Clean Architecture with ASP.NET Core

Steve Smith

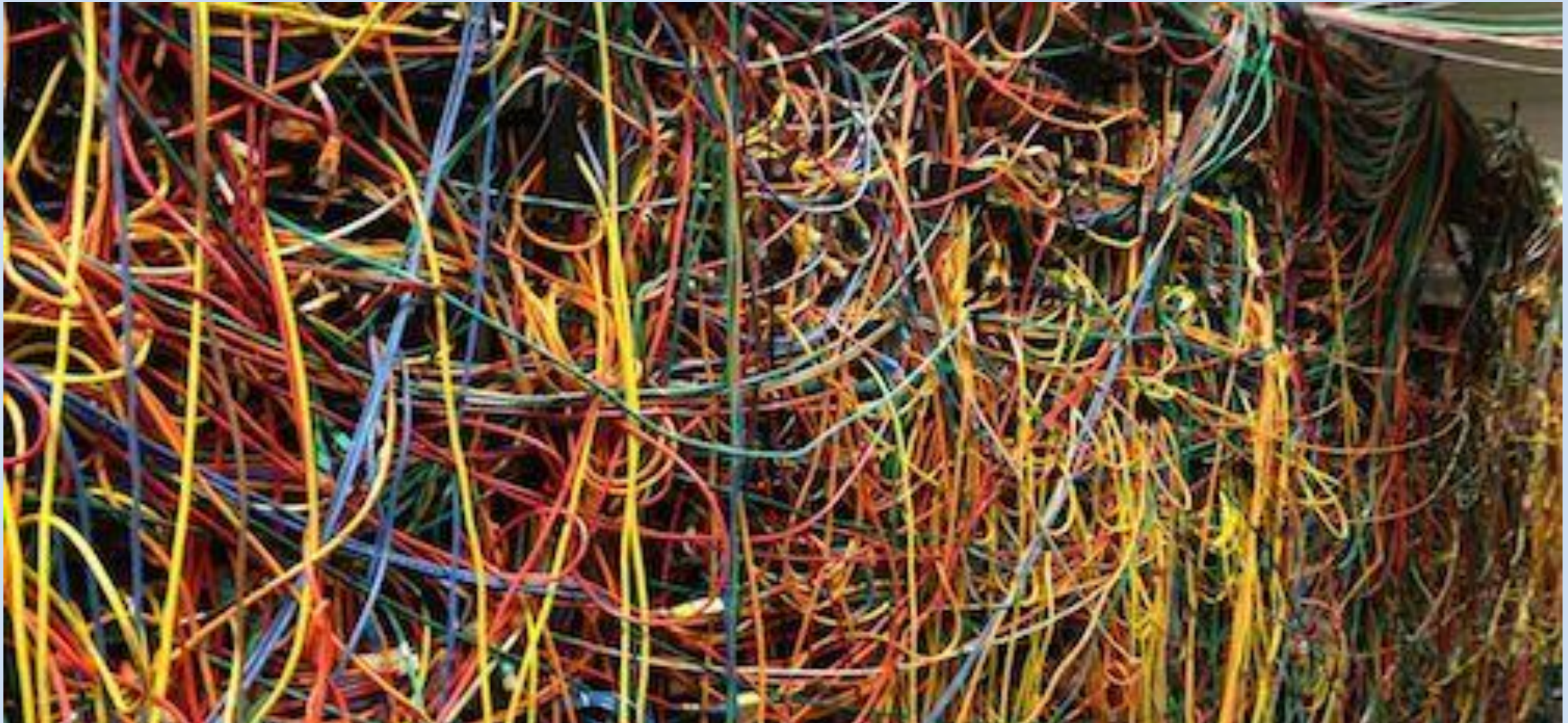
@ardalis

steve@nimblepros.com | NimblePros.com

NimblePros

Better software development. Delivered.

The Problem – Tightly Coupled Hardwired Dependencies



The Goal – Loosely Coupled Well-Organized Dependencies

Guiding Principles

Don’t take my word for it – here are several industry-accepted principles we can apply to our thinking about code organization and architecture

Separation of Concerns

Don't let your plumbing code pollute your software.

Separation of Concerns

- Avoid mixing different responsibilities in the same code structure

- Why?
  - Mixing responsibilities adds coupling between them where there should be none

- Implication
  - Not following Separation of Concerns often leads to Spaghetti Code

Spaghetti Code

Maintenance is easy with everything in one place.

The Big 3 Concerns (to keep separate)

- Data Access
- Business Rules / Domain Model
- User Interface

Single Responsibility

Avoid tightly coupling your tools together.

Single Responsibility Principle (SRP)

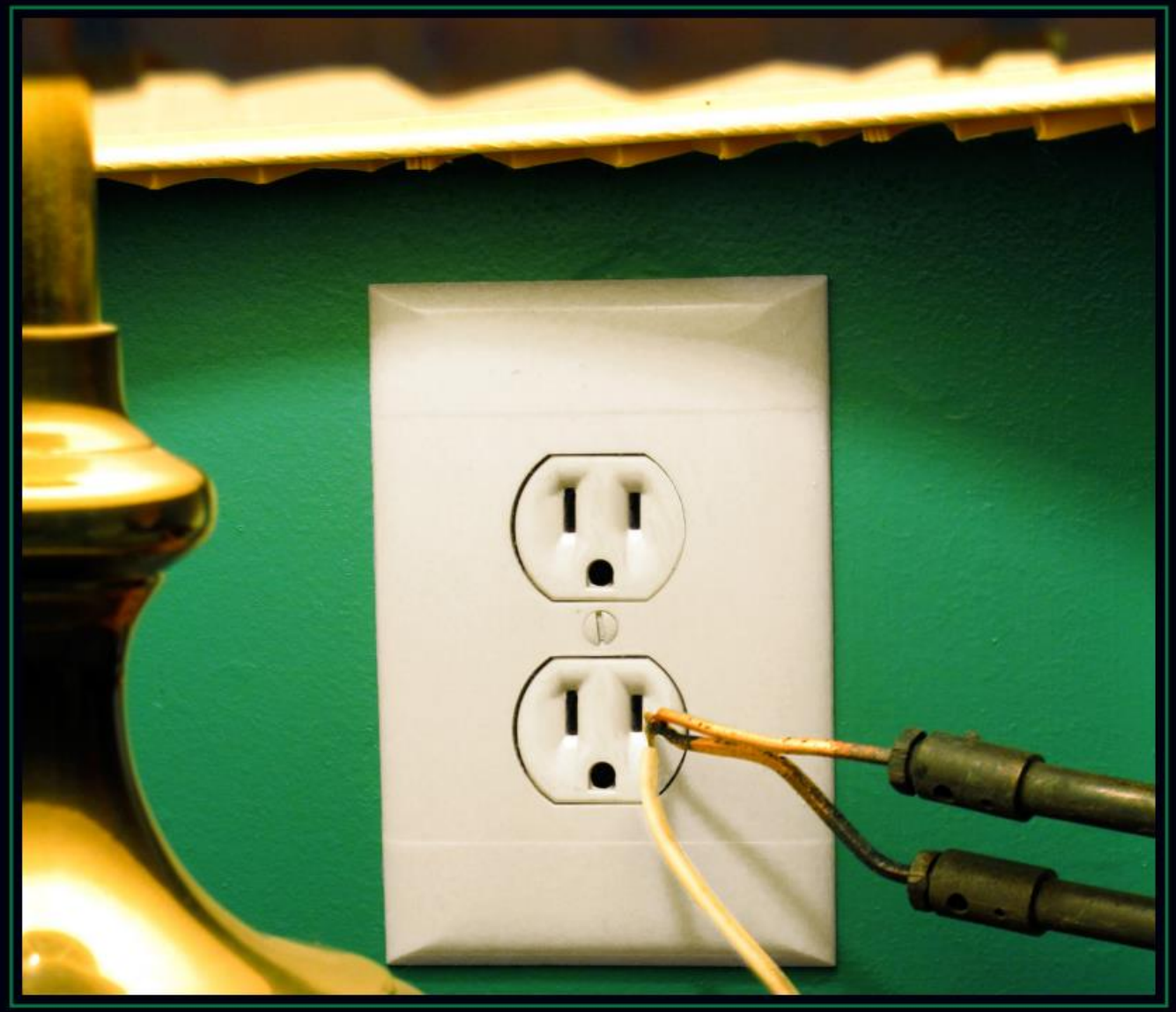
- Closely related to Separation of Concerns
- Classes should have just one single responsibility – a single reason to change

- Why?
  - Mixing responsibilities adds coupling between them where there should be none

- Implication
  - Applications will consist of more, smaller classes than otherwise

We Want to Avoid This

```
    public async Task CreateOrder(Cart cart, Customer customer)
    {

        try
        {
            Log("Starting order creation.");

             ValidateCart(cart);
             ValidateCustomer(customer);

            Order newOrder = ProcessCart(cart, customer);

            await  dbContext.Orders.AddAsync(newOrder);
            await _dbContext.SaveChangesAsync();

            await SendOrderConfirmationEmail(customer.Email);

             UpdateUI("Order created successfully.");
        }
        catch (Exception ex)
        {
            LogError("Error in CreateOrder: " + ex.Message);
             UpdateUI("An error occurred while creating the order.");
             // Additional error handling logic here
        }
    }
```



Dependency Inversion

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

Dependency Inversion Principle (DIP)

- High level modules should depend on abstractions, not low-level modules
- Low level modules should depend on shared abstractions, too
- Abstractions should not depend on details
- Details should instead depend on abstractions

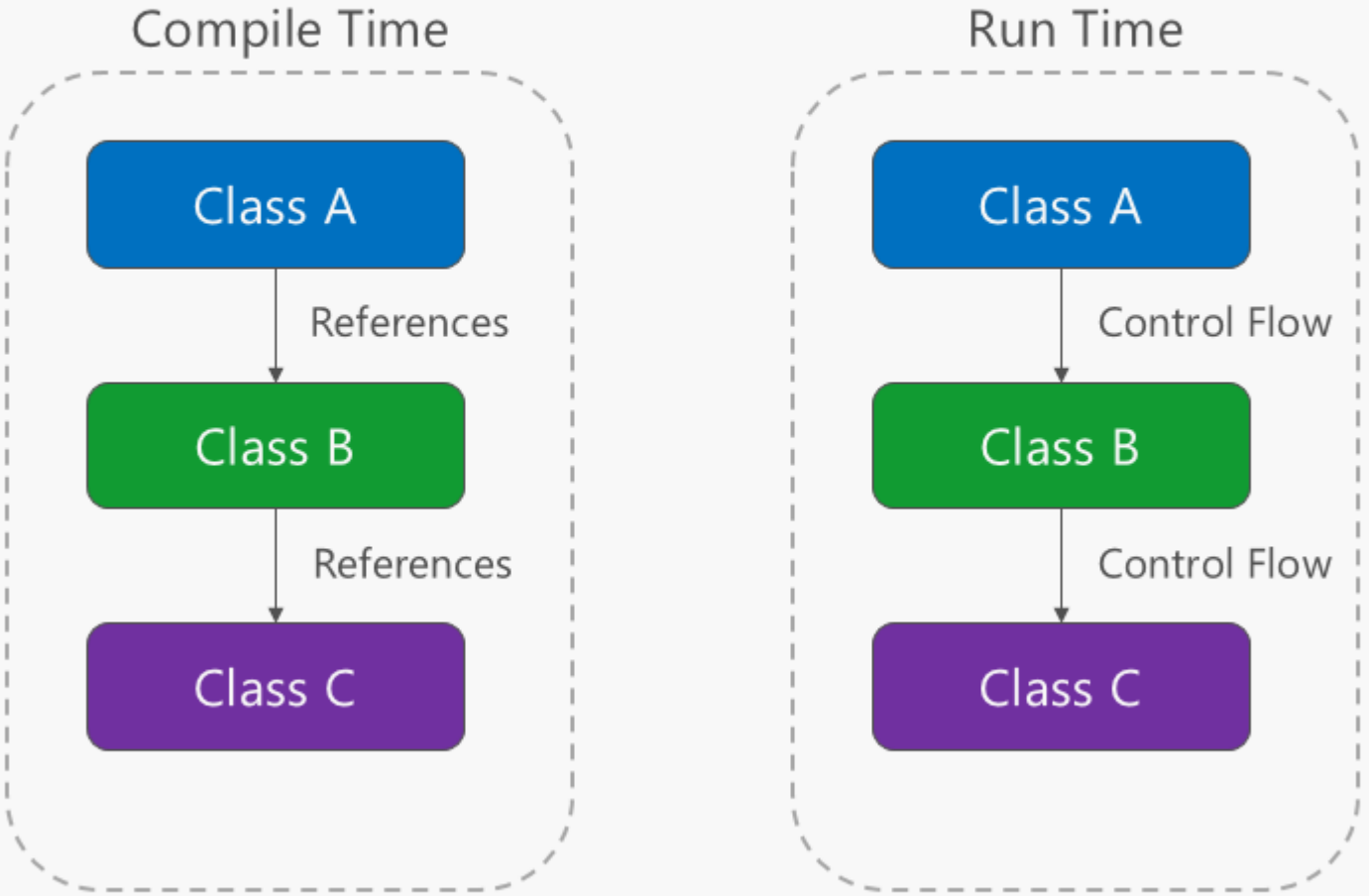
- Why?
  - Dependencies that flow toward low-level details result in code that is coupled to infrastructure concerns
- Implication
  - Applications should be organized so dependency flows toward abstractions and business logic, not away from them

Explicit Dependencies Principle

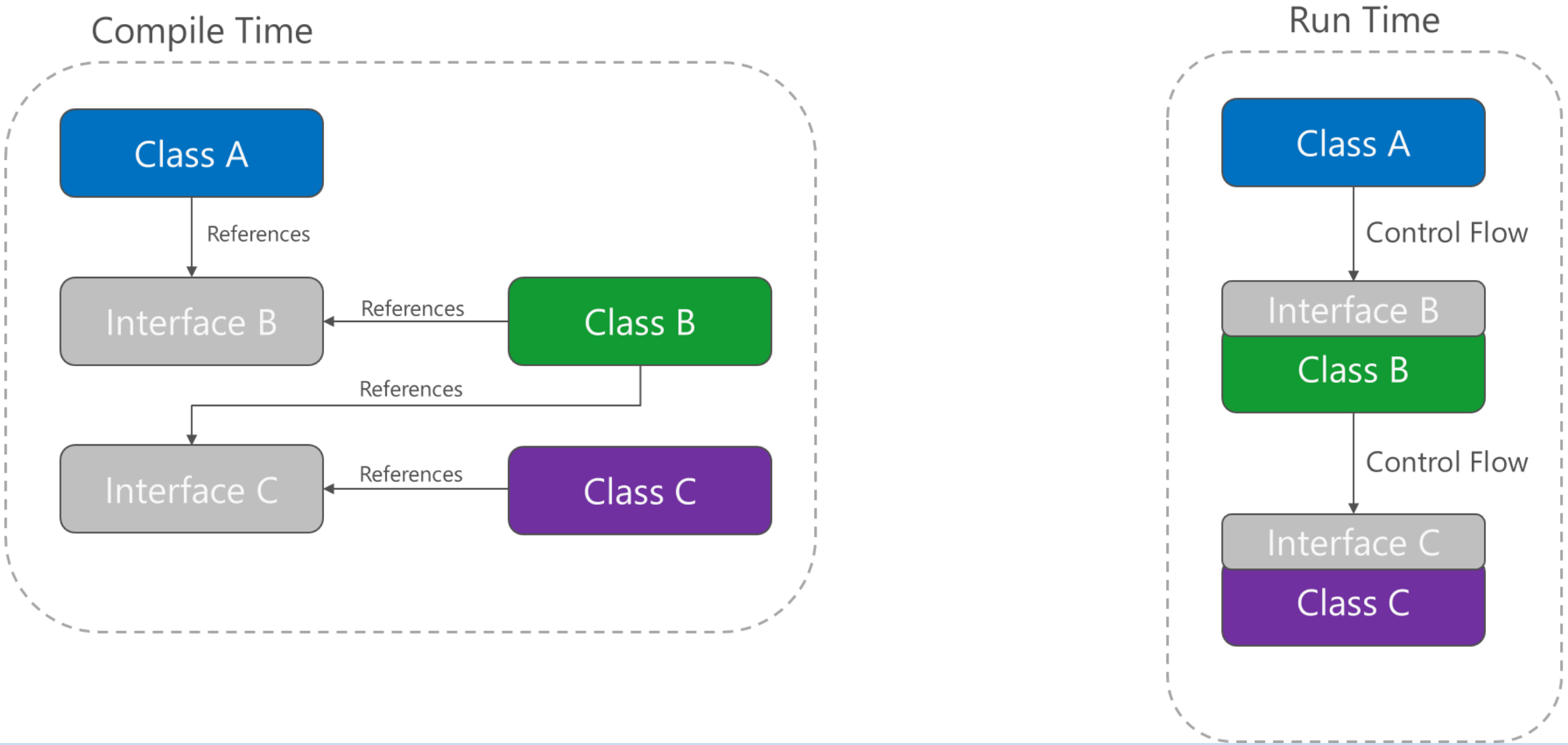
- Classes should request all dependencies via their constructor
- Make types honest, not deceptive
- Think of a class as if it were a cooking recipe
  - The constructor arguments are the required ingredients
  - Don’t surprise people trying to follow your recipe!

https://deviq.com/principles/explicit-dependencies-principle

Direct Dependency Graph



Inverted Dependency Graph



Make the right thing easy and the wrong thing hard

Force developers into the “pit of success!”

Otherwise, they may wind up in…


The pit of despair!
imgflip.com

Make the right thing easy and the wrong thing hard

UI Classes should not depend directly on Infrastructure classes

- How can our solution structure help enforce this?

Make the right thing easy and the wrong thing hard

Business logic and domain models should not depend directly on Infrastructure classes

- How can our solution structure help enforce this?

Make the right thing easy and the wrong thing hard

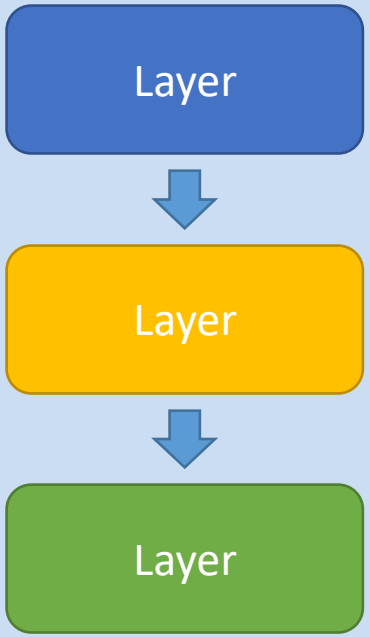
Less total code and less code repetition leads to fewer bugs and greater consistency

- How can our code organization help achieve this?

Make the right thing easy and the wrong thing hard

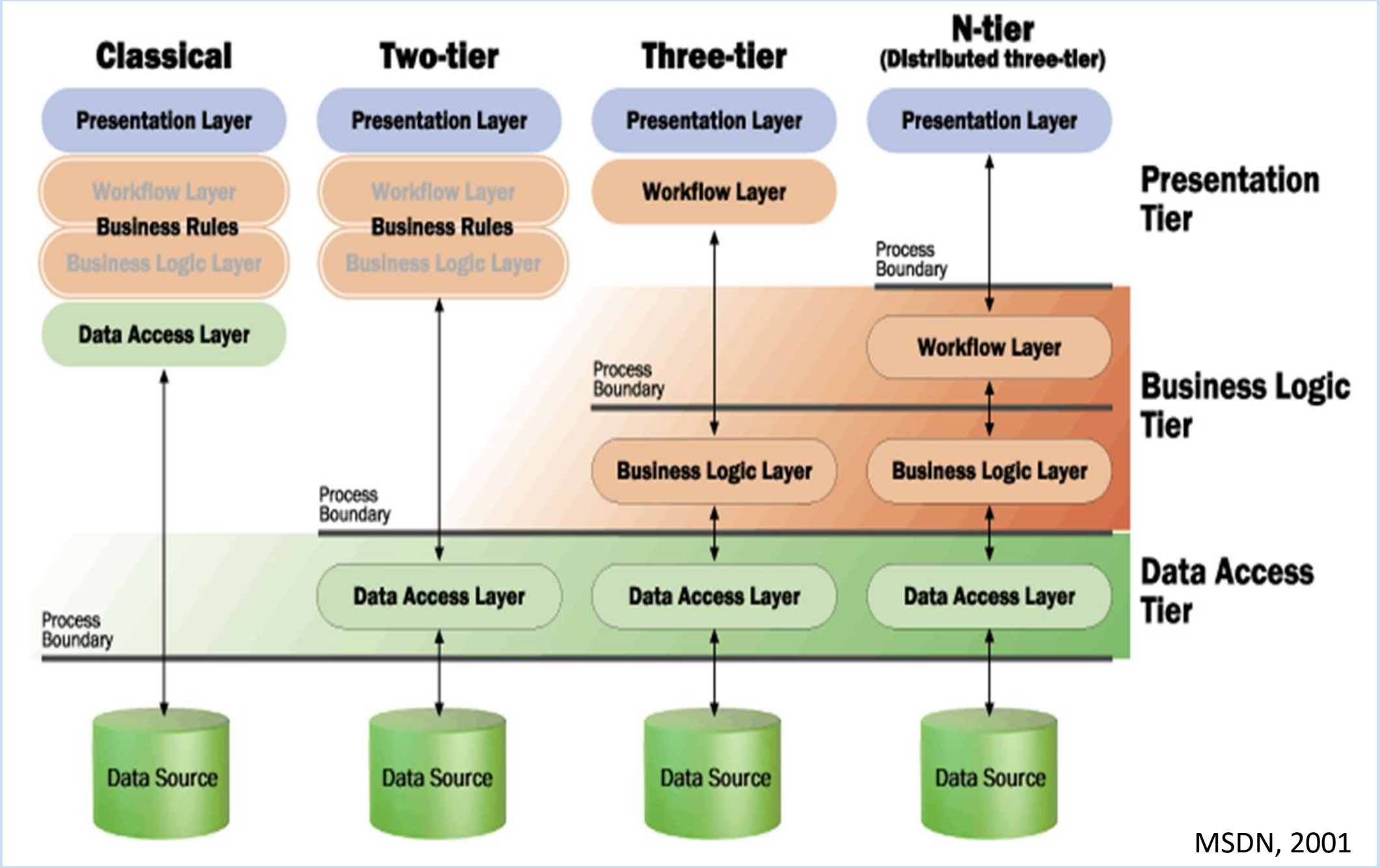
LINQ is great, but LINQ everywhere often means data access logic and/or business logic everywhere

- What patterns can we use to help tame repetitive use of LINQ everywhere in our application?

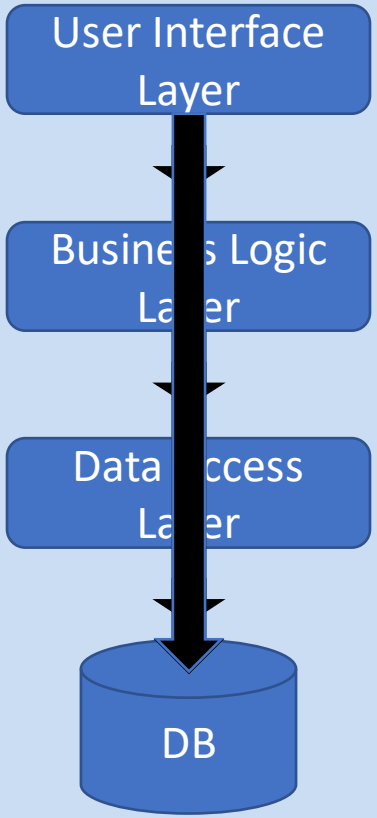


“Classic” N-Tier Architecture

Also referred to as “N-Layer Architecture”


Classical
Two-tier
Three-tier
N-tier
(Distributed three-tier)
Presentation Layer
Workflow Layer
Business Rules
Business Logic Layer
Data Access Layer
Process Boundary
Presentation Tier
Business Logic Tier
Data Access Tier
Data Source
MSDN, 2001

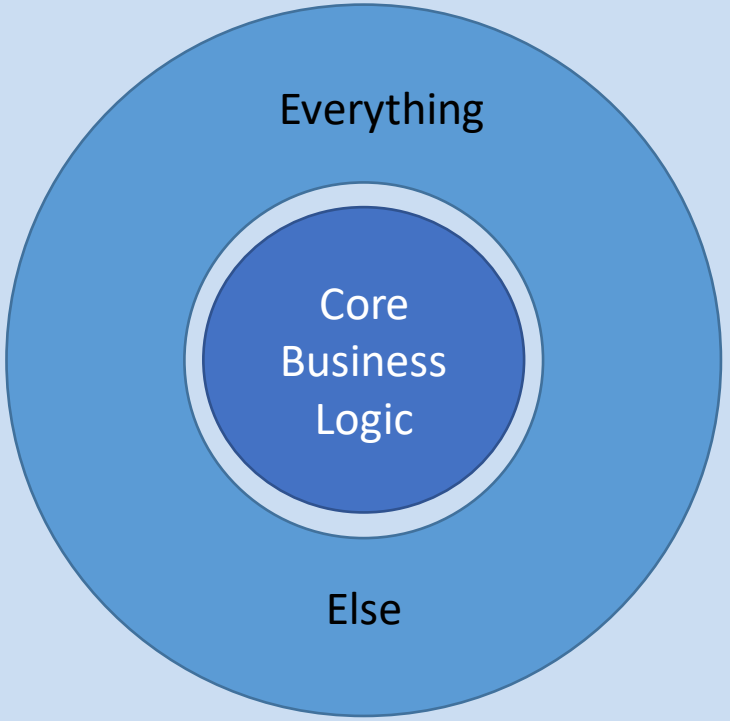
Dependencies are Transitive



Everything

Depends on the *database*


IVE JUST SUCKED ONE YEAR OF
YOUR LIFE AWAY.



Domain-Centric Design

Focus on the domain model and business logic, not infrastructure

Core Business Logic

- The domain model

- Abstractions and interfaces for all required infrastructure dependencies
  - Infrastructure adapters implement these interfaces
  - UI constructs consume these interfaces via dependency injection

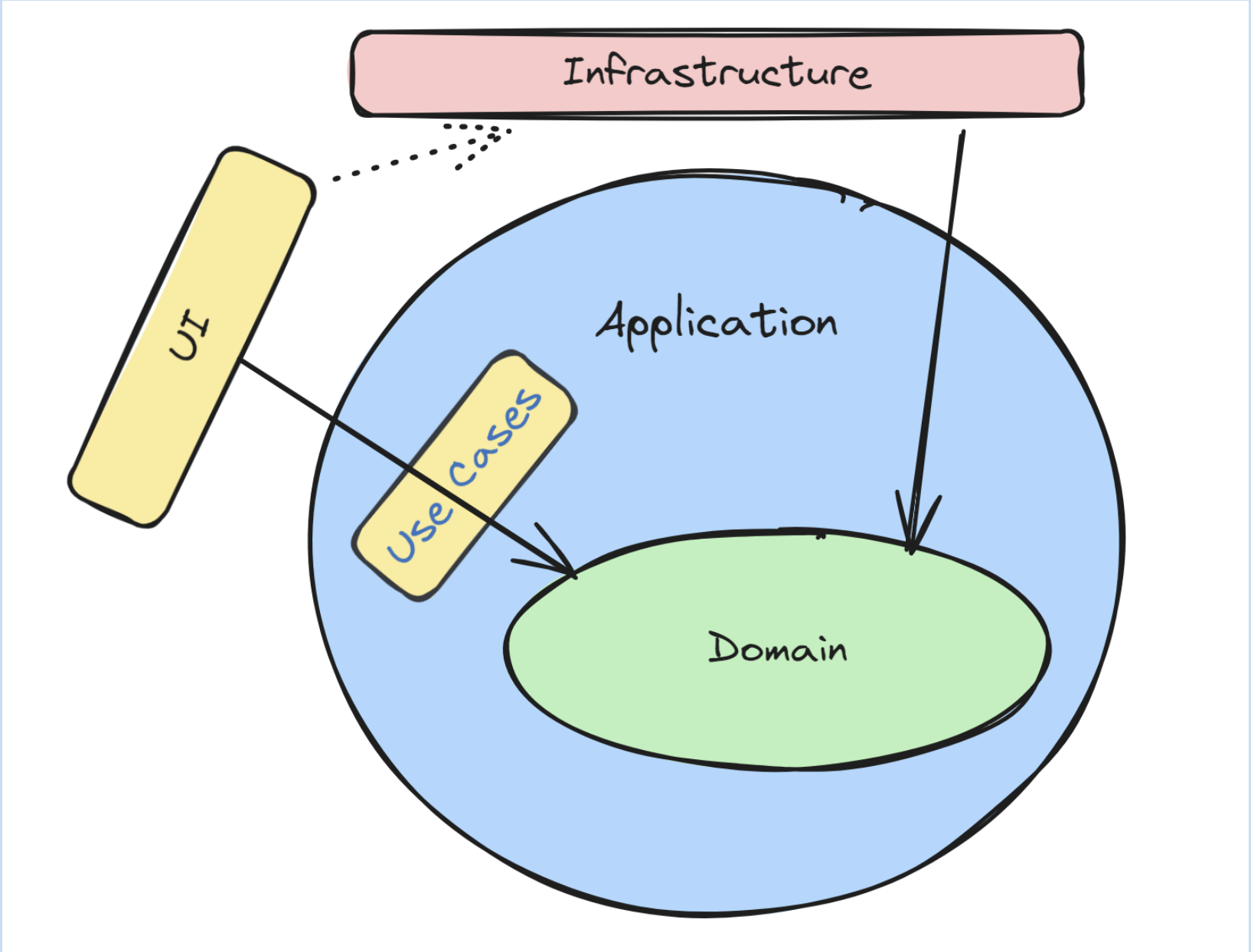
App Logic – Features or Use Cases

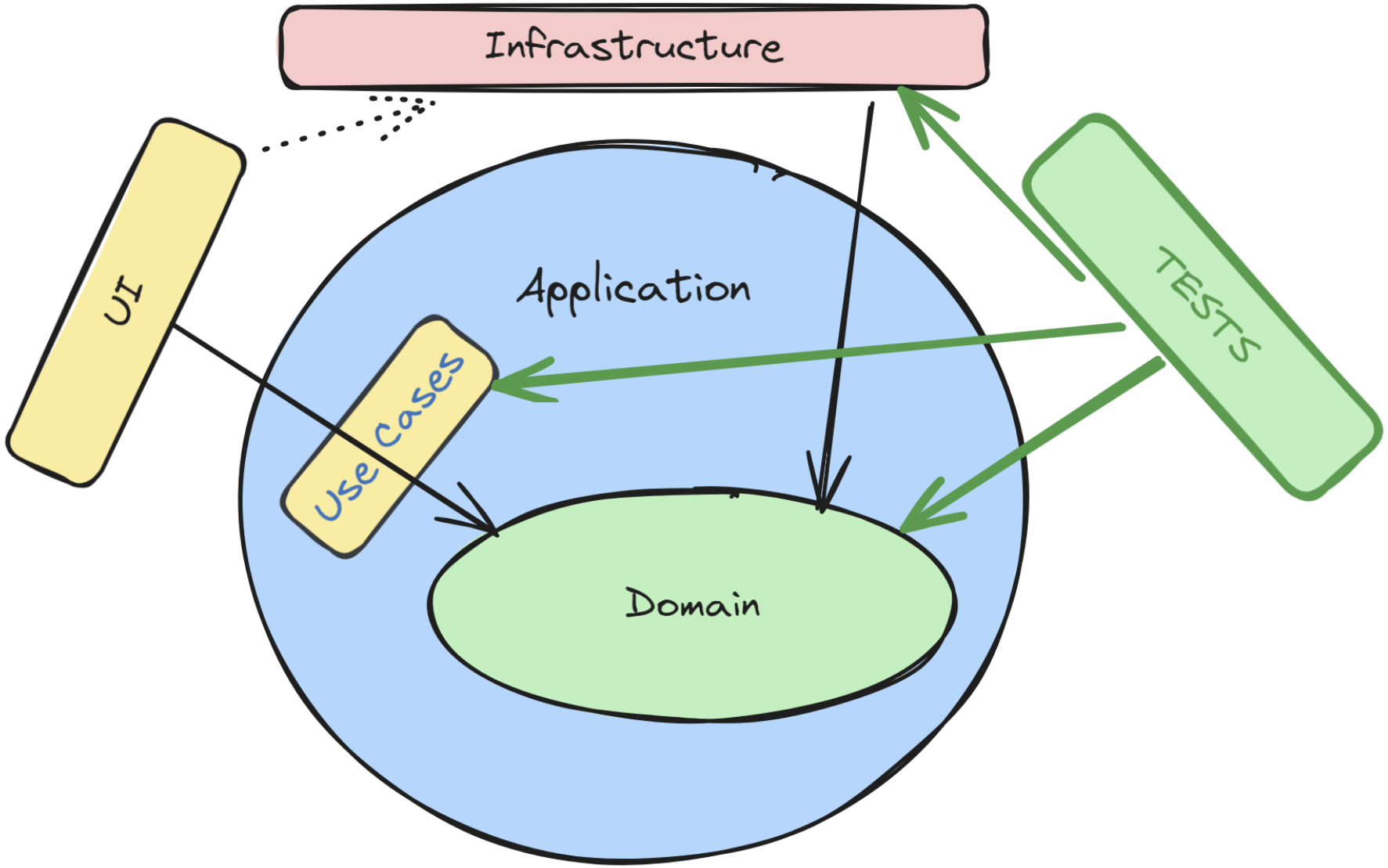
- Follow CQRS: Command/Query Responsibility Segregation
- Optional – many apps can skip this additional project/layer – do it in UI
- The commands
  - Load, create, and/or delete domain model type(s) (via Repository pattern)
  - Call methods on model object instances
  - Persist changes (via Repository pattern)
- The queries
  - Define a query service abstraction
  - Which returns custom DTOs representing the results
  - And are implemented in Infrastructure
  - Using what data access tech is most appropriate

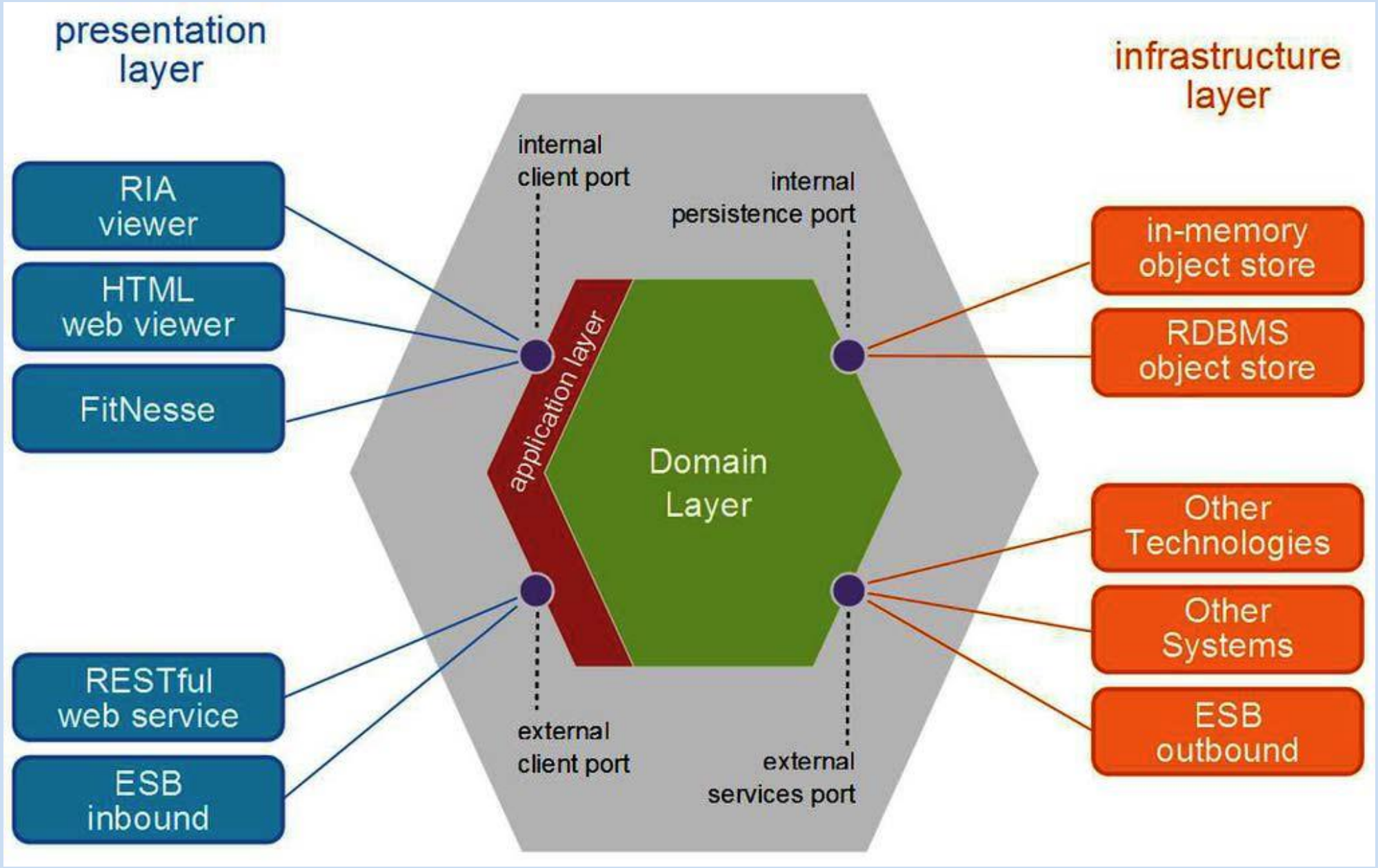
Typically do not use domain model

Clean Architecture

- a.k.a. **Onion Architecture**
  - (2009, Jeffrey Palermo)
- a.k.a. **Hexagonal Architecture**
  - (2005, Alistair Cockburn)
- a.k.a. **Ports and Adapters**
  - (probably the clearest name, 2005, Alistair Cockburn)
- Clean Architecture was introduced by Robert C. Martin in 2012


Infrastructure
UI
Application
Use Cases
Domain


Infrastructure
UI
Application
Use Cases
Domain
TESTS


presentation layer
infrastructure layer
RIA viewer
HTML web viewer
FitNesse
RESTful web service
ESB inbound
internal client port
internal persistence port
application layer
Domain Layer
external client port
external services port
in-memory object store
RDBMS object store
Other Technologies
Other Systems
ESB outbound

Rules of Clean Architecture

The Core of the Solution holds the domain model (and all business logic).

Rules of Clean Architecture

The Core of the Solution does not depend on external dependencies.

Rules of Clean Architecture

The Core of the Solution defines abstractions/interfaces, which are implemented in Core or Infrastructure

Rules of Clean Architecture

Avoid direct references to Infrastructure project types.

Exceptions:

App's Composition Root (startup DI wireup)

Integration/Functional Tests

Organizing ASP.NET Core Apps into Clean Architecture Solutions

The Core Project (domain model)

Interfaces

Entities

Value Objects

Aggregates

Domain Services

Exceptions

Domain Events

Event Handlers

Specifications

The UseCases Project (app model - optional)

Query Interfaces

DTOs

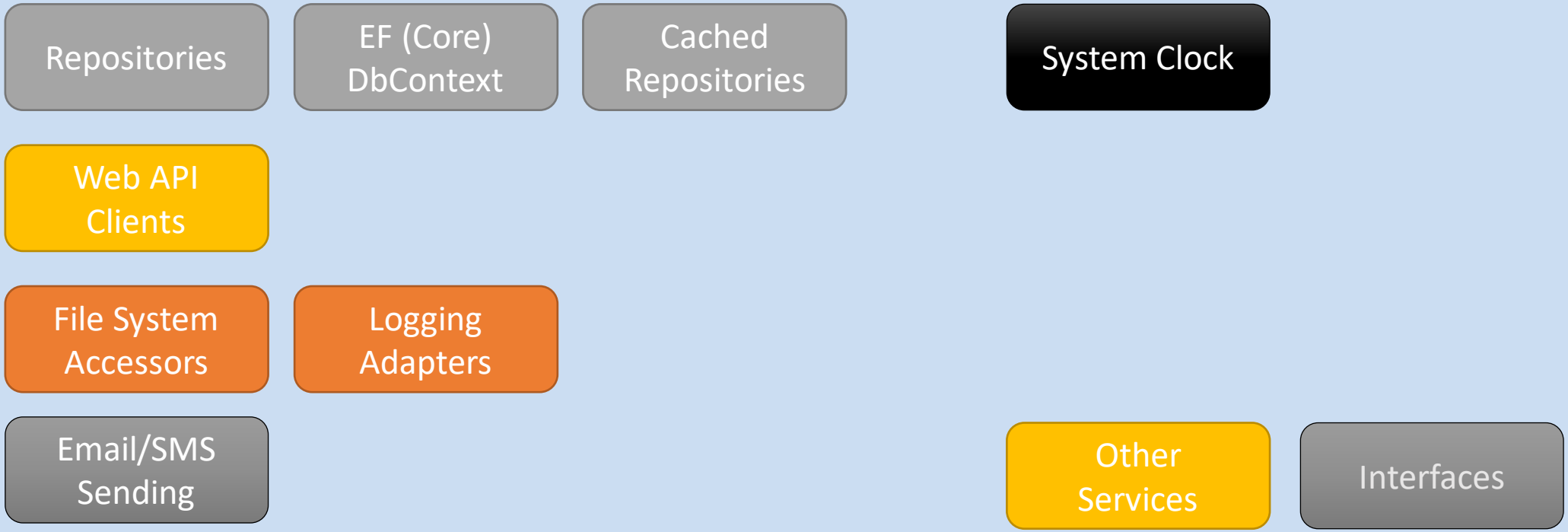
Queries

Query Handlers

Commands

Command Handlers

The Infrastructure Project (dependencies)



The Web Project (dependencies)

Controllers

API Endpoints

Views

Or

Razor Pages

ViewModels

ApiModels

BindingModels

Filters

Binders

Tag/Html Helpers

Other Services

Interfaces

Sharing Common Code Between Solutions

- Common types may be shared between solutions.
- Domain-Driven Design refers to this class library as a **Shared Kernel**.
- Shared Kernel is ideally distributed as a NuGet Package.

The Shared Kernel Package

Base Entity

Base Domain Event

Base Specification

Common Exceptions

Common Interfaces

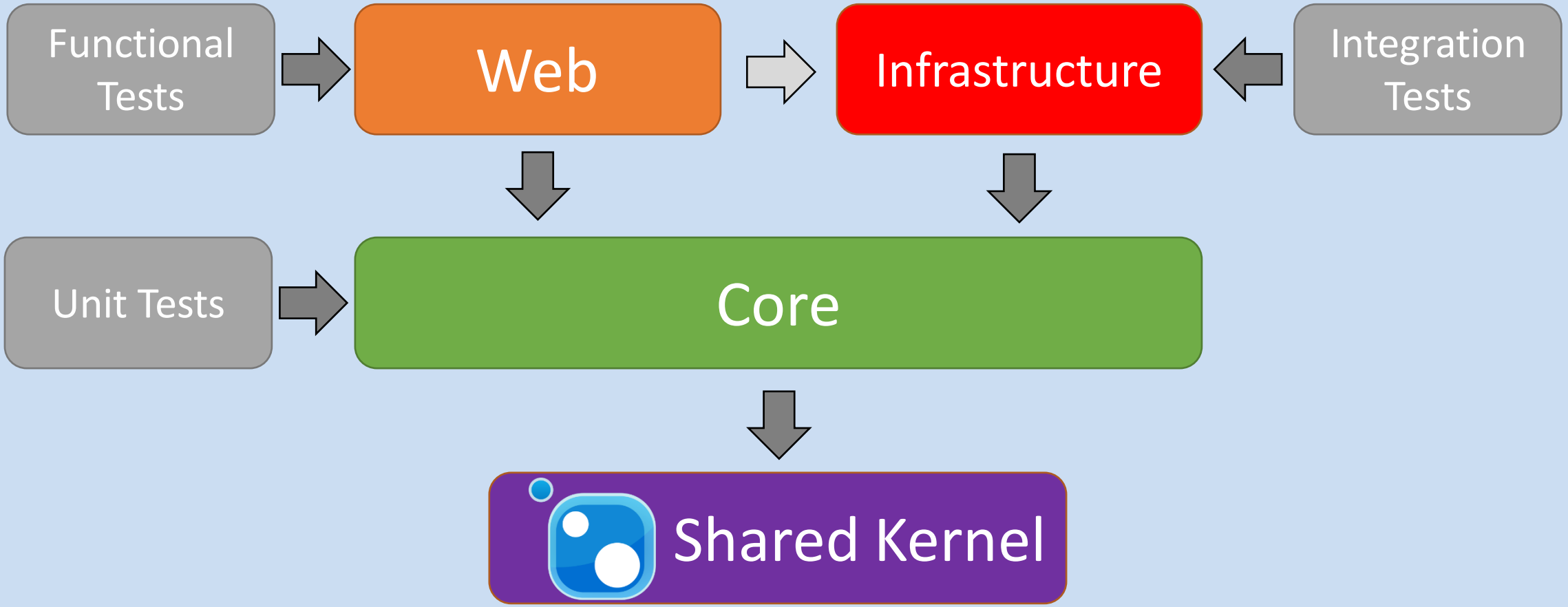
Common Auth e.g. User class

Common DI

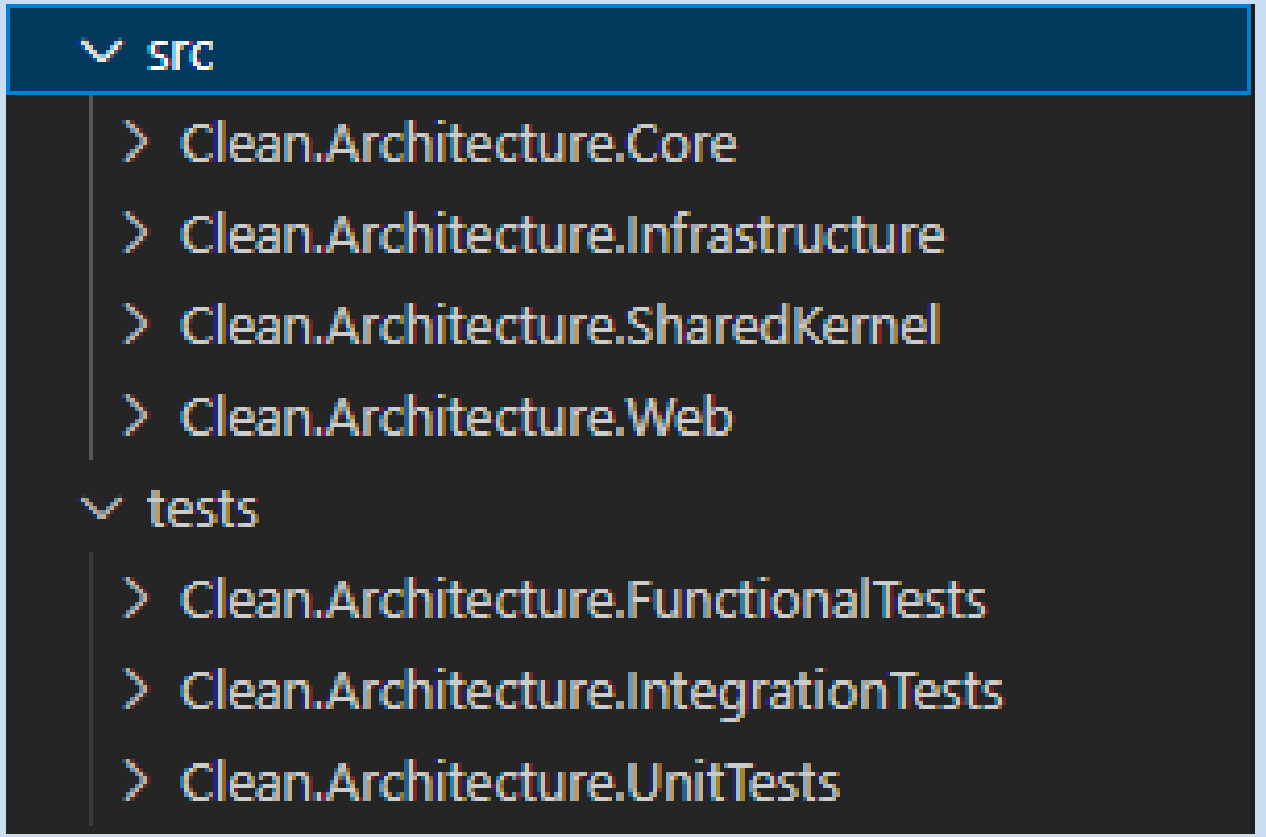
Common Logging

Common Guard Clauses

Overall Dependency Relationship



Overall Solution Structure



But I Hate Multi-Project Solutions!

- You can still follow these principles
- You lose compile-time checks
- You can still use ArchUnit.NET or similar to try to enforce where certain types should/should not be used

Clean Architecture Features

The key benefits of using a Clean Architecture approach.

Clean Architecture Features

Compile-time Dependency Policy

- Separate projects ensure developers cannot add dependencies creating cycles (e.g. Core -> Infrastructure -> Core)

Clean Architecture Features

Framework Independent

- Works with ASP.NET, ASP.NET Core, Worker Services, Java, etc.
- No reliance on proprietary codebase or software libraries

Clean Architecture Features

Database Independent

- Minimizes code with knowledge of data storage choices
- No dependency on any particular database or data access library

Clean Architecture Features

Modular

- Easily supports multiple adapters implementing any abstraction
- Allows use of separate service implementations in separate deployment environments (local, dev, test, stage, prod, etc)

Clean Architecture Features

Testable

- Business/domain logic is easily unit tested
- All other modules can be unit or integration tested depending on how modules are composed for each test scenario

Clean Architecture Drawbacks

Everything has tradeoffs.

Clean Architecture Drawbacks

- Learning curve
  - Use of abstractions and DI can make code execution paths less obvious
- More code
  - Use of multiple projects and interfaces/adapters adds to overall code and size of solution (at least at first)
  - *"It's too many projects!"* It's literally **3 projects**. (Give me a break.)

Recommendation

- Best to use for non-trivial, non-CRUD apps
- Works well with Domain-Driven Design

Clean Architecture Template

Using the dotnet CLI template

First, install the template from NuGet:

```
dotnet new -i Ardalis.CleanArchitecture.Template
```

You should see the template in the list of templates from `dotnet new` after this install successfully. Look for "Steve Smith Clean Architecture" with Short Name of "clean-arch".

Navigate to the directory where you will put the new solution.

Run this command to create the solution structure in a subfolder name `Your.ProjectName`:

```
dotnet new clean-arch -o Your.ProjectName
```

The `Your.ProjectName` directory and solution file will be created, and inside that will be all of your new solution contents, properly namespaced and ready to run/test!

Code Walkthrough

Summary

- Separate concerns by project, and class
- Invert dependencies, especially on infrastructure concerns
- Core project holds abstractions and business logic, zero dependencies
- Infrastructure holds adapters for abstractions; references Core
- UI consumes abstractions and domain model; no direct use of Infrastructure types
- Shared Kernel holds concepts shared between apps/solutions

Stickers!


NimblePros™
NimblePros™
NimblePros™
NimblePros™
NimblePros™

Resources

- Find me at ardalis.com or @ardalis
- Template: https://github.com/ardalis/cleanarchitecture
- Reference App: https://github.com/dotnet-architecture/eShopOnWeb

- Team Training/Mentoring: https://NimblePros.com
- Individual Developer Career Coaching: https://devBetter.com