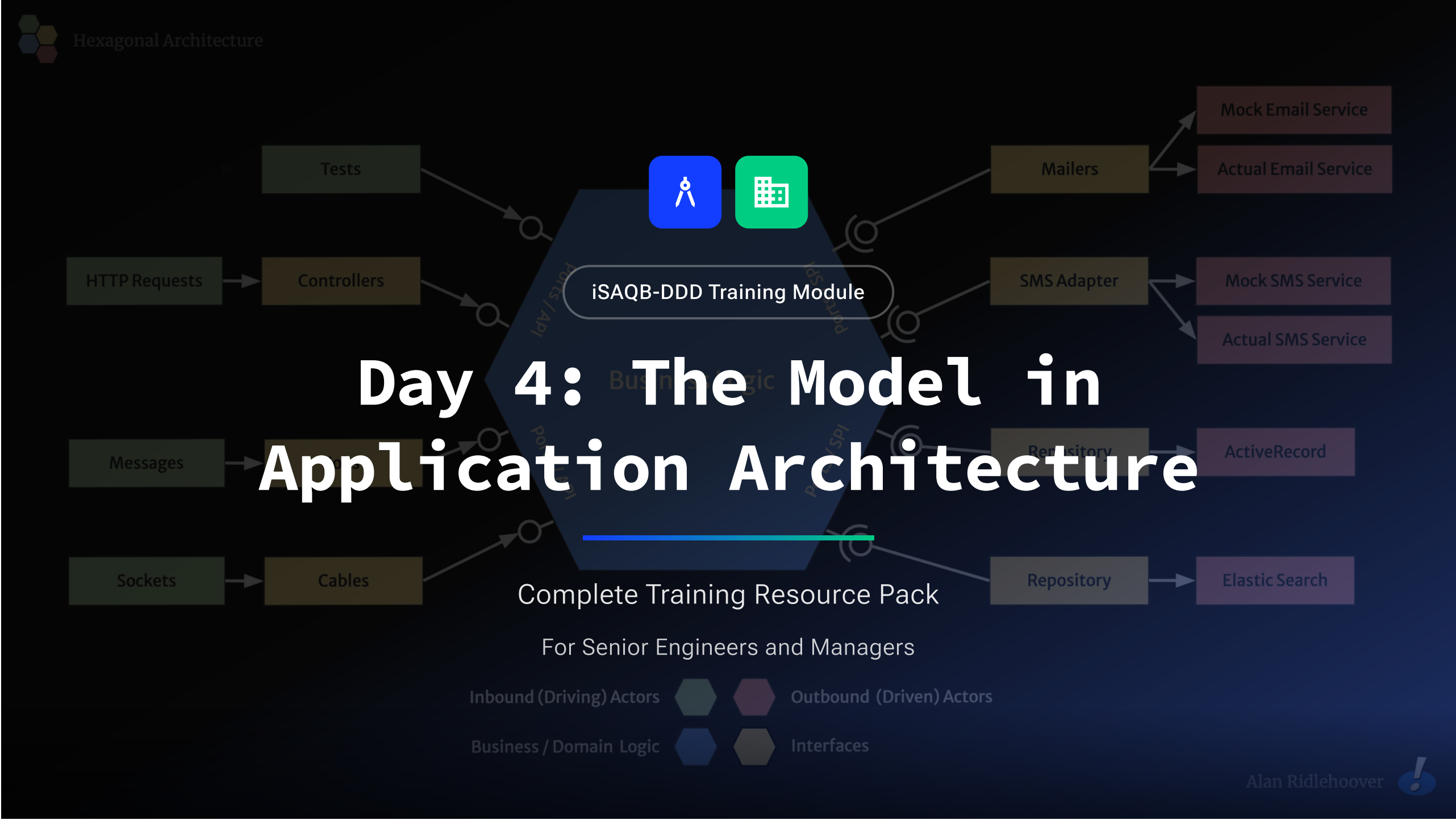

Hexagonal Architecture
Tests
HTTP Requests
Controllers
Messages
Sockets
Cables
Mailers
Mock Email Service
Actual Email Service
SMS Adapter
Mock SMS Service
Actual SMS Service
ActiveRecord
Repository
Elastic Search
iSAQB-DDD Training Module
Day 4: The Model in Application Architecture
Complete Training Resource Pack
For Senior Engineers and Managers
Inbound (Driving) Actors
Outbound (Driven) Actors
Business / Domain Logic
Interfaces
Alan Ridlehoover

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

Module 1: Hexagonal Architecture

What is Hexagonal Architecture?

Also known as Ports and Adapters

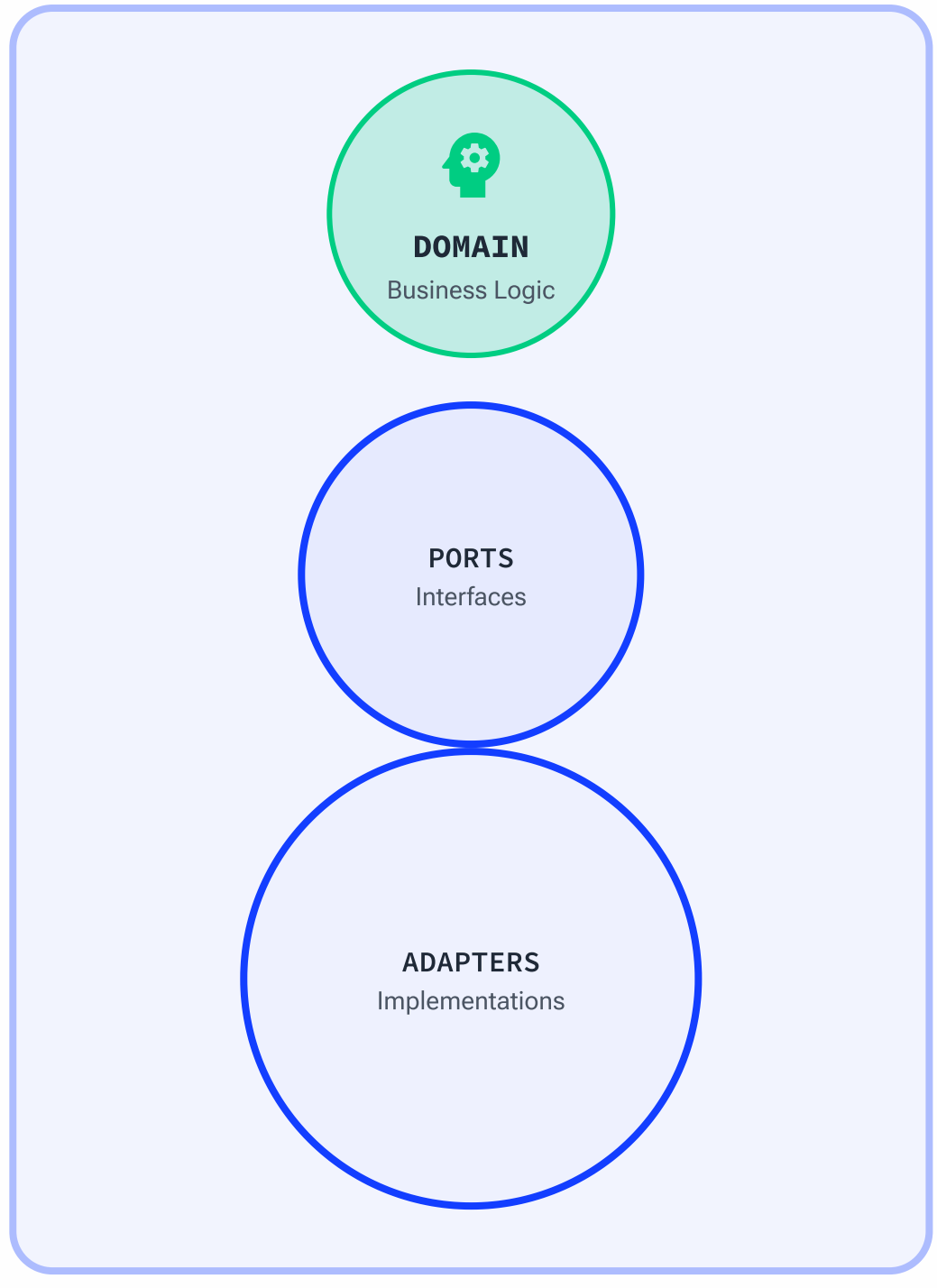
Separates domain logic from external concerns, making the core independent of UI, database, and infrastructure

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

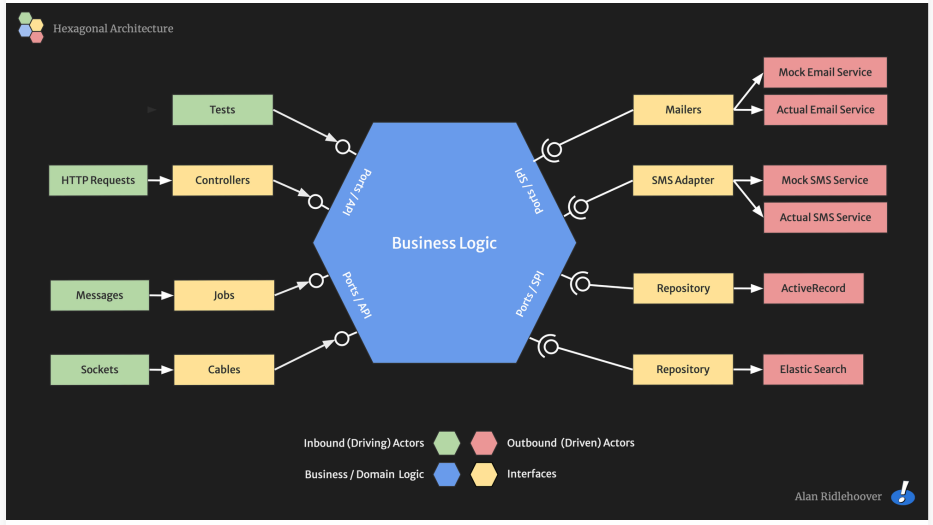
Architecture Visualization

Concentric Layers with Domain at Core



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

Ports and Adapters: The Foundation



PORTS

Interfaces for Domain Interactions

Inbound Ports

Receive requests from outside

- REST
- gRPC
- UI

Outbound Ports

Domain initiates actions

- Database
- External APIs
- Messaging

ADAPTERS

Implementations of Ports

Inbound Adapters

Call inbound ports

- Controllers
- View Models

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

Tech Stack

DDD	CQRS
AWS	gRPC

Results

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

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

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

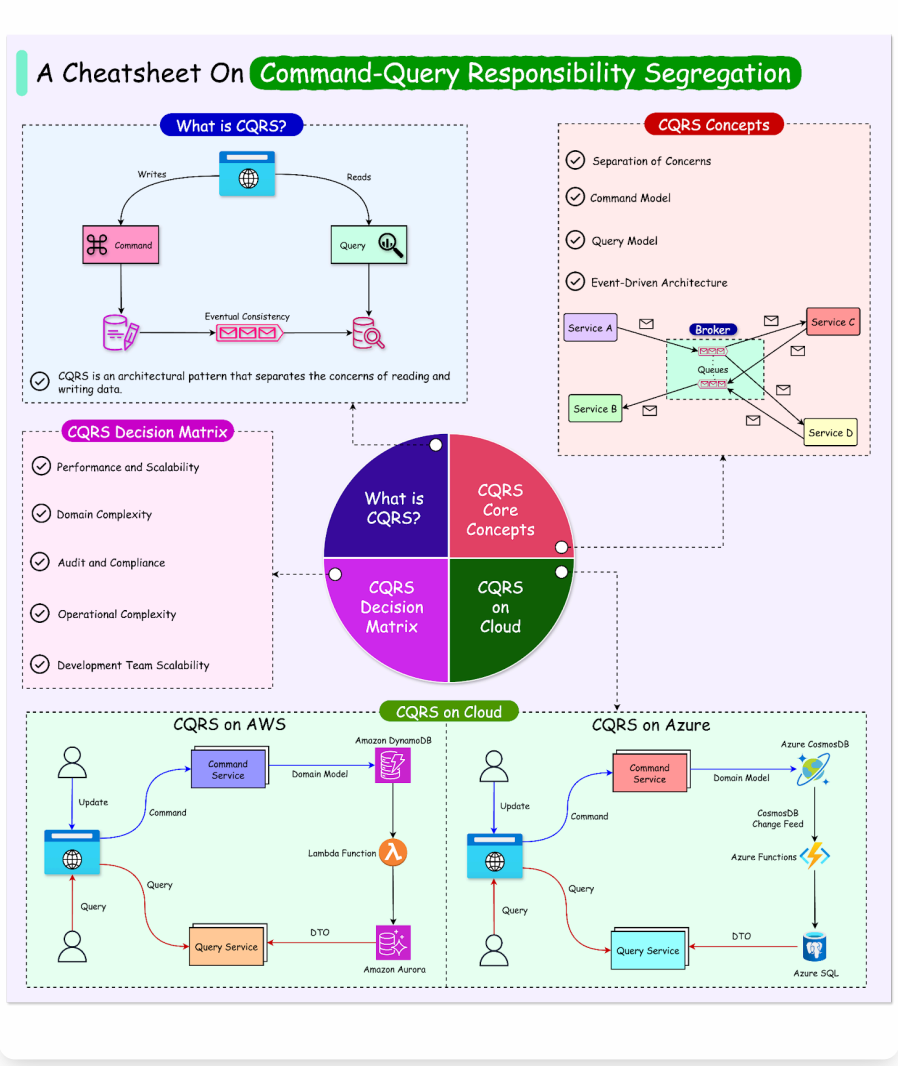
Scalability
Scale reads and writes independently

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

Eventual Consistency

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**

Timeline

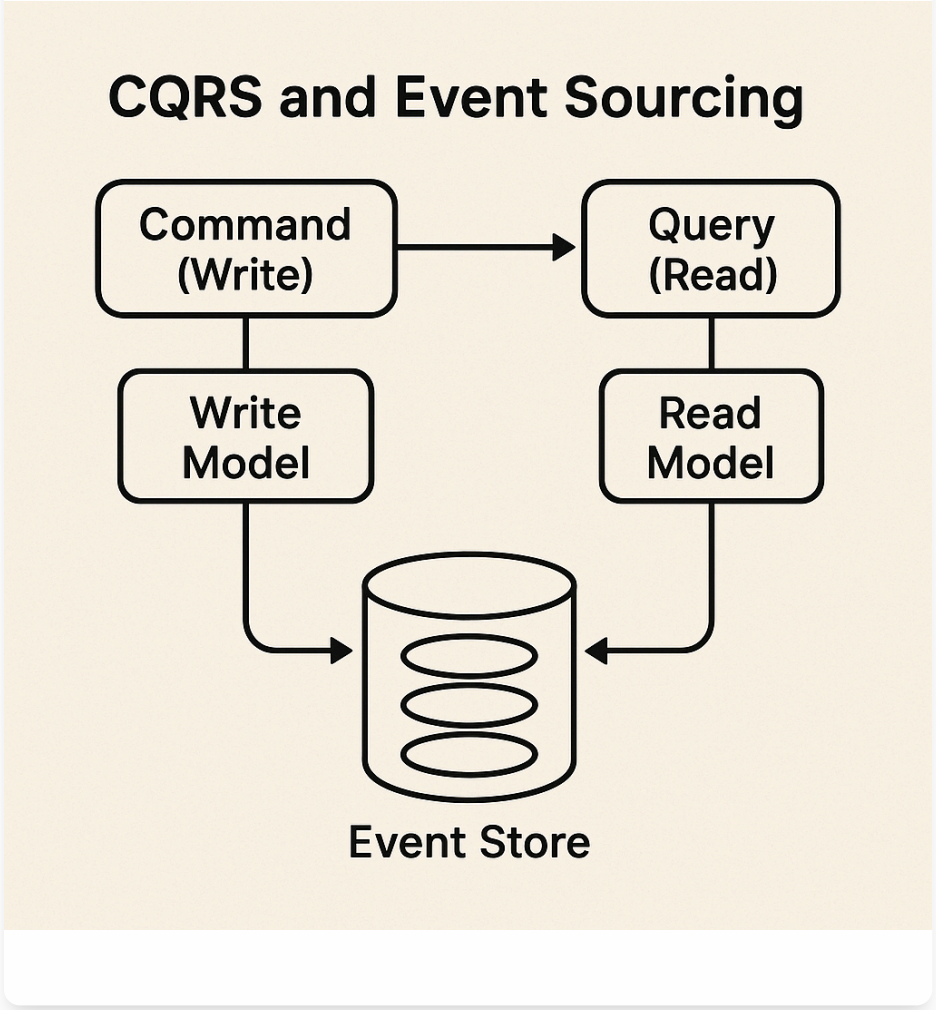
Strategies

- 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



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

Context

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

Architecture

Microservices-based	Service mesh
Kubernetes	Container-driven

Results Achieved

Faster Innovation | Better Scalability | Improved DX

Implementation

- **Vehicle Management** Bounded Context
- **Connectivity** Bounded Context
- **OTA Updates** Bounded Context
- **Diagnostics** Bounded Context

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

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** Complex: Simple/Complex
- **C** Consistency: Immediate/Eventual
- **T** Team: Expert/Novice

⚠ CQRS Pitfalls: What to Avoid

Common Pitfalls

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

⚠ 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

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

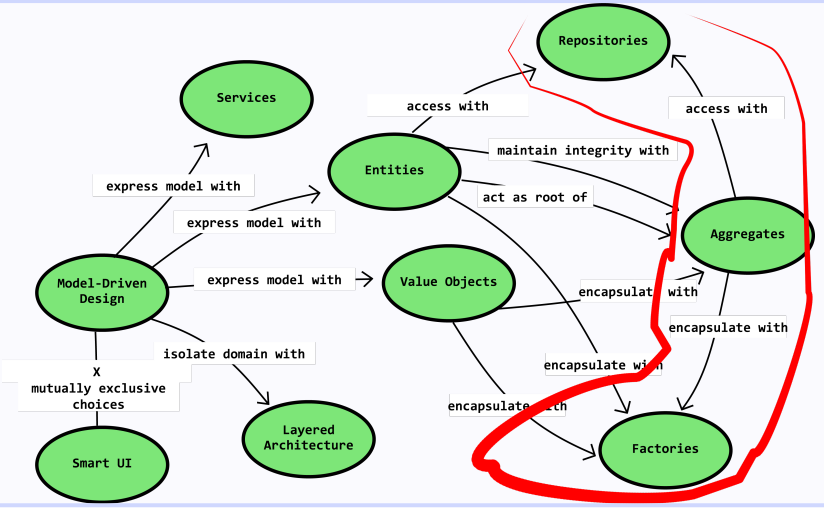
Cluster of domain objects treated as a **single unit** for data consistency

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

- 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

- Dependency inversion
- Separate concerns
- Protect domain logic
- Domain independence
- Enable testing
- Flexible infrastructure

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

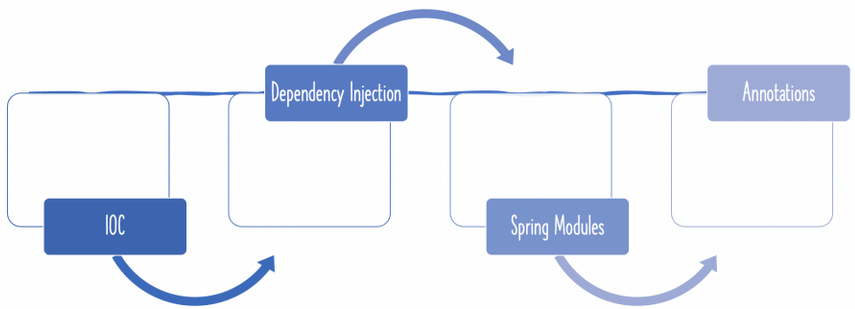
Decision Factors

- Business criticality
- Budget constraints
- Maintenance requirements
- Time to market
- Technical risk tolerance
- 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
- Enable testing
- Flexible implementations
- Centralized configuration

Key Concepts

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

Benefits

- Loose coupling
- Testability with mocks
- Runtime configuration
- 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

.NET

ASP.NET Core DI

Autofac

JavaScript

InversifyJS

TSyringe

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

Architecture

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

Implementation

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

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

Tech Stack

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

Event Flow

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

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

Mercedes-Benz

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

Comparison Metrics

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

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

2. **DDD + CQRS Extensive Use**
   BMW uses in every project

3. **Hexagonal / Service Mesh**
   Decoupling through adapters

4. **Cloud-Native Architectures**
   AWS, Azure, Kubernetes adoption

5. **Domain Separation Critical**
   Bounded contexts & aggregates

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
- CQRS Pattern
- Domain Events
- 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

Implementation Roadmap

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

CQRS Deep Dive: Advanced Patterns

Command Side Patterns

- **Command Handlers**
  Handle business operations
- **Validation Layer**
  Ensure data integrity
- **Domain Events**
  Publish state changes
- **Eventual Consistency**
  Async propagation to read model

Query Side Patterns

- **Read Model Optimization**
  Denormalized for fast queries
- **CQRS Read Handlers**
  Simple query logic
- **Caching Strategies**
  Redis, Memcached layers
- **Materialized Views**
  Pre-computed result sets

Synchronization Strategies

- **Outbox Pattern**
  Reliable event publishing
- **Change Data Capture (CDC)**
  Database transaction logs
- **Message Brokers**
  Kafka, RabbitMQ, Service Bus

Event Sourcing + CQRS

- **Event Store**
  Append-only log of events
- **Event Replay**
  Rebuild state from history
- **Temporal Queries**
  Query any point in time
- **Audit Trail**
  Complete event history

⬢ Hexagonal Architecture: Expert Implementation

Inbound Adapters

REST Controllers
HTTP endpoints

GraphQL Resolvers
Queries & mutations

Event Listeners
Kafka, RabbitMQ

CLI Commands
Batch jobs

Inbound Ports

IUseCase Interfaces
Define use case contracts

Command Handlers
Process domain commands

Query Handlers
Handle read requests

Outbound Ports

IRepository Interfaces
Data access contracts

IExternalService
Third-party APIs

IEventPublisher
Event broadcasting

Outbound Adapters

Database Implementations
EF Core, Dapper, MongoDB

External API Clients
HTTP, gRPC, WebSocket

Message Publishers
Kafka, RabbitMQ, Service Bus

Expert Notes

1. One port per use case
2. Adapters implement ports
3. DI resolves all dependencies

Event Sourcing vs CQRS: Critical Differences

Event Sourcing

- **All state changes as events**
  Immutable event store
- **Complete audit trail**
  No data loss
- **Time travel queries**
  Query any point in time
- **Event replay capability**
  Rebuild system state

CQRS

- **Separate read/write models**
  Optimize for different workloads
- **Optimize for workloads**
  Scale reads and writes independently
- **Not required for ES**
  Can be used with traditional storage
- **Simpler implementation**
  Lower complexity than ES

Key Insight

1. CQRS can use ES or not
2. ES benefits from CQRS separation

Decision Framework

→ **Event Sourcing** for audit & temporal queries

→ **CQRS** for read/write imbalance

Complexity Warning

Combination is powerful but adds **significant complexity**

⚠ Architecture Anti-Patterns: What to Avoid

Hexagonal Anti-Patterns

1. Too many ports (confusion)
2. Over-engineering simple apps
3. Not using ports for all external deps
4. Mixing concerns in one adapter

CQRS Anti-Patterns

1. CQRS everywhere (premature optimization)
2. Tight coupling between read/write models
3. Ignoring eventual consistency
4. No synchronization strategy
5. Not handling conflicts

DDD Anti-Patterns

1. Anemic domain model
2. Universal language
3. Big aggregates
4. Direct database access

Consequences

- Increased complexity
- Tight coupling
- Maintenance nightmares
- Loss of agility

Prevention Strategies

1. Start simple, add complexity as needed
2. Use architecture decision frameworks
3. Measure before optimizing
4. Document architecture rationale
5. Review with team regularly
6. Test architectural decisions

Migrating to DDD: Practical Approach

Strangler Fig Pattern

Gradually extract **bounded contexts** from monolith while keeping both systems running

Migration Steps

1. Identify bounded contexts in monolith
2. Define new architecture boundaries
3. Implement **Anti-Corruption Layer**
4. Extract first bounded context
5. Continue extraction iteratively
6. Decommission old monolith

Migration Techniques

- **Feature flags**
- **Parallel run**
- **Blue-green deployment**
- **API Gateway routing**

Supporting Tools

- **Service mesh** - Zero-trust networking
- **API gateways** - Request routing
- **Load balancers** - Traffic distribution

Real-World Example

`BMW Unified Configurator` - Migrated legacy system to microservices using DDD + CQRS

Day 4 Summary: Key Takeaways

Day 4 Learning Goals: ✓ All Achieved

- Design port-adapter architecture
- Distinguish DDD from BDD
- Identify DDD patterns
- Apply architecture patterns

1 Hexagonal Architecture

Ports & Adapters pattern

2 CQRS Pattern

Command-Query separation

3 DDD vs BDD

Key differences identified

4 DDD Tactical Patterns

Aggregates, Repositories, Services

5 Architecture Comparisons

Layered, Hexagonal, Onion, Clean

6 Dependency Injection

IoC containers and implementation

Industry Case Studies Covered

BMW
Unified Configurator

Mercedes-Benz
Connected Car Platform

Tesla
Autopilot System

Mini-Project

Connected Car Telematics System

Tools & Resources

Qlerify, DDD Toolbox, Context Mapper

Learning Goals Achievement Mapping

1 Design port-adapter architecture

- **Module 1:** Hexagonal Architecture
  Slides 3-8
- **Module 6:** Dependency Injection
  Slides 29-31

2 Distinguish DDD from BDD

- **Module 3:** DDD vs BDD
  Slides 18-19

3 Identify DDD patterns in applications

- **Module 4:** DDD Tactical Patterns
  Slides 20-24
- **Module 2:** CQRS Pattern
  Slides 9-17

4 Apply architecture patterns to different contexts

- **Module 5:** Architecture Comparisons
  Slides 25-28
- **Module 7:** Industry Case Studies
  Slides 32-35

All Learning Goals Achieved

Comprehensive coverage across 10 modules with 51 slides of dense, information-rich content

→ Ready for Day 5: Strategic Design

What to Expect

1. Domain **modeling in depth**
2. Bounded **context design**
3. Context **mapping**
4. Domain **events**
5. Integration **patterns**

Day 4 Recap Checklist

- Review **Hexagonal Architecture**
- Review **CQRS patterns**
- Review **DDD tactical patterns**
- Review **industry case studies**
- Review **mini-project requirements**

Self-Study Topics

1. Read `Martin Fowler's Bounded Context` article
2. Watch `Alberto Brandolini's` EventStorming videos
3. Practice `context mapping` with Qlerify
4. Read `"Domain-Driven Design Distilled"`

Expert Tips

- **Connect Day 4** patterns to Day 5 strategic design
- Practice **context mapping** before Day 5
- Review **case studies** for architectural insights
- Complete **mini-project** to apply learnings

Self-Study Topics: Deepen Your Knowledge

1 Advanced DDD

- **Bounded Context** mapping
- **EventStorming** workshops
- Context Mapping **patterns**
- Strategic Design **patterns**

2 CQRS Deep Dive

- **Event Sourcing** implementations
- Consistency **patterns**
- Scaling **CQRS** systems
- Testing **strategies**

3 Architecture Patterns

- **Saga** pattern
- **Outbox** pattern
- **CDC** (Change Data Capture)
- Event-driven **architecture**

4 Automotive Case Studies

- Study `BMW` Unified Configurator
- Study `Mercedes` Connected Car
- Study `Tesla` Autopilot

`Tip:` Practice context mapping with `Qlerify` before Day 5

Glossary: Day 4 Key Terms

Hexagonal Architecture

Ports and Adapters pattern, domain at center

CQRS

Command Query Responsibility Segregation, read/write separation

DDD

Domain-Driven Design, focus on domain complexity

BDD

Behavior Driven Development, specification by example

Aggregate

Cluster of domain objects treated as unit

Repository

Collection-like interface for data access

Factory

Object creation pattern, ensures valid state

Domain Service

Stateless operations not belonging to entity

Value Object

Immutable, defined by attributes

Entity

Has identity, lifecycle, mutable behavior

Bounded Context

Explicit boundary, own language, distinct model

References & Further Reading

Books

- **"Domain-Driven Design"** by Eric Evans
- **"Implementing DDD"** by Vaughn Vernon
- **"Building Microservices"** by Sam Newman

Articles

- Martin Fowler - **Bounded Context**
- InfoQ - **Architecture Decision Framework**
- Microsoft Learn - **Clean Architecture**

Tools

- `Qlerify` - AI-powered DDD modeling
- `DDD Toolbox` - Web application
- `Context Mapper` - DDD examples

Blogs & Resources

- Jimmy Bogard
- Udi Dahan
- Greg Young
- InfoQ Architecture Guide
- 5 Best Tools for DDD

`Tip:` Visit `dddcommunity.org` for more resources

Thank You & Next Steps

Key Takeaways

- Architecture patterns **require context**
- `DDD + CQRS + Hexagonal` powerful together
- Start simple, add **complexity as needed**
- Learn from **industry leaders**
- Practice with **mini-project**

Next Steps

1. Review all slides
2. Complete Day 4 recap
3. Start mini-project
4. Prepare for Day 5
5. Join DDD community

Contact & Community

iSAQB-DDD Training | DDD Community | Qlerify Tool Team

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

DDD Patterns

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

Concepts

- Port
- Adapter
- Invariant
- Ubiquitous Language
- Subdomain
- 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)