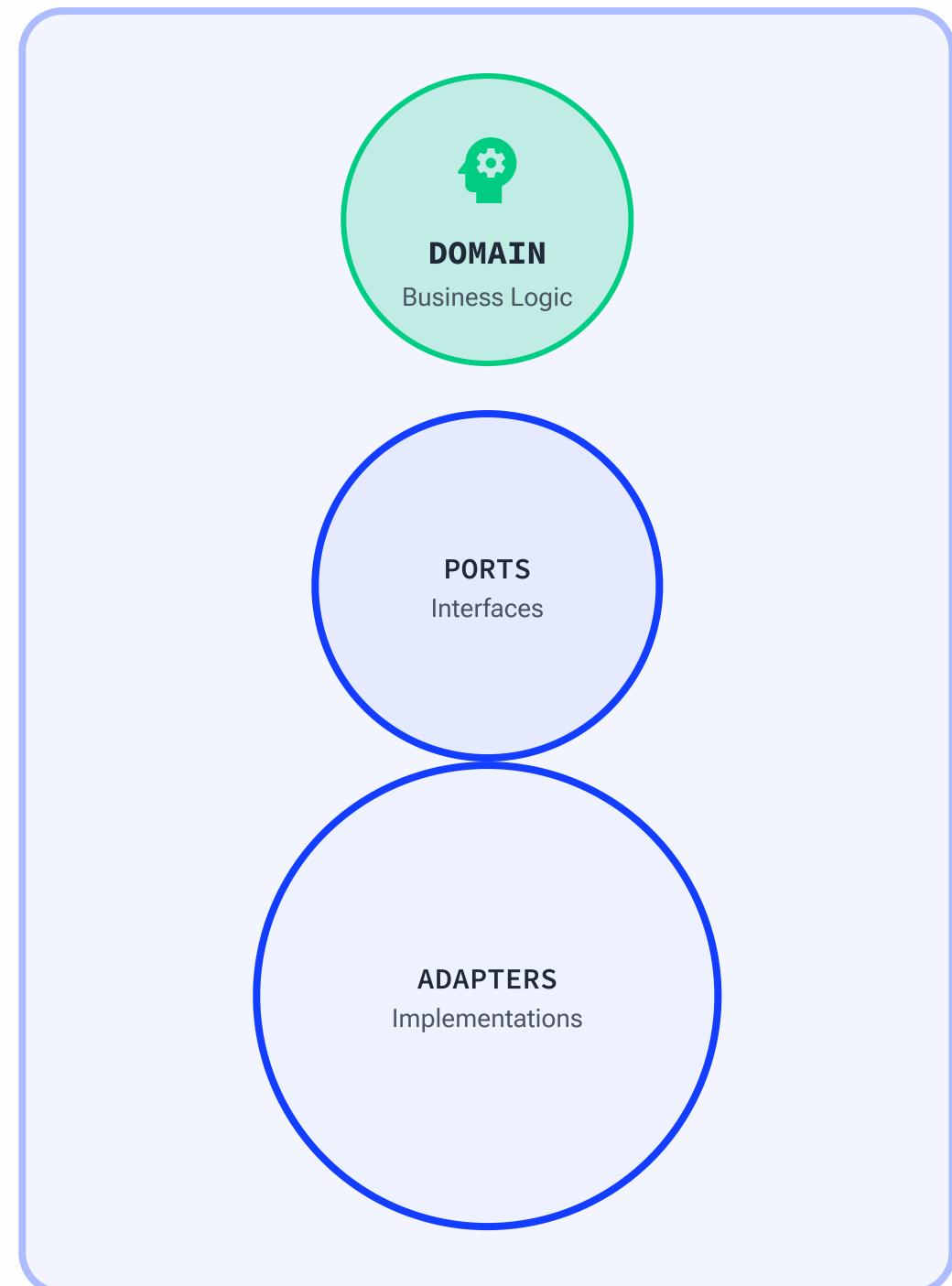
Handle concrete external system interactions

4 Dependency Inversion

Domain doesn't depend on adapters

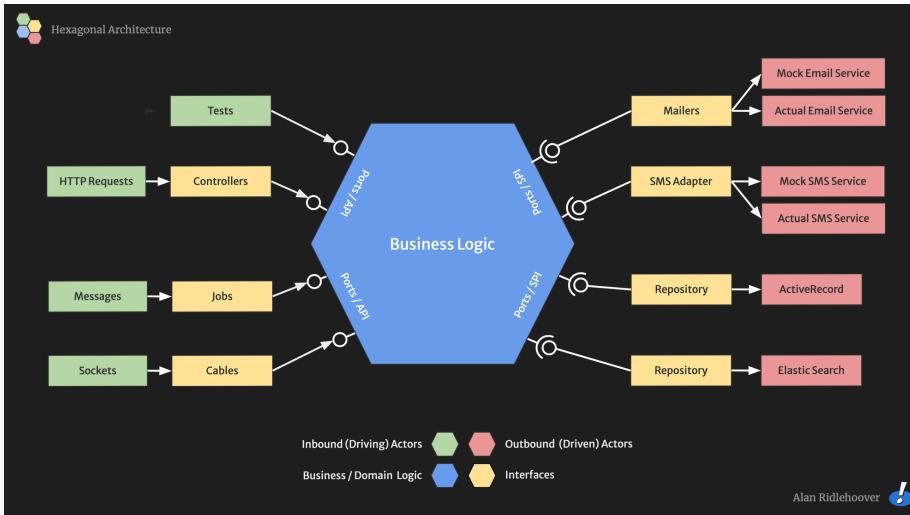
Architecture Visualization

Concentric Layers with Domain at Core



💡 Key Insight: Dependencies flow **inward** to the domain

Ports and Adapters: The Foundation



Ports

Inbound Ports

Receive requests from outside

REST gRPC UI

Interfaces for Domain Interactions

Outbound Ports

Domain initiates actions

Database External APIs Messaging

Adapters

Inbound Adapters

Call inbound ports

Controllers View Models

Implementations of Ports

Outbound Adapters

Implement outbound ports

Repositories Client Wrappers

Key Principle

Adapters depend on ports (interfaces), not the domain. This enables swapping implementations without affecting business logic.

← Adapter Types & Examples

INBOUND ADAPTERS

Driving - Receive Requests

REST Controllers

Receive HTTP requests from clients

GraphQL Resolvers

Handle GraphQL queries and mutations

CLI Commands

Command-line interface interactions

Event Listeners

Process external events from message queues

UI Components

Direct user interactions and forms

OUTBOUND ADAPTERS

Driven - Domain Initiates Actions

Database Repositories

Persist and retrieve domain data

External API Clients

Call third-party services and partners

Message Publishers

Send domain events to event streams

File System Access

Read/write files and documents

Email Services

Send notifications and alerts

Industry Case Study: BMW Group

⚙️ Unified Configurator Platform

Next-generation automotive car configuration system supporting multiple BMW Group brands

🏗️ Architecture

- ✓ Microservices design
- ✓ Hexagonal patterns
- ✓ AWS cloud-native

🏆 Results

- 1 Improved scalability
- 2 Faster delivery
- 3 Better maintainability
- 4 Reduced coupling

🔧 Tech Stack



⚠️ Brands Supported

BMW MINI Rolls-Royce

Complex product configuration rules across brands

💡 Key Learnings

- 💡 Domain separation critical for multi-brand
- 💡 Ports enable comprehensive testing
- 💡 Adapters facilitate tech changes

▶️ Watch Implementation

YouTube: BMW Unified Configurator on AWS

⚖️ Hexagonal Architecture Analysis



Benefits

- 1 Domain independence from infrastructure
- 2 Easy testing with **mock adapters**
- 3 Flexibility to change **implementations**
- 4 Clear **separation of concerns**
- 5 Supports **multiple interfaces** simultaneously
- 6 Better **testability** in isolation



Trade-offs

- 1 Increased **initial complexity**
 - 2 More **boilerplate code**
 - 3 Steeper **learning curve**
 - 4 Risk of **over-engineering** for simple apps
 - 5 Additional **indirection layers**
- Recommendation:** Use for complex domains with multiple external dependencies

⌚ When to Use Hexagonal Architecture

Use When

 Multiple external dependencies (databases, APIs, messaging)

 Frequent technology stack changes

 Complex domain logic requiring isolation

 Need for comprehensive testing

 Multiple user interfaces (web, mobile, API)

 Domain evolves independently of infrastructure

Avoid When

 Simple CRUD applications

 Single external dependency

 Tight deadlines with limited resources

 Team unfamiliar with pattern

 Small prototype/MVP

☒ Decision Framework

● 3+ criteria = **Recommended**

● 2+ avoid criteria = **Simplify**

Module 2: CQRS Pattern

i What is CQRS?

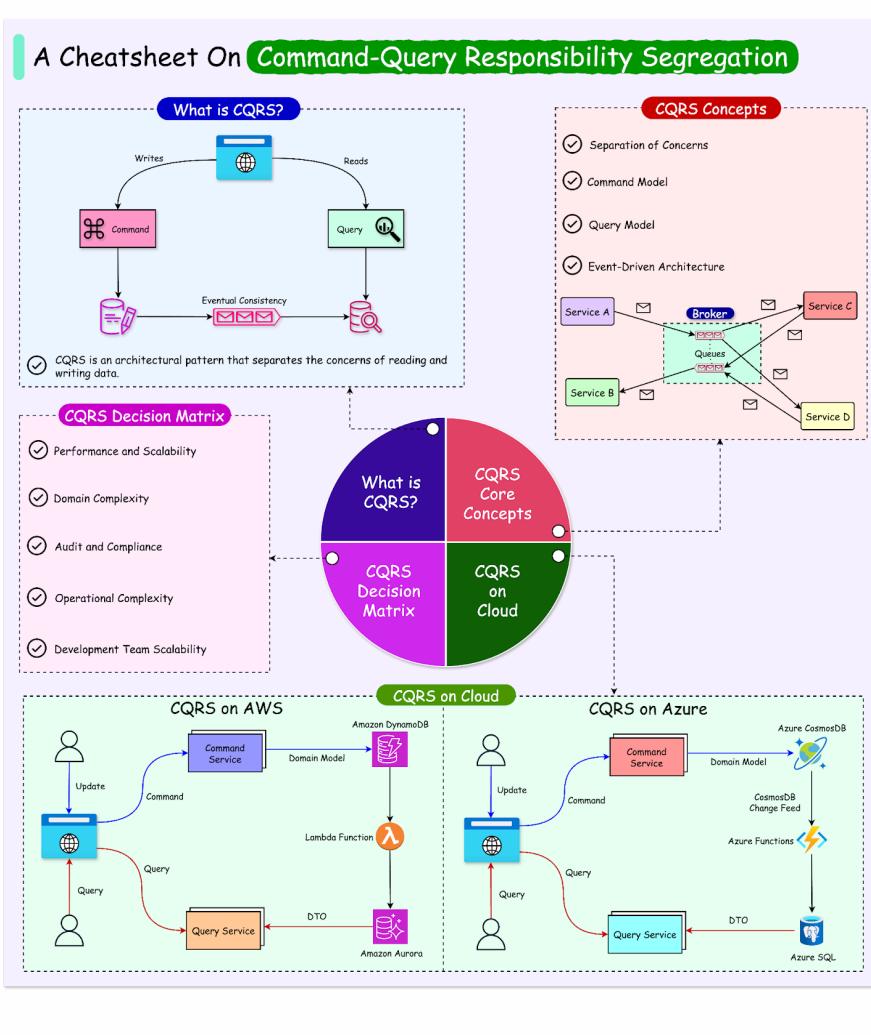
Command Query Responsibility Segregation

- 1 Separates operations for modifying data (Commands) from operations for reading data (Queries)
- 2 Based on CQS (Command Query Separation) by Bertrand Meyer
- 3 Extended by Greg Young for distributed systems
- 4 Uses different models for reads vs writes

Key Benefits

-  **Scalability**
Scale reads and writes independently
-  **Performance**
Optimize for specific operations
-  **Complexity Management**
Separate concerns effectively
-  **Business Alignment**
Match natural business operations

→ CQRS Separation



Commands (Write Side)

- ✓ Change state
- ✓ No return value
- ✓ Business operations
- ✓ Validation

Examples:

CreateOrder UpdateCustomer ProcessPayment

Queries (Read Side)

- ✓ Read state
- ✓ Return data
- ✓ Read-optimized models
- ✓ Denormalized data

Examples:

GetOrderDetails SearchProducts GetUserOrders

⚡ CQRS Models: Separate Data Models

Write Model

✓ Transactional **consistency**

☰ Normalized schema

☷ Domain-driven **design**

⚙️ Complex **business rules**

EXAMPLE:

↔ Order aggregate with events

Read Model

⌚ Read-**optimized** schema

▣ Denormalized for performance

═ Query-focused **indices**

⟳ Eventual **consistency**

EXAMPLE:

↔ OrderSummaryView for fast queries

WRITE MODEL



EVENTS



READ MODEL

⟳ Eventual Consistency

➊ Definition

Temporary period where **read and write models** may not be synchronized

How It Works

- 1 Command executed
- 2 State changes in **write model**
- 3 Event generated
- 4 Event propagates
- 5 Read model updated



🔧 Strategies

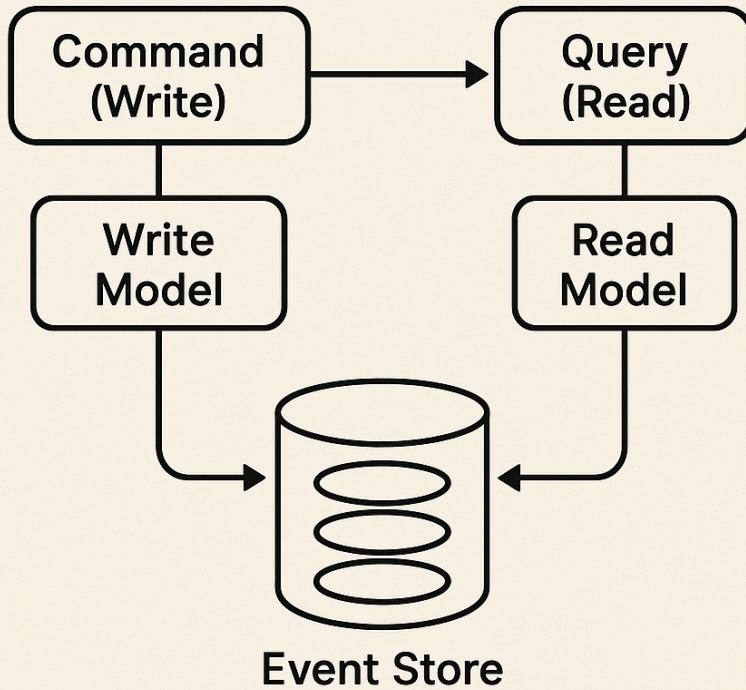
- ✓ Projection updates (background)
- ✓ Poll-based checks
- ✓ Event-driven propagation
- ✓ Version control for conflicts

⚠ Challenges

- ❗ User confusion
- ❗ Synchronization complexity
- ❗ Retry logic needed

✨ Advanced Pattern: CQRS with Event Sourcing

CQRS and Event Sourcing



⌚ Event Sourcing

All state changes stored as **immutable events**

- Complete audit trail
- Temporal queries
- Event replay capability
- Debugging aid

↗ CQRS + Event Sourcing

- 1 **Write model:** Event store
- 2 **Read model:** Projections from events
- 3 **Events** drive both sides

#[Use Cases

Banking Logistics Healthcare

⚠ Caution

- Event versioning
- Eventual consistency
- Increased complexity

🚗 Case Study: Mercedes-Benz Connected Car

❶ Context

- ✓ Next-gen connected car platform
- ✓ Vehicle telematics & diagnostics
- ✓ Over-the-air updates (OTA)

❷ Architecture

Microservices-based

Service mesh

Kubernetes

Container-driven

❸ Results Achieved

Faster Innovation

Better Scalability

Improved DX

❹ Implementation

Vehicle Management Bounded Context

Connectivity Bounded Context

OTA Updates Bounded Context

Diagnostics Bounded Context

❺ Tech Stack

Kong Gateway

HashiCorp Consul

AWS

Microsoft Azure

Case Study: BMW Group CQRS Implementation

Job Posting Requirements

C# Developer Azure Cloud (G1786)

↔ DDD - Domain-Driven Design

↗ CQRS - Command Query Segregation

➥ Factory Pattern

≡ Repository Pattern

Application Context

BMW Unified Configurator Platform

Software-Defined Vehicle (SDV) delivery platform

Implementation Details

- 1 Separate **command and query handlers**
- 2 **Aggregate roots** design
- 3 **Domain events** handling
- 4 **Event sourcing** capabilities

BMW Practice

Uses **CQRS-like pattern** in every project

Tactical DDD for large projects with complex business logic

② When to Use CQRS

✓ Use When

- Read/write **workload imbalance**
- Complex **business rules**
- Different data models needed
- High read/write volume
- **Temporal queries** required
- Audit trail requirements
- **Scalability** demands

✗ Avoid When

- ✗ Simple **CRUD operations**
- ✗ Low data volume
- ✗ **Immediate consistency** required
- ✗ Tight deadlines
- ✗ Team unfamiliarity

📊 Decision Matrix

S Scale: **High/Low**

C Consistency:
Immediate/Eventual

C Complex: **Simple/Complex**

T Team: **Expert/Novice**

⚠ CQRS Pitfalls: What to Avoid

❗ Common Pitfalls

- ✗ Over-engineering simple apps
- ✗ Ignoring eventual consistency
- ✗ Event versioning complexity
- ✗ Operational complexity
- ✗ Premature optimization
- ✗ No synchronization strategy
- ✗ Testing challenges

⚠ Anti-patterns

- 1 CQRS without separation
- 2 Synchronous propagation
- 3 Direct DB access from query side
- 4 Ignoring command validation
- 5 Not handling conflicts

🛡 Prevention Strategies

▶ Start Simple

Begin with basic separation, add complexity as needed

▶ Measure First

Optimize based on actual performance metrics

▶ Test Thoroughly

Unit test commands, integration test queries

▶ Document Clearly

Explain eventual consistency to stakeholders

💡 Rule of Thumb: Use CQRS when complexity justifies overhead

Module 3: DDD vs BDD

What is BDD?

Behavior Driven Development - Specification by Example

- Focus on system behavior
- Given-When-Then format
- Ubiquitous language
- Dev & Stakeholder collaboration
- Living documentation

Tools

Cucumber

SpecFlow

RSpec

Example Scenario

GIVEN user is logged in

WHEN user requests order

THEN order details displayed

Benefits

- ✓ Clear requirements
- ✓ Better communication
- ✓ Automated tests as documentation

← DDD vs BDD: Critical Distinctions

III Comparison Matrix

Aspect	DDD	BDD
Focus	Domain complexity	System behavior
Language	Ubiquitous language	Given-When-Then
Team	Domain experts + devs	All stakeholders
Artifacts	Bounded contexts, Aggregates	Scenarios, Features
Testing	Unit + integration	Behavior tests
Timing	Design phase	Development + testing
Application	Complex domains	All projects

↑ Combining DDD & BDD

→ DDD

Domain modeling

→ BDD

Behavior specification

💡 Best Practice

Use DDD for complex domain modeling

Use BDD for behavior specification

Together they provide **comprehensive** approach

Module 4: DDD Tactical Patterns - Aggregates

Definition

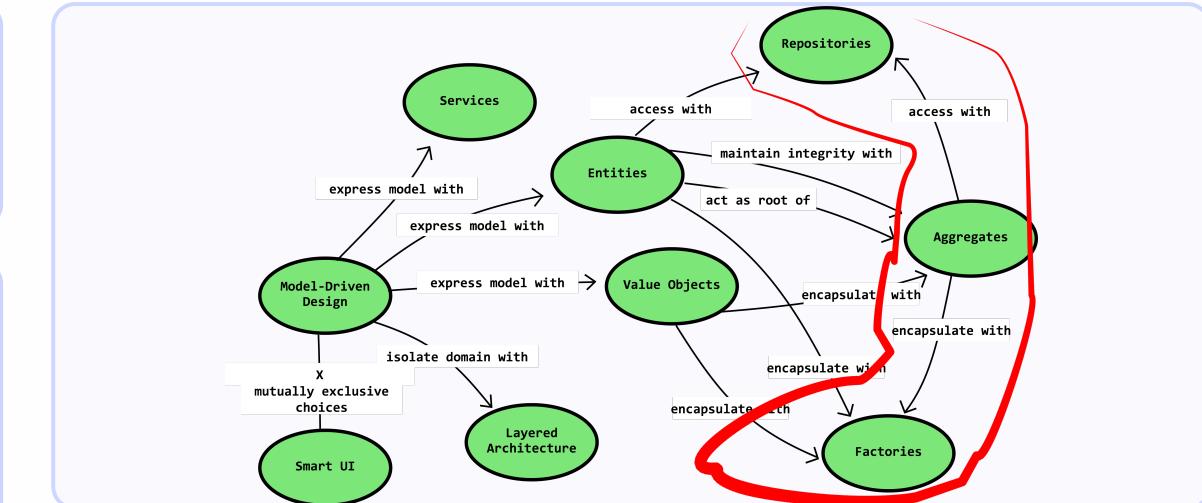
Cluster of domain objects treated as a **single unit** for data consistency

Characteristics

- **Aggregate root** as entry point
- **Internal consistency** boundary
- **Transactional** boundary
- Enforce **invariants**
- Protect **integrity**

Example: Order Aggregate

Order + OrderItems (accessed via Order root only)



Design Rules

- 1 One aggregate per transaction
- 2 Access through root only
- 3 **Small aggregates** (1-5 entities)
- 4 Eventual consistency for cross-aggregate

☰ Repositories: Collection-like Interface

☰ Purpose

Encapsulate **data access**, abstract **storage mechanism**

☱ Responsibilities

- + Add aggregate
- Remove aggregate
- 🔍 Find by ID
- ≡ Query by criteria

❖ Types

Generic Repository

Specific Repository

In-Memory Repository

↔ Implementation Steps

- 1 Define interface in **domain** layer
- 2 Implement in **infrastructure**
- 3 Use **dependency injection**

🏷️ Code Example

```
interface OrderRepository {  
    add(order: Order): void;  
    findById(id: string): Order;  
    findByCustomer(customerId: string): Order[];  
}
```

Factories: Object Creation Pattern

Purpose

Encapsulate **complex object creation**, ensure **valid aggregates**

Responsibilities

- + Create new aggregates
- ⌚ Reconstruct from persistence
- ✓ Ensure invariants
- ⌚ Hide complexity

Types

Simple Factory Methods

Abstract Factories

Builder Pattern

Examples

`OrderFactory.createOrder(customer, items)`

`CustomerFactory.createFromLegacyData(data)`

When to Use

- 1 Complex creation logic
- 2 Invariant enforcement
- 3 Multiple constructors

🔍 Domain Services: Stateless Operations

📄 Definition

Stateless operations that **don't naturally belong** to any entity or value object

⚡ Use Cases

- ↗ Cross-aggregate operations
- 🔗 External service integration
- ✖ Complex calculations
- ⌚ System-wide policies

🛍 Examples

`PaymentService.processPayment(order)`

`EmailService.sendNotification(user, message)`

✅ Characteristics

- 1 **Stateless** - no internal state
- 2 **Pure functions** - deterministic output
- 3 **Domain logic** only
- 4 **No data access** - use repositories

▶ Naming Convention

`XxxService` (`PaymentService`, `PricingService`, etc.)

Tactical Patterns: VO, Entities, BCs

❖ Value Objects

- ∅ Immutable, no identity
- ≡ Defined by attributes
- ↔ Money(amount, currency)

👤 Entities

- ⌚ Identity + lifecycle
- ✍ Mutable behavior
- ↔ Customer, Order

▣ Bounded Contexts

- ▣ Explicit boundaries
- ☒ Own language
- ▲ Distinct models



▣ Relationships

- 1 Entity contains Value Objects
- 2 Aggregate contains Entities + VOs
- 3 Bounded Context contains Aggregates

❖ Module 5: Architecture Comparisons – Layered

📄 Definition

Traditional **N-tier architecture** with clear layer separation

▀▀ Layers

- 1 **Presentation** Layer (UI, API)
- 2 **Business Logic** Layer (Domain)
- 3 **Data Access** Layer (Repositories, ORM)
- 4 **Database** Layer

↓ Direction

Top-down dependencies: Presentation → Business → Data → DB

👍 Pros

- ✓ Simple and familiar
- ✓ Easy to understand
- ✓ Clear responsibilities

👎 Cons

- ✗ Tight coupling between layers
- ✗ Hard to test in isolation
- ✗ Limited flexibility

█ Use Cases

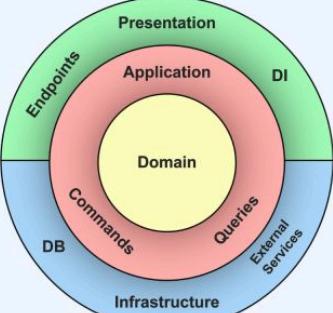
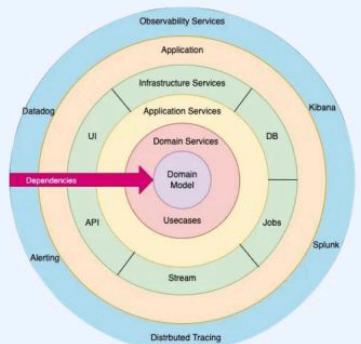
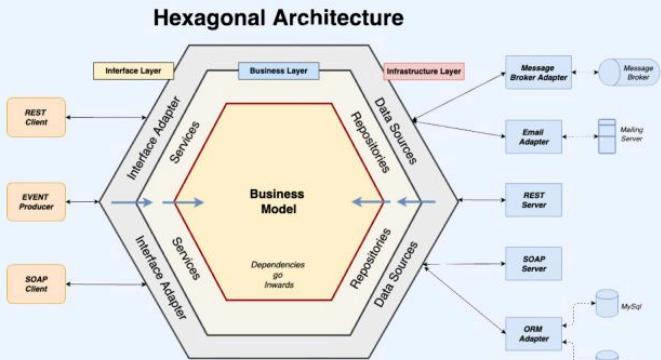
- █ Simple applications
- █ Small teams
- █ Tight deadlines

Modern Architectures: Comparison

Hexagonal, Onion & Clean Architecture



Hexagonal Architecture



Hexagonal

Ports/adapters
Domain at center
No specific layers

Onion

Concentric layers
Domain at core
Dependency inversion

Clean

Similar to Onion
Use case interactions
Focus on boundaries

→ Key Differences

- 1 Hexagonal emphasizes **ports/adapters**
- 2 Onion uses **layered structure**
- 3 Clean focuses on **use cases**

✓ Similarities

- | | | |
|--|--|--|
| <ul style="list-style-type: none">✓ Dependency inversion✓ Separate concerns | <ul style="list-style-type: none">✓ Protect domain logic✓ Domain independence | <ul style="list-style-type: none">✓ Enable testing✓ Flexible infrastructure |
|--|--|--|



@MilanJovanović



Architecture Decision Matrix

Criteria	Layered	Hexagonal	Onion	Clean
Complexity	Low	Medium	Medium	High
Learning Curve	Easy	Medium	Medium	Hard
Testing	Hard	Easy	Easy	Easy
Flexibility	Low	High	High	High
Maintainability	Low	High	High	Very High
Team Experience	Any	Exp/Med	Exp/Med	Expert
Project Size	Small-Med	Med-High	Med-High	Large
Timeline	Tight	Med-Long	Med-Long	Flexible

Decision Tree

- 1 Start with **Layered Architecture**
- 2 Add patterns as needed
- 3 Migrate to **Modern Architecture**

Key Principle

Choose architecture based on **complexity**, **team expertise**, and **project constraints**

Recommendation

Layered for simple projects, Hexagonal/Onion/Clean for complex domains

How to Choose Architecture

1 Assess Requirements

- 💼 Domain complexity
- 🎓 Team skills & experience
- ⌚ Project constraints
- ↗ Scalability needs

2 Evaluate Options

- ➡ Compare architectures using criteria table
- 📊 Analyze trade-offs & pros/cons
- ⚖ Weigh benefits vs. costs

3 Make Decision

- ✓ Choose based on analysis
- 📝 Document decision rationale
- 👥 Get stakeholder buy-in

4 Evolve & Adapt

- 🚀 Start simple, iterate incrementally
- ➕ Add architectural patterns as needed
- 🕒 Migrate to modern architecture over time
- ⟳ Refactor based on lessons learned

Decision Factors

- | | |
|----------------------------|----------------------------|
| 💡 Business criticality | 🕒 Time to market |
| 💻 Budget constraints | ⚠ Technical risk tolerance |
| 🔍 Maintenance requirements | ⌚ Performance SLA |

Architecture Signature

Each architecture has its "**DNA**" - define yours early to guide decisions consistently



Module 6: Dependency Injection (DI)

📄 What is DI?

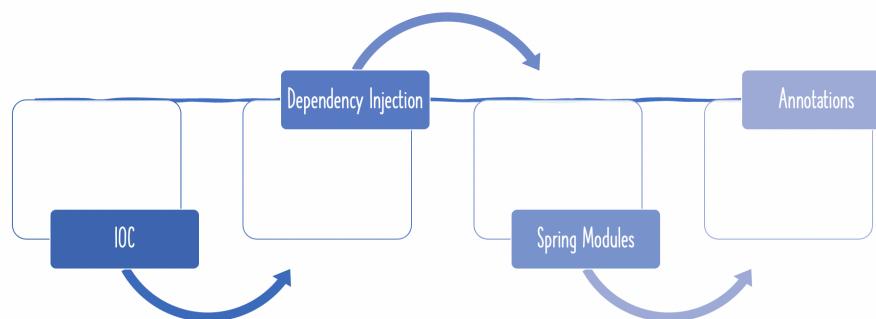
Design pattern implementing [Inversion of Control \(IoC\)](#)

🌀 Purpose

- 🔗 Decouple components
- ➡ Flexible implementations
- 🌐 Enable testing
- ⚙️ Centralized configuration

↔ Key Concepts

- 1 Dependencies are injected, not created
- 2 Constructor injection
- 3 Setter injection
- 4 Interface-based programming



✅ Benefits

- 🔗 Loose coupling
- 🔧 Runtime configuration
- 💡 Testability with mocks
- 📅 Lifespan management



Inversion of Control Container

Definition

Framework managing **object lifecycles** and **dependencies**

Injection Types

- 1 Constructor injection (preferred)
- 2 Setter/property injection
- 3 Method injection
- 4 Interface injection

Responsibilities

- + Object creation & resolution
- ⌚ Lifespan management
- ⚙ Configuration & binding

Popular Containers

Java

Spring Framework
Guice

JavaScript

InversifyJS
TSyringe

.NET

ASP.NET Core DI
Autofac

Python

Pin
dependency_injector

Code Example

- 1 Container registers IOrderRepository
- 2 Injects OrderRepositoryImpl
- 3 Lifespan: Singleton, Transient, Scoped

💡 **Key Principle:** Don't use DI for everything - apply where coupling creates problems

⊕ DI Best Practices

Implementation Patterns

- 1 Constructor injection (preferred)
- 2 Interface-based programming
- 3 Lifespan management
- 4 Auto-registration

ⓘ Lifespan Options

Singleton Transient Scoped

⊜ Best Practices

- ✓ Use **interfaces** not concretions
- ✓ Keep constructors **simple**
- ✓ Avoid **service locator** pattern
- ✓ Configure **explicitly**
- ✓ Test with **mock frameworks**
- ✓ Document **clearly**

↔ Code Example

```
public class OrderService {  
    private readonly IOrderRepository _repo;  
    public OrderService(IOrderRepository repo)  
        => _repo = repo;  
}
```

🚗 Case Study: Tesla Autopilot

⌚ Context

- ✓ Self-driving system
- ✓ Real-time sensor processing
- ✓ AI/ML integration

🔄 Implementation

- Modular microservices for independent updates
- Clean interfaces
- Explicit data contracts

▪️ Architecture

- ️ Microservices based design
- ⚡ Sensor events trigger AI processing
- ⌚ Millisecond-level responses

💡 Key Learnings

- ✓ Hardware-software synergy
- ✓ Cross-domain thinking
- ✓ Performance optimization

⚠ Challenges

- ❗ Two-board architecture complexity
- ❗ Memory capacity increase

Tesla Autopilot - Technical Deep Dive

System Architecture

- 1 Foundation Inference Infrastructure team
- 2 Modular **microservices** for independent updates
- 3 **Distributed systems** with message queues

Event Flow

-  Sensor events trigger processing
-  AI processing (neural networks)
-  Millisecond-level responses

Tech Stack

-  Custom **hardware architecture**
-  **Real-time** processing engines
-  Event-driven **microservices**

Patterns Used

- 1 **Event-driven** architecture
- 2 **CQRS-like** separation
- 3 **Hexagonal-like** components

 Ref: Tesla System Design interview guides

VW Group Competitors: Architecture Comparison

BMW Group

-  Unified Configurator Platform
-  DDD + CQRS patterns
-  Microservices on AWS
-  SDV Delivery Platform

Comparison Metrics

Criteria	BMW	MB
Complexity	High	Medium
Innovation Focus	Configurator	Connectivity
Platform	AWS	Azure + AWS
Approach	DDD-heavy	Service Mesh

Mercedes-Benz

-  Connected Car Platform
-  Service Mesh (Kong)
- Kubernetes orchestration
-  HashiCorp Consul
-  AWS + Microsoft Azure

Key Difference

-  **BMW:** Focus on vehicle configuration and customization
-  **Mercedes:** Focus on connected car services and networking
-  Both using **Microservices + Service Mesh** patterns

Automotive Industry Architectures: Key Learnings

1 Microservices Enable Updates

Independent deployment & scaling

3 Hexagonal / Service Mesh

Decoupling through adapters

5 Domain Separation Critical

Bounded contexts & aggregates

2 DDD + CQRS Extensive Use

BMW uses in every project

4 Cloud-Native Architectures

AWS, Azure, Kubernetes adoption

6 Event-Driven Architecture

Scalable & decoupled systems

↗ Common Thread

- ✓ **Flexibility** first priority
- ✓ **Scalability** through design
- ✓ **Developer Experience** drives adoption

↗ Industry Trend

Software-Defined Vehicles (SDV) driving architecture evolution

- Modular microservices standard
- DDD patterns in core systems
- Event-driven for real-time

Module 8: Tools & Resources



Qlerify

- ❖ AI-powered DDD modeling
- ❖ Process & domain models
- ❖ Data schemas, Given-When-Then

[Ref #37:](#) Complete DDD tool suite with AI features



Context Mapper

- ❖ DDD examples & patterns
- ❖ Cargo domain example
- ❖ Model visualization tools

[Ref #65:](#) Reference implementations & mapping tools



DDD Toolbox

- ❖ Strategic design tools
- ❖ Web application interface
- ❖ Context mapping support
- ❖ Event storming collaboration

[Ref #38:](#) Modern web app for DDD practitioners



GitHub Resources

- ❖ ThreeDotsLabs Wild Workouts Go
- ❖ Community DDD repositories
- ❖ Production-ready examples

[Ref #61:](#) Wild Workouts Go DDD example project

DDD Tooling Ecosystem

GitHub

-  Repositories & examples
-  Community discussions
-  Pull request reviews

Stack Overflow

-  Q&A solutions
-  Problem-specific search
-  Expert community validation

DDD Community

-  Conferences & meetups
-  Open-source projects
-  Learning materials

Medium

-  Technical articles
-  Case studies & tutorials
-  Author insights

Reddit */r/DomainDrivenDesign*

-  Active community
-  Success stories
-  Experience sharing

Learning Resources

-  InfoQ Architecture Guide
-  Martin Fowler's Patterns
-  Microsoft Learn DDD
-  5 Best Tools for DDD

Conferences

-  DDD Europe Conference
-  Local DDD meetups
-  Workshops & training



Module 9: Mini-Project – Connected Car Telematics System

🚗 Project Title

Build a Connected Car **Telematics System** using DDD principles

💼 Business Context

Automotive company developing **telematics platform** for fleet management, remote diagnostics, and predictive maintenance

⚠ Technical Stack

- ✓ Hexagonal Architecture
- ✓ Domain Events
- ✓ CQRS Pattern
- ✓ Microservices

⚡ Requirements

- 1 Vehicle **data collection** from sensors
- 2 **Real-time telemetry** streaming
- 3 **Driver behavior** analysis
- 4 **Maintenance scheduling** alerts
- 5 **Fleet management** dashboard

📊 Learning Goals

- Apply **Hexagonal Architecture** patterns
- Implement **CQRS** for read/write separation
- Use **DDD tactical patterns** (Aggregates, Repositories, Services)
- Design **Bounded Contexts** for clear separation

1 Domain Modeling

Identify Bounded Contexts

Vehicle Management Telematics Fleet
Maintenance

Define Aggregates

Vehicle TelemetryData Driver Schedule

Design Value Objects

GPSLocation SensorReading

2 Architecture Design

Design Hexagonal Architecture

Domain at center, ports/adapters

Define Ports

Inbound
↓ (REST, GraphQL)
Outbound
↑ (Database, MQTT)

Plan Adapters

Implement external system interfaces

3 Implementation

Set Up Project Structure

Follow hexagonal layout

Implement Domain Layer

Aggregates, VOs, Services

Add Ports & Adapters

Interfaces and implementations

Configure DI Container

Wiring and lifespans

Tip: Start with Phase 1, validate with stakeholders before moving to Phase 2

Project Deliverables

Deliverables

- 1 **Domain Model Diagram** (bounded contexts, aggregates)
- 2 **Architecture Design** (hexagonal diagram, ports/adapters)
- 3 **Code Implementation** (domain layer, ports, adapters)
- 4 **Documentation** (README, API docs, deployment guide)
- 5 **Tests** (unit tests, integration tests)

Submission Format

-  GitHub repository with clear structure
-  Include comprehensive README.md
- Version control with meaningful commits

Evaluation Criteria

- | | |
|--|---|
| 
40% | DDD Pattern Correctness
Bounded contexts, aggregates, repositories, domain services |
| 
25% | Architecture Quality
Hexagonal pattern, ports/adapters design, decoupling |
| 
20% | Code Quality
Clean code, SOLID principles, test coverage |
| 
15% | Documentation
Architecture diagrams, API documentation, README |

 **Bonus Points:** Using recommended tools (Qlerify, DDD Toolbox)

Module 10: Comprehensive Glossary

Architecture

- Hexagonal
- Onion
- CQRS
- DI
- Layered
- Clean
- Event Sourcing
- IoC

DDD Patterns

- Aggregate
- Factory
- Value Object
- Bounded Context
- ACL
- PL
- Repository
- Domain Service
- Entity
- Context Map
- OHS

Concepts

- Port
- Invariant
- Subdomain
- Adapter
- Ubiquitous Language
- Core Domain

Key Definitions

- ❖ **Hexagonal Architecture**
Ports & Adapters pattern with domain at center
- ❖ **CQRS**
Command Query Responsibility Segregation

- ❖ **Domain-Driven Design**
Focus on business domain logic

- ❖ **Dependency Injection**
IoC pattern for object creation

Patterns & Principles

- **Bounded Context:** Explicit boundary within which model applies
- **Aggregate Root:** Entry point for aggregate manipulation
- **Ubiquitous Language:** Common language shared by team and domain
- **Port:** Interface for domain interaction
- **Adapter:** Implementation of port for external system
- **Invariant:** Business rule that must always be true
- **ACL:** Anti-Corruption Layer for external integration
- **OHS:** Open Host Service for published API

Module 11: Learning Goals Achievement

1

Design Port-Adapter Architecture

Implement hexagonal architecture with ports and adapters

- ✓ Modules 1-3: [Hexagonal Architecture](#)
- ✓ Module 5: [Architecture Comparisons](#)
- ✓ Module 6: [Dependency Injection](#)

2

Distinguish DDD from BDD

Understand differences between domain and behavior-driven design

- ✓ Module 3: [DDD vs BDD](#)
- ✓ Module 4: [Tactical Patterns](#)

3

Identify Patterns in DDD Applications

Recognize and apply tactical DDD patterns correctly

- ✓ Module 4: [Tactical Patterns](#)
- ✓ Module 7: [Industry Case Studies](#)
- ✓ Module 9: [Mini-Project](#)

4

Apply Patterns to Contexts

Choose appropriate architecture based on context

- ✓ Module 5: [Decision Framework](#)
- ✓ Module 7: [Industry Examples](#)
- ✓ Module 9: [Connected Car Project](#)



All learning goals met through comprehensive modules, case studies, and hands-on mini-project



Module 12: Day 5 Preparation

⌚ Day 4 Recap Checklist

- 1 Hexagonal architecture understood
- 2 CQRS principles mastered
- 3 DDD vs BDD distinguished
- 4 Tactical patterns identified
- 5 Architecture choices evaluated
- 6 Dependency Injection implemented
- 7 Case studies reviewed

📖 Self-Study Topics

- 1 Domain-Driven Design (Eric Evans)
- 2 Clean Architecture (Robert C. Martin)
- 3 Microservices Patterns (Martin Fowler)
- 4 Event Sourcing (Greg Young)

→ Day 5 Preview

-  **Distributed Systems**
Message queues, service discovery
-  **Event-Driven Architecture**
Async communication, eventual consistency
-  **Cloud-Native Patterns**
Kubernetes, service mesh, observability

💡 Preparation Tips

- ✓ Review DDD tactical patterns (Aggregates, Repositories)
- ✓ Practice CQRS with simple examples
- ✓ Explore DDD tools (Qlerify, DDD Toolbox)
- ✓ Start mini-project: Connected Car Telematics

Summary: Day 4 Complete

💡 Key Takeaways

- 1 Port-adapter architecture protects domain logic
- 2 CQRS separates reads/writes for performance
- 3 DDD & BDD serve different purposes
- 4 Tactical patterns guide implementation
- 5 Architecture choice depends on context
- 6 Dependency Injection enables loose coupling
- 7 Automotive industry uses these patterns extensively

📘 Resources Covered

- | | |
|--------------------------|----------------------------|
| ✓ Hexagonal Architecture | ✓ CQRS Pattern |
| ✓ DDD vs BDD | ✓ Architecture Comparisons |
| ✓ Tactical Patterns | ✓ Dependency Injection |
| ✓ Case Studies | ✓ Tools & Platforms |
| ✓ Mini-Project Guide | ✓ Glossary |
| ✓ BMW/MB/Tesla | |

💼 Learning Goals Met

- ✓ Design port-adapter architecture ✓
- ✓ Distinguish DDD from BDD ✓
- ✓ Identify patterns in DDD applications ✓
- ✓ Apply architecture patterns to contexts ✓

🛠 Ready for Day 5

Mini-Project: Connected Car Telematics System - Hands-on implementation of Day 4 concepts

Comprehensive References

Books

- 1 [Domain-Driven Design](#) (Eric Evans)
- 2 [Clean Architecture](#) (Robert C. Martin)
- 3 [Implementing DDD](#) (Vaughn Vernon)

Articles

- ✓ [InfoQ Architecture Guide](#)
- ✓ [Microsoft Learn DDD](#)
- ✓ [5 Best Tools for DDD](#)
- ✓ [DDD Tools Comprehensive Guide](#)

Online Resources

- 1 [Qlerify](#) - AI-powered DDD modeling
- 2 [DDD Toolbox](#) - Strategic design tools
- 3 [Context Mapper](#) - DDD examples
- 4 [ThreeDotsLabs](#) - Wild Workouts Go

Community

- 💬 [r/DomainDrivenDesign](#)
- 📅 [DDD Community Forums](#)
- 🌐 [DDD Europe Conference](#)

Learning Platforms

- ▶ [YouTube: Architecture Tutorials](#)
- ↔ [GitHub: DDD Repositories](#)
- ≡ [Medium: DDD Articles](#)
- ❓ [Stack Overflow: DDD Questions](#)

Key Takeaways

- ✓ 45 comprehensive slides
- ✓ All learning goals met
- ✓ Hands-on mini-project included
- ✓ Industry case studies (BMW, MB, Tesla)