Clean Architecture with ASP.NET Core

in 2 hours

Ardalis

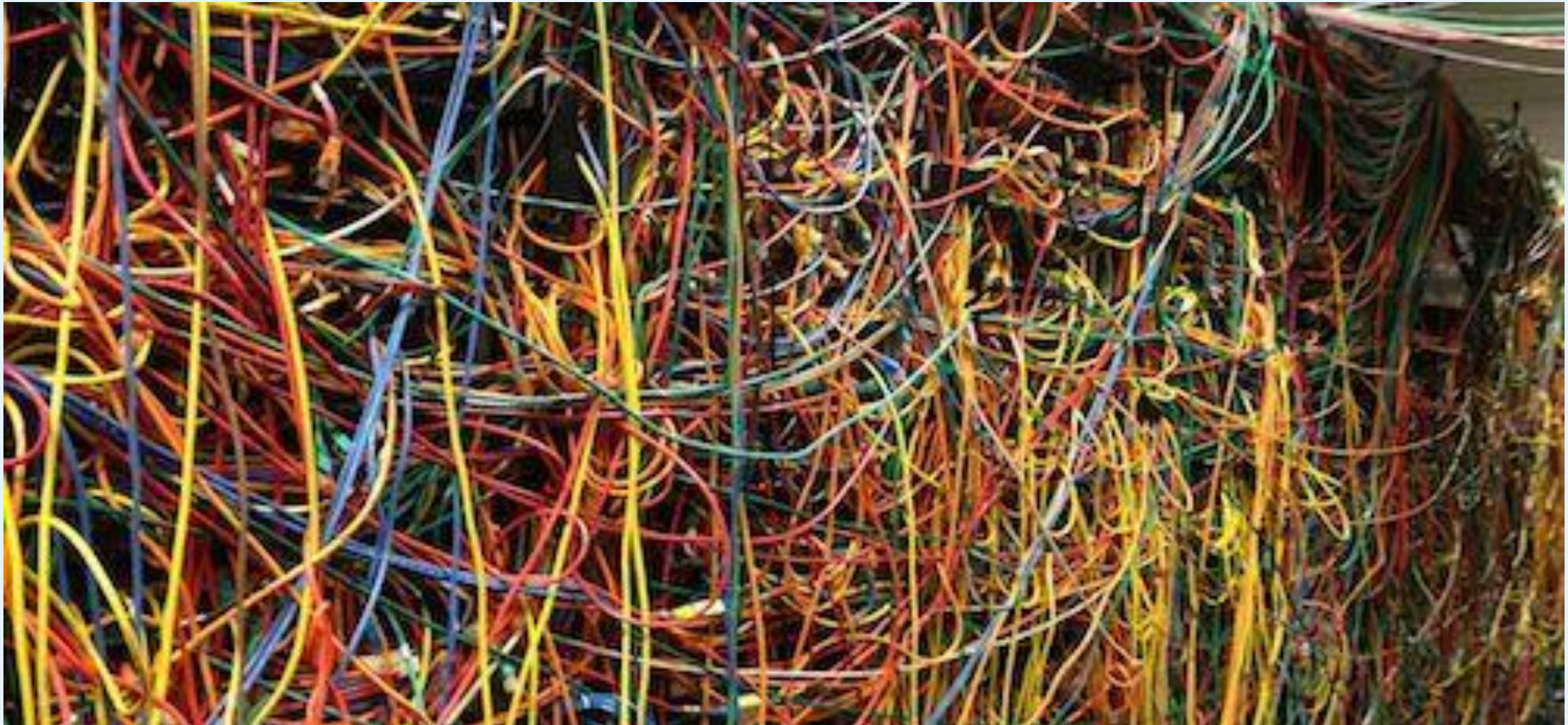
NimblePros.com | Ardalis.com

@ardalis.com | YouTube.com/ardalis

NimblePros

Better software development. Delivered.

The Problem – Tightly Coupled Hardwired Dependencies



The Goal – Loosely Coupled Well-Organized Dependencies

First Law of Software Architecture

"Everythin

It depends.

-off."

Corollary:

"If an architect thinks they've discovered something that isn't a trade-off, more likely they just haven't yet identified the trade-off."

Fundamentals of Software Architecture, Mark Richards and Neal Ford

A Brief History Lesson

Or, what led to this architecture being like… this?

Client/Server Beginnings

Data is King

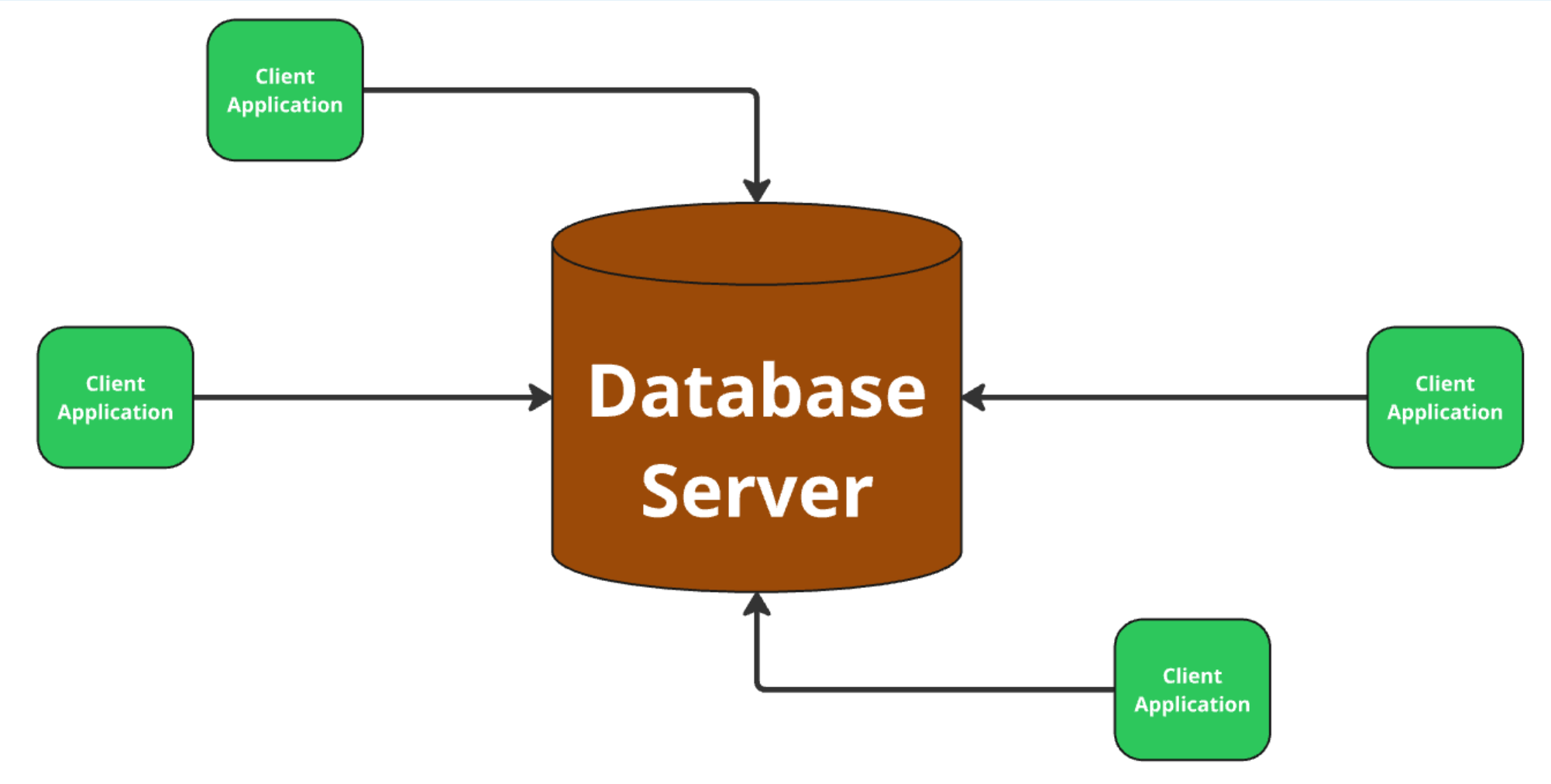
- MS Access
- FoxPro/Visual FoxPro
- dBase
- Clipper
- Paradox
- Clarion
- FileMaker Pro
- PowerBuilder

- Oracle
- Microsoft SQL Server
- IBM DB2
- Sybase SQL Server
- Informix

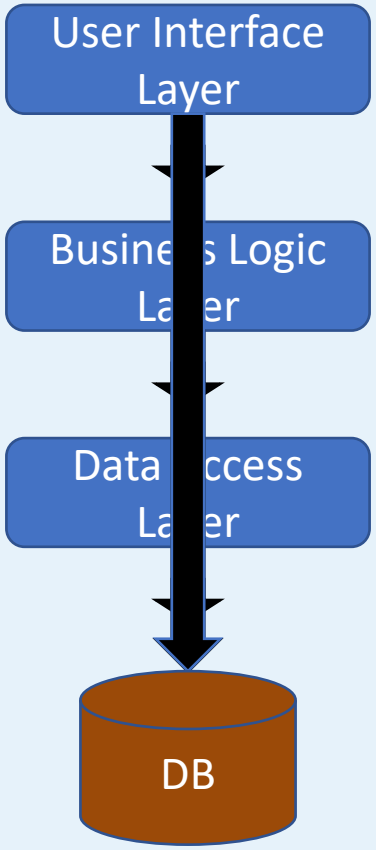
THE DBA!

thenounproject.com/icon/hero-4159954/

Database-Centric



Early Layered Approaches Didn't Solve the Dependency Problem



Everything

Depends on the *database*

True Story:
Trying to Unit Test ASP.NET Apps in 2006

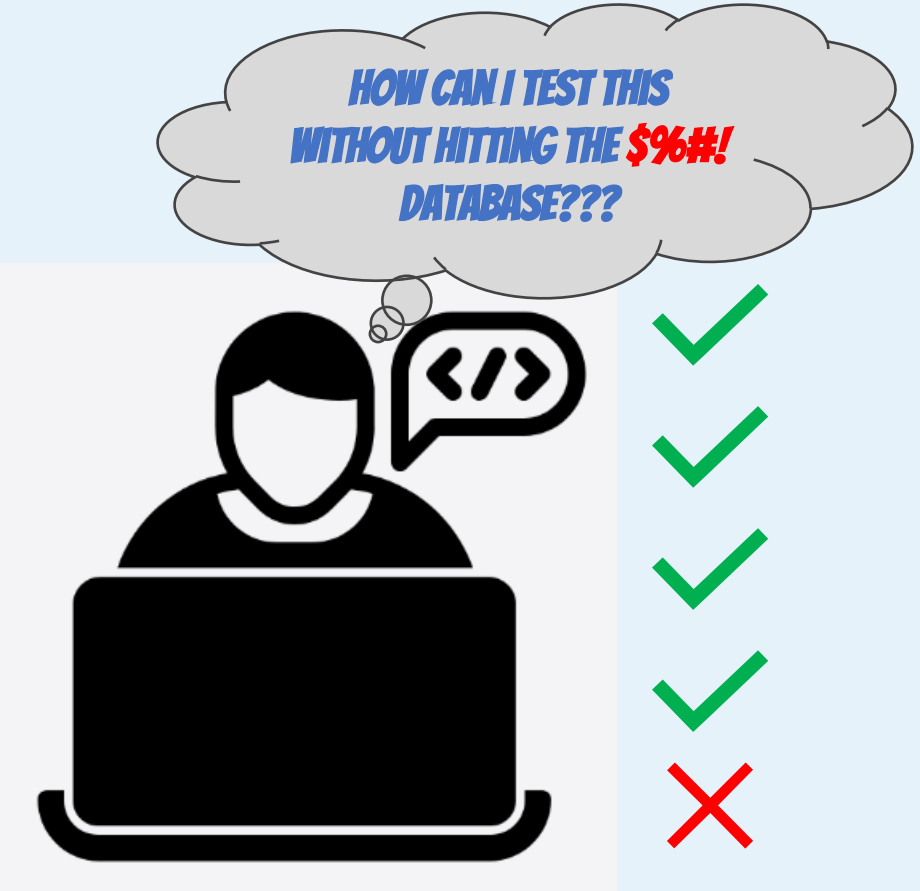
Doing it al

I wanted t
to make su

But halfwa
important

How the h
populatin

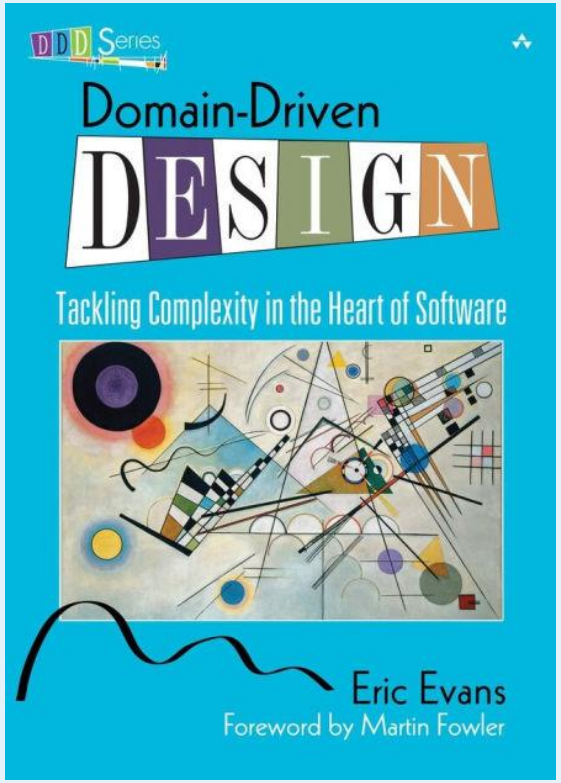
A Move Toward Quality and Automation



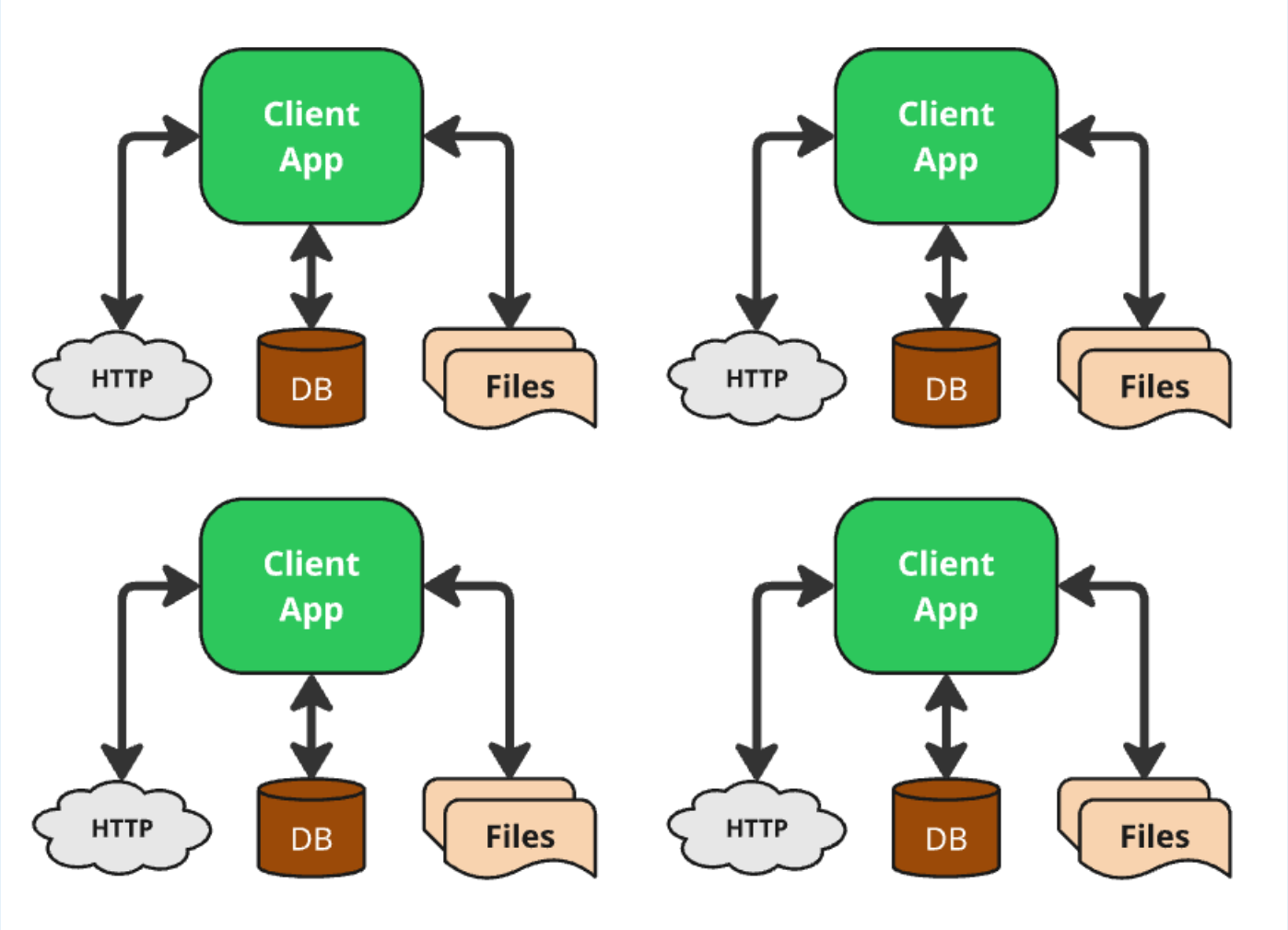
CruiseControl (ThoughtWorks)

thenounproject.com/icon/programmer-7961919/

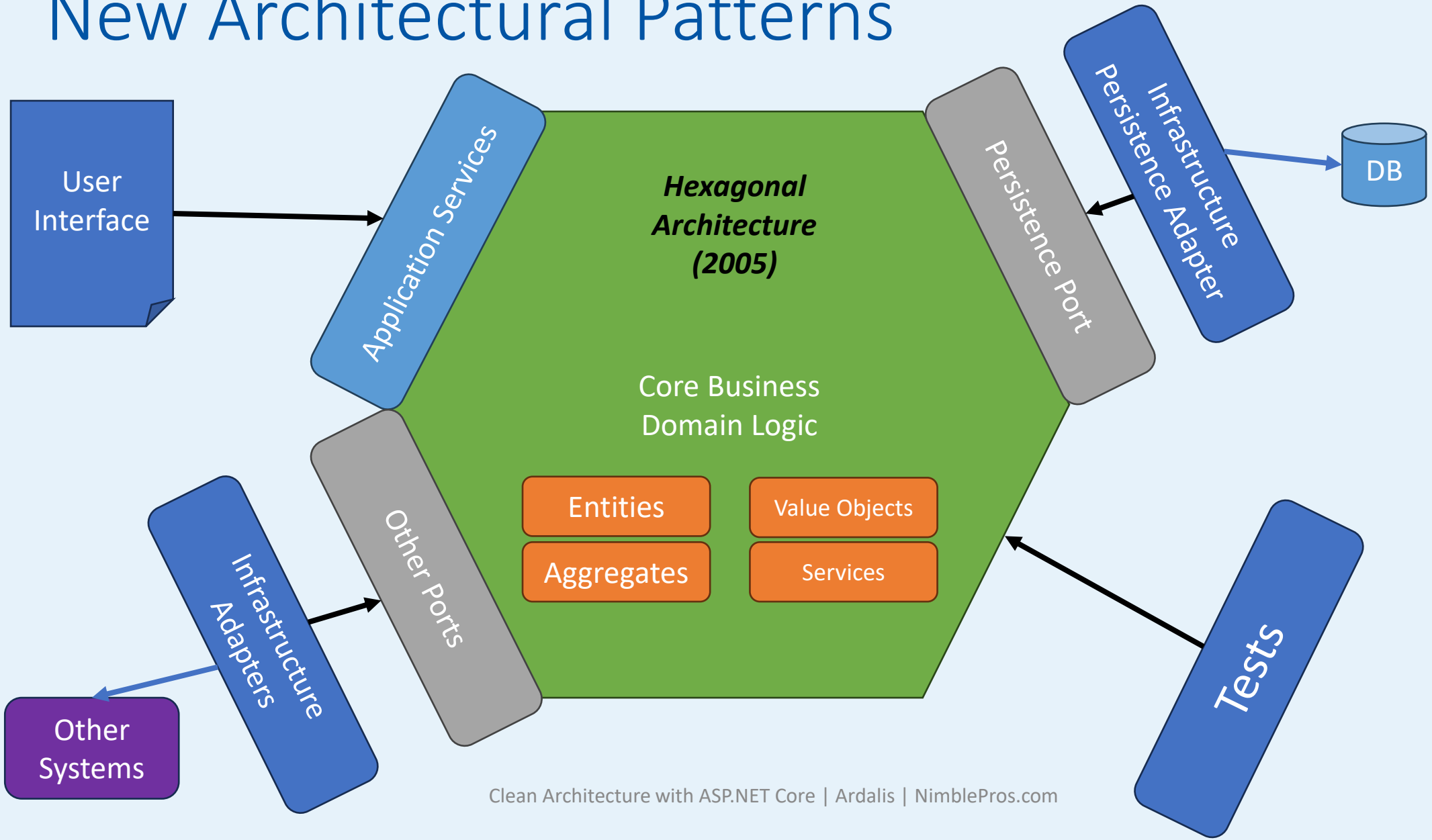
Enter a Domain-Centric Approach



2003



New Architectural Patterns



Principles

Of Clean Architecture

Separation of Concerns

Don't let your plumbing code pollute your software.

DevIQ.com/principles/separation-of-concerns

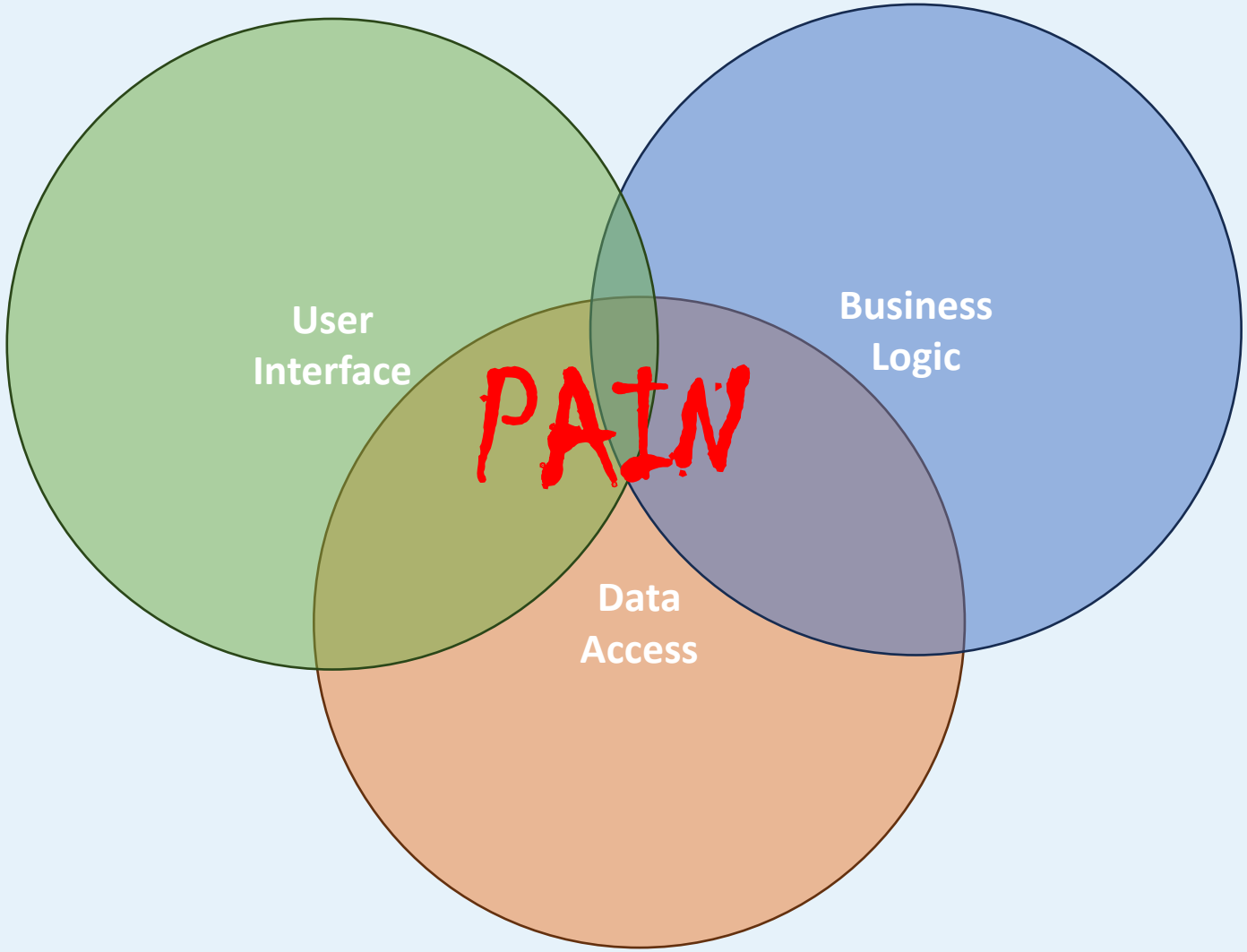
Why Separate Concerns?

Spaghetti Code

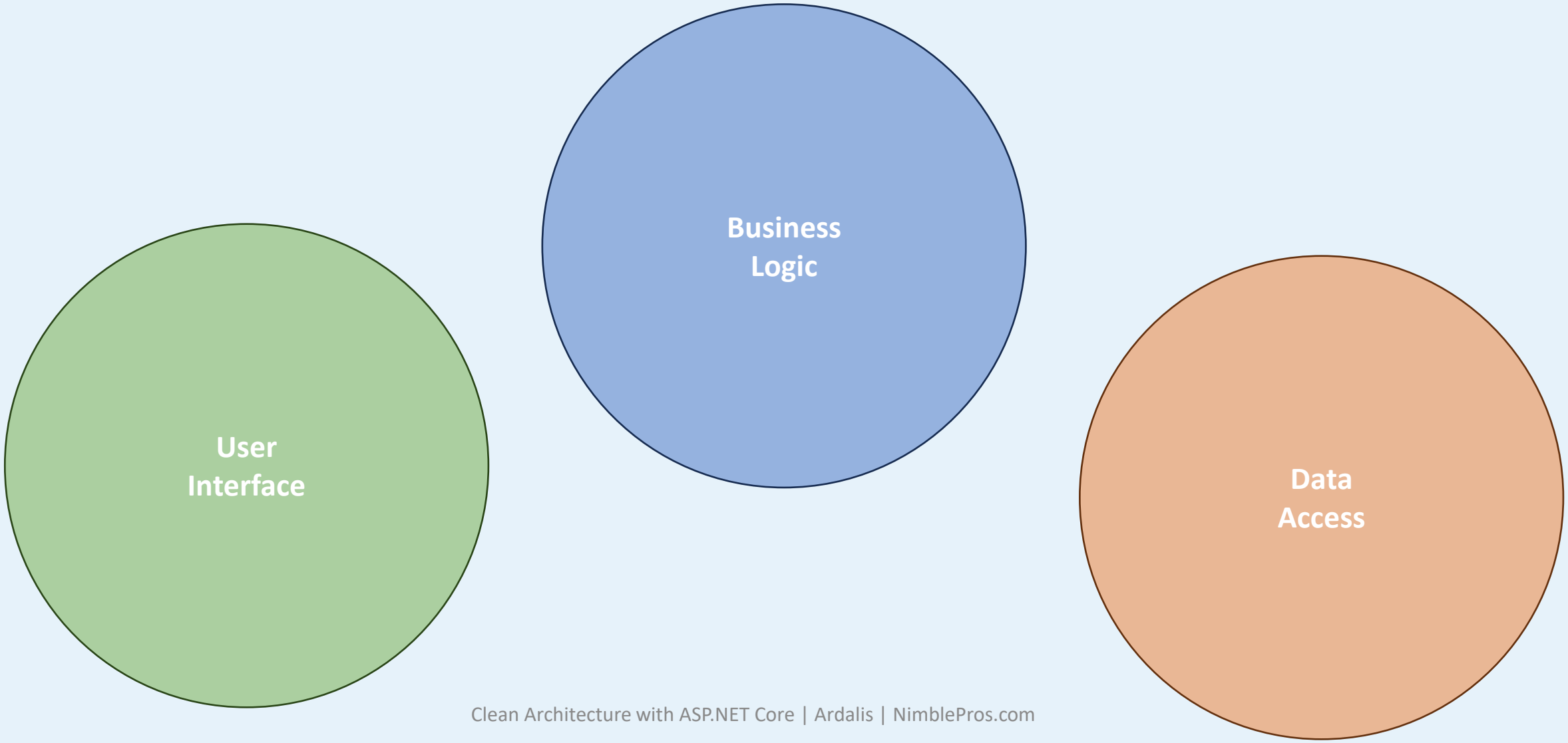
Maintenance is easy with everything in one place.

DevIQ.com/antipatterns/spaghetti-code

The Big 3 Concerns



You Gotta Keep ‘em Separated



Single Responsibility

Avoid tightly coupling your tools together.

DevIQ.com/principles/single-responsibility-principle

Single Responsibility Principle (SRP)

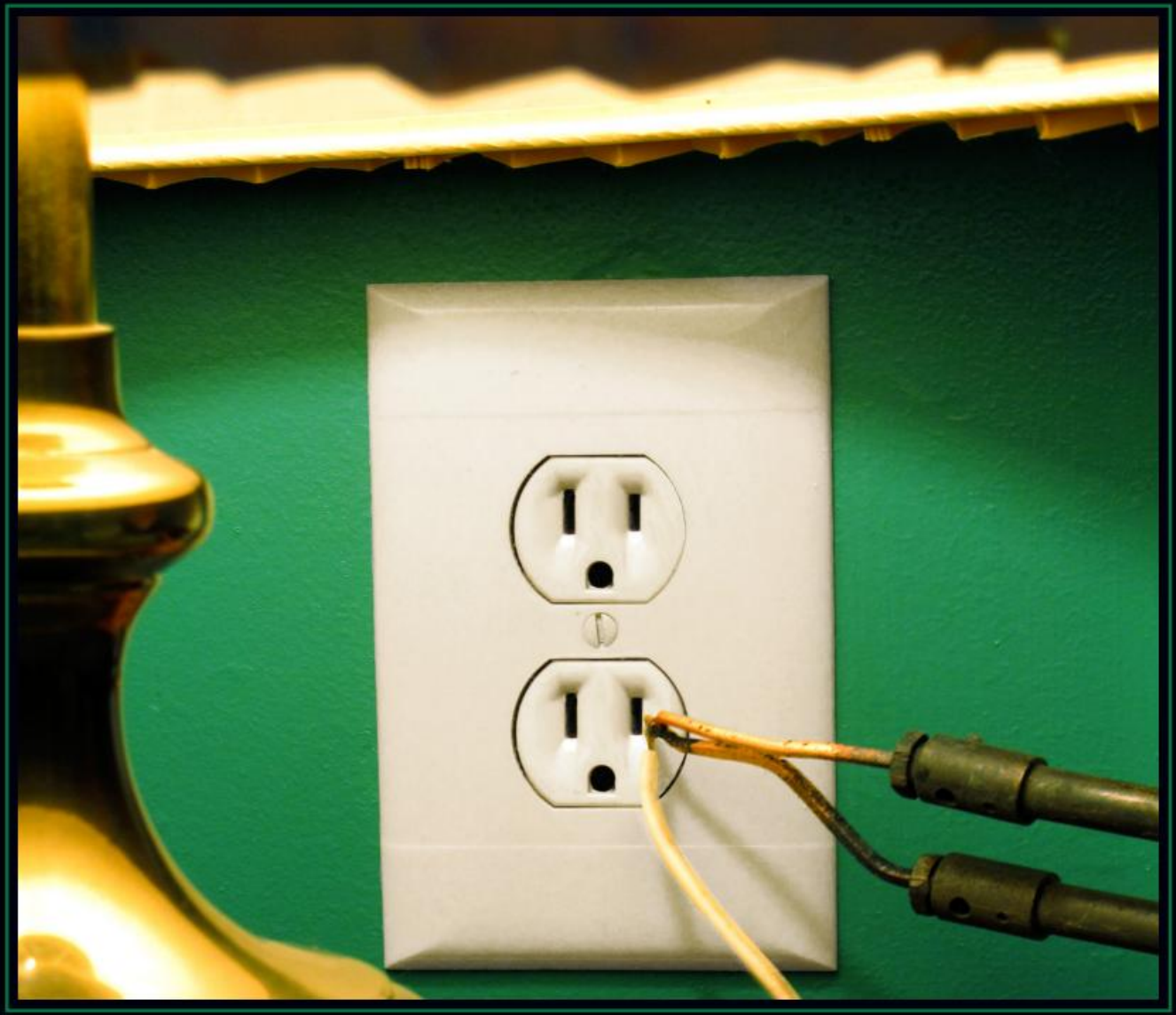
Classes should have a single reason to change

Single Responsibility Principle (SRP)

Too many responsibilities adds coupling and reduces cohesion

If I follow SRP…

My solution will have more, smaller types.



Dependency Inversion

Would you solder a lamp directly to the electrical wiring in a wall?

DevIQ.com/principles/dependency-inversion-principle

Dependency Inversion Principle (DIP)

High level modules should depend on abstractions, not low-level modules.

Low level modules should also depend on abstractions!

Abstractions should not depend on details;

Details should instead depend on abstractions!

Explicit Dependencies Principle

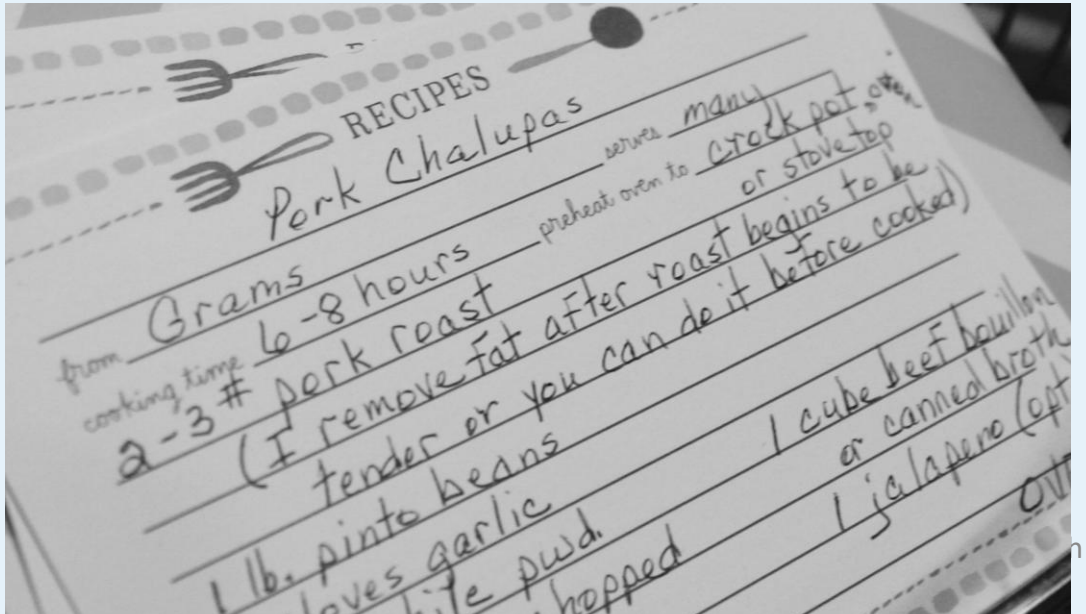
Classes should request all dependencies via their constructor

DevIQ.com/principles/explicit-dependencies-principle

Explicit Dependencies Principle

Make Types Honest About Their Dependencies

Avoid Runtime Surprises!



Constructor Parameters Are Like **Recipe Ingredients**

DevIQ.com/principles/explicit-dependencies-principle

Make the right thing easy and the wrong thing hard

Let the compiler help you – that's its job.

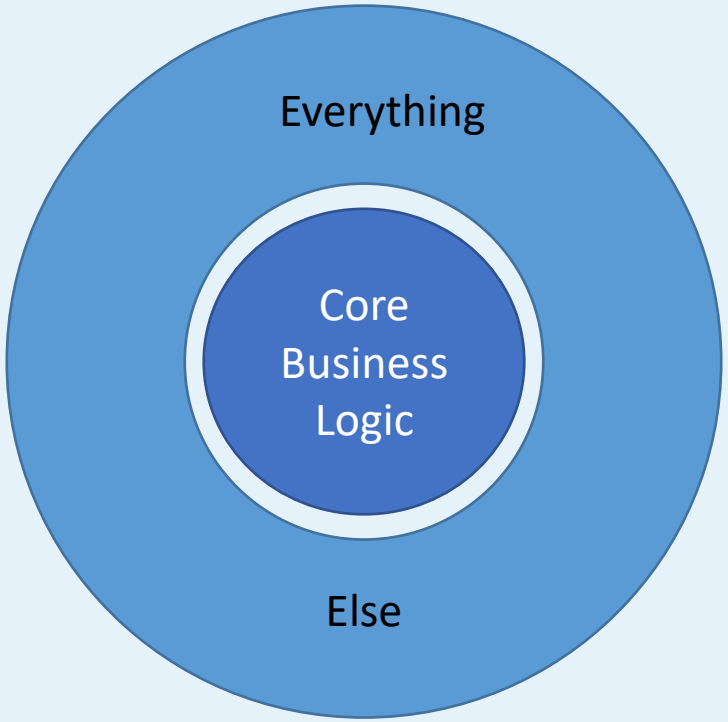
Force developers into the “pit of success!”

The default path should be the right path.

The Dependency Rule

Source code dependencies point inward toward **high level** concerns.

External dependencies and UI concerns are kept **at the edge**.



Domain-Centric Design

Focus on the domain model and business logic, not infrastructure

Core Business Logic

The domain model

Abstractions and interfaces for working with the domain model and any infrastructure required

Use Cases Project

Follow CQRS: Command/Query Responsibility Segregation

Defines **Messages** (Commands, Queries) and their related types (DTOs)

May also publish Events

Use Cases - Commands

- Load, create, and/or delete domain model type(s) (via abstractions)
- Call methods on domain model object instances
- Persist changes (via abstractions)

Use Cases - Queries

- Fetch domain model types via abstractions and map them to DTOs to return

OR

- Define a custom query service abstraction
  - Which returns custom DTOs representing the results
  - And is implemented in Infrastructure using whatever data access tech is most appropriate

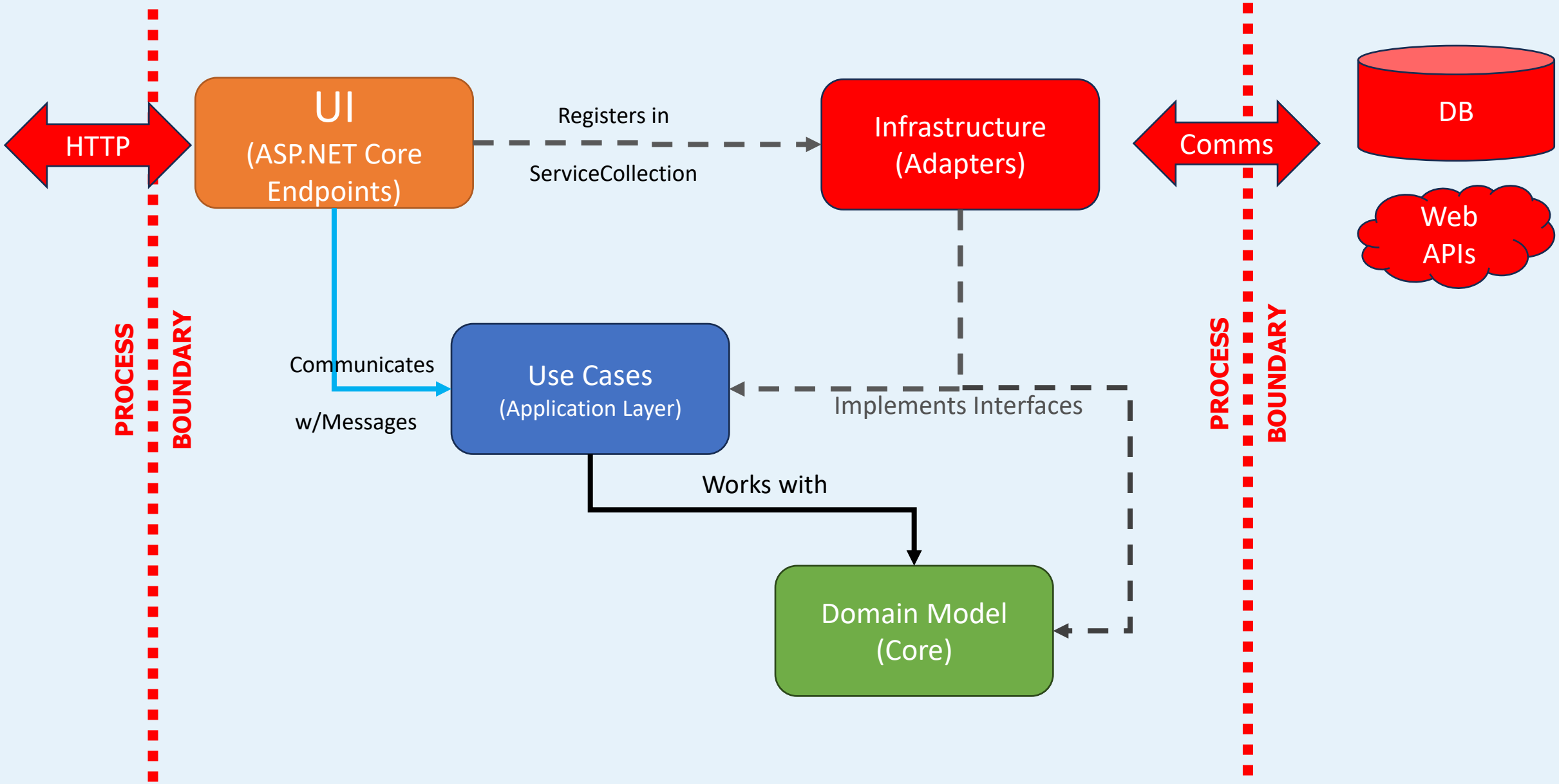
Infrastructure Project

- Defines **adapters** for any out-of-process communication

- Implements Interfaces defined in **Core** or **Use Cases**

Clean Architecture Organization

Typically just 4 projects


HTTP
UI
(ASP.NET Core
Endpoints)
Registers in
ServiceCollection
Infrastructure
(Adapters)
Comms
DB
Web
APIs
PROCESS
BOUNDARY
Communicates
w/Messages
Use Cases
(Application Layer)
Implements Interfaces
PROCESS
BOUNDARY
Works with
Domain Model
(Core)

The Core Project (domain model)

Interfaces

Entities

Value Objects

Aggregates

Domain Services

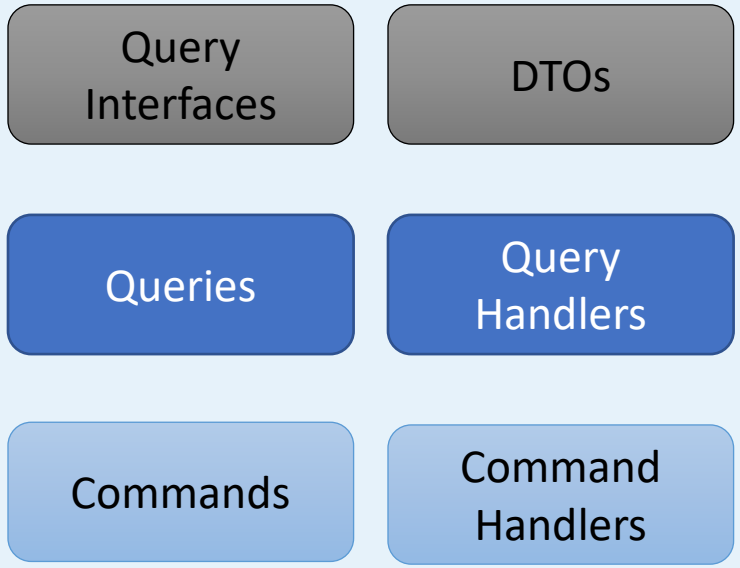
Exceptions

Domain Events

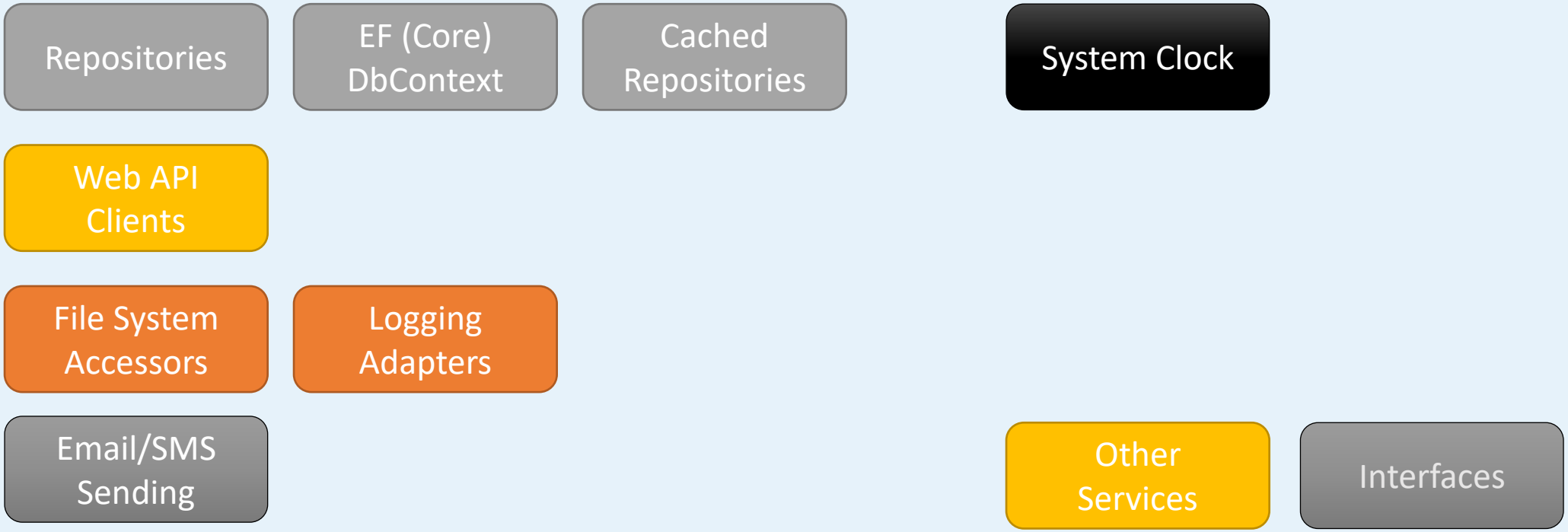
Event Handlers

Specifications

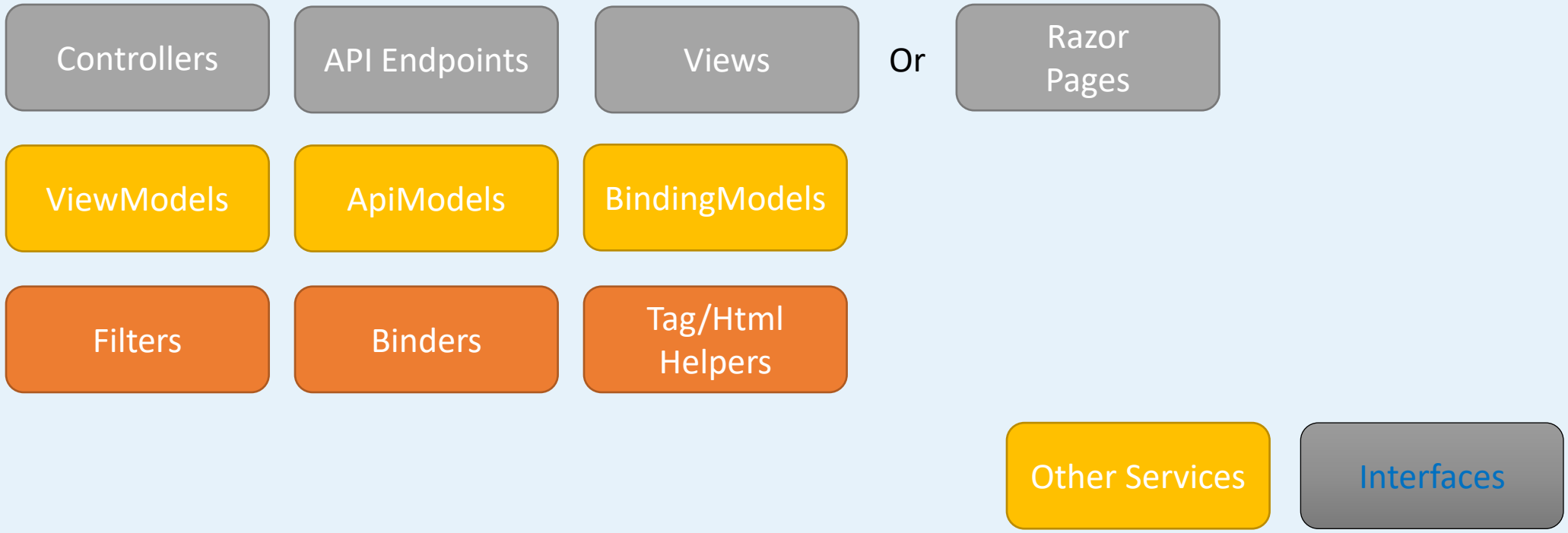
The UseCases Project



The Infrastructure Project (dependencies)



The Web Project (dependencies)



Sharing Common Code Between Applications / Solutions

Domain-Driven Design refers to this as a **Shared Kernel**

- Holds **common types** to be shared between solutions
- Ideally distributed as a **NuGet Package**
  - Once stable, to allow opt-in to new versions

The Shared Kernel Package

Base Entity

Base Domain Event

Base Specification

Common Exceptions

Common Interfaces

Common Auth e.g. User class

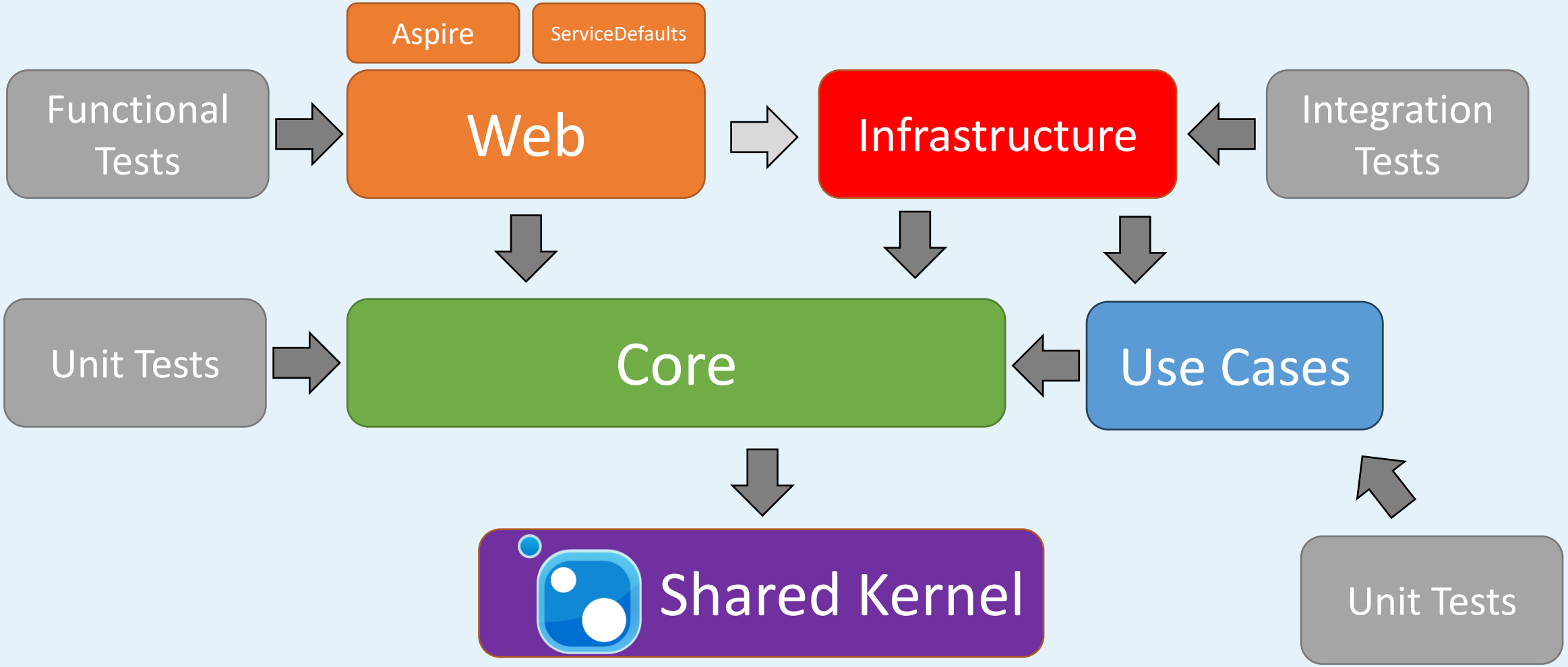
Common DI

Common Logging

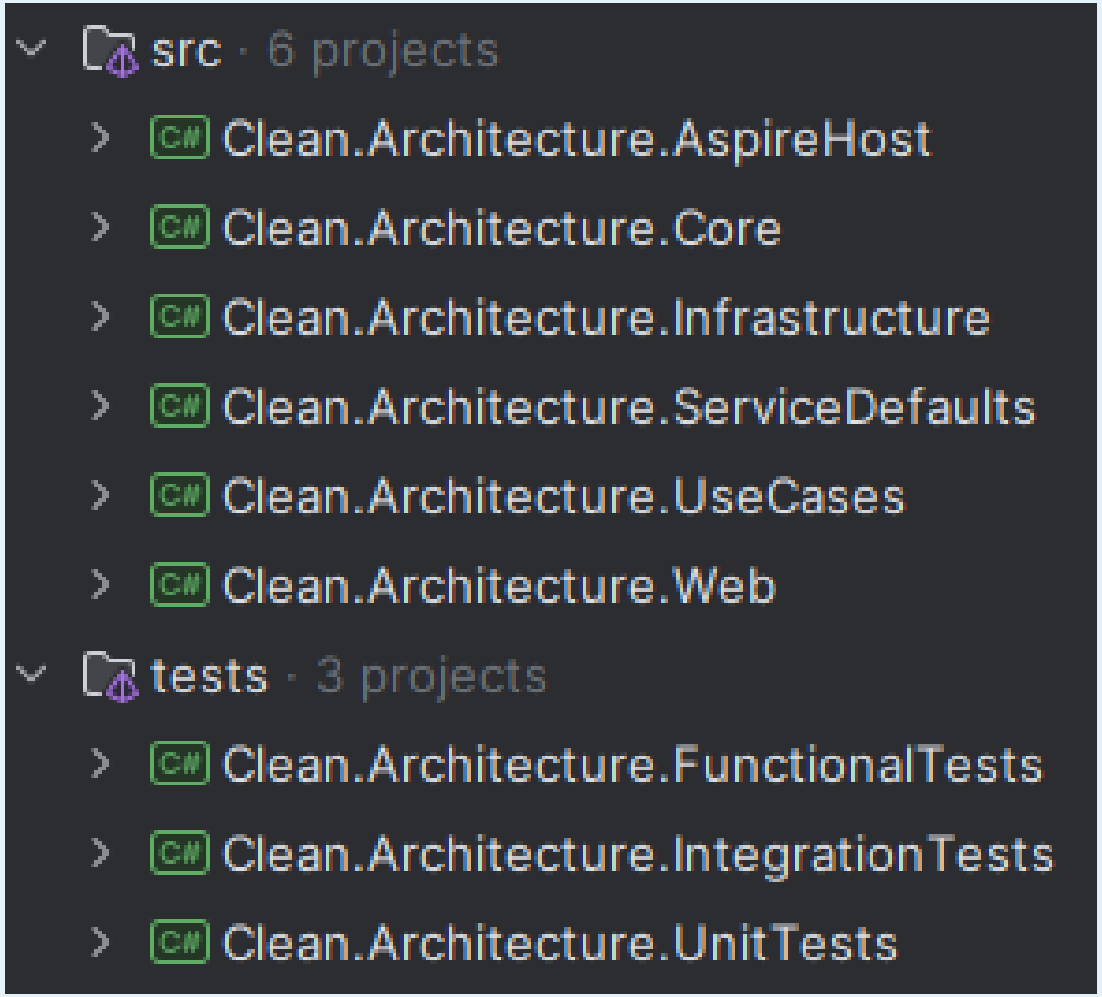
Common Guard Clauses

Common Behaviors

Overall Dependency Relationship

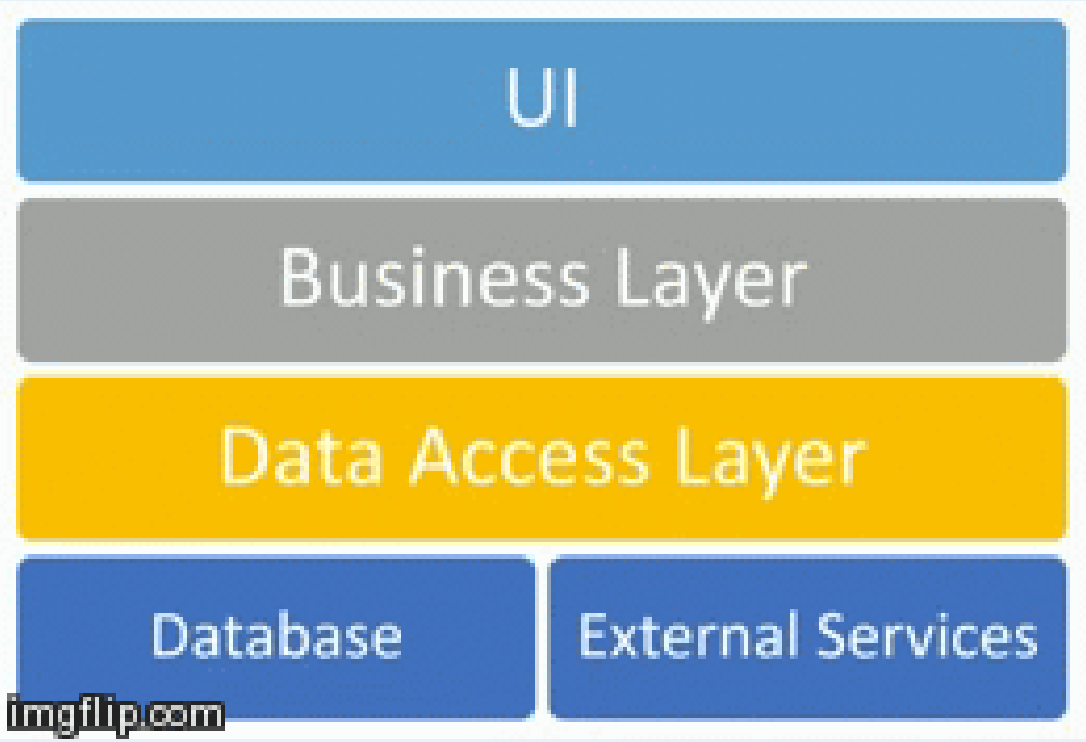


Overall Solution Structure



Deliver in Vertical Slices

- Don’t build one horizontal layer at a time!
- **Vertical Slices** referred to feature delivery long before it was popularized as an “architecture”



deviq.com/practices/vertical-slices

Demos, Code, and Questions

What about Vertical Slice Architecture (VSA)?

- Aka **Feature Folders**

- One Project (typically)
- Co-locate API endpoint, DTOs, and (if using them) messages and their handlers in one folder per use case
- (Usually) Prefer fewer abstractions (but still use DI)

- Not **everything** is in a feature folder
  - Domain models, DbContext, etc are still shared
- Less focus on dependency control/management

But we really hate projects…

- Use one project and folders and namespaces instead

- Don't rely on compiler and keywords like **`internal`**

Single Project Clean Architecture

- Enforce the **Dependency Rule** and other architecture rules in unit tests instead of using projects – still gain the dependency management benefits!

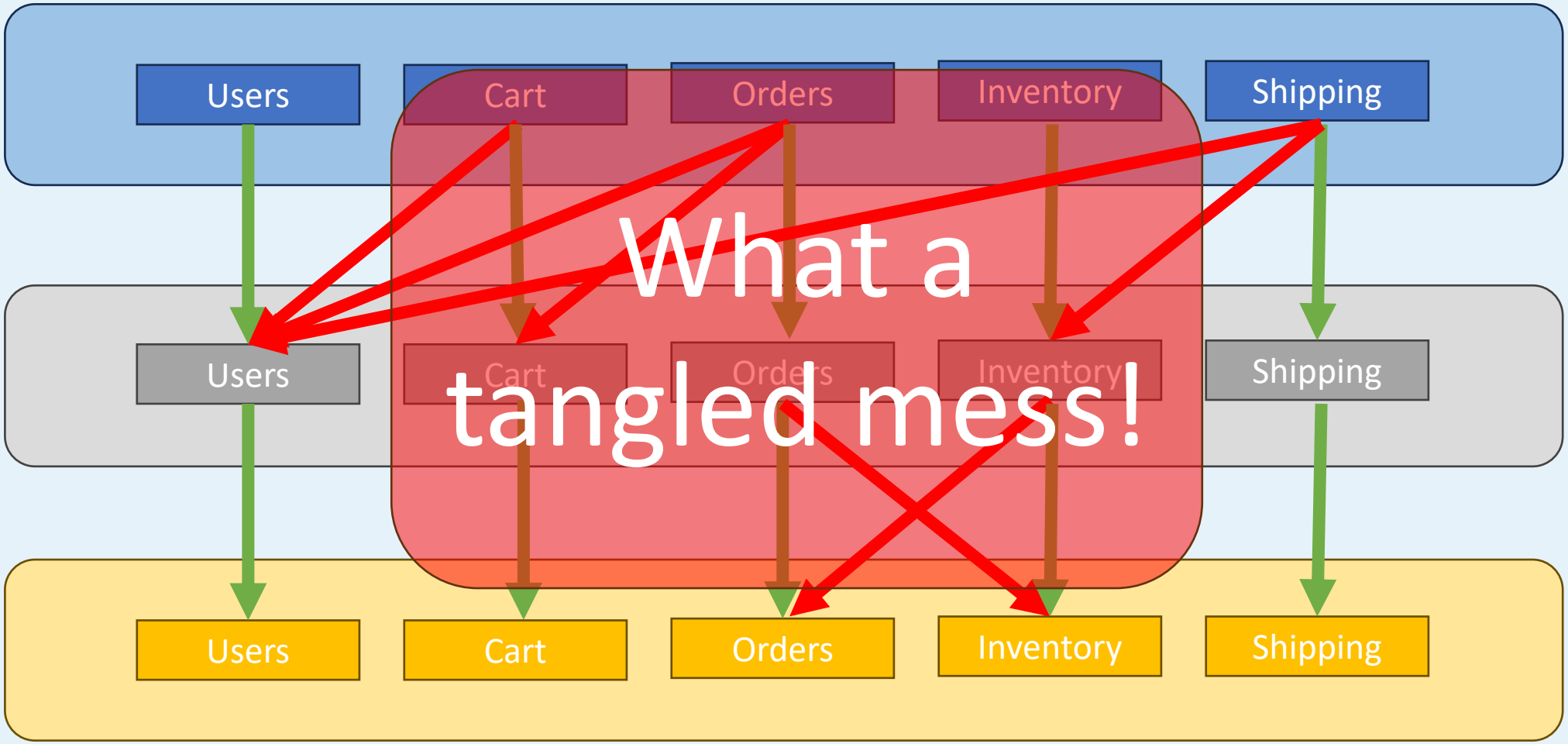
- Use **ArchUnit.Net** (github.com/TNG/ArchUnitNET)
  - Or github.com/BenMorris/NetArchTest

- Be sure to at least run these tests in your CI build!

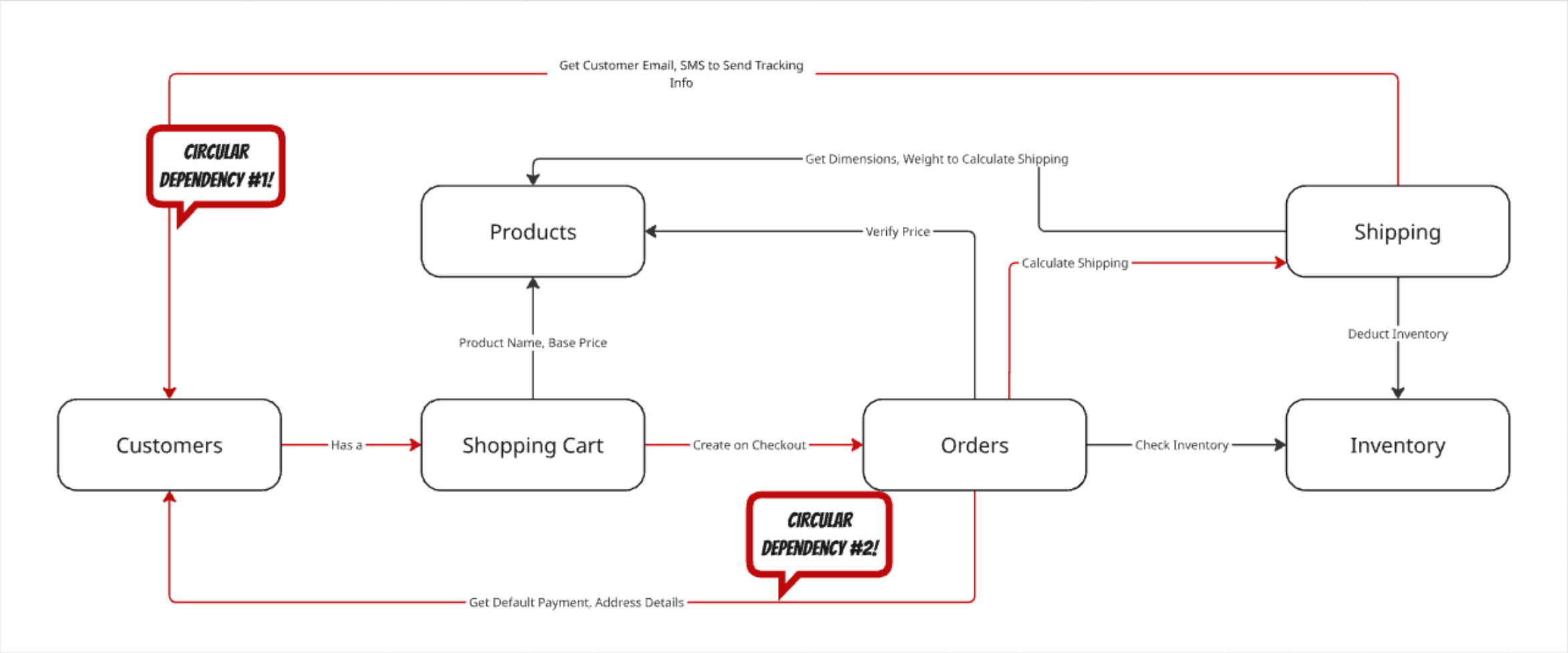
ArchUnitNet Test Example

```
[Fact]
public void DomainTypesShouldNotReferenceInfrastructure()
{
  var domainTypes = Types().That()
    .ResideInNamespace("RiverBooks.OrderProcessing.Domain.*")
    .As("OrderProcessing Domain Types");

  var infrastructureTypes = Types().That()
    .ResideInNamespace("RiverBooks.OrderProcessing.Infrastructure.*")
    .As("Infrastructure Types");

  var rule = domainTypes.Should().NotDependOnAny(infrastructureTypes);

  rule.Check(Architecture);
}
```

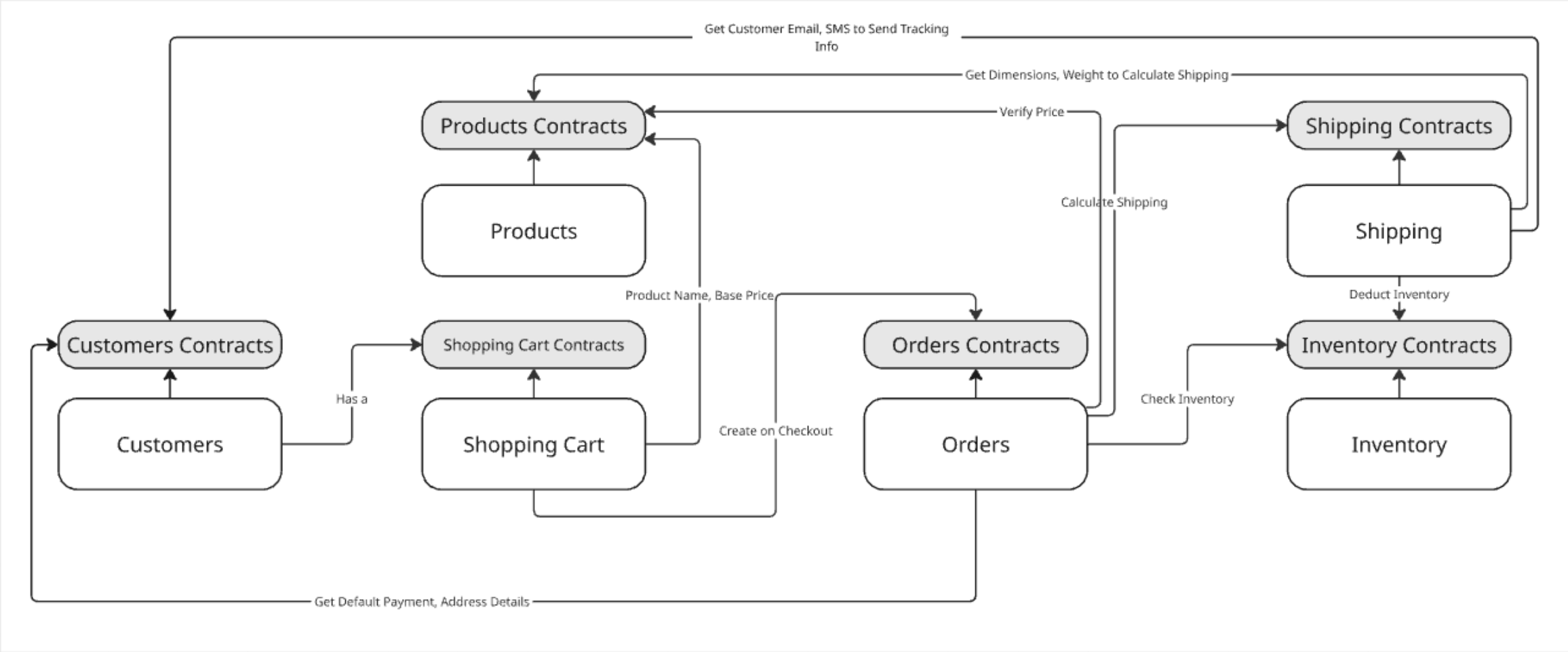
The Problem with Layered Architecture



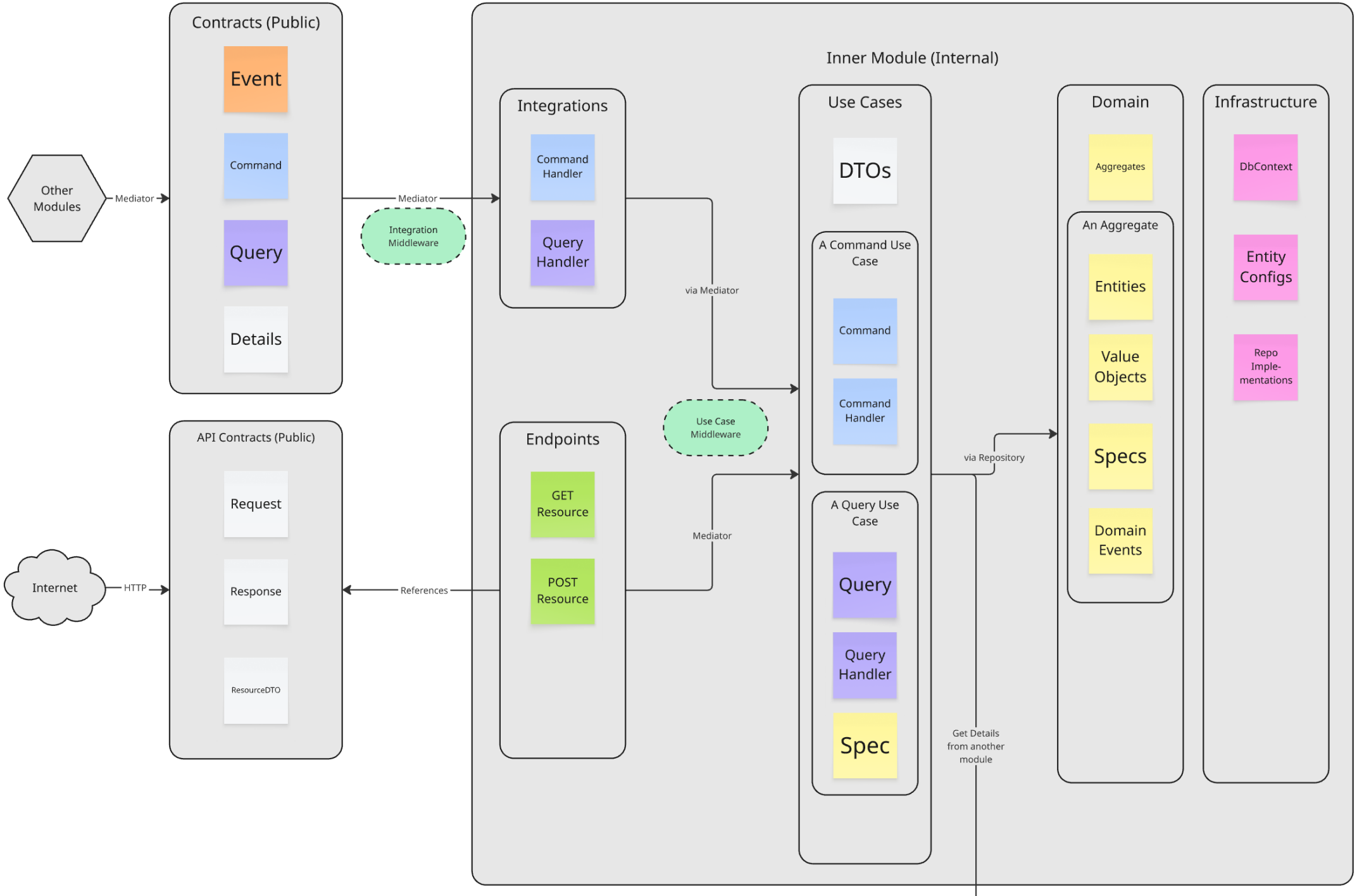
Libraries


Get Customer Email, SMS to Send Tracking Info
CIRCULAR DEPENDENCY #1!
Get Dimensions, Weight to Calculate Shipping
Products
Verify Price
Shipping
Calculate Shipping
Product Name, Base Price
Deduct Inventory
Customers
Has a
Shopping Cart
Create on Checkout
Orders
Check Inventory
Inventory
CIRCULAR DEPENDENCY #2!
Get Default Payment, Address Details

Modules (inner implementation library + public contracts library grouped together)



A Modular Monolith Module



Courses!
DomeTrain.com

bit.ly/3T1pC17

Resources