Applying Clean Architecture to ASP.NET Core Apps

STEVE SMITH

ARDALIS.COM | @ARDALIS | STEVE@ARDALIS.COM

MENTOR | TRAINER | COACH | FORCE MULTIPLIER

Learn More After Today

1) Pluralsight

- N-Tier Apps with C# https://www.pluralsight.com/courses/n-tier-apps-part1
- Domain-Driven Design Fundamentals https://www.pluralsight.com/courses/domain-driven-design-fundamentals

2) Microsoft FREE eBook/Sample App

- eShopOnWeb eCommerce Sample https://ardalis.com/architecture-ebook

Microsoft
Architecting Modern Web Applications with ASP.NET Core and Microsoft Azure
Steve Smith

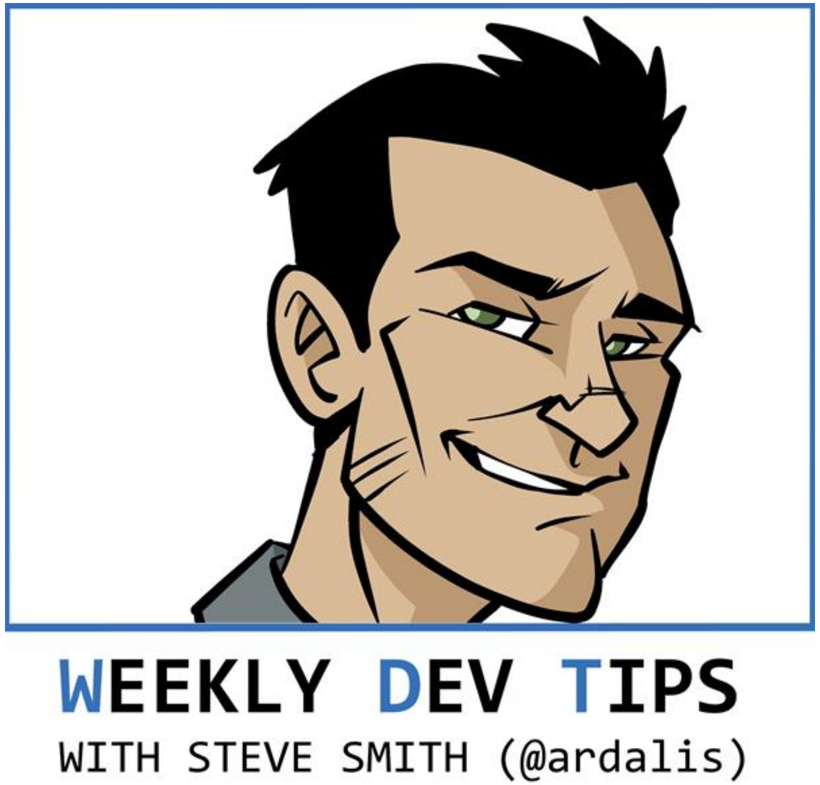
3) Contact me for mentoring/training for your company/team

- Developer Career Mentoring at **devBetter.com**

Weekly Dev Tips

Podcast and Newsletter

- Ardalis.com/tips
- WeeklyDevTips.com
- (I have stickers if you're into that)
- Streaming at twitch.tv/ardalis Fridays



Questions

HOPEFULLY YOU'LL KNOW THE ANSWERS WHEN WE'RE DONE

Why do we separate applications into multiple projects?

What are some principles we can apply when organizing our software modules?

How does the organization of our application’s solution impact coupling?

What problems result from certain common approaches?

How does Clean Architecture address these problems?

How does ASP.NET Core help?

Principles

A BIT OF GUIDANCE

Separation of Concerns

Don’t let your plumbing code pollute your software.

Separation of Concerns

Avoid mixing different code responsibilities in the same (method | class | project)

Separation of Concerns

The Big Three™

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

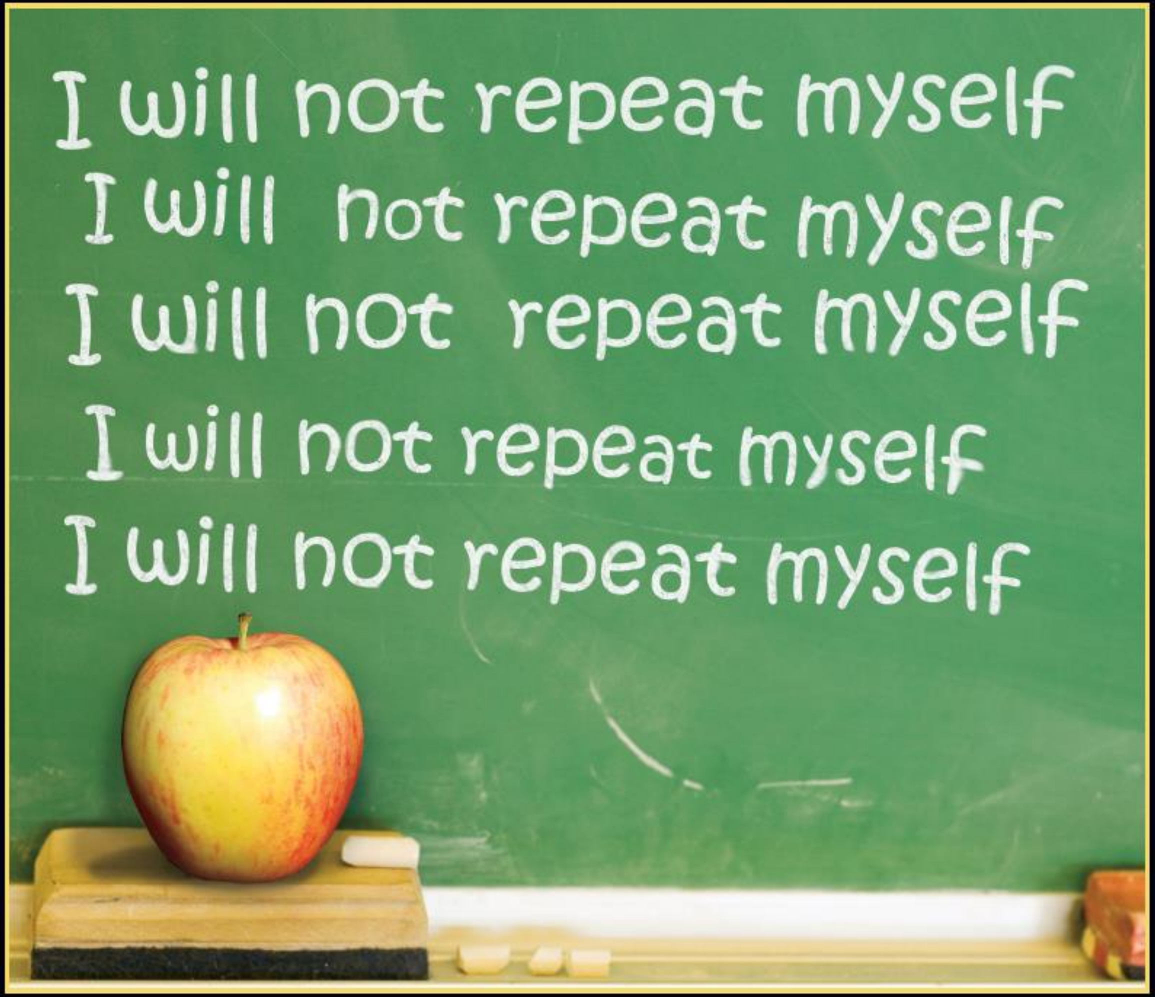
SINGLE RESPONSIBILITY

Avoid tightly coupling your tools together.

Single Responsibility

Works in tandem with Separation of Concerns

Classes should focus on a single responsibility – a single reason to change.

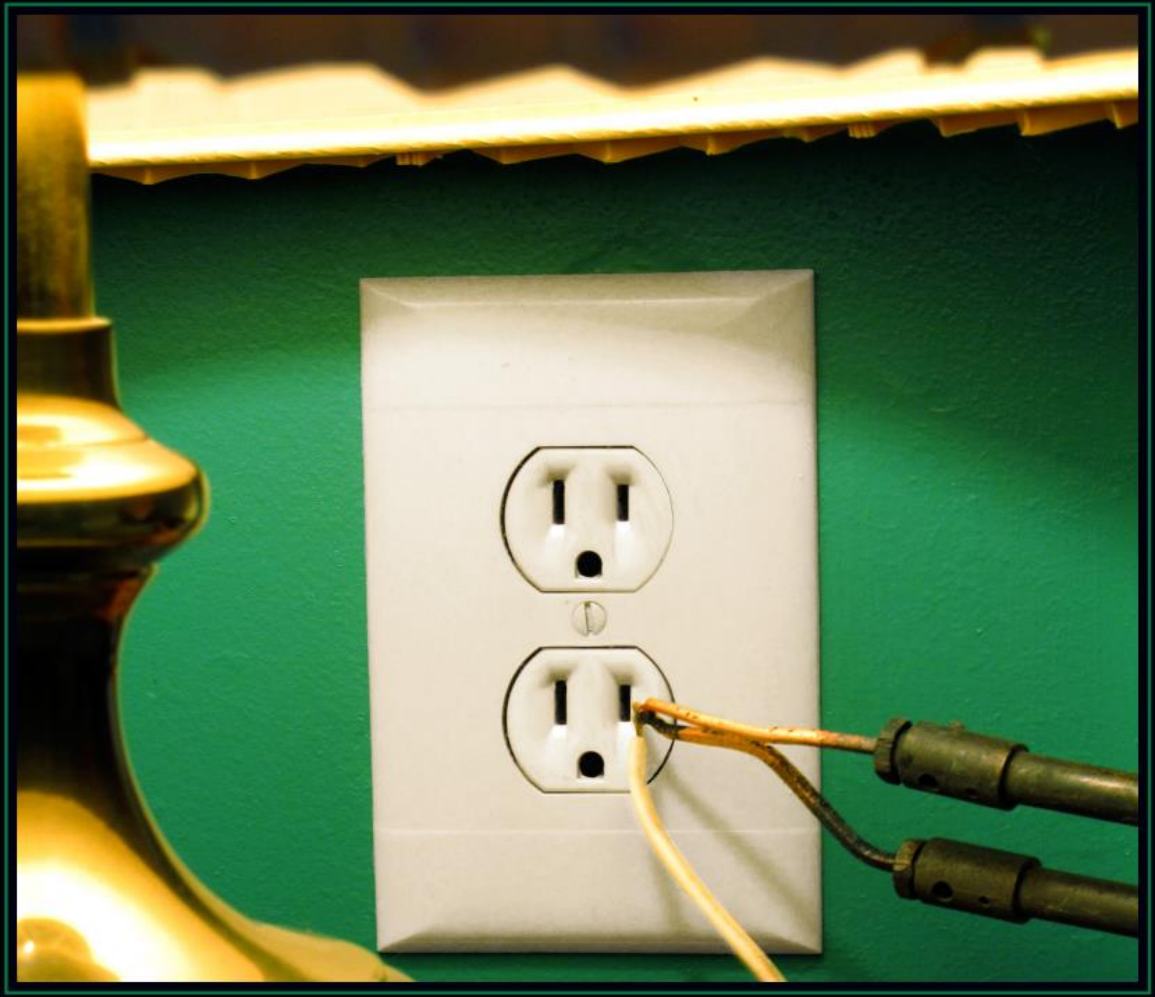


DON'T REPEAT YOURSELF

Repetition is the root of all software evil.

Following Don’t Repeat Yourself…

- Refactor *repetitive code* into functions
- Group functions into cohesive classes
- Group classes into folders and namespaces by
  - Responsibility
  - Level of abstraction
  - Etc.
- Further group class folders into projects



DEPENDENCY INVERSION

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

Invert (and inject) Dependencies

Both high level classes and implementation-detail classes should depend on abstractions (interfaces).

Invert (and inject) Dependencies

Classes should follow Explicit Dependencies Principle:

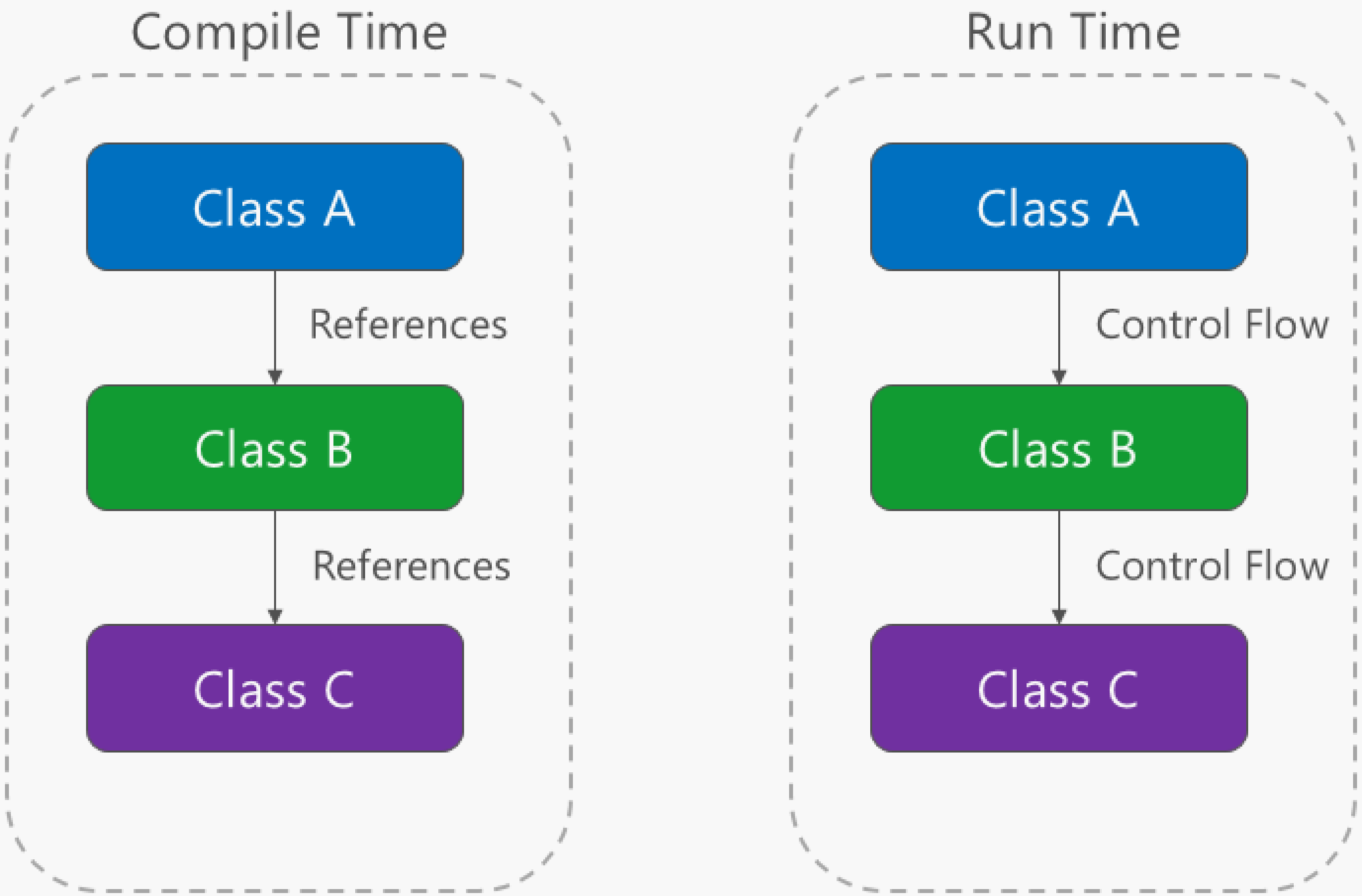
- Request **all** dependencies via their constructor
- Make your types honest, not deceptive

Invert (and inject) Dependencies

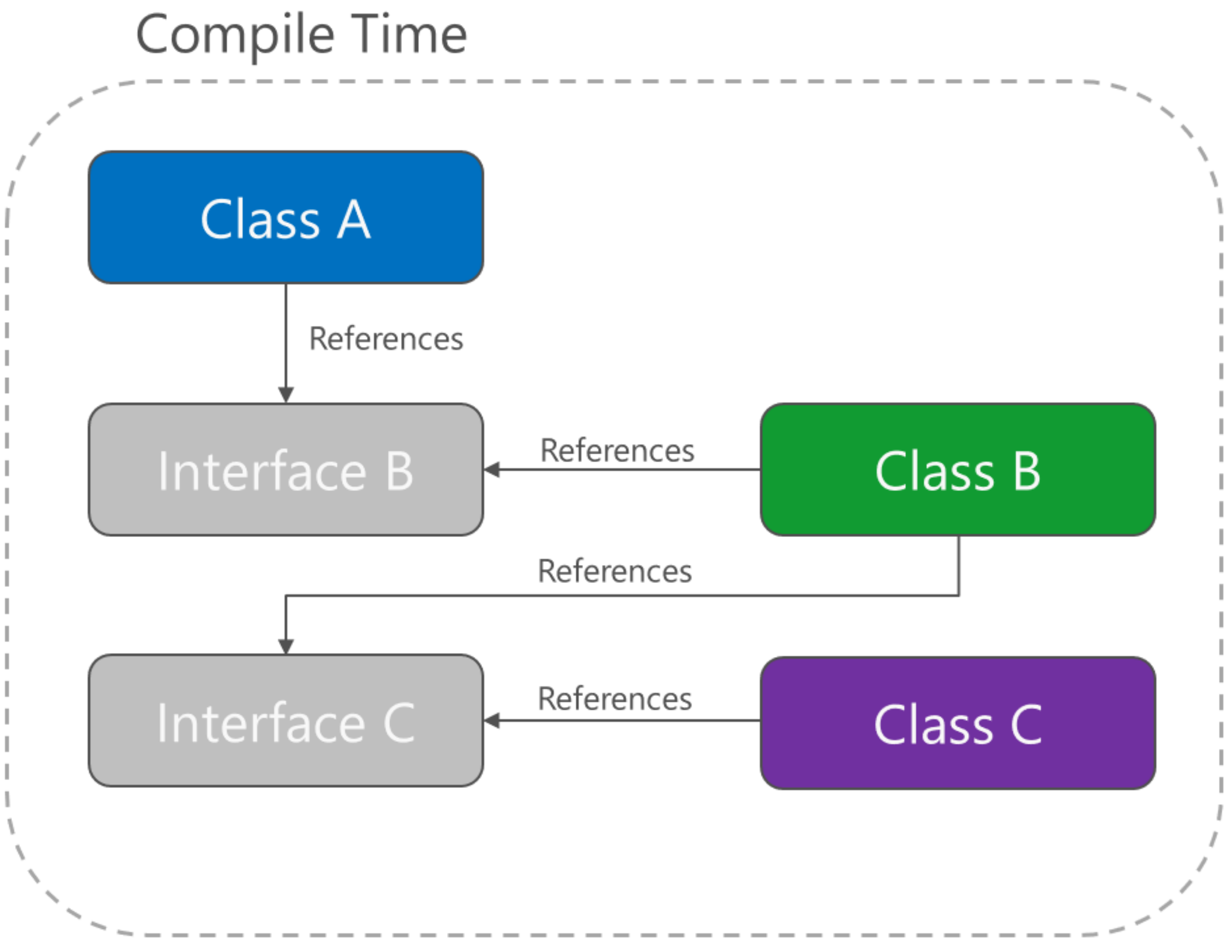
Corollary: **Abstractions/interfaces must be defined somewhere accessible by**:

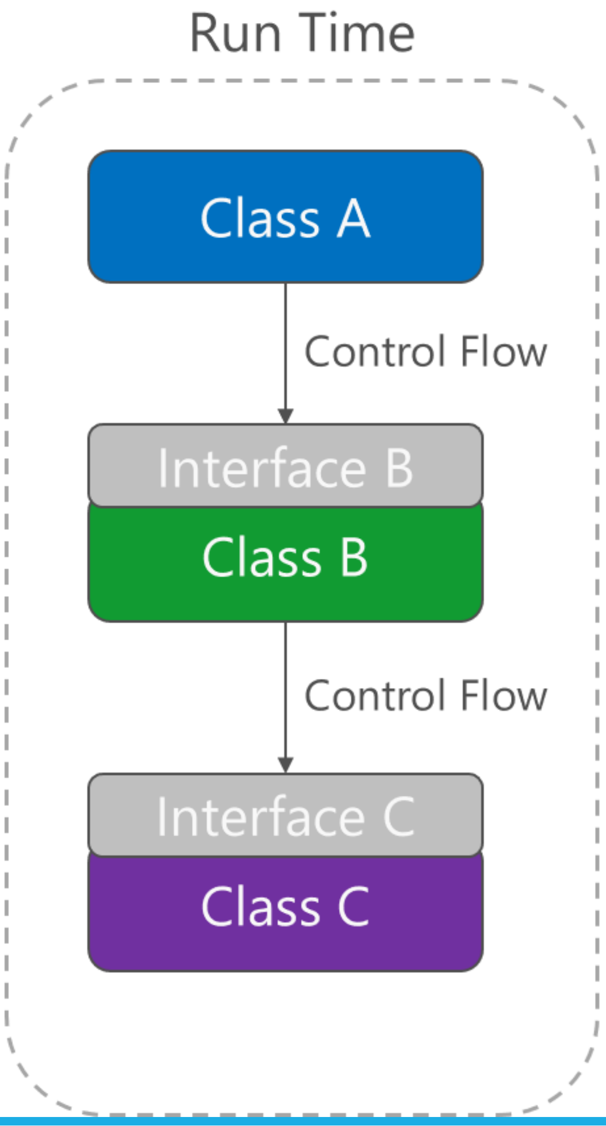
- Low level implementation services
- High level business services
- User interface entry points

Direct Dependency Graph



Inverted Dependency Graph





Make the right thing easy and the wrong thing hard

FORCE DEVELOPERS INTO A "PIT OF SUCCESS"

Make the right thing easy and the wrong thing hard.

UI classes shouldn't depend directly on infrastructure classes

- How can we structure our solution to help enforce this?

Make the right thing easy and the wrong thing hard.

Business/domain classes shouldn’t depend on infrastructure classes

- How can our solution design help?

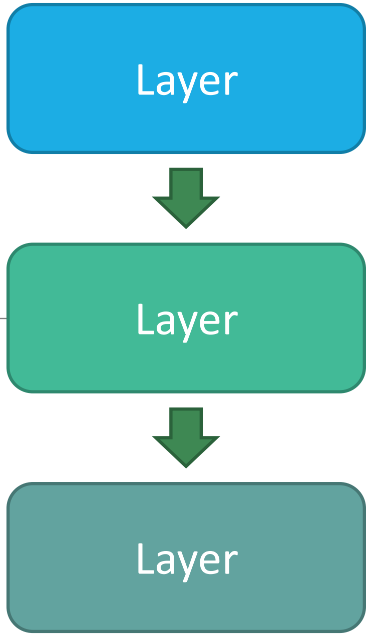
Make the right thing easy and the wrong thing hard.

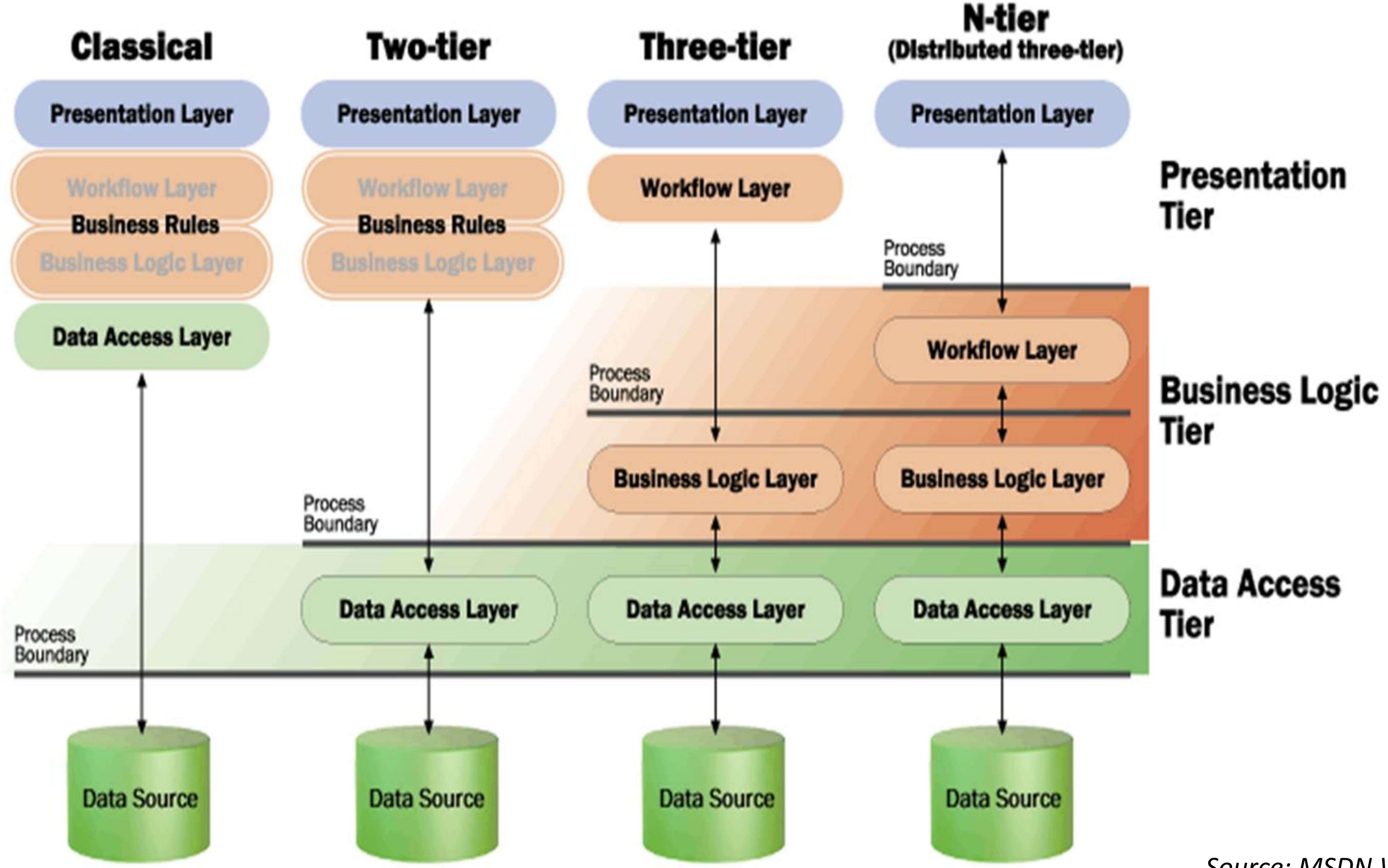
Repetition of (query logic, validation logic, policies, error handling, anything) is a problem

- What patterns can we apply to make avoiding repetition easier than copy/pasting?

“Classic” N-Tier Architecture

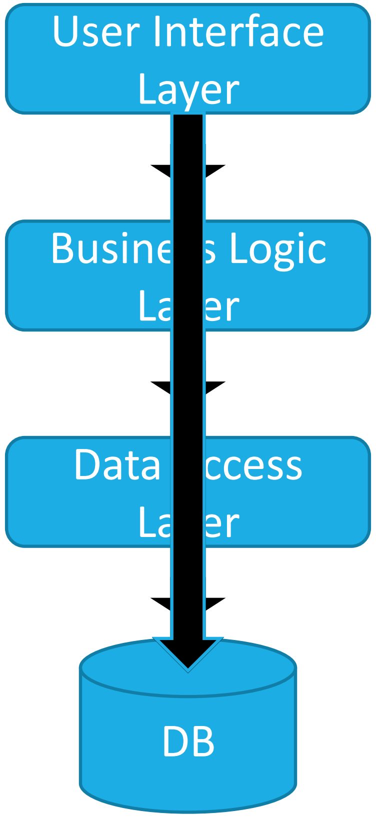
OR N-LAYER





Source: MSDN Website, 2001

Transitive Dependencies

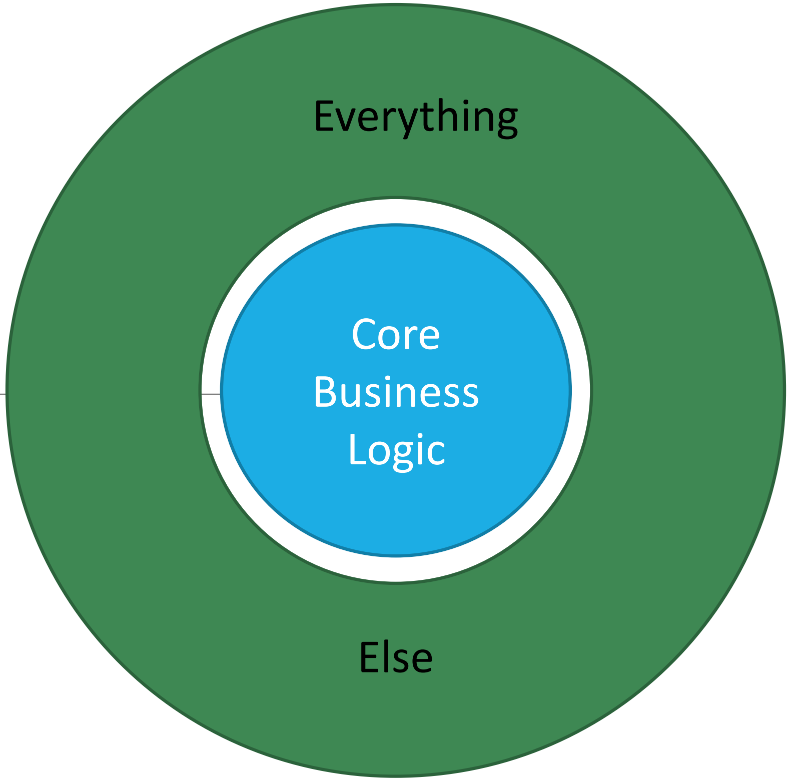


Everything

Depends on the *database*

Domain-Centric Design

AND THE CLEAN ARCHITECTURE



Domain Model

Not just business logic, but also:

A model of the problem space composed of Entities, Interfaces, Services, and more.

Interfaces define contracts for working with domain objects

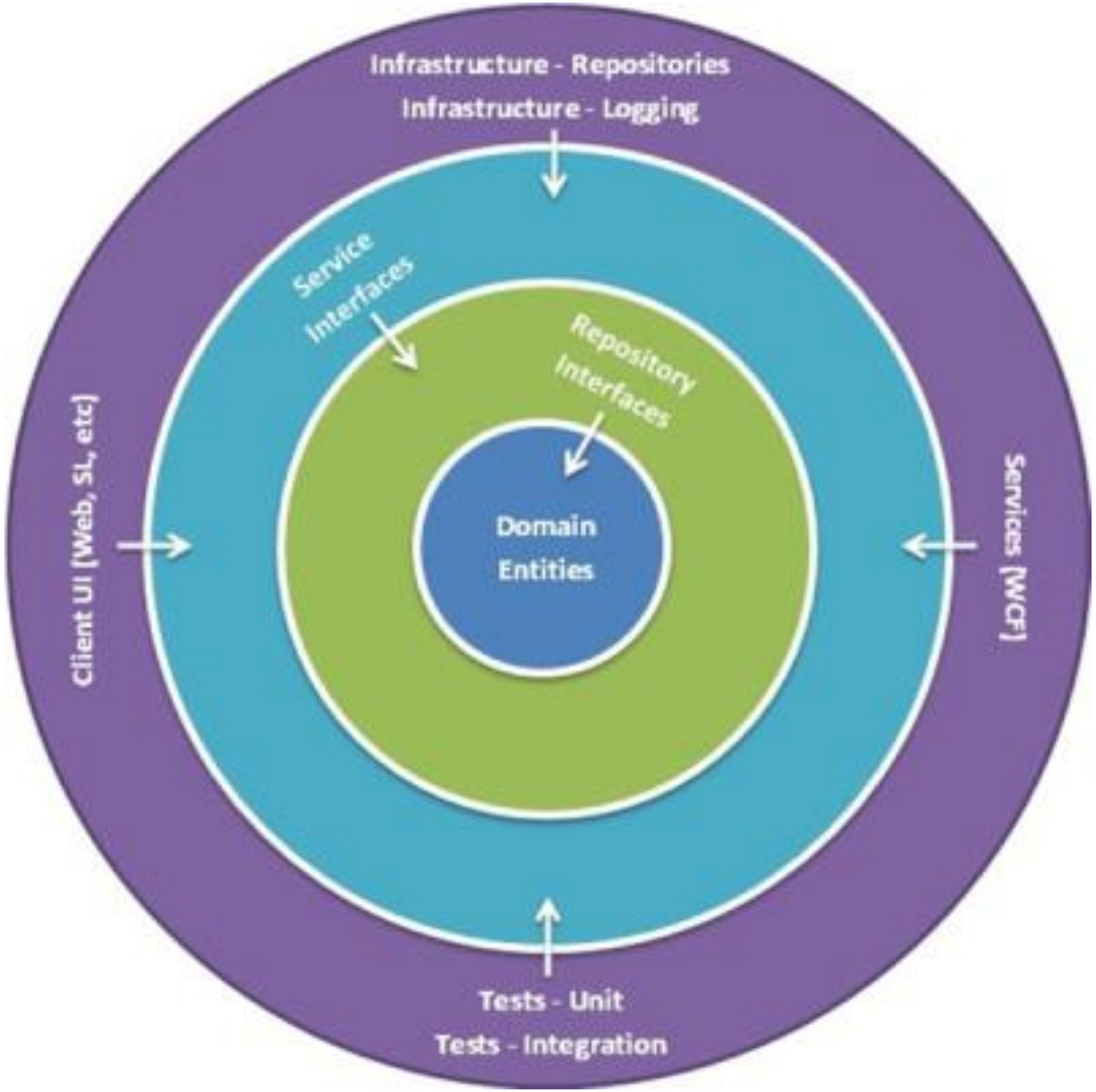
Everything in the application (including infrastructure and data access) depends on these interfaces and domain objects

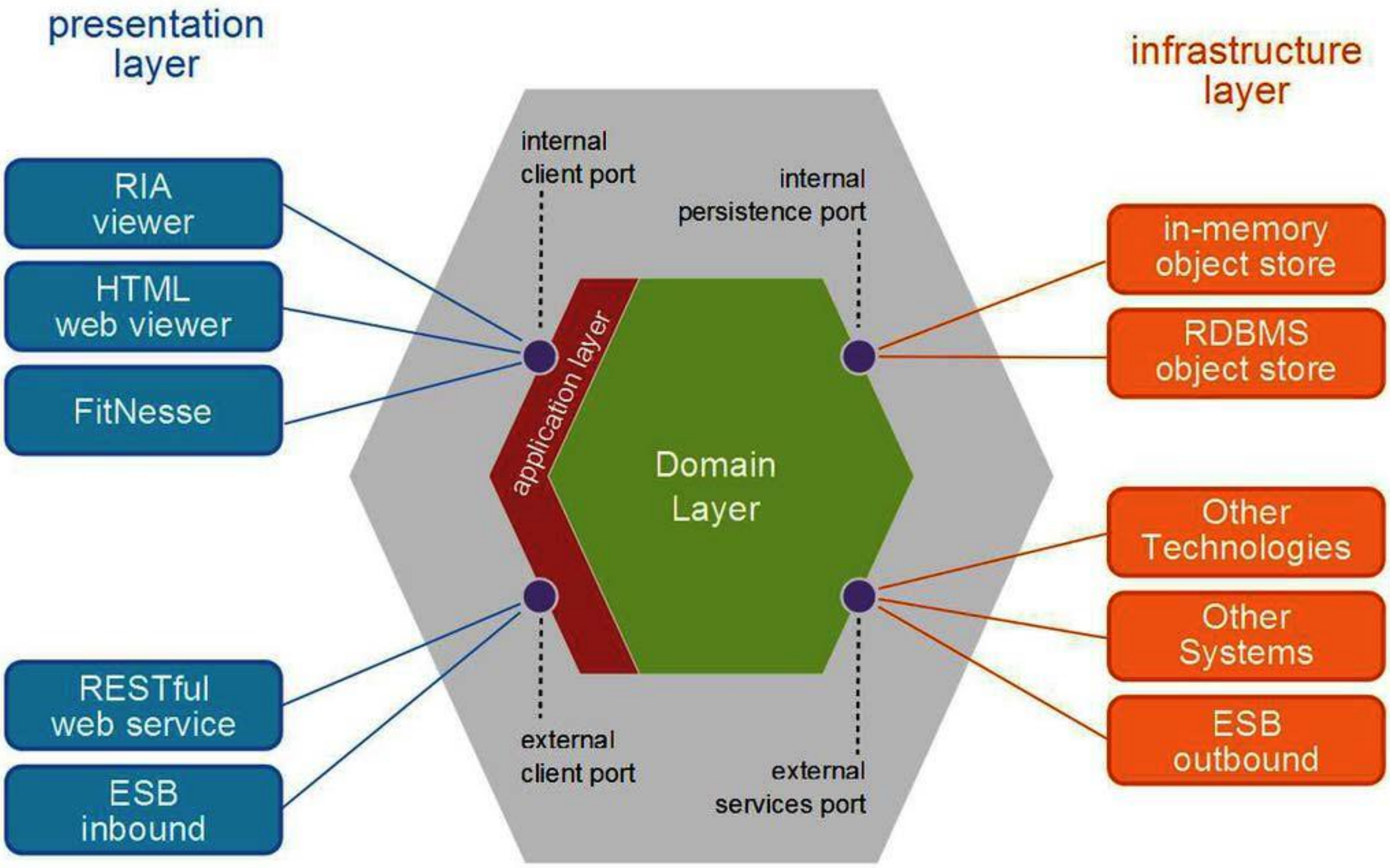
Clean Architecture

Onion Architecture

Hexagonal Architecture

Ports and Adapters


Infrastructure - Repositories
Infrastructure - Logging
Service Interfaces
Repository Interfaces
Domain Entities
Client UI (Web, SL, etc)
Services (WCF)
Tests - Unit
Tests - Integration


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

Clean Architecture “Rules”

1. You do not talk about Clean Architecture.

Clean Architecture “Rules”

~~1. You do not talk about Clean Architecture.~~

Clean Architecture “Rules”

The Application Core contains the Domain Model

Clean Architecture “Rules”

All projects depend on the Core project;
dependencies point inward toward this core

Clean Architecture “Rules”

Inner projects define `interfaces`;

Outer projects **implement** them

Clean Architecture "Rules"

Avoid direct dependency on the Infrastructure project
(except from Integration Tests and possibly Startup.cs)

Clean Architecture Features

Framework Independent

- You can use this architecture with ASP.NET (Core), Java, Python, etc.
- It doesn’t rely on any software library or proprietary codebase.

Clean Architecture Features

Database Independent

- The vast majority of the code has no knowledge of persistence details.
- This knowledge may exist in just one class, in one project that no other project references.

Clean Architecture Features

UI Independent

- Only the UI project cares about the UI.
- The rest of the system is UI-agnostic.

Clean Architecture Features

Testable

- Apps built using this approach, and especially the core domain model and its business rules, are easy to test.

Refactoring to a Clean Architecture

Best to start from a properly organized solution

- See https://github.com/ardalis/CleanArchitecture

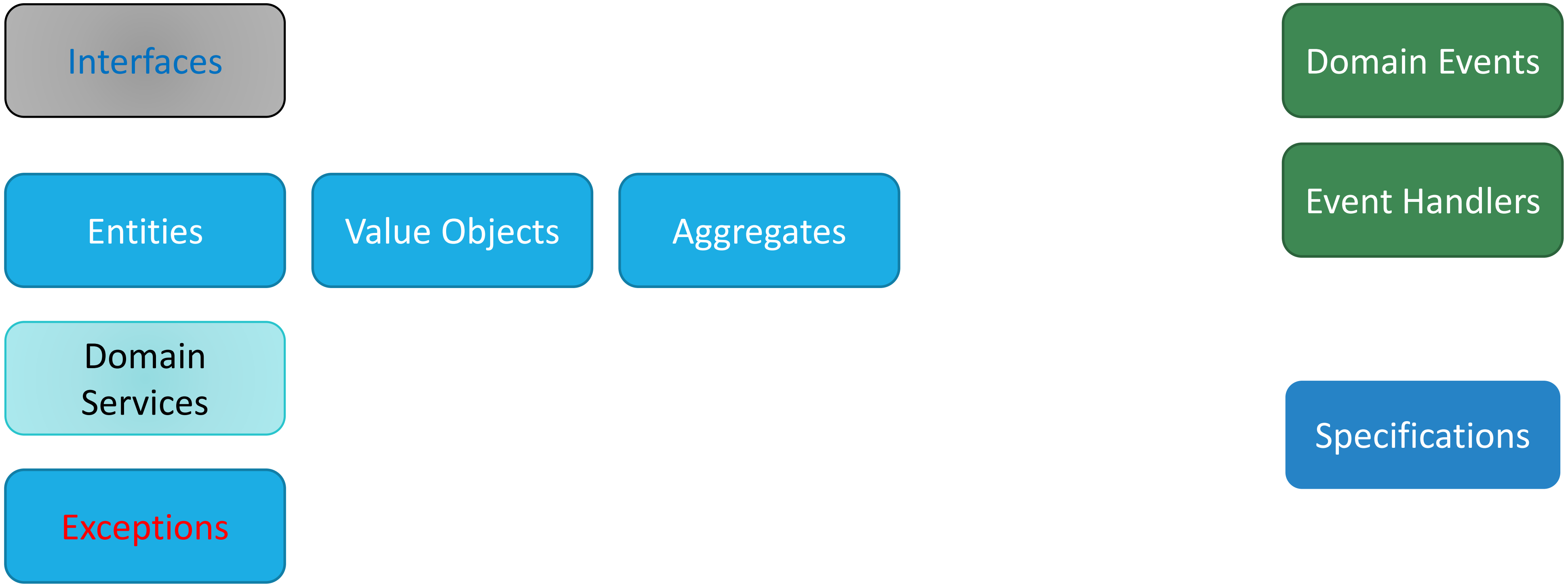
Next-best: Start from an application consisting of just a single project

Most difficult: Large, existing investment in multi-layer architecture without abstractions or DI

The Core Project (domain model)

Minimal dependencies – none on *Infrastructure*.

What Goes in Core:



Interfaces

Entities

Value Objects

Aggregates

Domain Services

Exceptions

Domain Events

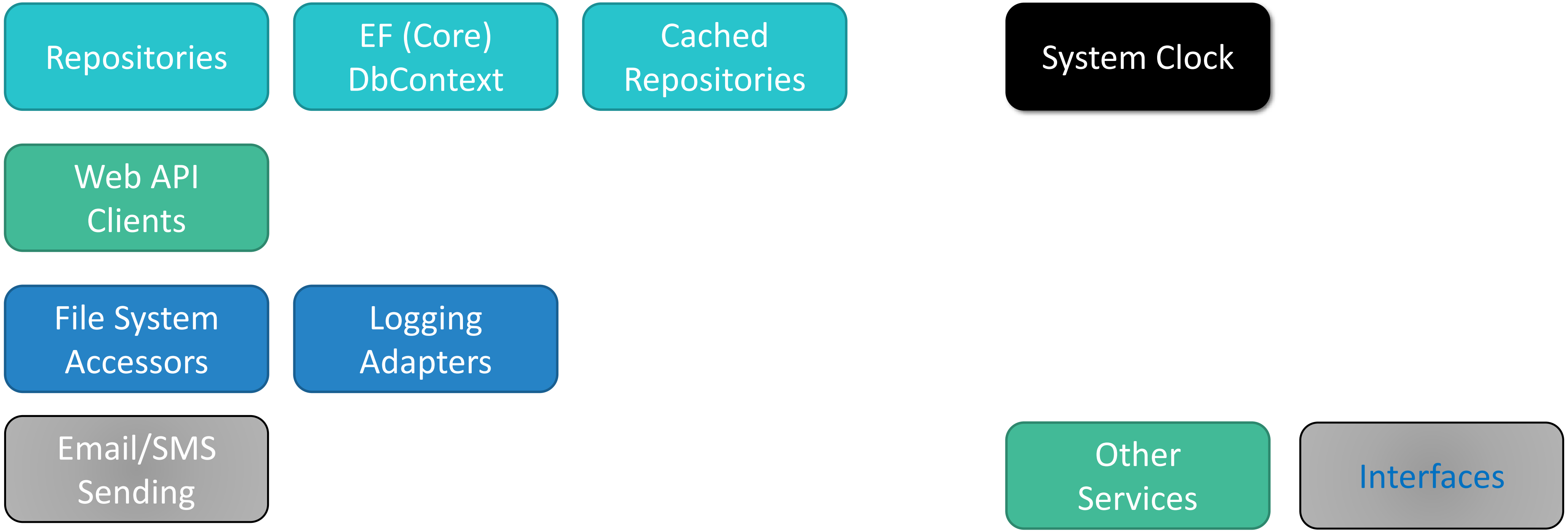
Event Handlers

Specifications

The Infrastructure Project

All dependencies on out-of-process resources.

What Goes in Infrastructure:



Repositories

EF (Core) DbContext

Cached Repositories

System Clock

Web API Clients

File System Accessors

Logging Adapters

Email/SMS Sending

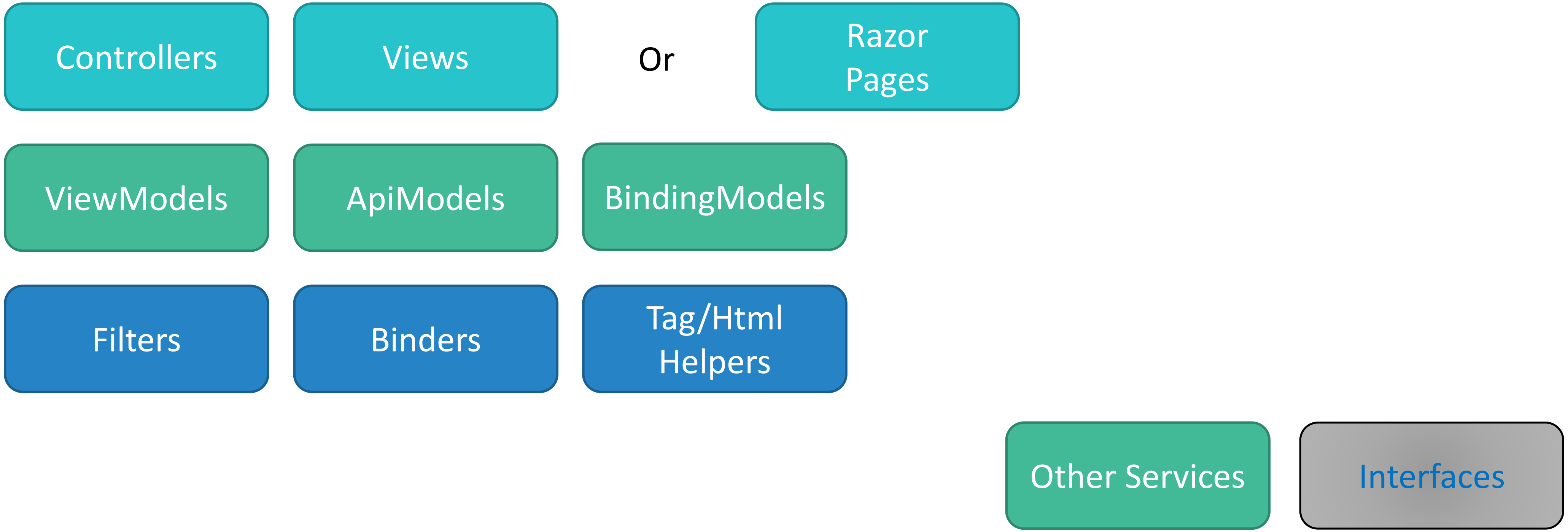
Other Services

Interfaces

The Web Project

All dependencies on out-of-process resources.

What Goes in Web:



Controllers

Views

Or

Razor Pages

ViewModels

ApiModels

BindingModels

Filters

Binders

Tag/Html Helpers

Other Services

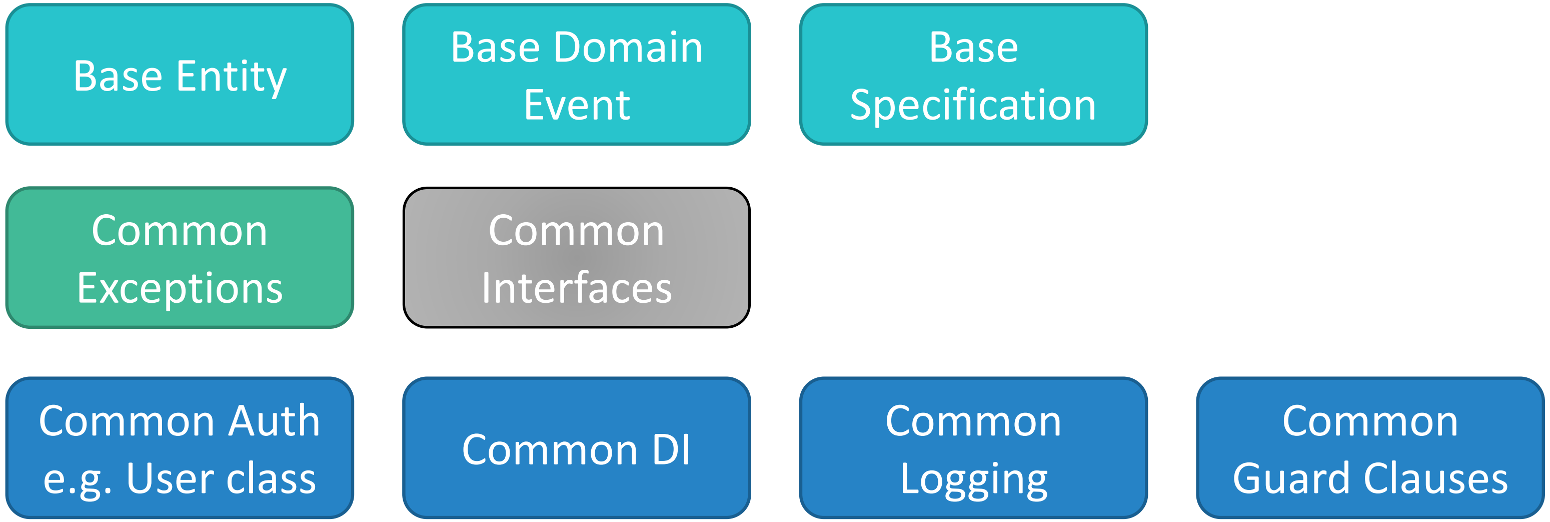
Interfaces

Sharing Between Solutions:
Shared Kernel

Common Types May Be Shared Between Solutions. Will be referenced by Core project(s).

Ideally distributed as **Nuget Packages**.

What Goes in Shared Kernel:



Base Entity

Base Domain Event

Base Specification

Common Exceptions

Common Interfaces

Common Auth e.g. User class

Common DI

Common Logging

Common Guard Clauses

Guard Clauses?

BAD EXAMPLE

```
public void ProcessOrder(Order order, Custom customer)
{
  if(order != null)
  {
    if(customer != null)
    {
      // process order here
    } else {
      throw new ArgumentNullException(nameof(customer), customer);
    }
  } else {
    throw new ArgumentNullException(nameof(order), order);
  }
}
```

Guard Clauses?

```
public void ProcessOrder(Order order, Customer customer)
{
   if(order == null) throw new ArgumentNullException(nameof(order),
order);

   if(customer==null) throw new ArgumentNullException(nameof(customer),
customer);

    // process order here
}
```

Guard Clauses?

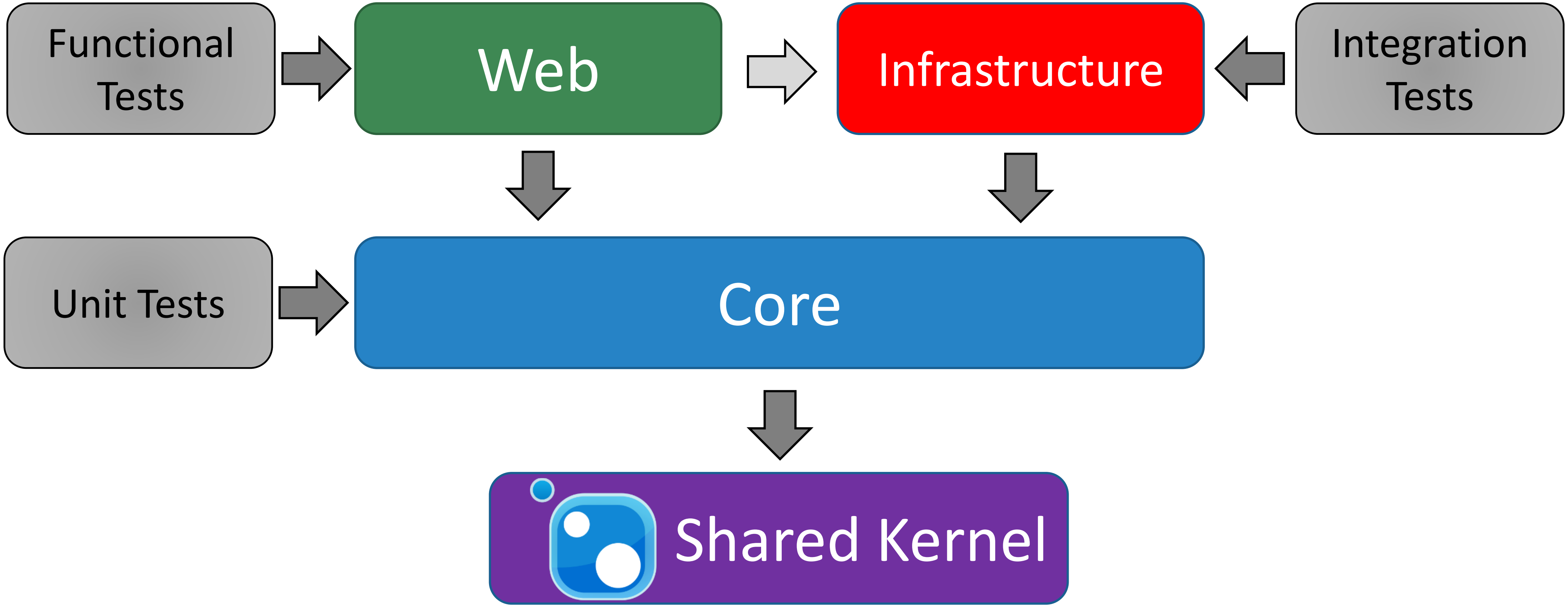
Simple checks for input that use common rules and exceptions.

Nuget Package: Ardalis.GuardClauses (https://github.com/ardalis/GuardClauses)

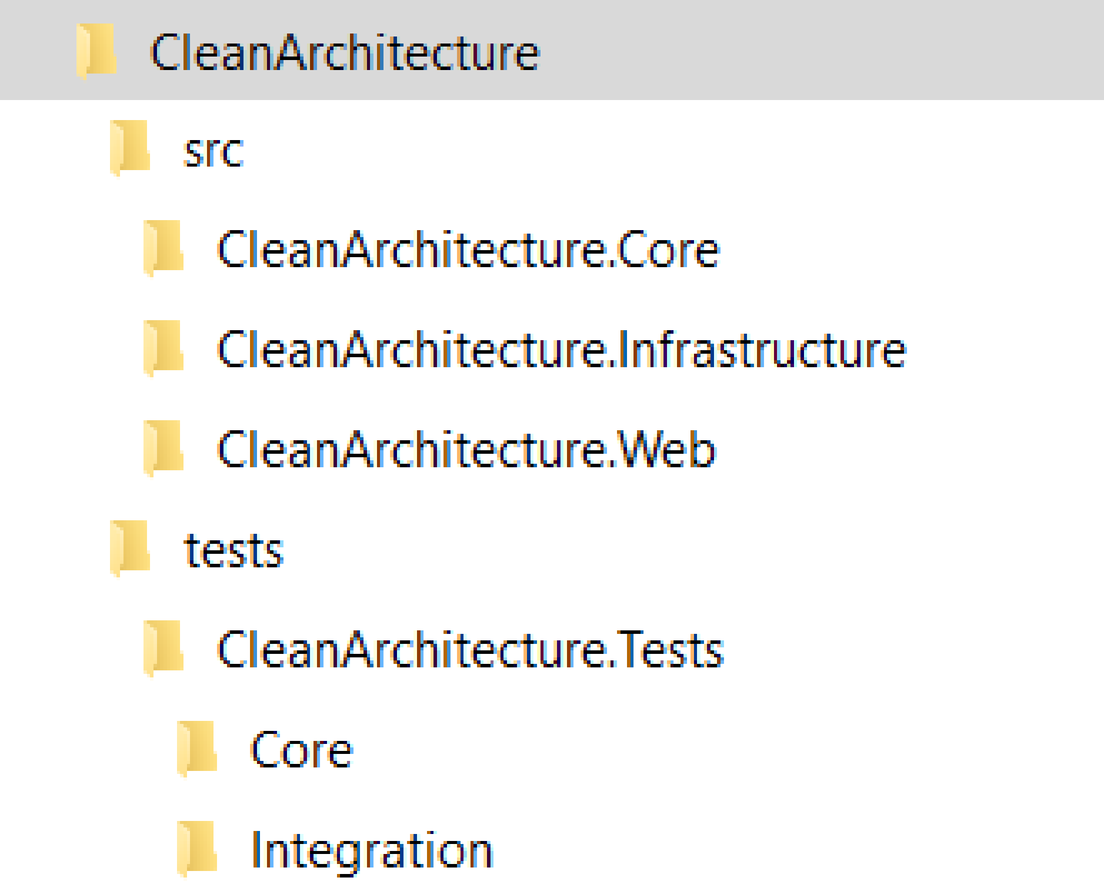
Example:

```
public void ProcessOrder(Order order, Customer customer)

{

    Guard.Against.Null(order, nameof(order));
    Guard.Against.Null(customer, nameof(customer));
    // process order here

}
```

Solution Structure – Clean Architecture



Typical (Basic) Folder Structure



- CleanArchitecture
  - src
    - CleanArchitecture.Core
    - CleanArchitecture.Infrastructure
    - CleanArchitecture.Web
  - tests
    - CleanArchitecture.Tests
      - Core
      - Integration

What belongs in actions/handlers?

Controller Actions (or Page Handlers) should:

1) Accept task-specific types (ViewModel, ApiModel, BindingModel)

2) Perform and handle model validation (ideally w/filters)

3) “Do Work” (*More on this in a moment*)

4) Create any model type required for response (ViewModel, ApiModel, etc.)

5) Return an appropriate Result type (View, Page, Ok, NotFound, etc.)

“Do Work” – Option One

Repositories and Entities

1) Get entity from an injected Repository

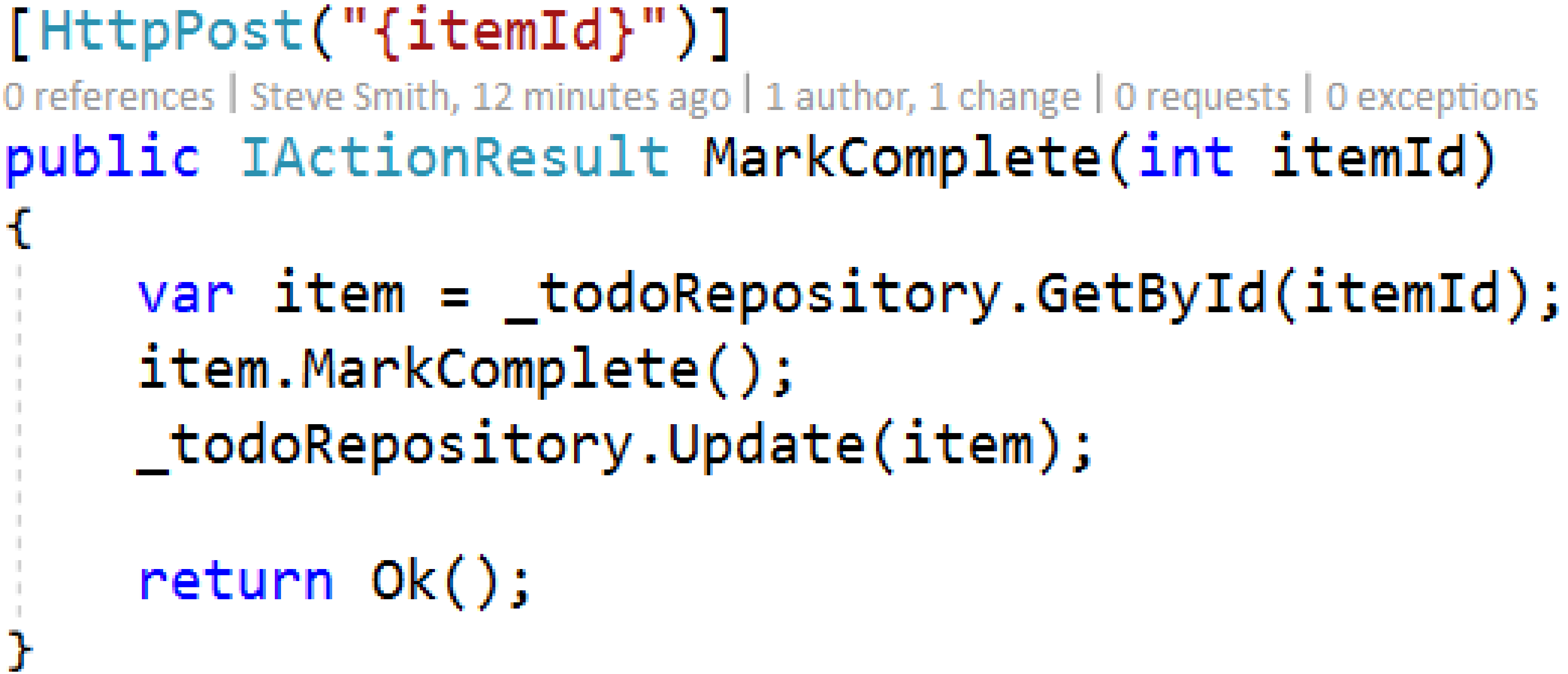
2) Work with the entity and its methods.

3) Update the entity’s state using the Repository

Great for simple operations

Great for CRUD work

Requires mapping between web models and domain model within controller



```
[HttpPost("{itemId}")]
0 references | Steve Smith, 12 minutes ago | 1 author, 1 change | 0 requests | 0 exceptions
public IActionResult MarkComplete(int itemId)
{
    var item = _todoRepository.GetById(itemId);
    item.MarkComplete();
    _todoRepository.Update(item);

    return Ok();
}
```

“Do Work” – Option Two

Work with an application service.

1) Pass ApiModel types to service

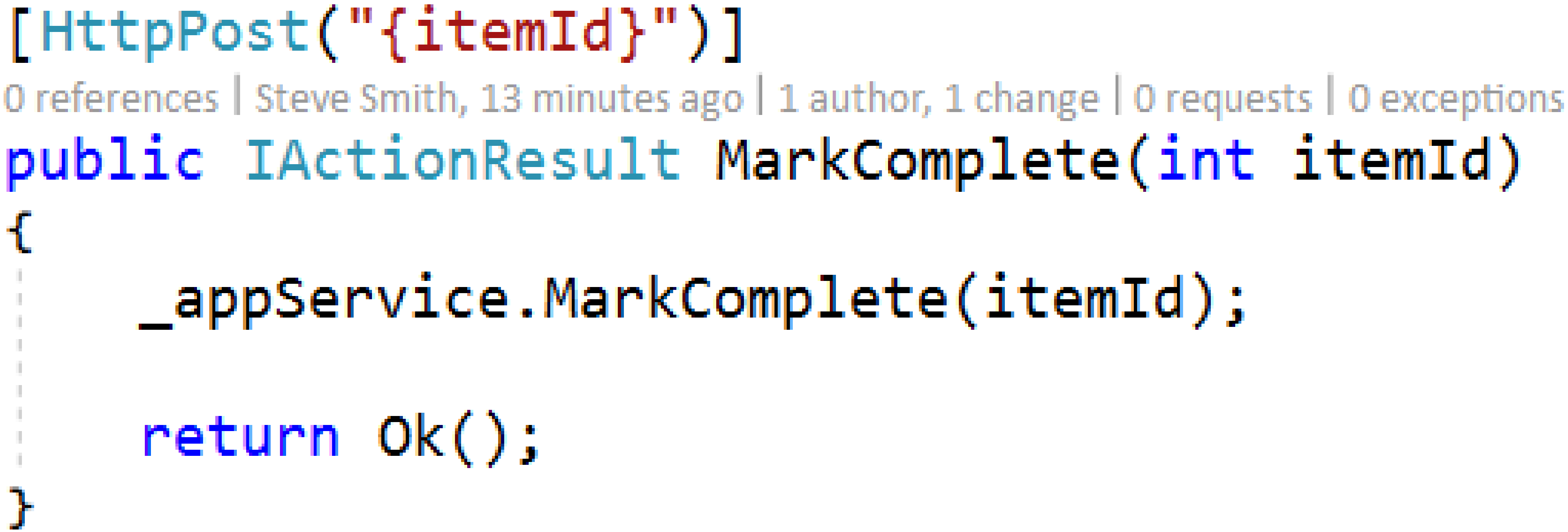
2) Service internally works with repositories and domain model types.

3) Service returns a web model type

Better for more complex operations

Application Service is responsible for mapping between models

Keeps controllers lightweight, and with fewer injected dependencies



```
[HttpPost("{itemId}")]
0 references | Steve Smith, 13 minutes ago | 1 author, 1 change | 0 requests | 0 exceptions
public IActionResult MarkComplete(int itemId)
{
    _appService.MarkComplete(itemId);

    return Ok();
}
```

“Do Work” – Option Three

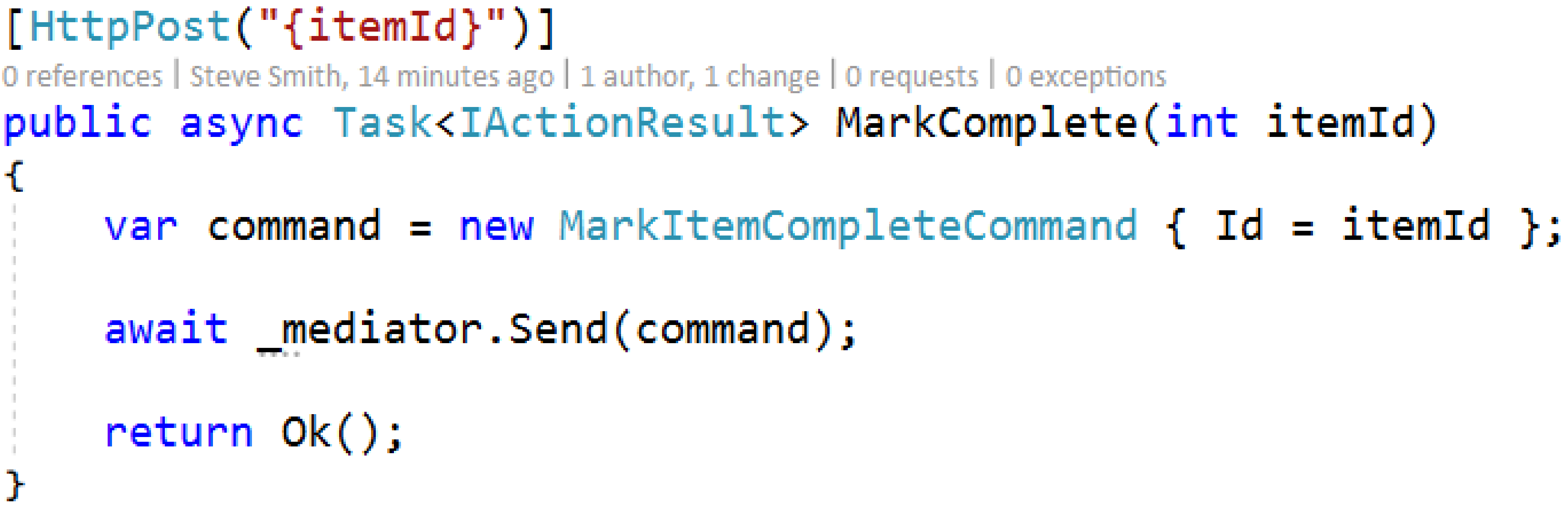
Work with commands and a tool like Mediatr

1) Use ApiModel types that represent commands (e.g. RegisterUser)

2) Send model-bound instance of command to handler using `_mediator.Send()`

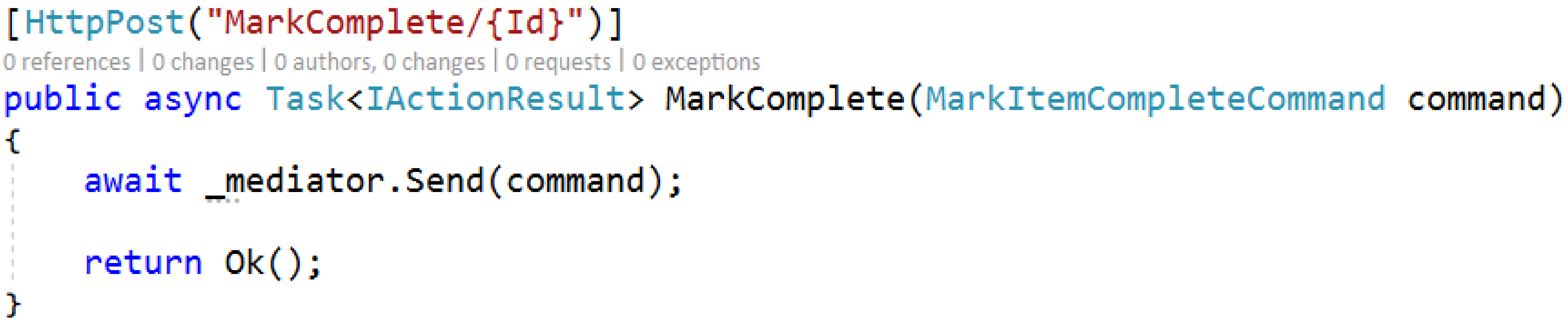
No need to inject separate services to different controllers – Mediatr becomes only dependency.

Instantiate Appropriate Command



```
[HttpPost("{itemId}")]
0 references | Steve Smith, 14 minutes ago | 1 author, 1 change | 0 requests | 0 exceptions
public async Task<IActionResult> MarkComplete(int itemId)
{
    var command = new MarkItemCompleteCommand { Id = itemId };

    await _mediator.Send(command);

    return Ok();
}
```

Resolve Command w/Model Binding



```
[HttpPost("MarkComplete/{Id}")]
0 references | 0 changes | 0 authors, 0 changes | 0 requests | 0 exceptions
public async Task<IActionResult> MarkComplete(MarkItemCompleteCommand command)
{
    await _mediator.Send(command);

    return Ok();
}
```

Code Walkthrough

GITHUB.COM/ARDALIS/CLEANARCHITECTURE

Resources

Clean Architecture Solution Template

https://github.com/ardalis/cleanarchitecture

For Worker Services: https://github.com/ardalis/CleanArchitecture.WorkerService

Online Courses (Pluralsight and DevIQ)

SOLID Principles of OO Design	https://ardalis.com/ps-stevesmith
N-Tier Architecture in C#	https://ardalis.com/ps-stevesmith
DDD Fundamentals	https://ardalis.com/ps-stevesmith
ASP.NET Core Quick Start	http://aspnetcorequickstart.com/

Weekly Dev Tips Podcast	http://www.weeklydevtips.com/

Microsoft Architecture eBook/sample	http://aka.ms/WebAppArchitecture
Group Coaching for Developers	https://devbetter.com/

Thanks!

Steve Smith

steve@ardalis.com

@ardalis

WEEKLY DEV TIPS

WITH STEVE SMITH (@ardalis)