

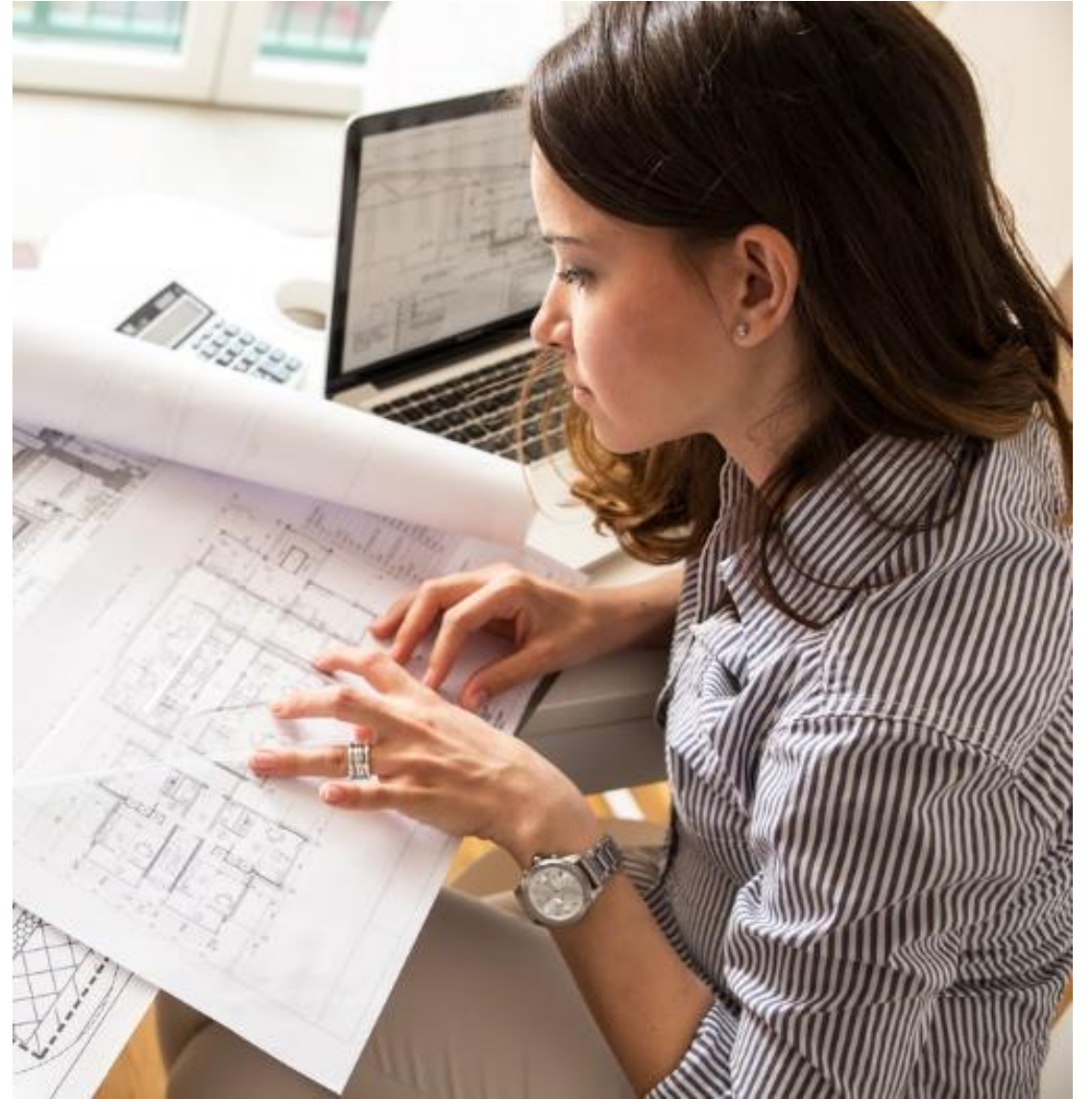
Clean Architecture

Patterns, Practices, and Principles

Matthew Renze

@matthewrenze

#iadnug













About Me

- Independent software consultant
- Education
 - B.S. in Computer Science (ISU)
 - B.A. in Philosophy (ISU)
- Community
 - Pluralsight Author
 - ASP Insider
 - Public Speaker
 - Open-Source Software

IOWA STATE
UNIVERSITY



Overview

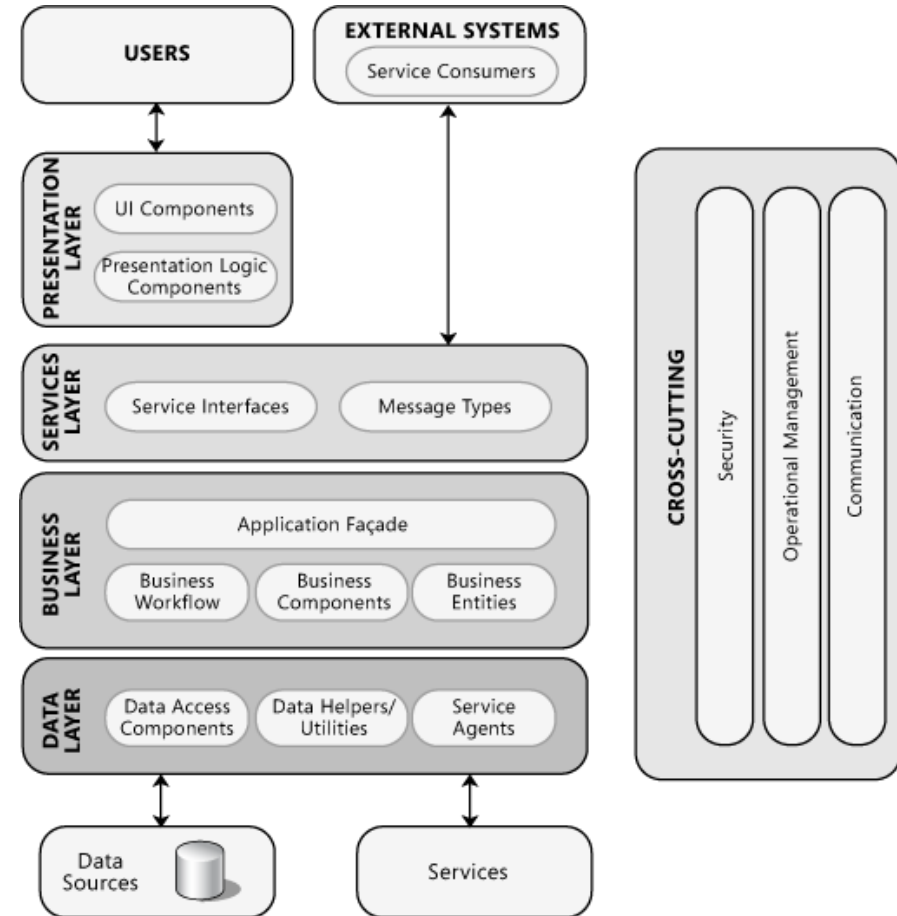
- Clean Architecture
- Domain-Centric Architecture
- Application Layer
- Commands and Queries
- Functional Organization
- Microservices

Focus

- Enterprise Architecture
- Line-of-Business Applications
- Modern equivalent of 3-Layer
- Generally applicable
- 6 Key Points
- Q & A

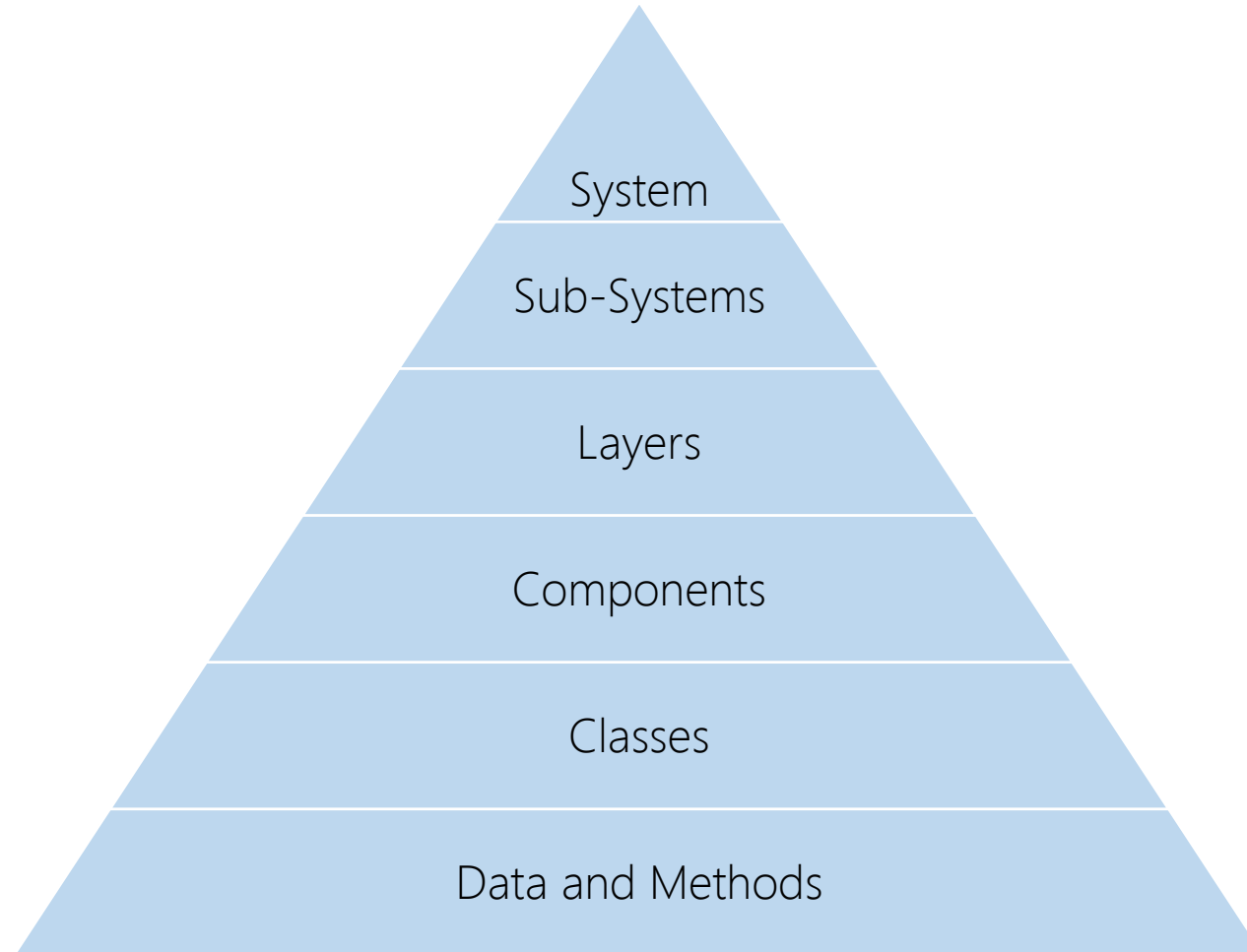
What is Software Architecture?

- High-level
- Structure
- Layers
- Components
- Relationships

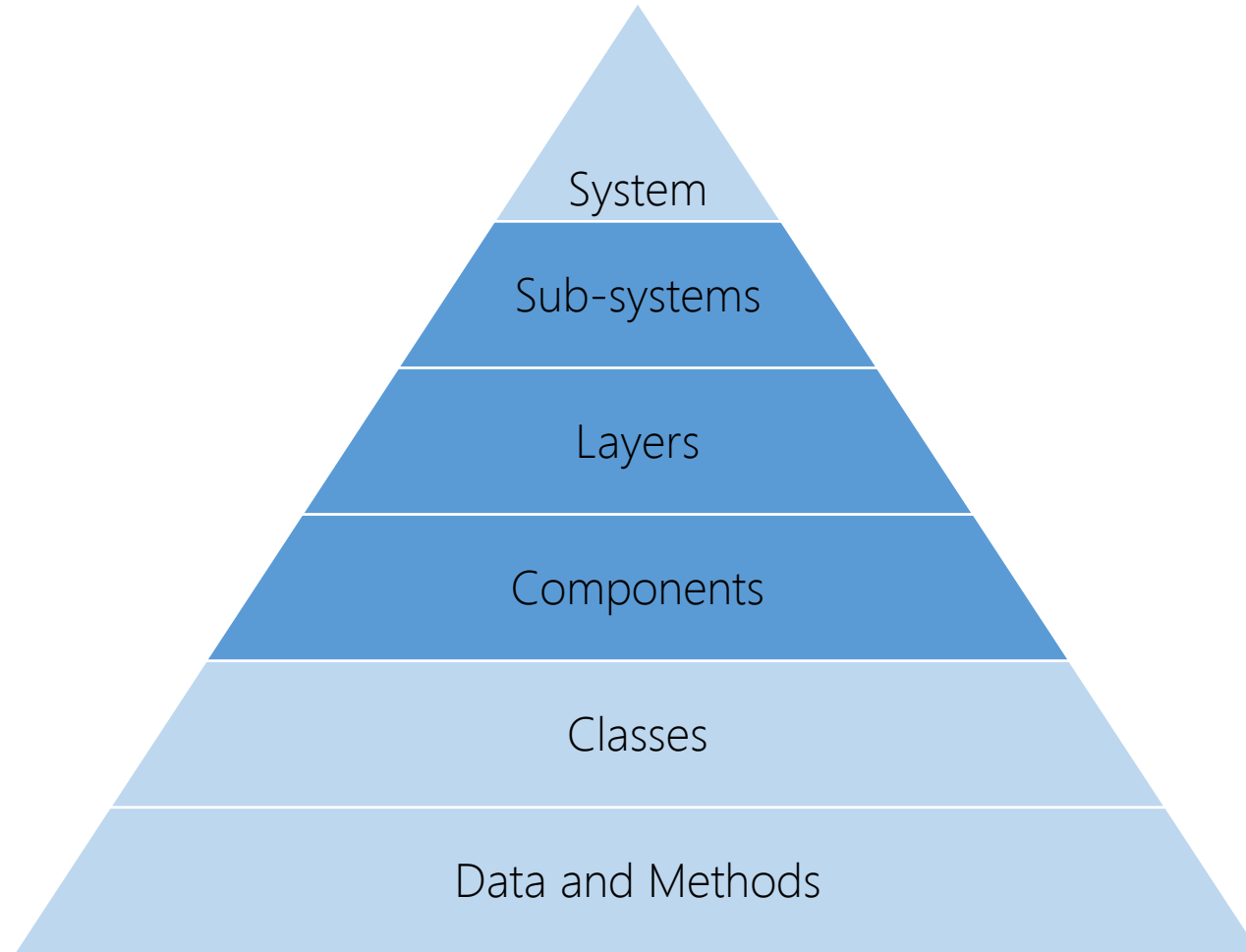


Source: <http://msdn.microsoft.com/en-us/library/ff650706.aspx>

Levels of Architectural Abstraction



Levels of Architectural Abstraction



Messy vs Clean Architecture

Messy vs Clean Architecture



Messy vs Clean Architecture



What is Bad Architecture?

- Complex
- Inconsistent
- Incoherent
- Ridged
- Brittle
- Untestable
- Unmaintainable



What is Clean Architecture?

- Simple
- Understandable
- Flexible
- Emergent
- Testable
- Maintainable



What is Clean Architecture?

Architecture that is designed
for the inhabitants of the
architecture, not for the
architect, or the machine

Why is Clean Architecture Important?

- Cost/benefit
- Minimize cost to maintain
- Maximize business value

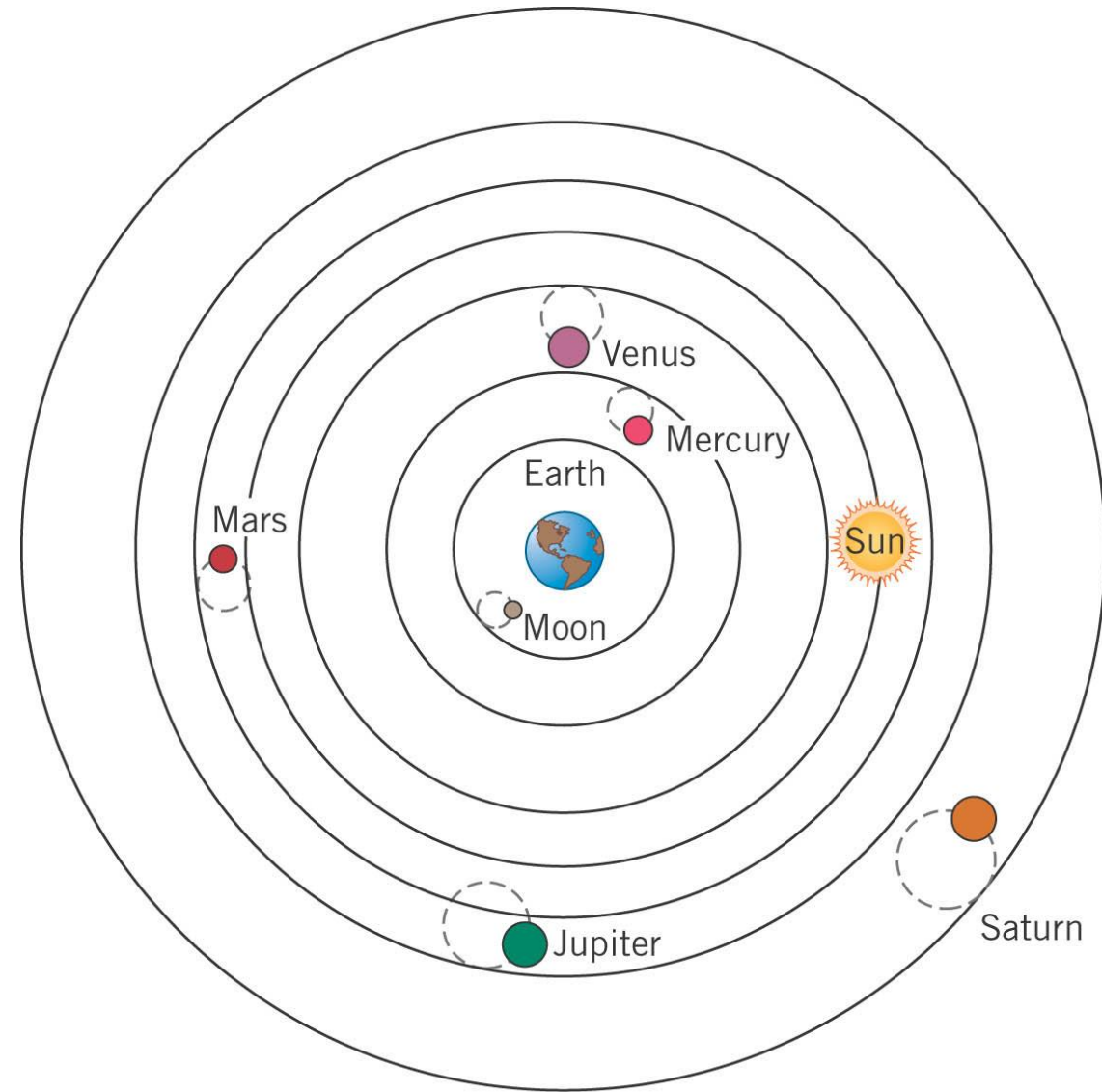
How Do We Create Clean Architecture?

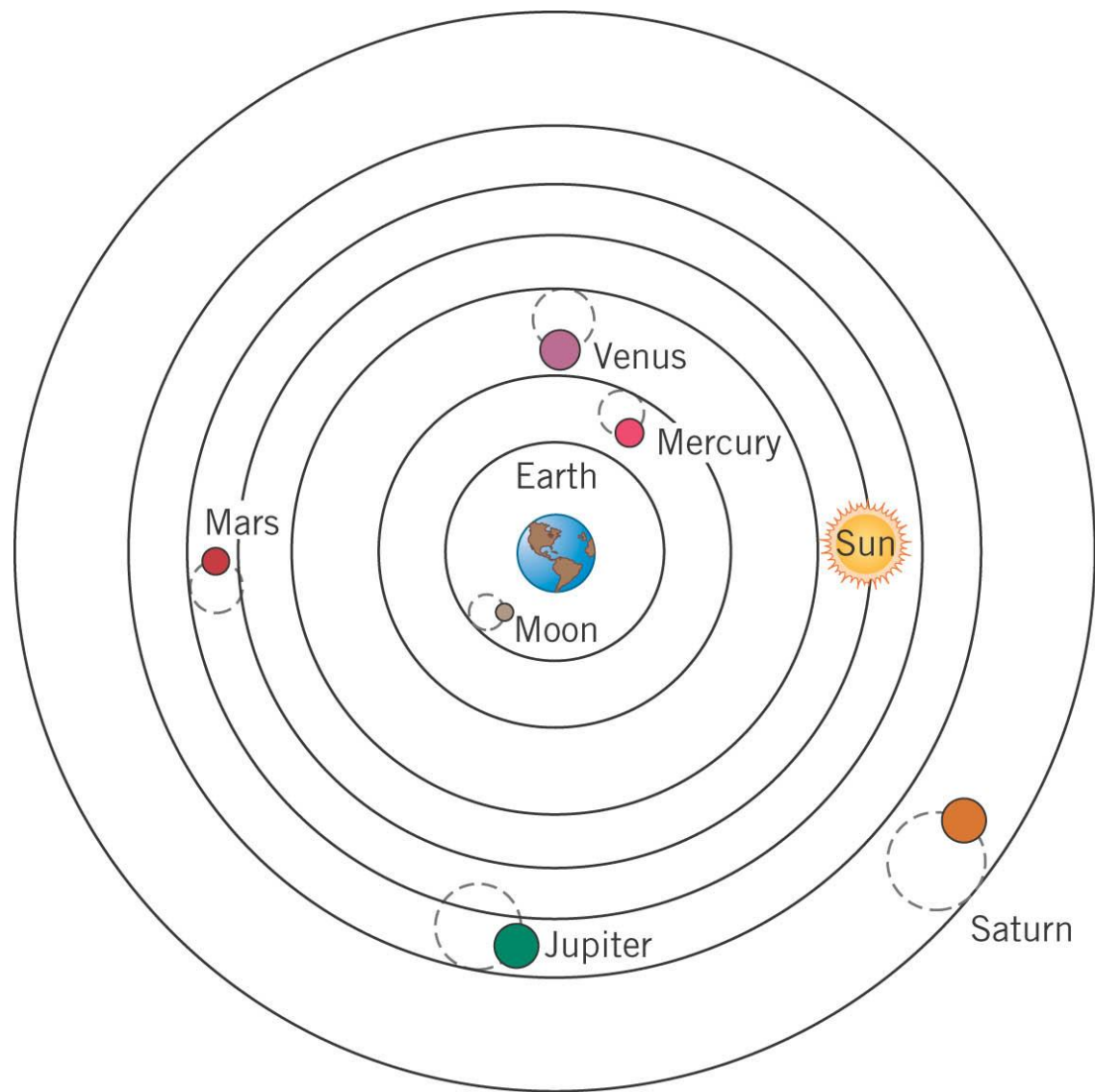
- Design for inhabitants first
- Focus on the domain
- Use an application layer
- Separate commands and queries
- Organize via functional cohesion
- Divide at bounded contexts

Decisions, Decisions, Decisions...

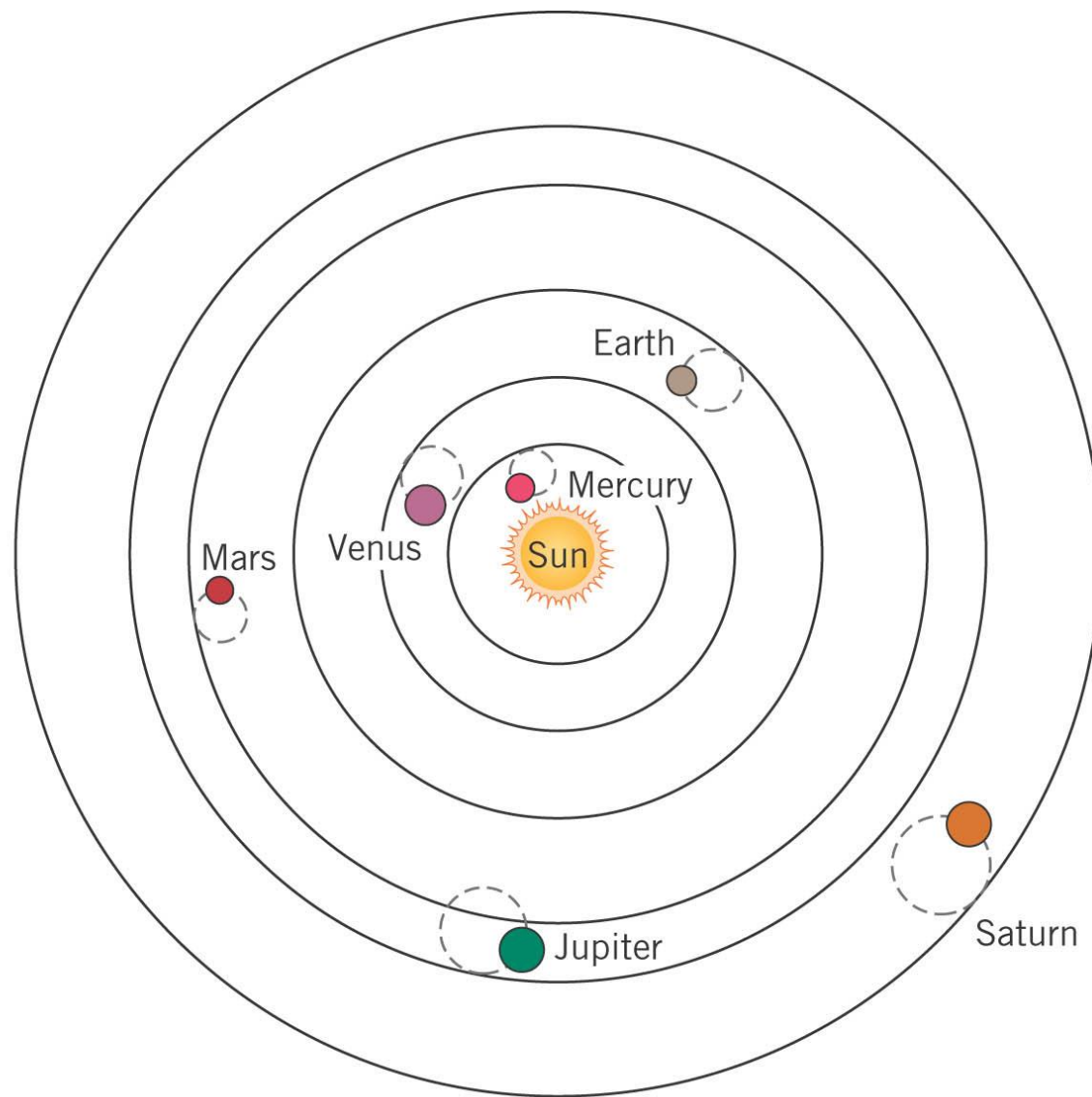
- Context is king
- All decisions are a tradeoff
- Use your best judgement

Domain-Centric Architecture



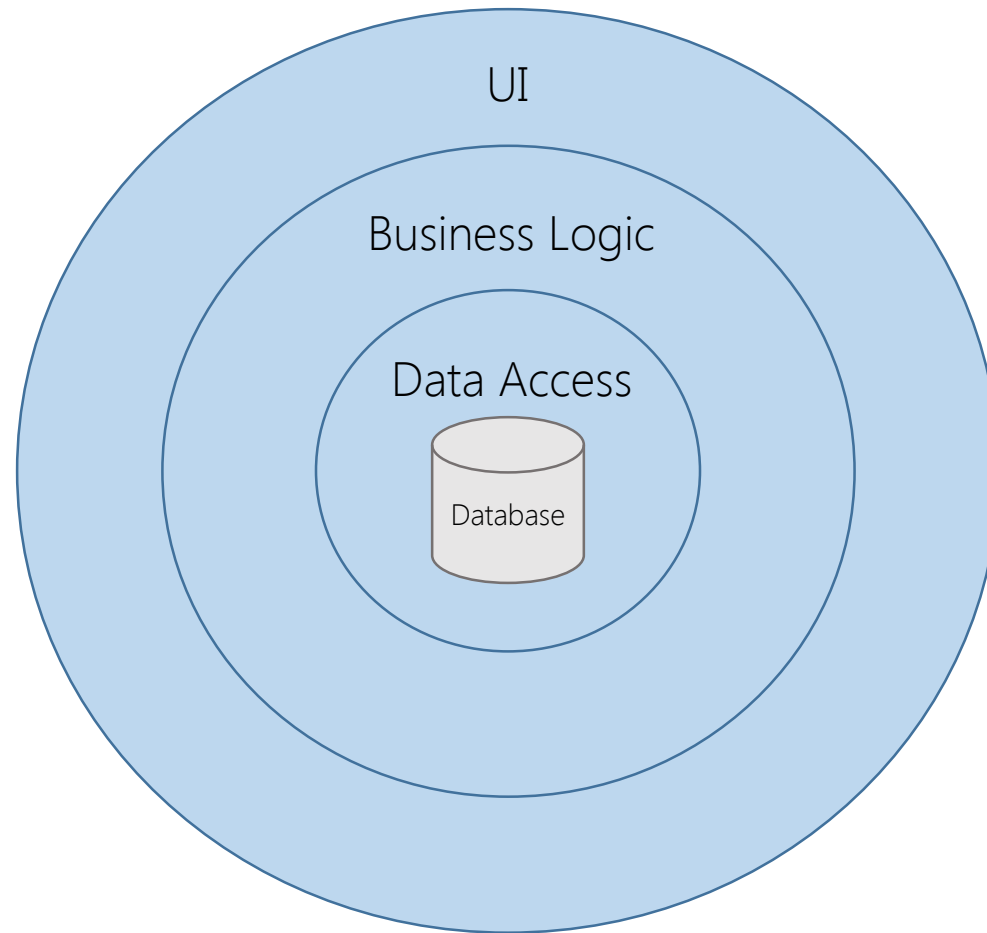


Geocentric Model

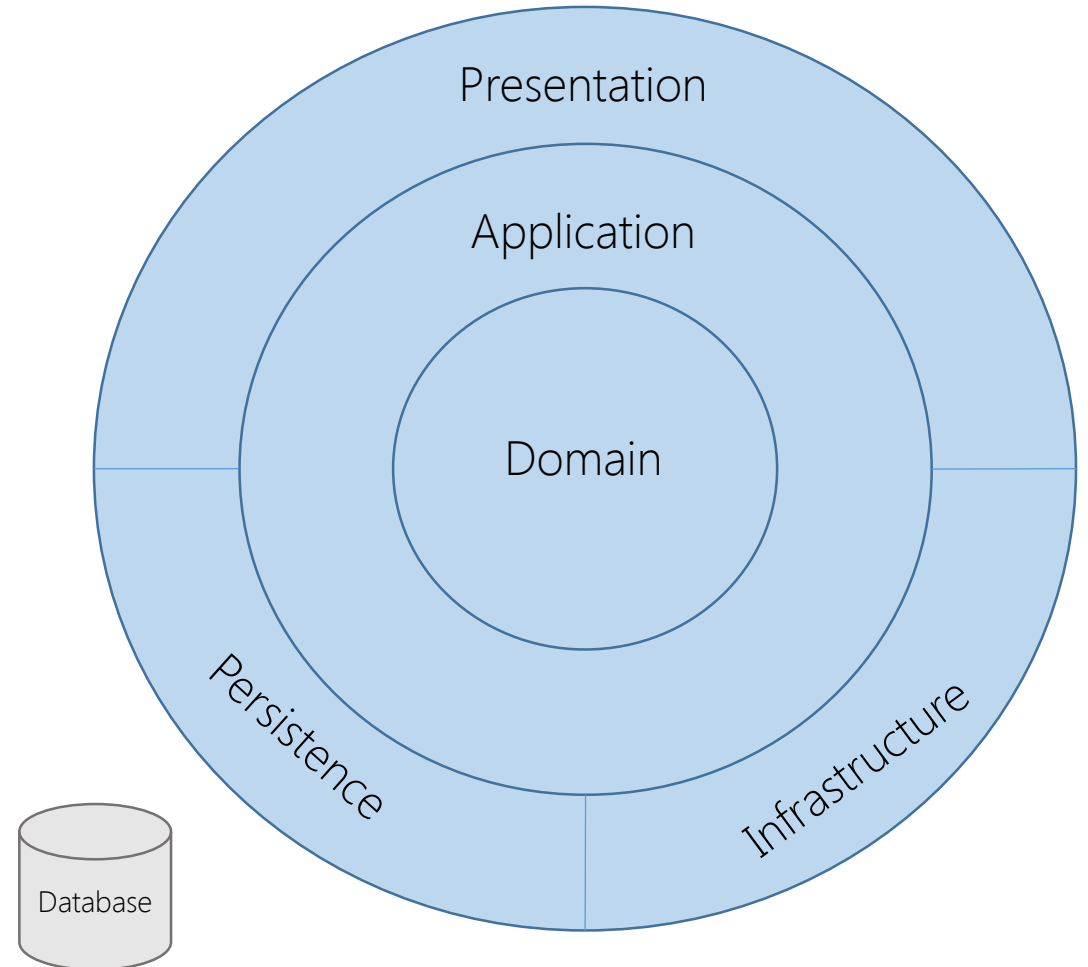
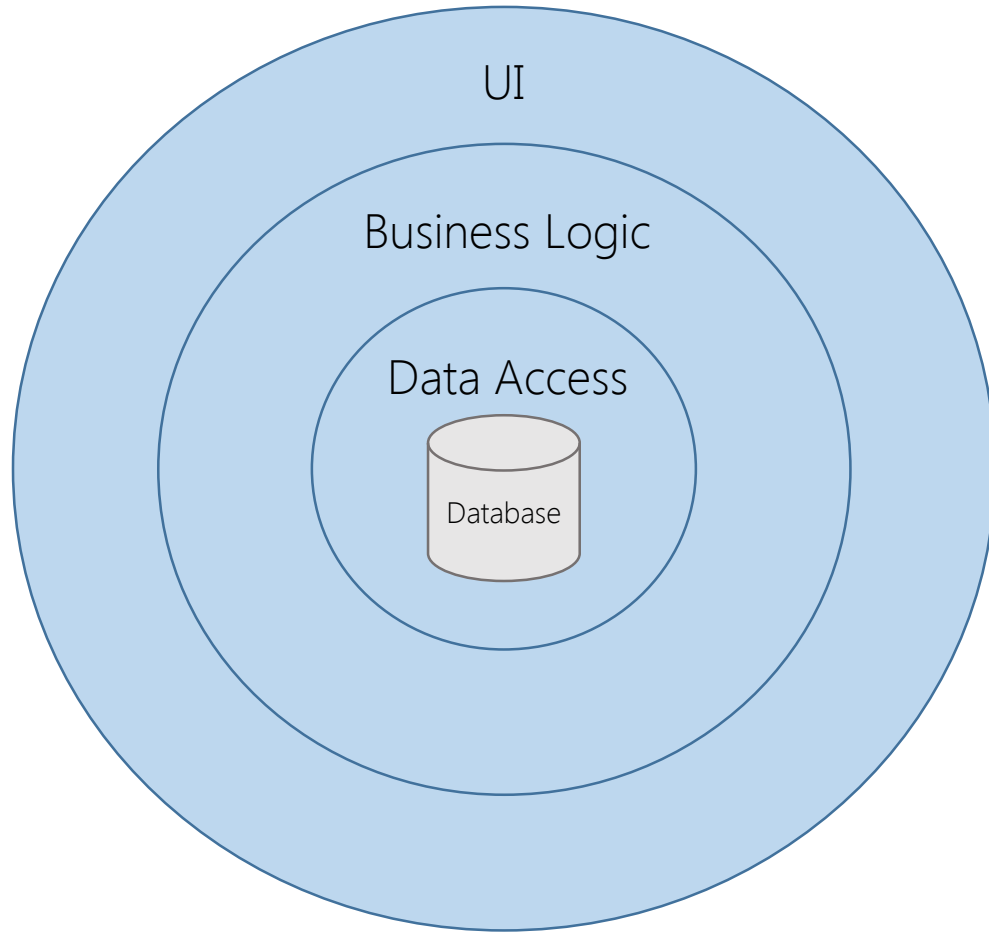


Heliocentric Model

Classic 3-Layer Database-centric Architecture



Database- vs. Domain-centric Architecture



“The first concern of the architect is to make sure that the house is usable, it is not to ensure that the house is made of brick.”

– Uncle Bob

Essential vs. Detail

- Space is essential
- Usability is essential
- Building material is a detail
- Ornamentation is a detail

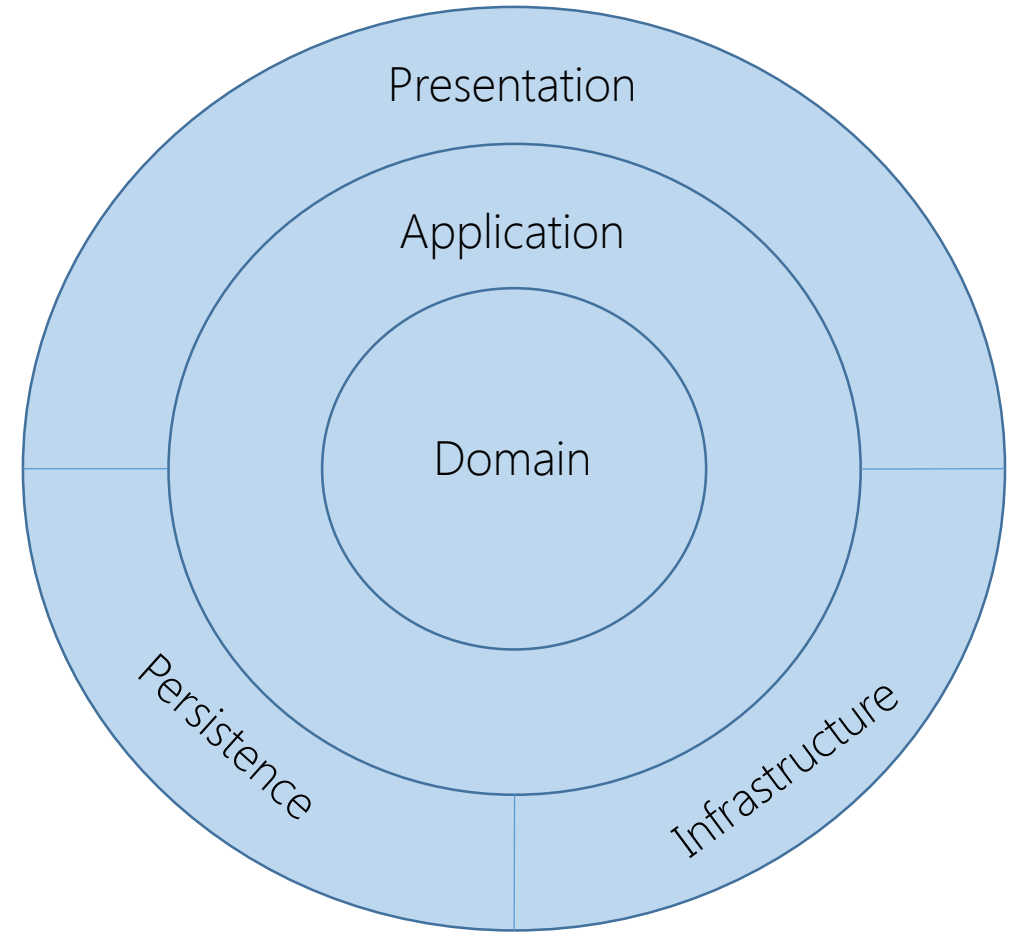
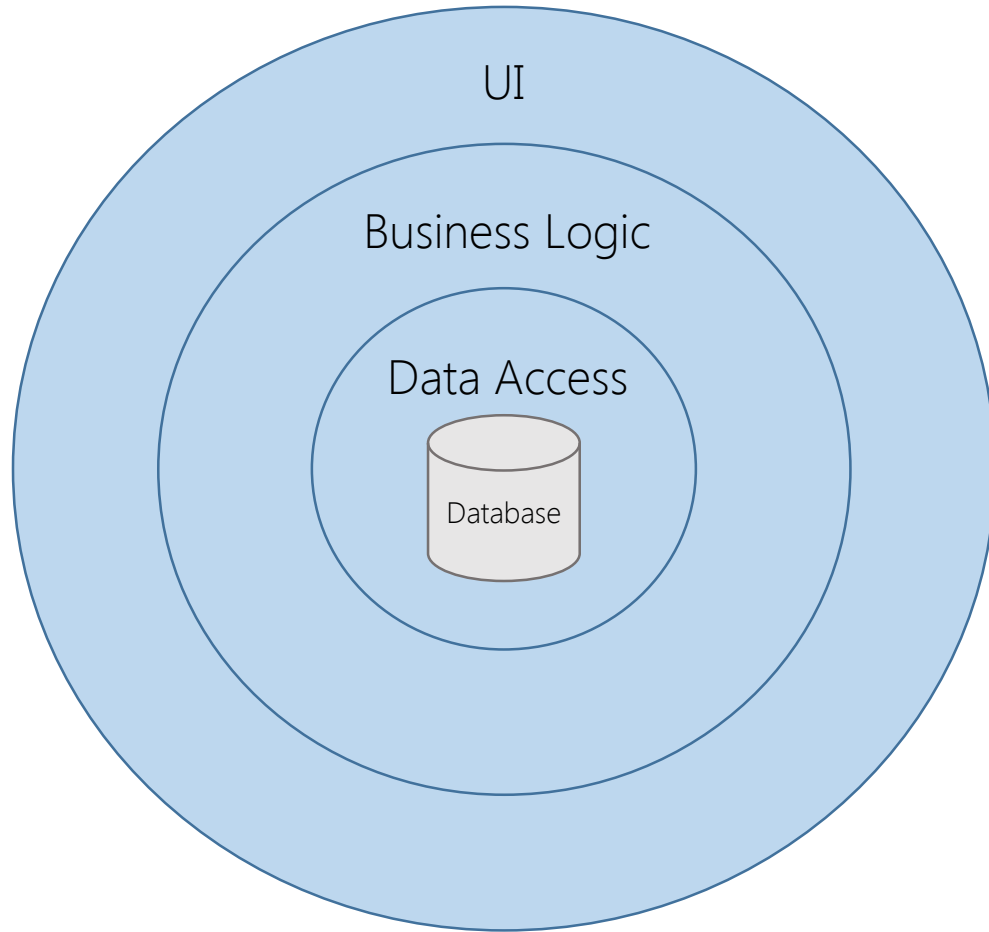


Source: <http://www.whitegadget.com/attachments/pc-wallpapers/85254d1320380902-house-house-wallpaper.jpg>

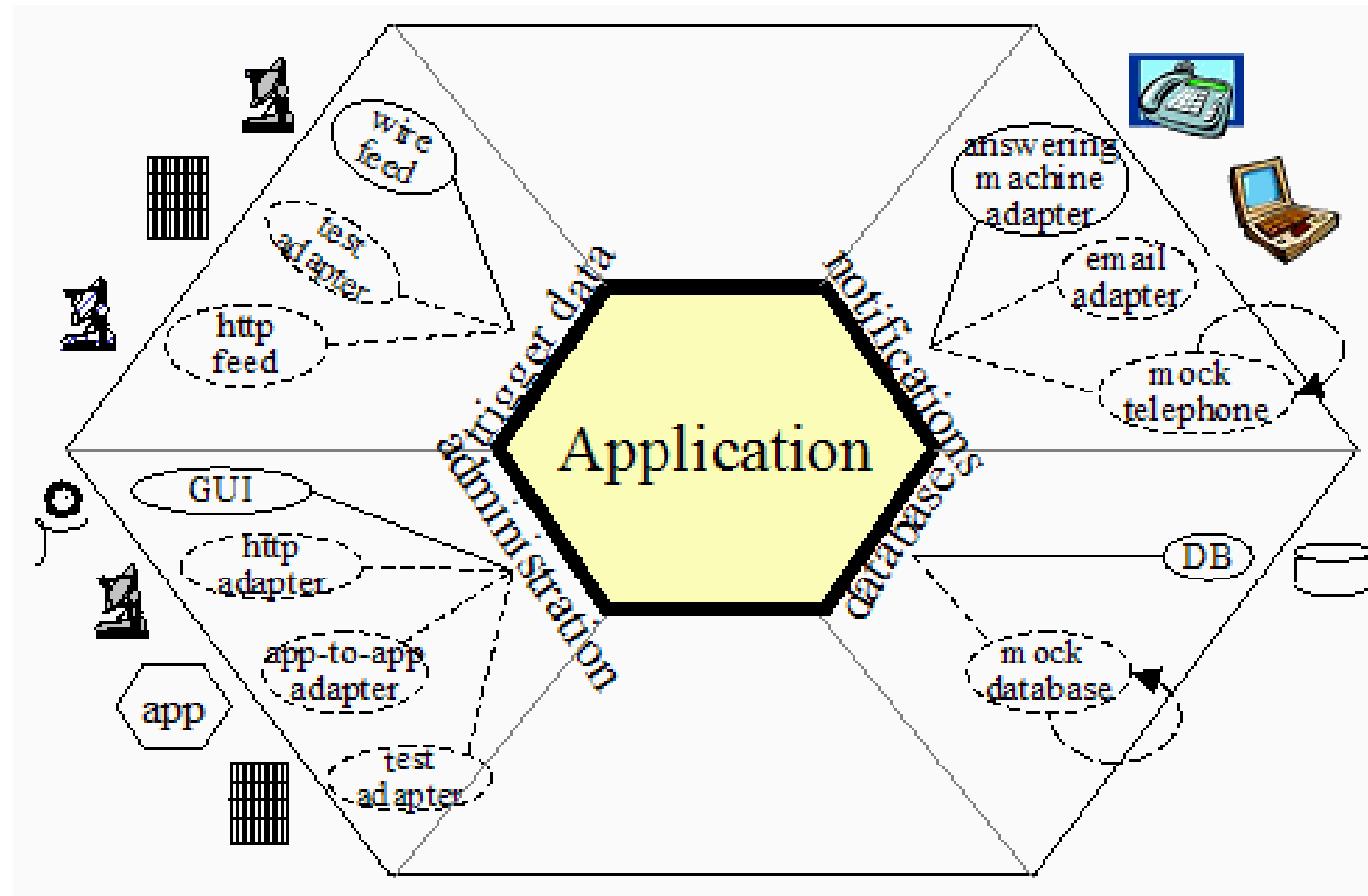
Essential vs. Detail

- Domain is essential
- Use cases are essential
- Presentation is a detail
- Persistence is a detail

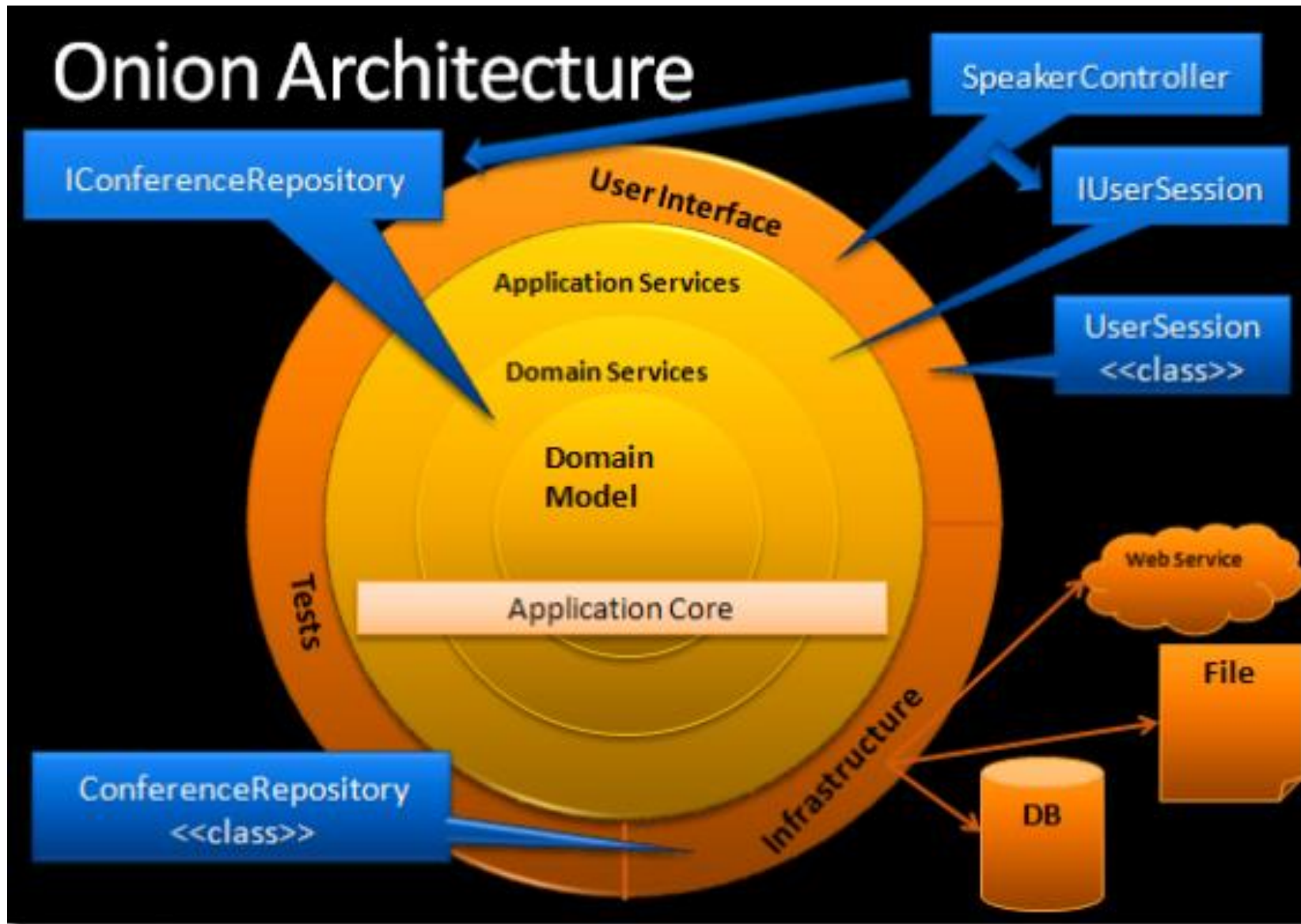
Database- vs. Domain-centric Architecture



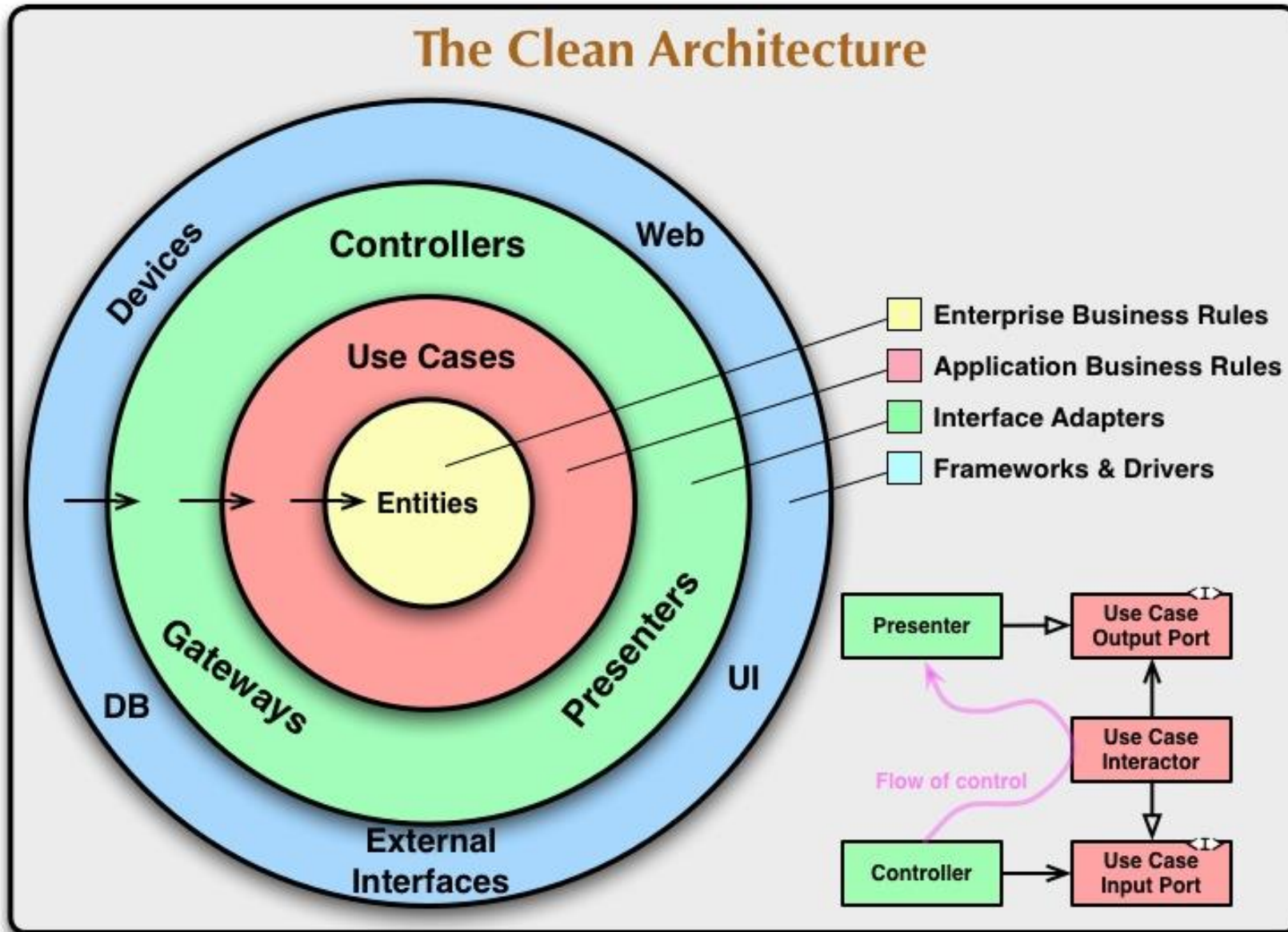
Hexagonal Architecture



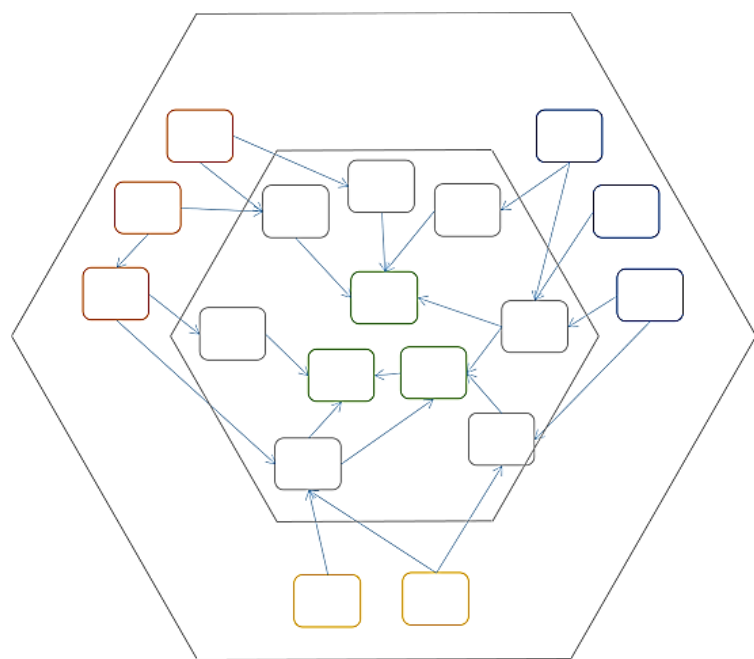
Source: <http://alistair.cockburn.us/Hexagonal+architecture>



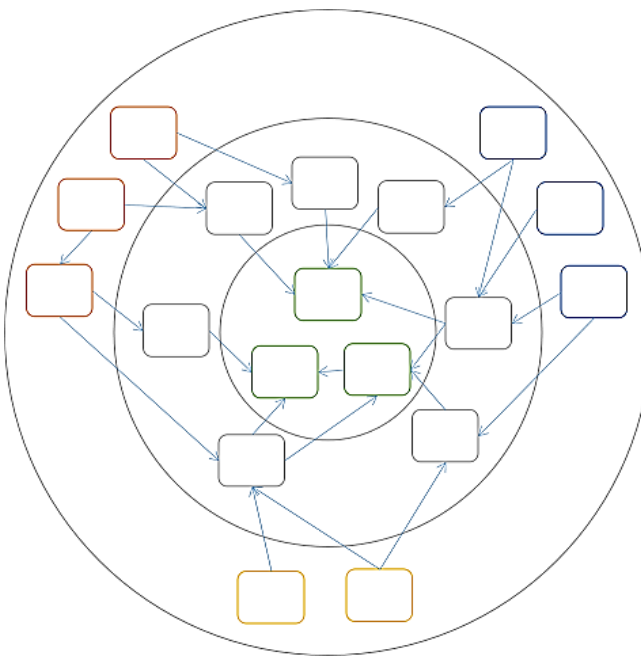
Source: <http://jeffreypalermo.com/blog/the-onion-architecture-part-2/>



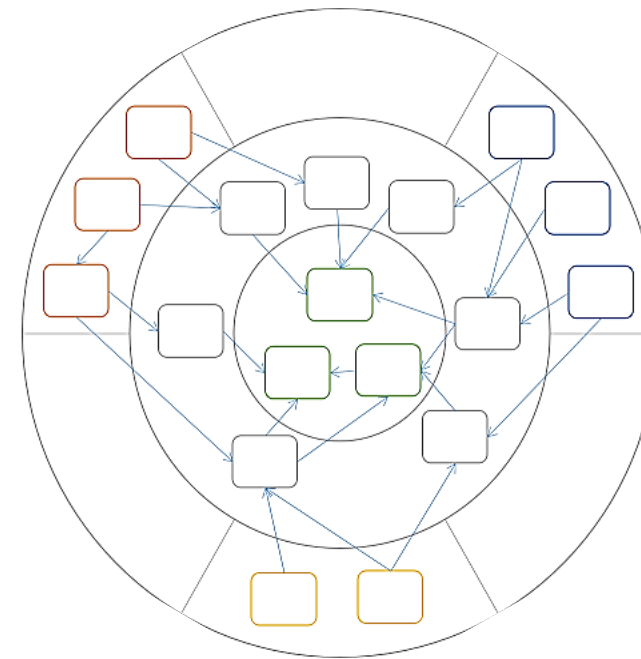
It's All the Same Thing



Hexagonal



Onion



Clean

Why Use Domain-Centric Architecture?

Pros

- Focus on essential
- Less coupling to details
- Necessary for DDD

Why Use Domain-Centric Architecture?

Pros

- Focus on essential
- Less coupling to details
- Necessary for DDD

Cons

- Change is difficult
- Requires extra thought
- Cost may not be justified

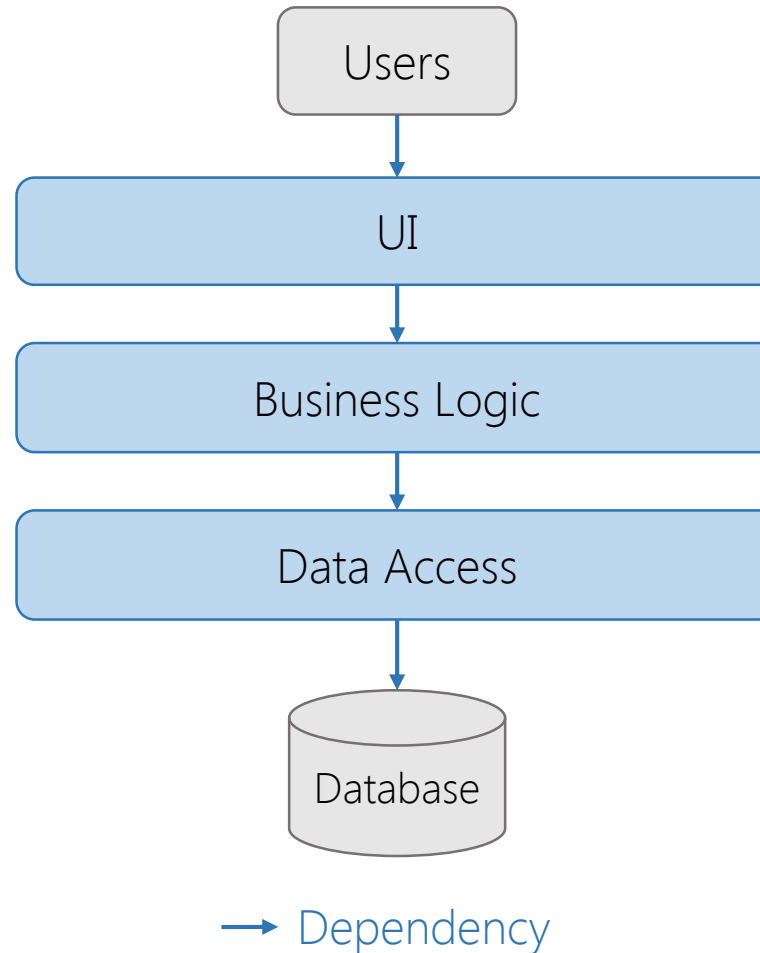
Application Layer

What are Layers?

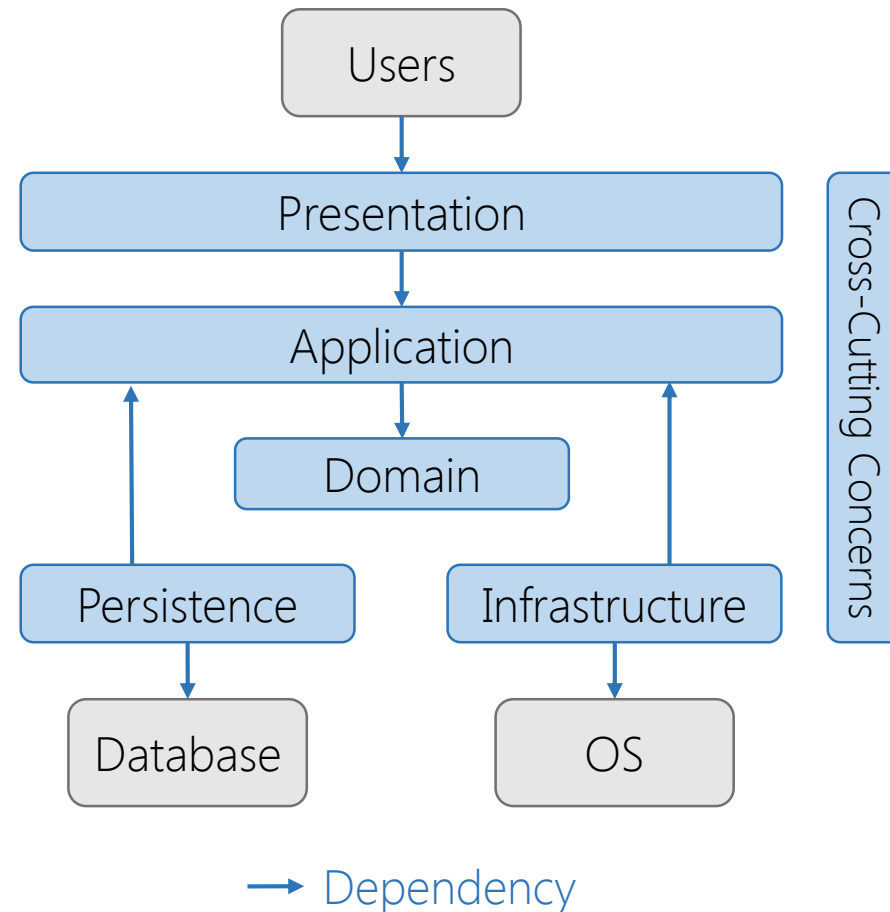
- Levels of abstraction
- Single-Responsibility Principle
- Developer roles / skills
- Multiple implementations
- Varying rates of change

Note: Layers != tiers

Classic 3-Layer Architecture

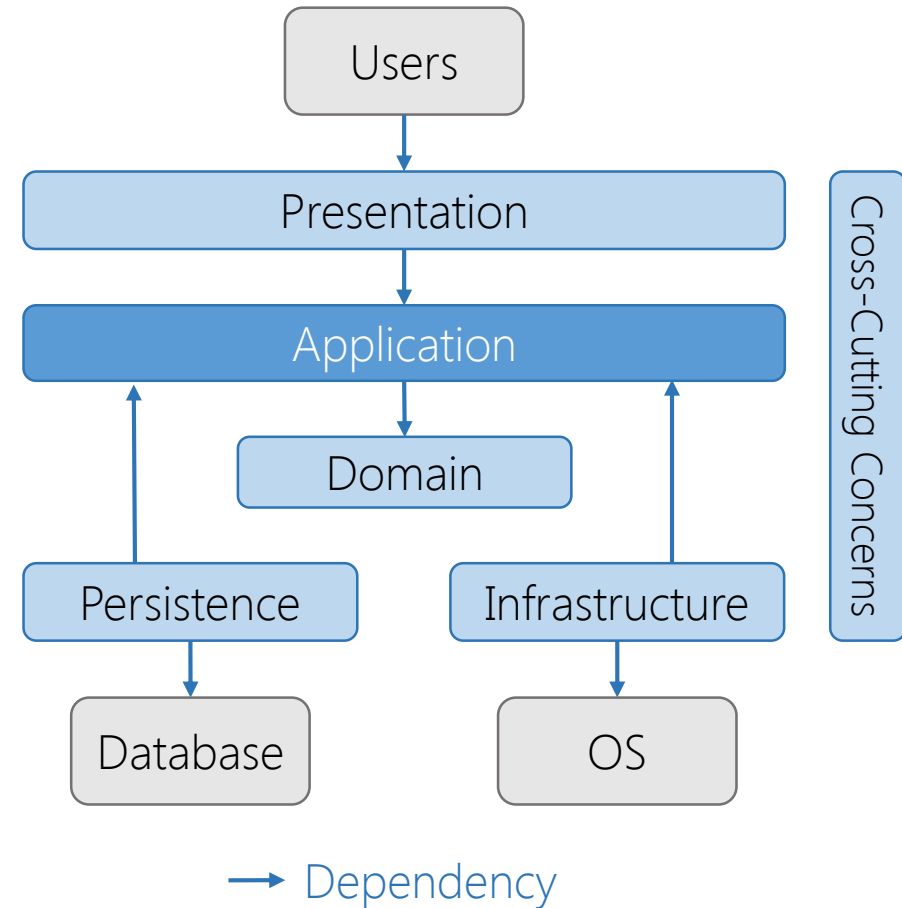


Modern 4-Layer Architecture



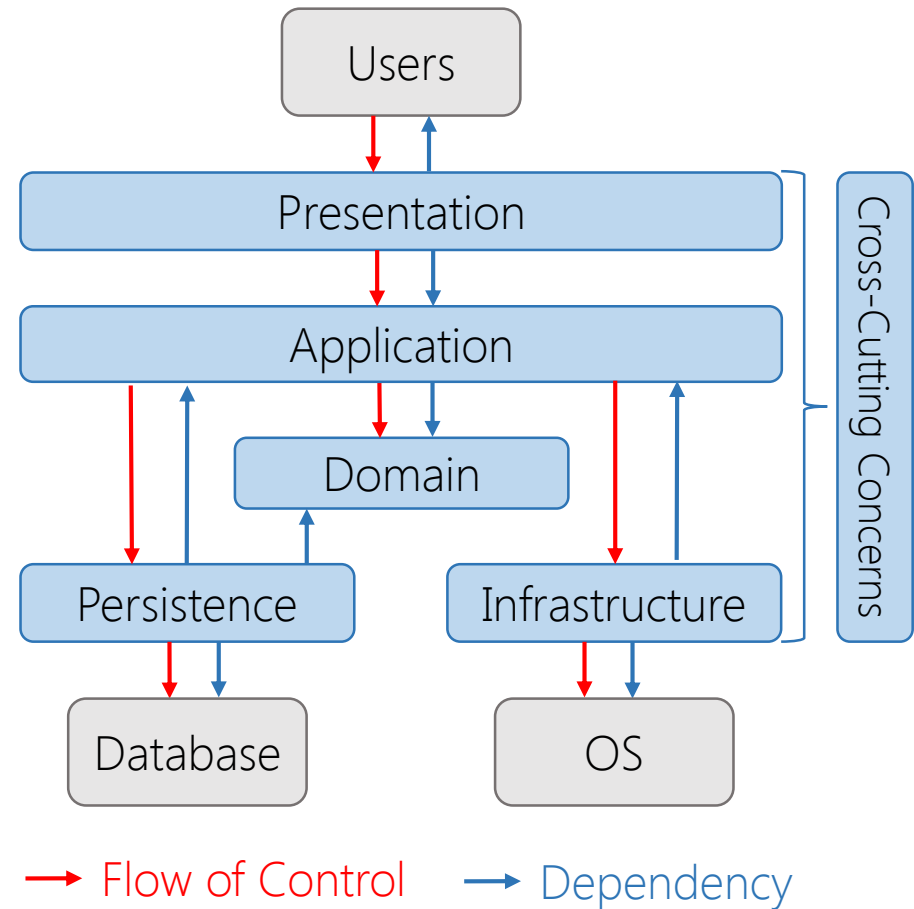
Application Layer

- Implements use cases
- High-level application logic
- Knows about lower layers
- No knowledge of upper layers
- Contains interfaces for details



Layer Dependencies

- Dependency inversion
- Inversion of control
- Independent deployability
- Flexibility and maintainability



Why Use an Application Layer?

Pros

- Focuses on use cases
- Very easy to understand
- Follows DIP

Why Use an Application Layer?

Pros

- Focuses on use cases
- Very easy to understand
- Follows DIP

Cons

- Additional cost
- Requires extra thought
- IoC is counter-intuitive

Commands and Queries

Command-Query Separation

Command

- Does something
- Modifies state
- Should not return a value

Command-Query Separation

Command

- Does something
- Modifies state
- Should not return a value

Query

- Answers a question
- Does not modify state
- Always returns a value

Command-Query Separation

Command

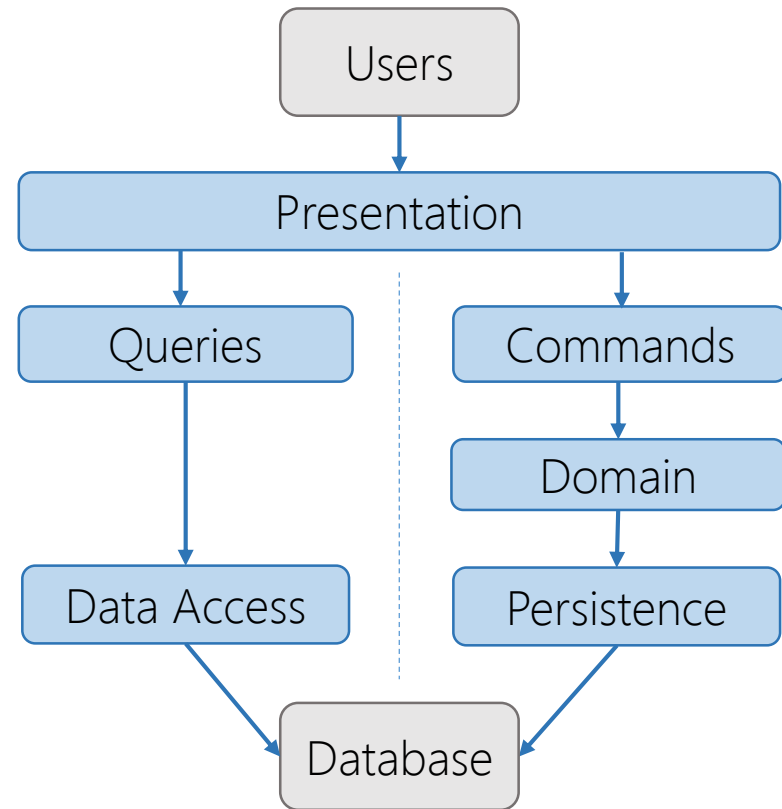
- Does something
- Modifies state
- Should not return a value
(ideally)

Query

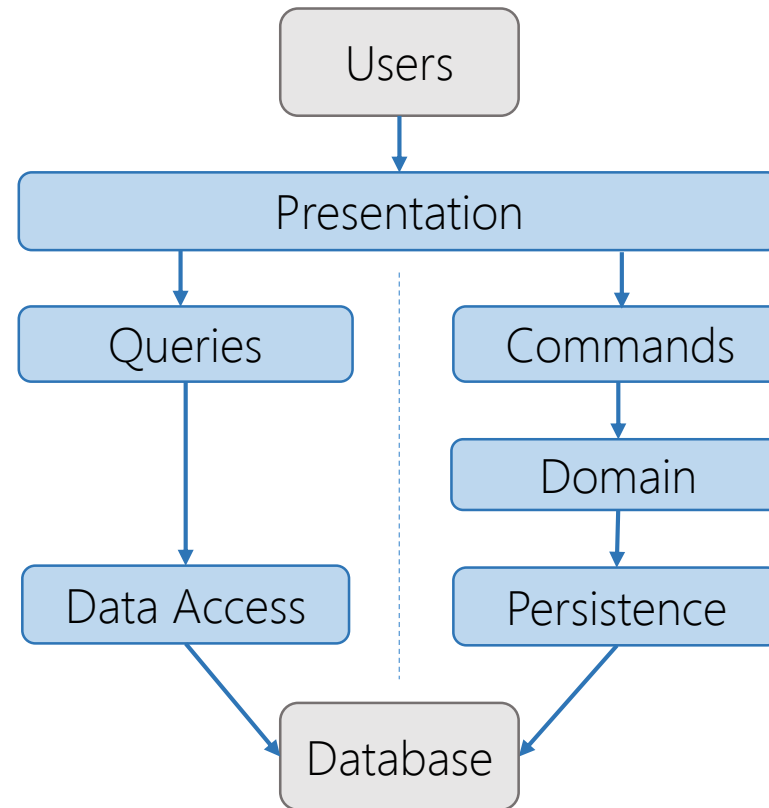
- Answers a question
- Does not modify state
- Always returns a value

Avoid mixing the two!

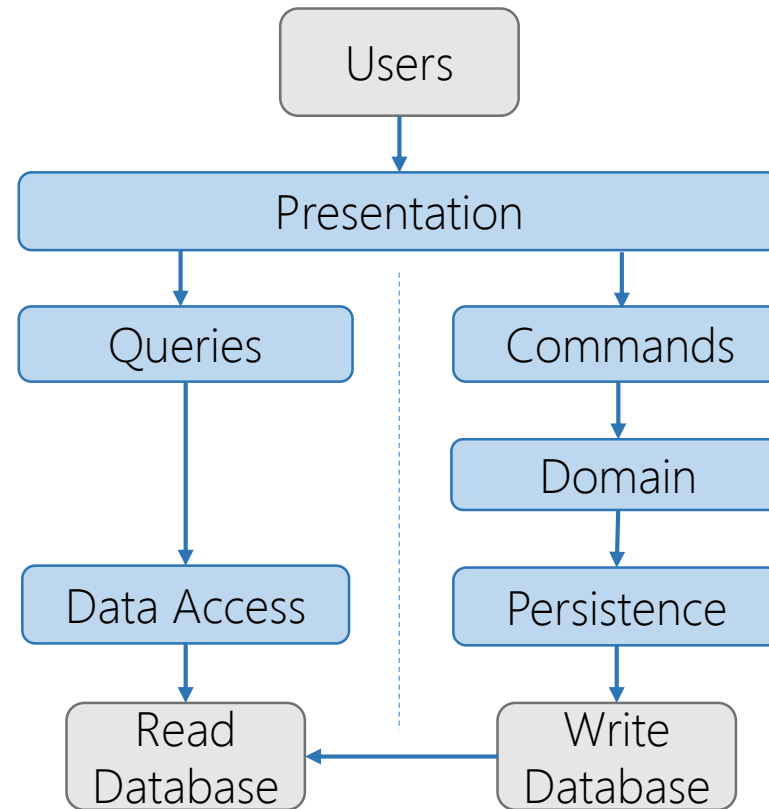
CQRS Architectures



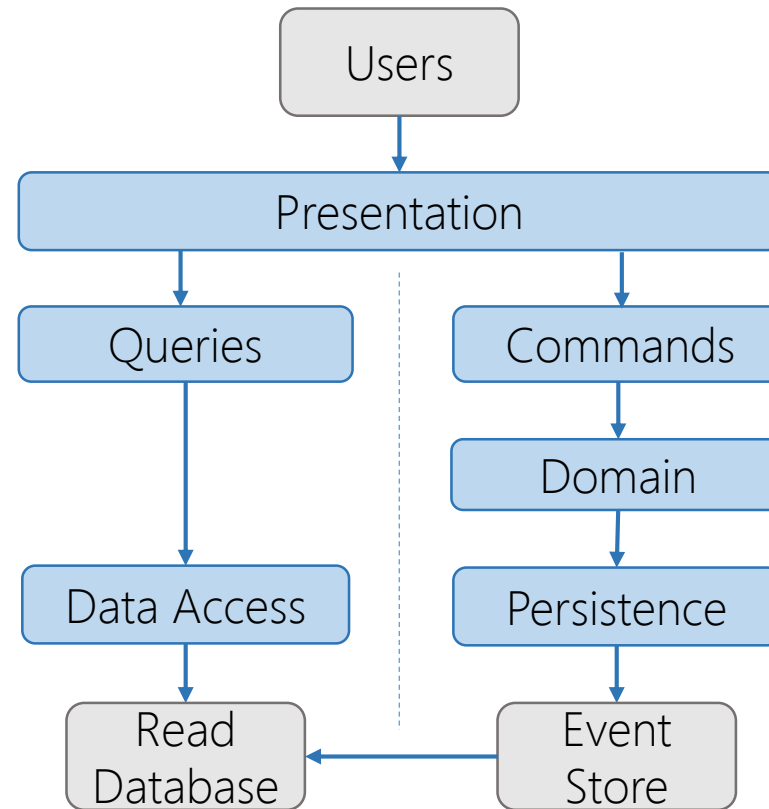
CQRS Type 1 – Single Database



CQRS Type 2 – Read/Write Databases

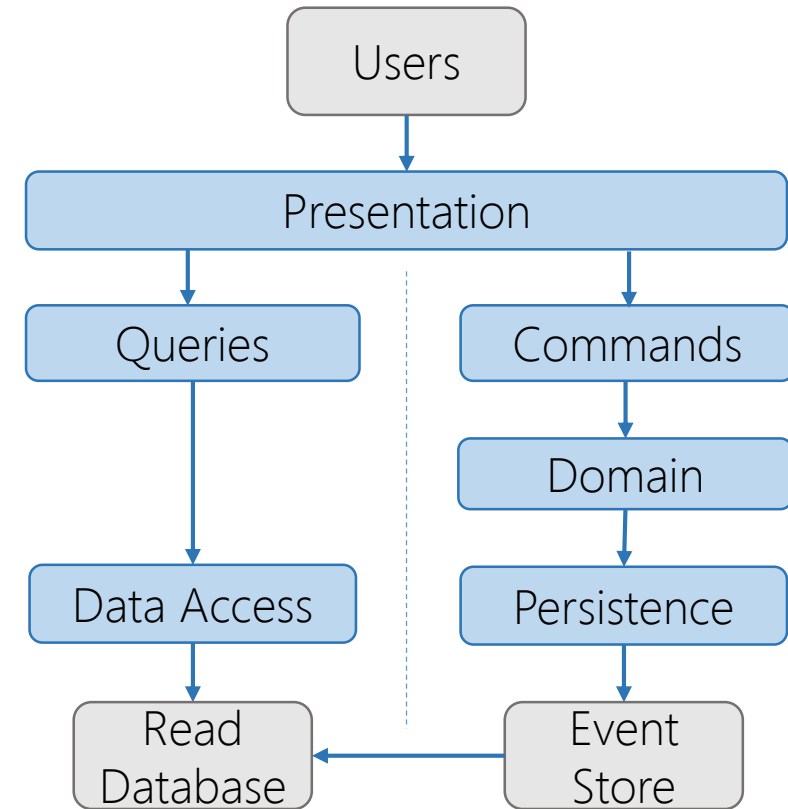


CQRS Type 3 – Event Sourcing



CQRS Type 3 – Event Sourcing

- Complete audit trail
- Point-in-time reconstruction
- Replay events
- Rebuild production database



Why Use CQRS?

Pros

- Efficient domain-centric design
- Simpler to understand
- Optimized performance

Why Use CQRS?

Pros

- Efficient domain-centric design
- Simpler to understand
- Optimized performance

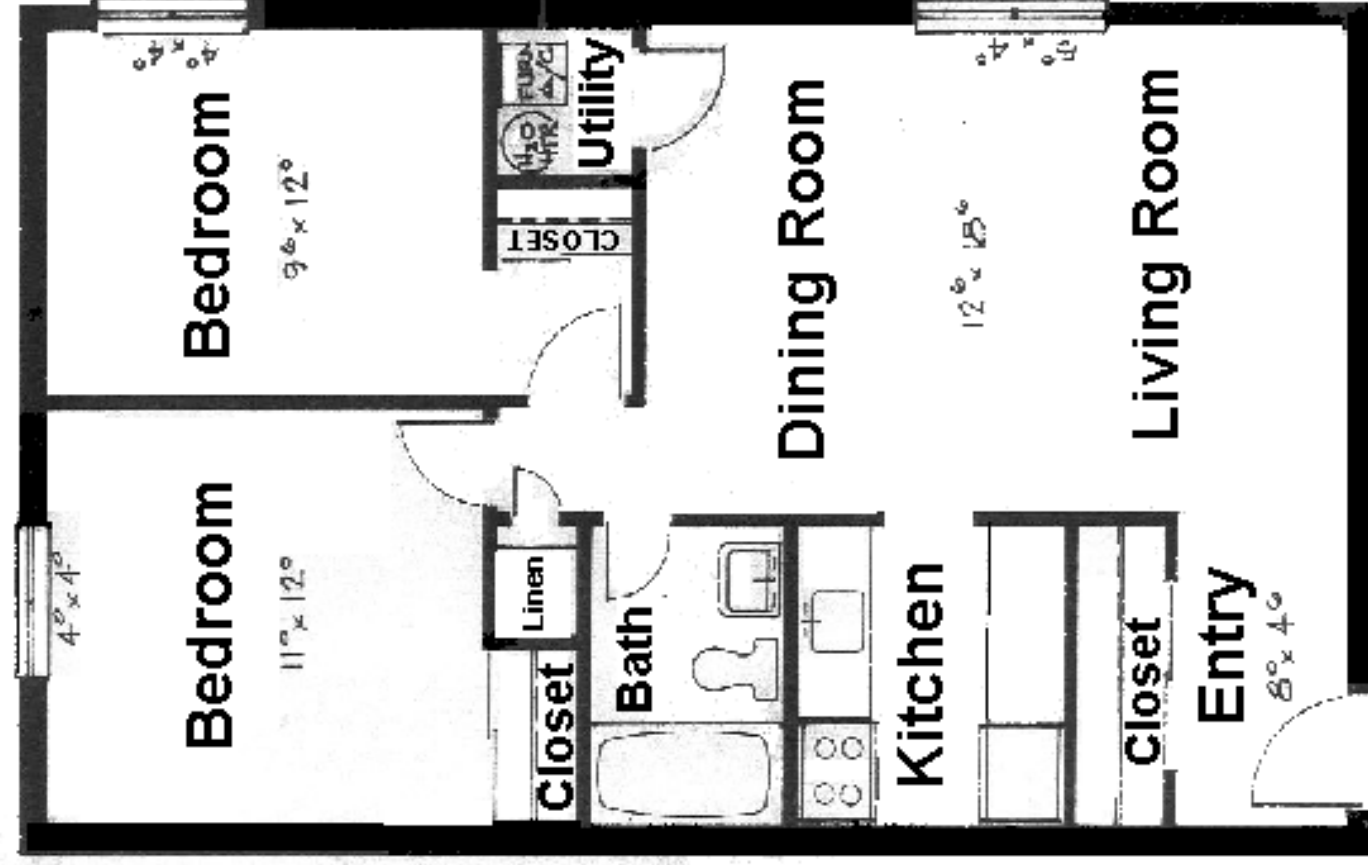
Cons

- Inconsistent design
- Type 2 is more complex
- Type 3 might be overkill

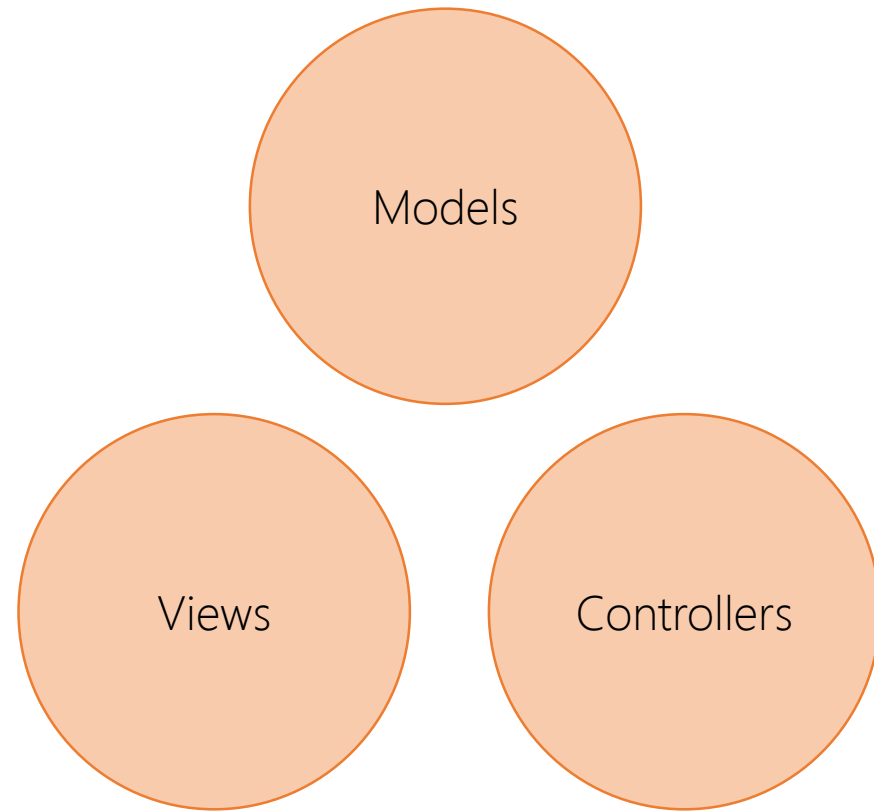
Functional Organization

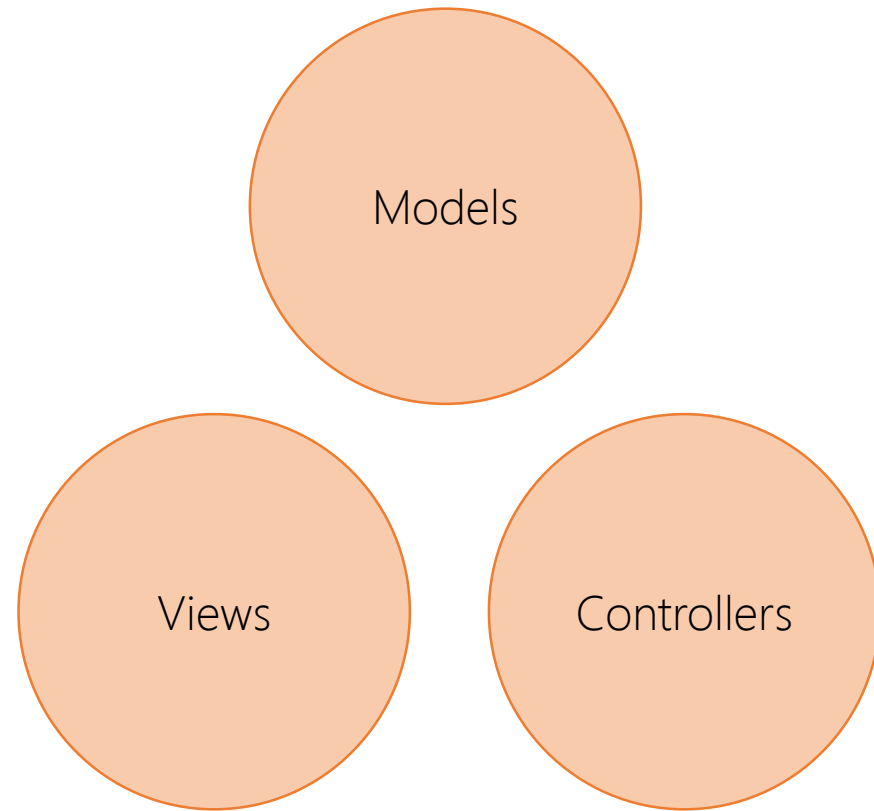
“The architecture should scream
the intent of the system!”

– Uncle Bob

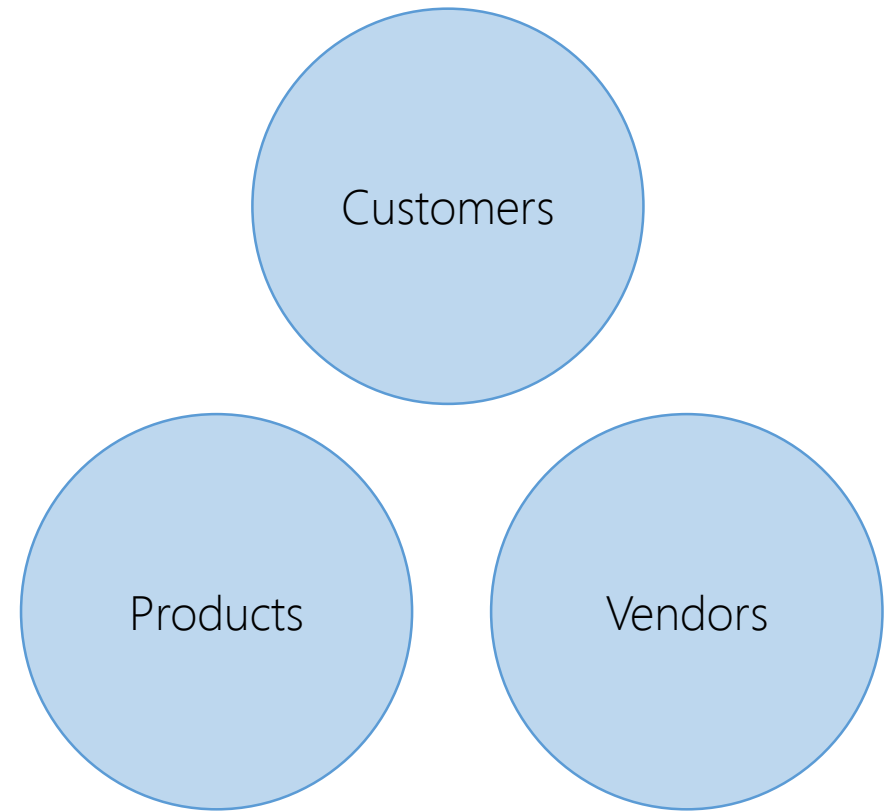


Material	Quantity	Cost
Appliances	5	\$5,000
Cabinets	10	\$2,500
Doors	15	\$750
Fixtures	12	\$2,400
Floors	9	\$4,000
Walls	20	\$10,000
Windows	8	\$2,500





VS





Content



Controllers



Models



Scripts



Views



Content



Controllers



Models



Scripts



Views

vs



Customers



Employees



Products



Sales



Vendors

So what?

Why Use Functional Organization

Pros

- Spatial locality
- Easy to navigate
- Avoid vendor lock-in

Why Use Functional Organization

Pros

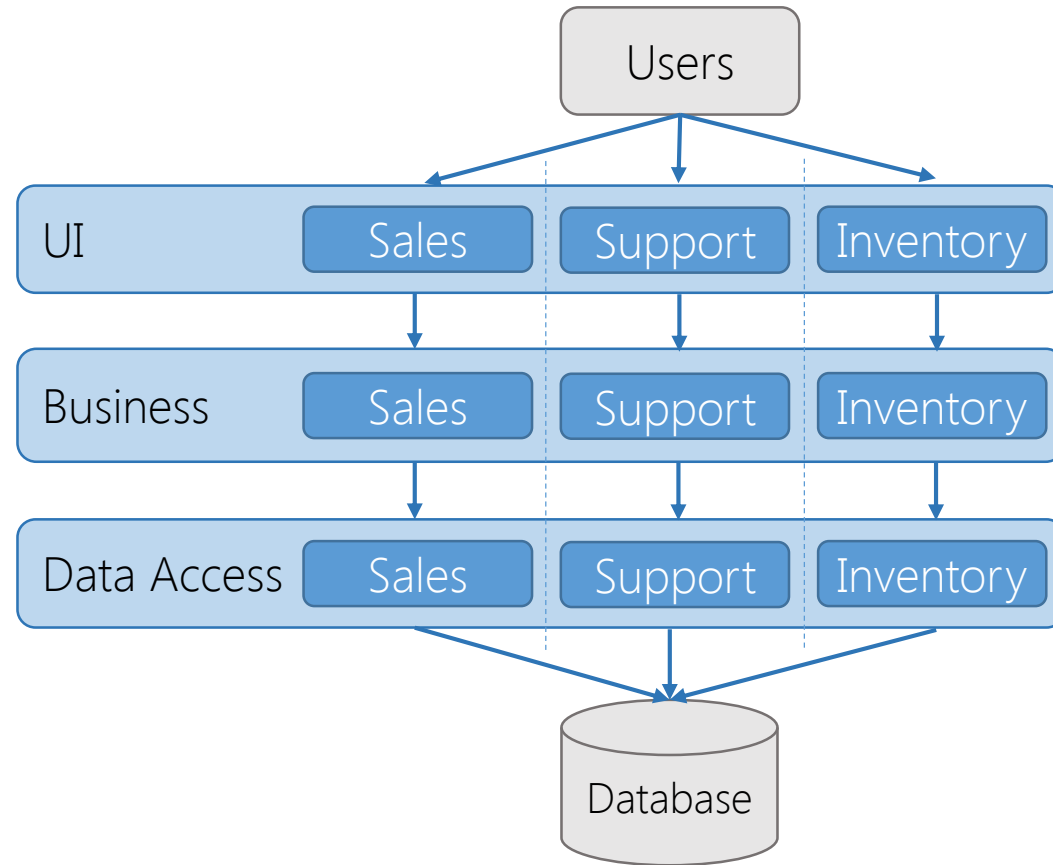
- Spatial locality
- Easy to navigate
- Avoid vendor lock-in

Cons

- Lose framework conventions
- Lose automatic scaffolding
- Categorical is easier at first

Microservices

Components



Problem Domain

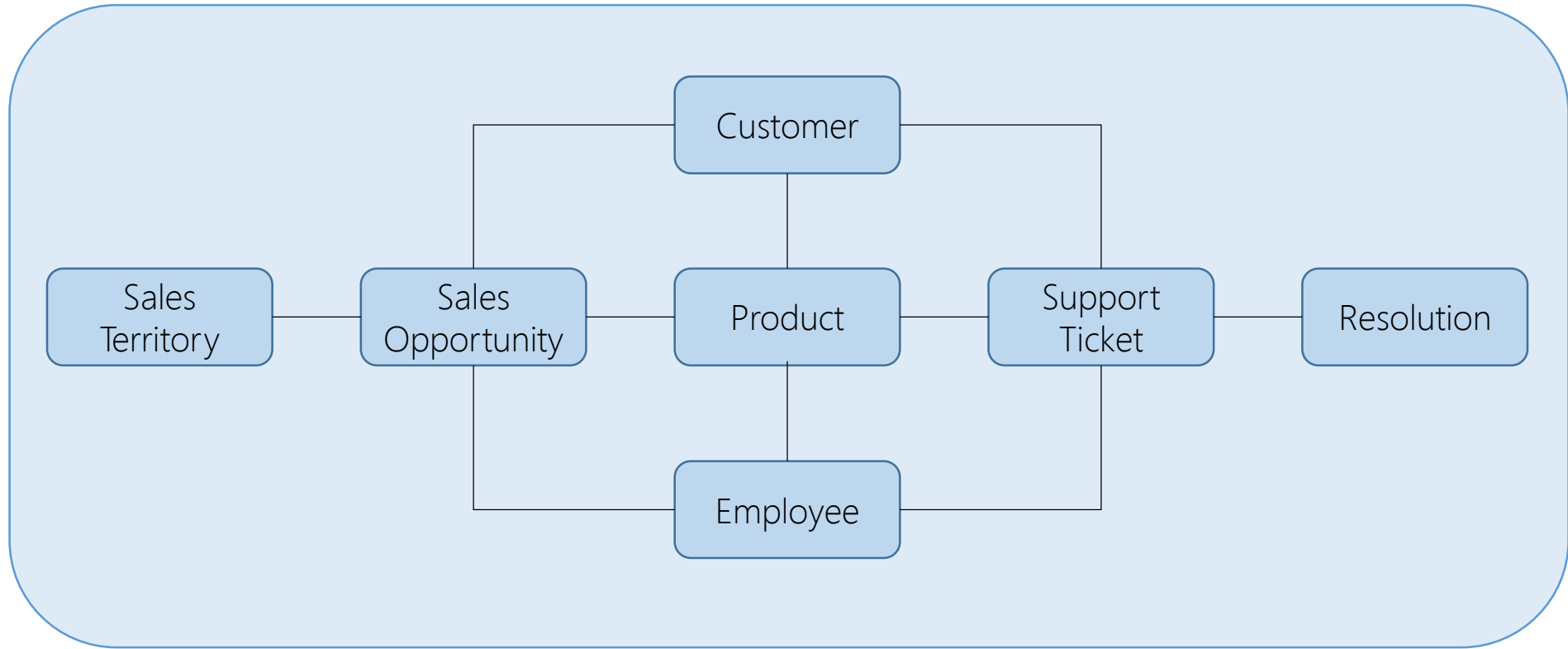
Sales

- Sales Opportunity
- Contact
- Sales Person
- Product
- Sales Territory

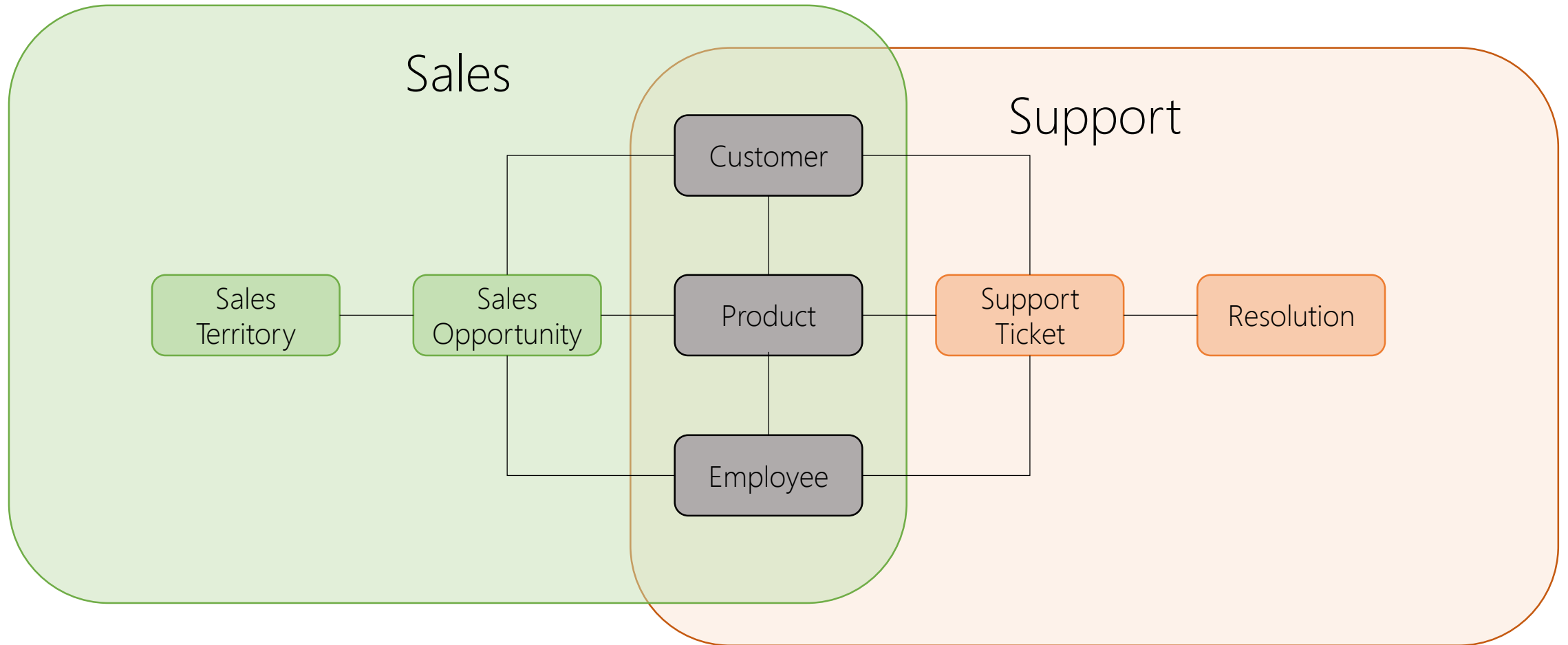
Support

- Support Ticket
- Customer
- Support Person
- Product
- Resolution

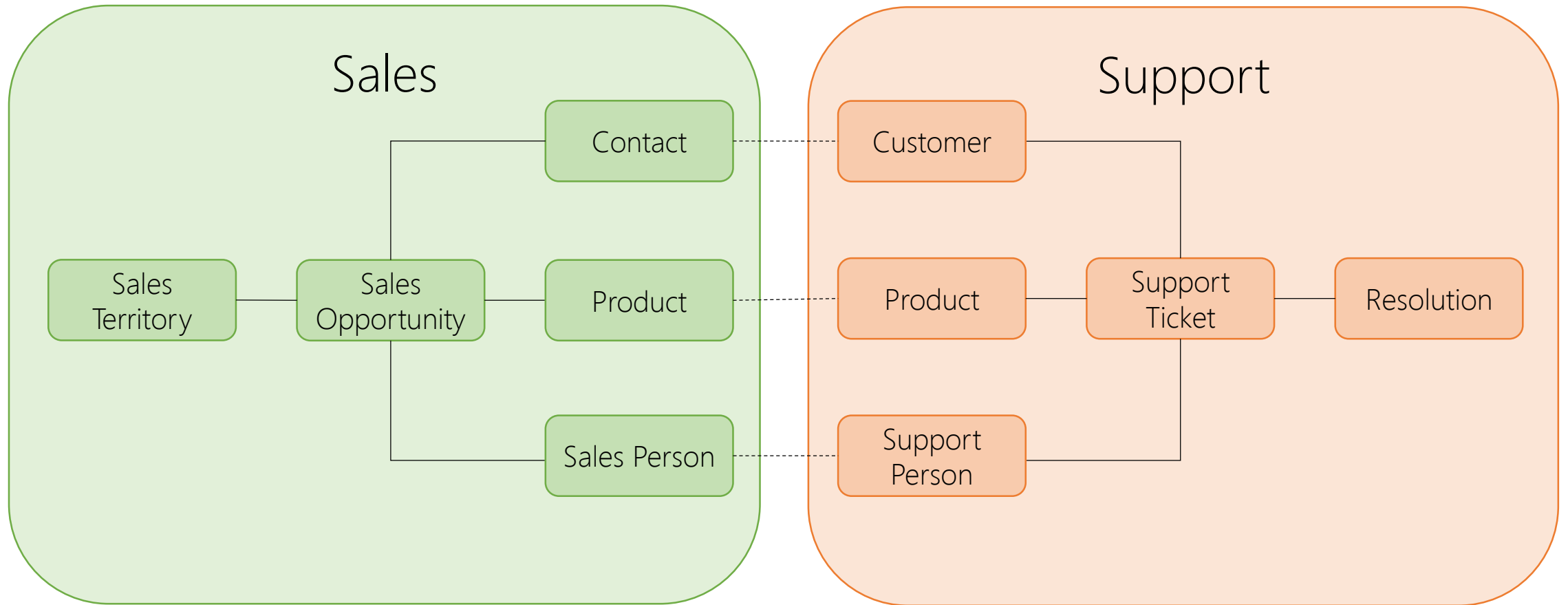
Single Domain Model



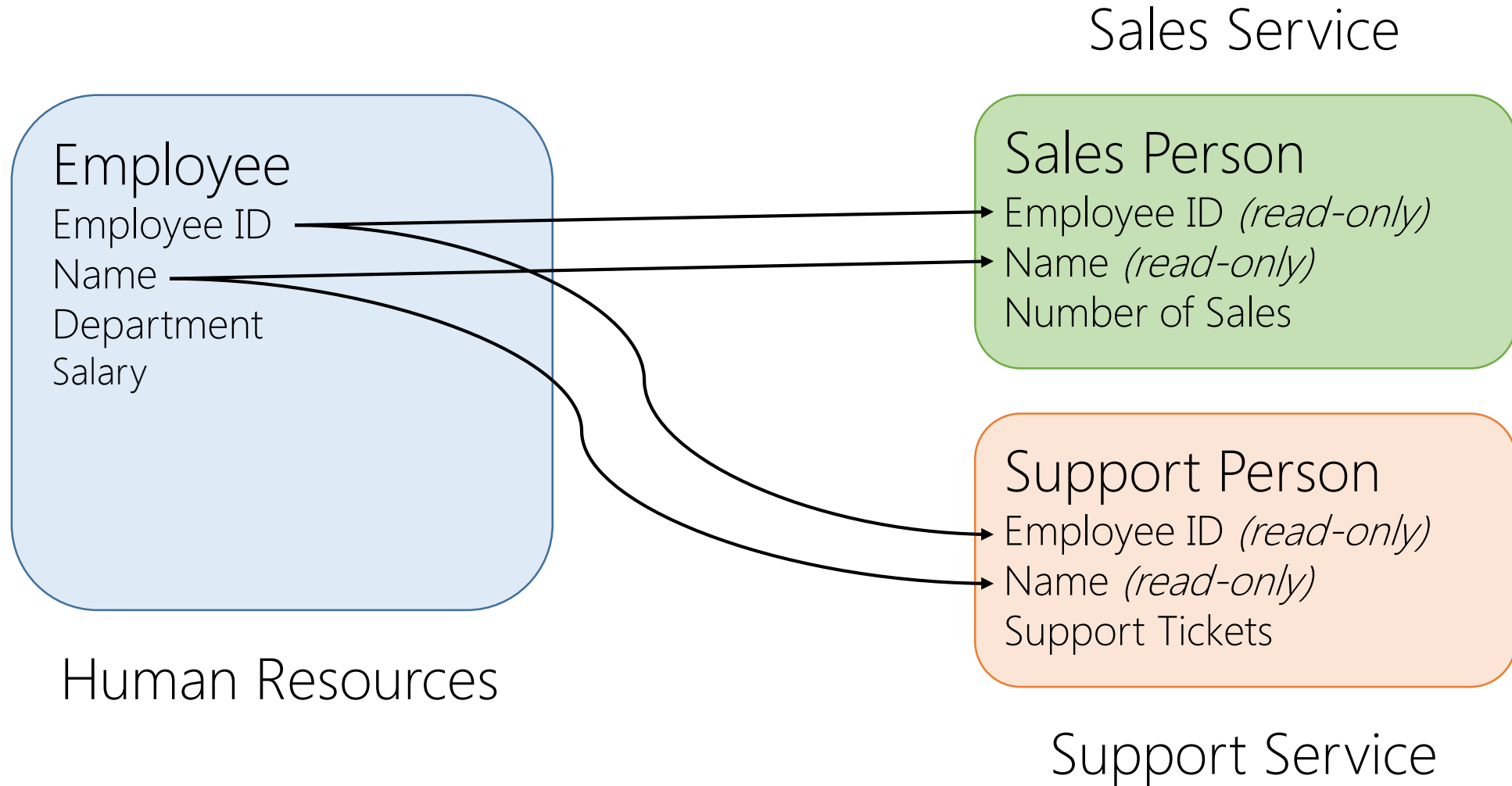
Bounded Contexts



Bounded Contexts

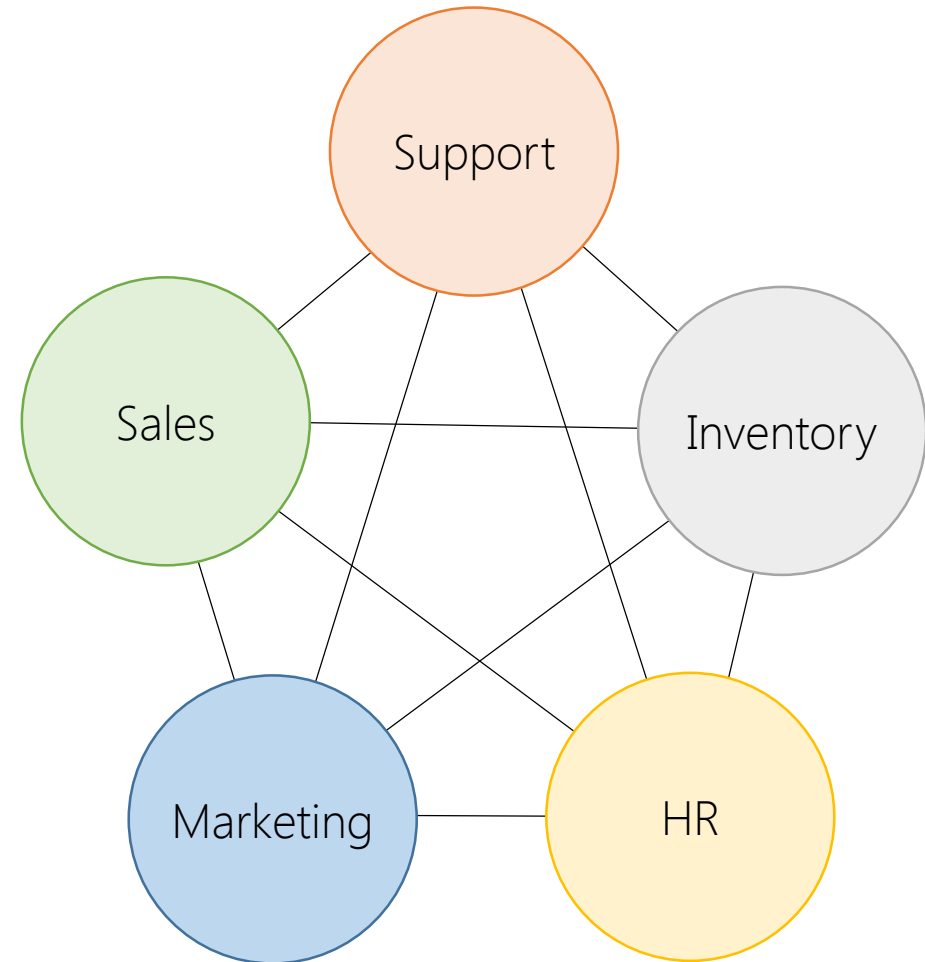


Bounded Contexts



Microservice Architectures

- Subdivide system
- Bounded contexts
- Small teams
- Independent
- Similar to SOA



Why Use Microservices?

Pros

- Less cost for large domains
- Smaller teams
- Independence

Why Use Microservices?

Pros

- Less cost for large domains
- Smaller teams
- Independence

Cons

- Only for large domains
- Higher up-front cost
- Distributed system costs

Conclusion

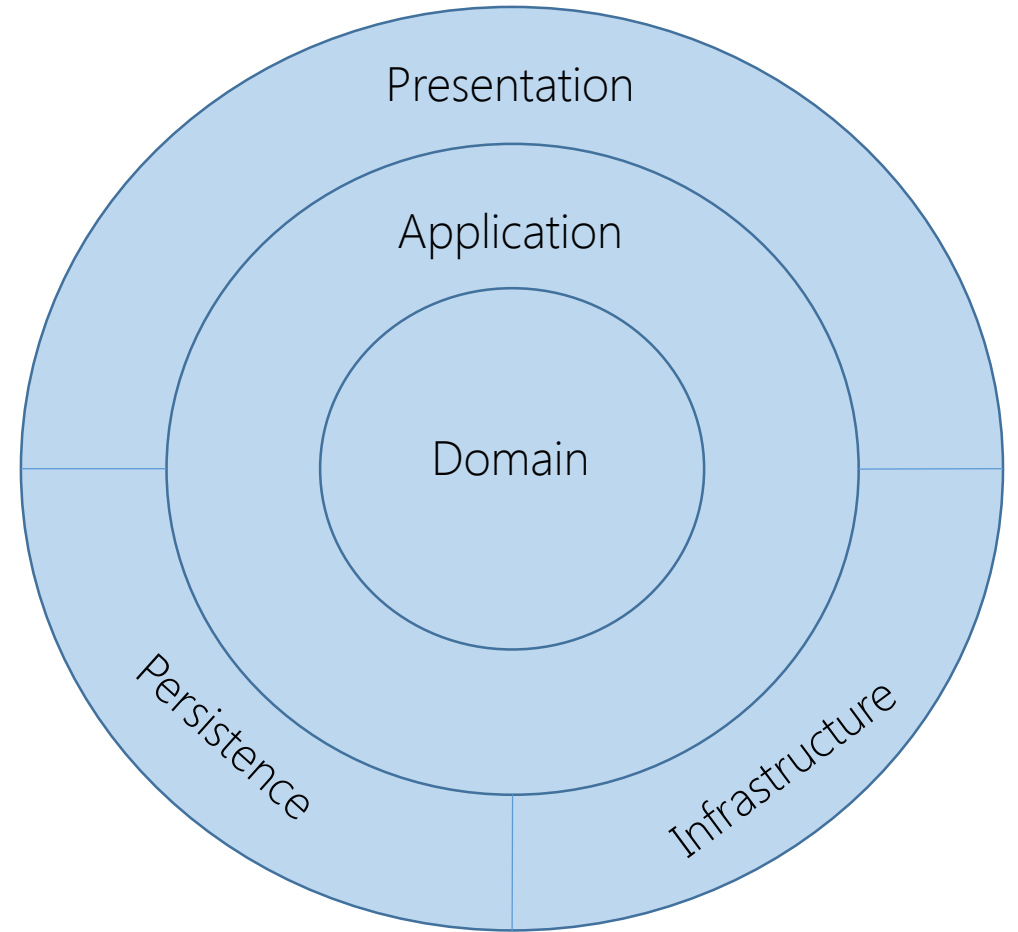
Summary

- Focus on the inhabitants



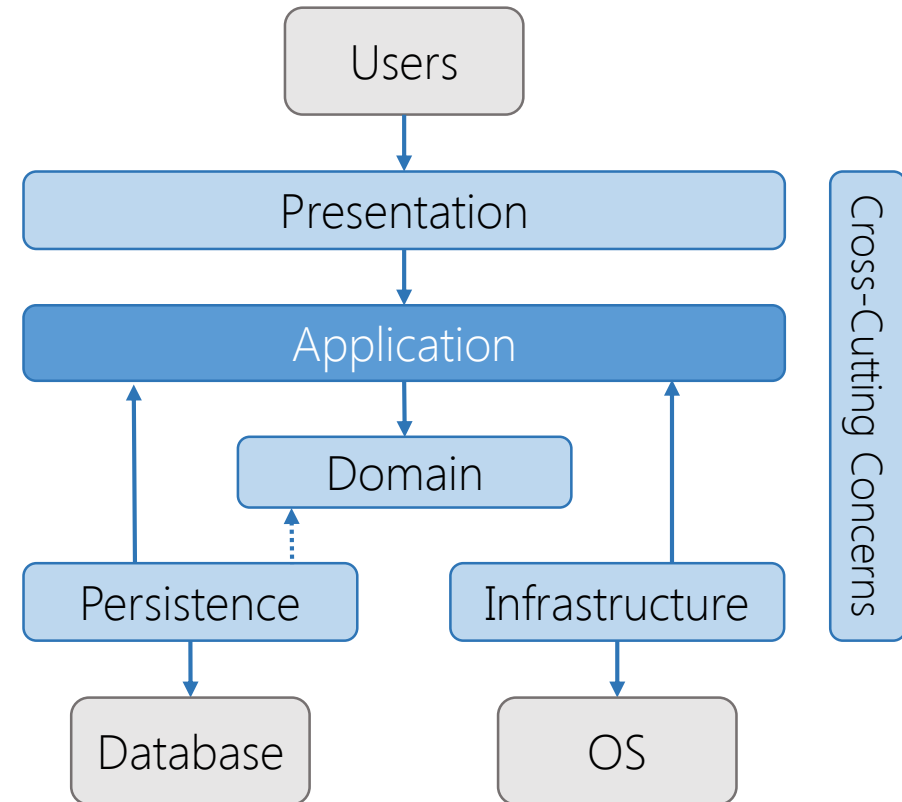
Summary

- Focus on the inhabitants
- Domain-centric Architecture



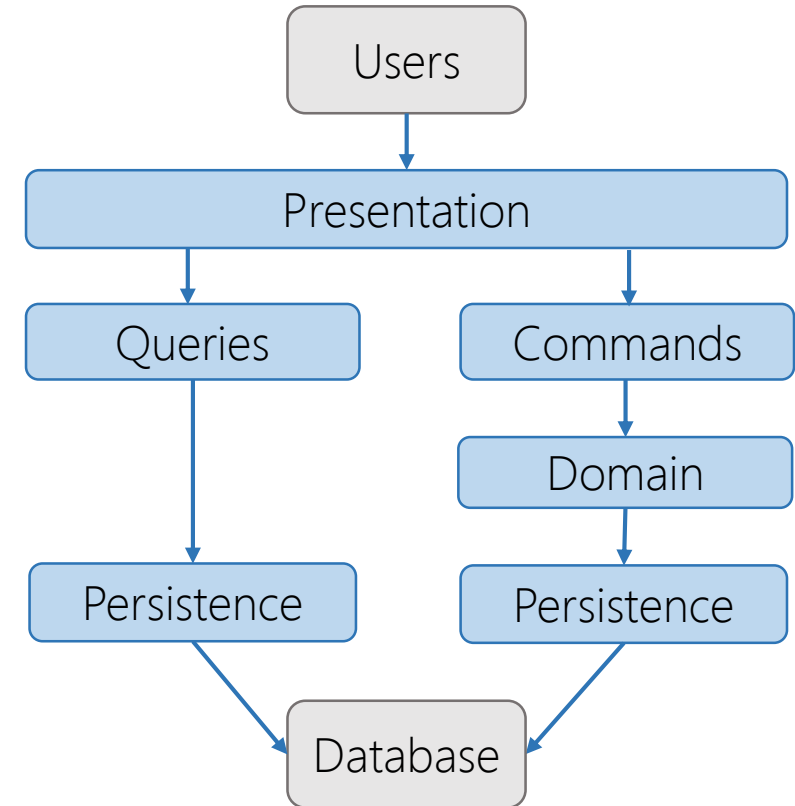
Summary

- Focus on the inhabitants
- Domain-centric Architecture
- Application Layer



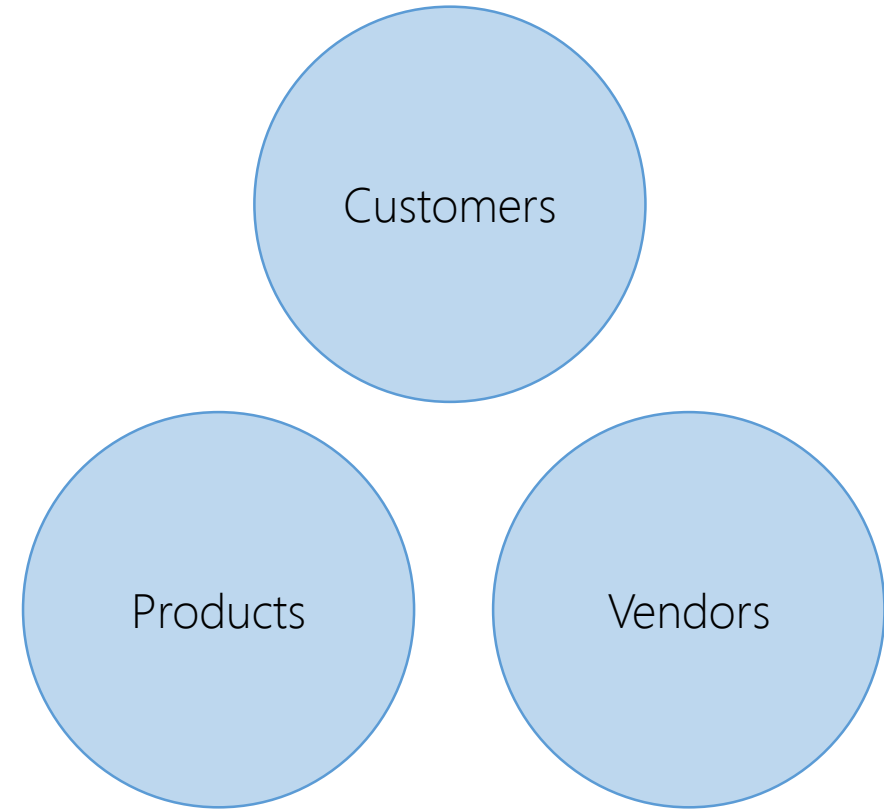
Summary

- Focus on the inhabitants
- Domain-centric Architecture
- Application Layer
- Commands and Queries



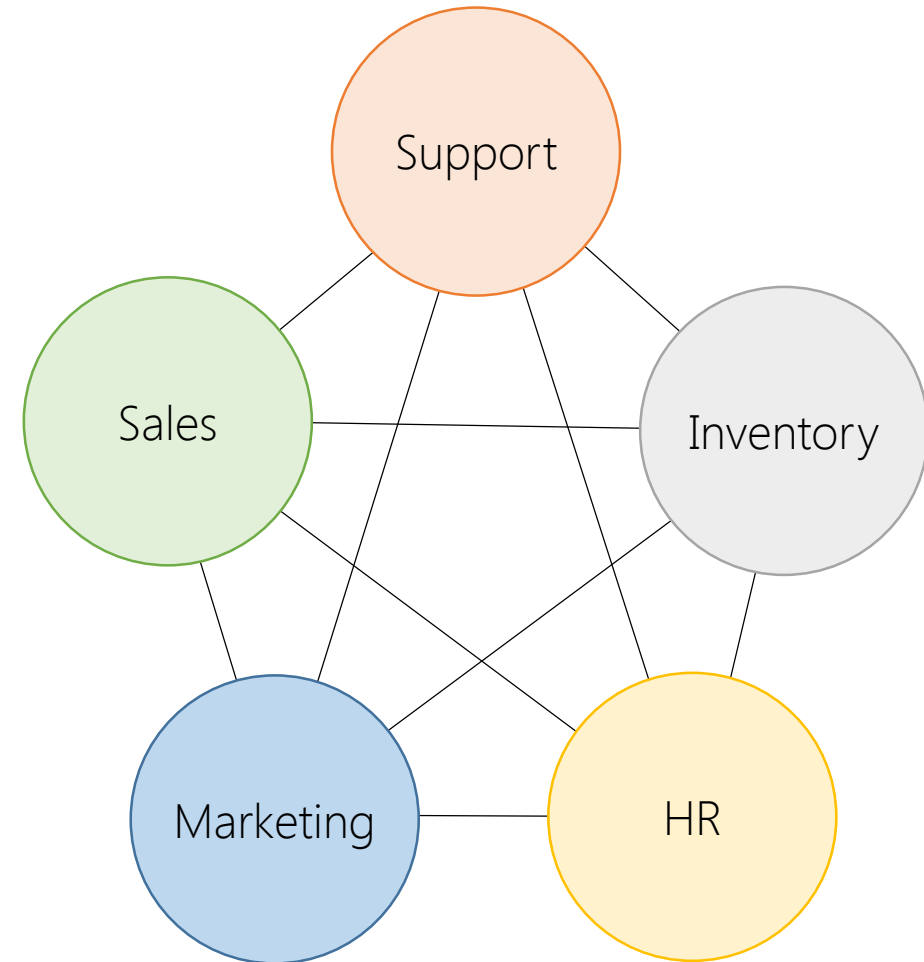
Summary

- Focus on the inhabitants
- Domain-centric Architecture
- Application Layer
- Commands and Queries
- Functional Cohesion

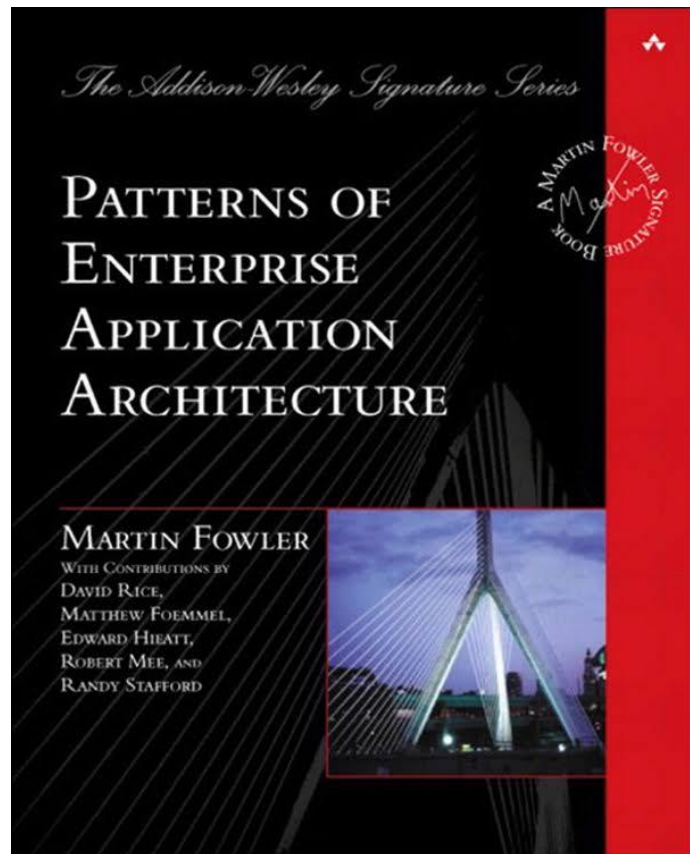


Summary

- Focus on the inhabitants
- Domain-centric Architecture
- Application Layer
- Commands and Queries
- Functional Cohesion
- Bounded Contexts



Where to Go Next?



Martin Fowler

Where to Go Next?

Uncle Bob presents the
Clean Code
Video Series

The banner features a green circular icon with a white checkmark and the words 'CLEAN CODE' repeated around its perimeter. To the right of the icon is a small portrait of Uncle Bob and a video player control bar with a timestamp of -00:14:19.

Episode 1 - Clean Code	Episode 12 - The Interface Segregation Principle
Episode 2 - Names++	Episode 13 - The Dependency Inversion Principle
Episode 3 - Functions	Episode 14 - SOLID Case Study
Episode 4 - Function Structure	Episode 15 - SOLID Components
Episode 5 - Form	Episode 16 - Component Cohesion
Episode 6 - TDD - Part 1	Episode 17 - Component Coupling
Episode 6 - TDD - Part 2	Episode 18 - Component Case Study
Episode 7 - Architecture	Episode 19 - Advanced TDD - Part 1
Episode 8 - SOLID Foundations	Episode 19 - Advanced TDD - Part 2
Episode 9 - The Single Responsibility Principle	
Episode 10 - The Open-Closed Principle	
Episode 11 - The Liskov Substitution Principle	

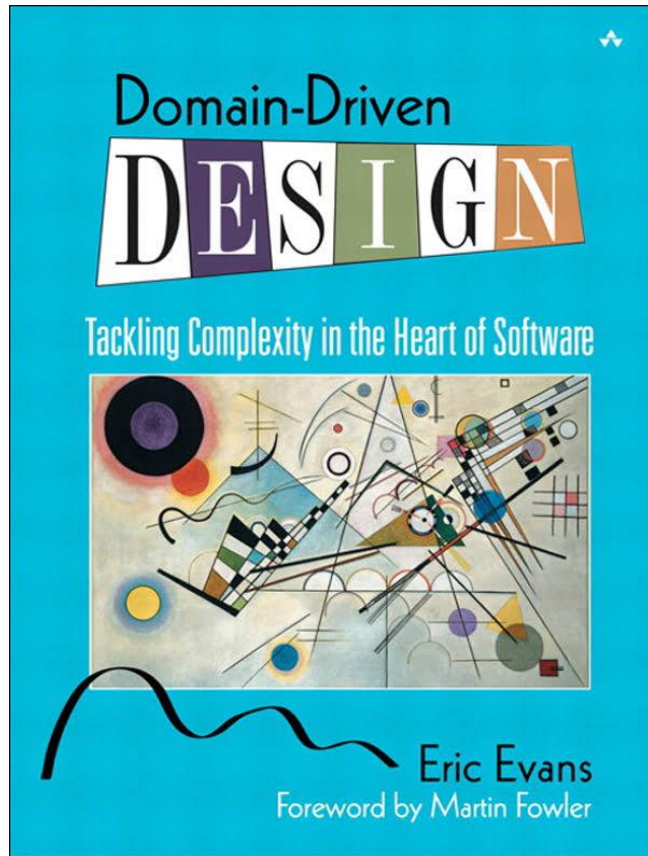
Episode 20 - Clean Tests

<http://cleancoders.com/>



Robert C. Martin

Where to Go Next?



Eric Evans

Where to Go Next?



Greg Young



Udi Dahan

Feedback

- Feedback is very important to me
- One thing you liked?
- One thing I could improve?

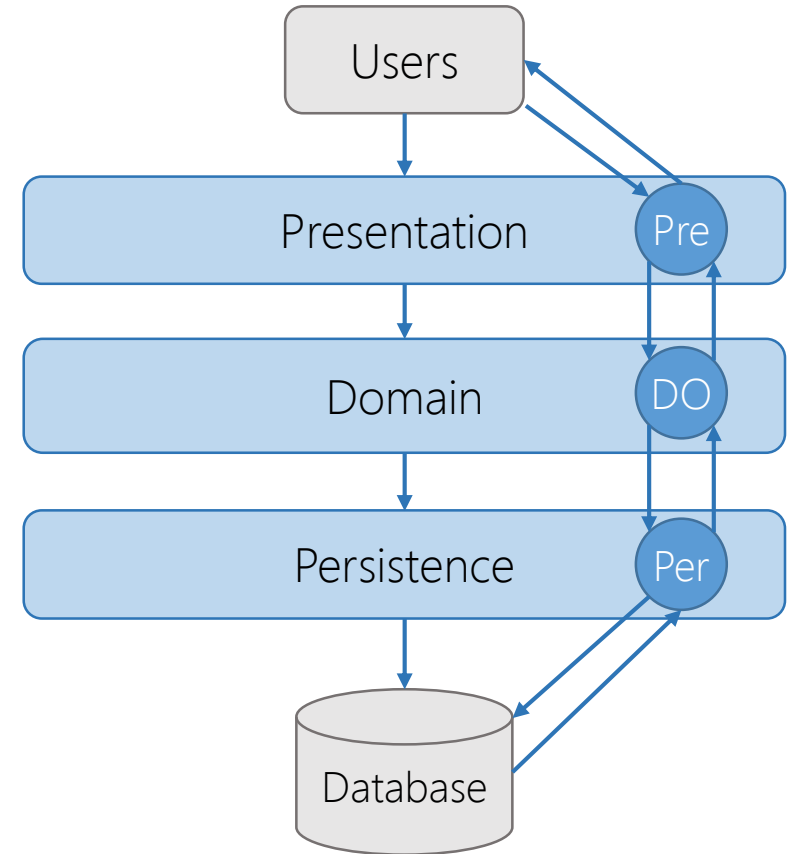
Contact Info

Matthew Renze
@matthewrenze
matthew@renzeconsulting.com

Renze Consulting
www.renzeconsulting.com

Layer Separation

- Separation of concerns
- Avoid abstraction leakage
- Multiple object representations
- Note: Add application layer and DTOs



Cost-Benefit of Layers

- Layers have a high cost
- Objects must be adapted
- Start with the minimum
- Typically 3-4 layers
- NOTE: Need to add application layer and DTOs

