Clean Architecture with ASP.NET Core

STEVE SMITH

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

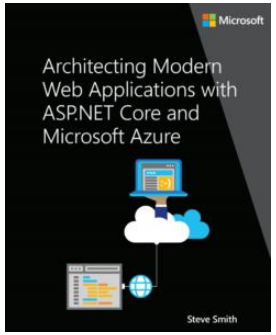
DEVIQ.COM

Learn More After Today

1) Pluralsight
   - N-Tier Apps with C# http://bit.ly/PS-NTier1
   - Domain-Driven Design Fundamentals http://bit.ly/ddd-fundamentals

2) DevIQ
   - ASP.NET Core Quick Start http://aspnetcorequickstart.com

3) Microsoft FREE eBook/Sample App
   - eShopOnWeb eCommerce Sample https://ardalis.com/architecture-ebook



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

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?

Oh yeah, and isn’t ASP.NET Core pretty cool? :)

Principles

A BIT OF GUIDANCE

Separation of Concerns

Don't let your plumbing code pollute your software.

telerik
deliver more than expected

Separation of Concerns

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

The Big Three™

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

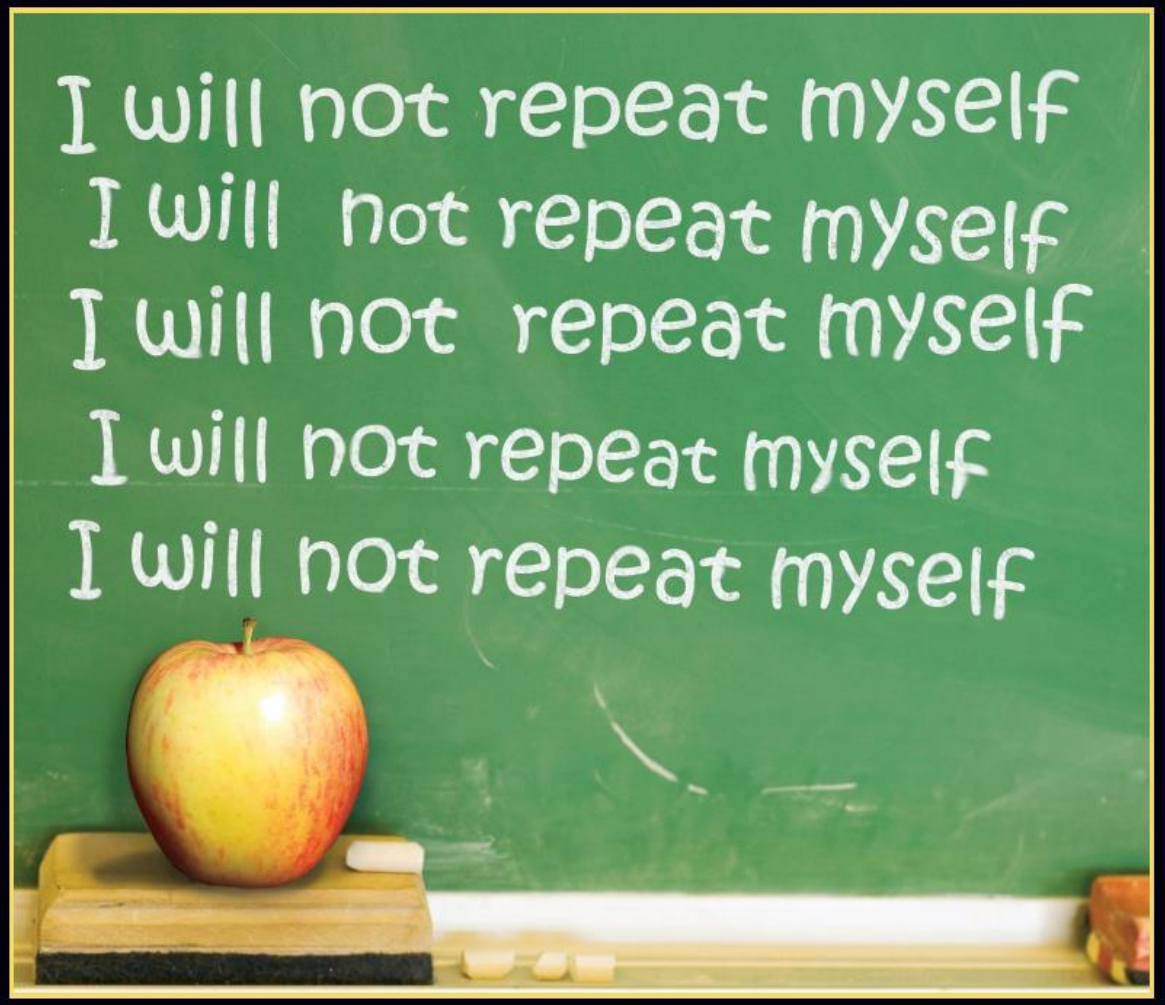
Single Responsibility

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.

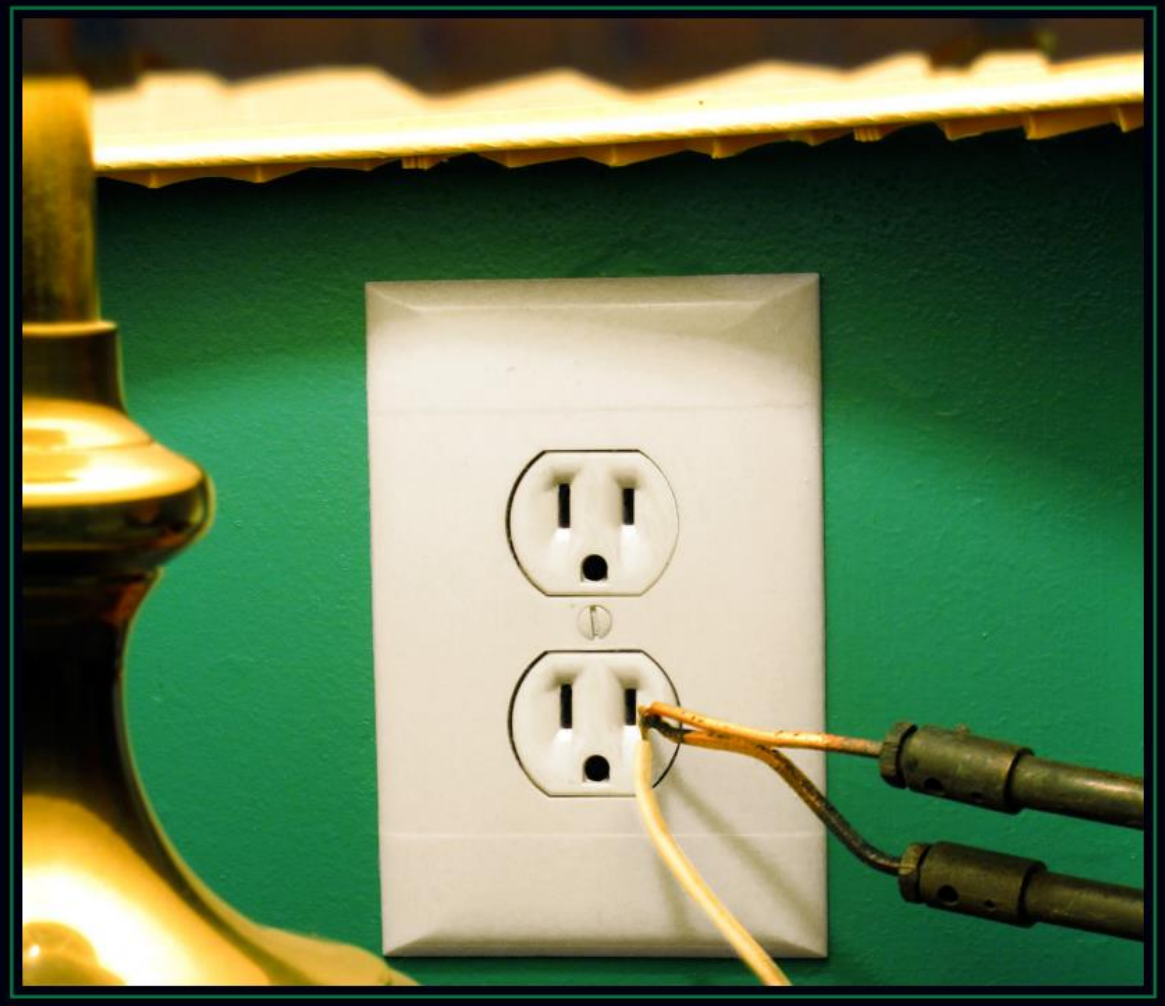


Don't Repeat Yourself

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).

- Classes should follow Explicit Dependencies Principle:
  - Request **all** dependencies via their constructor.

Corollary

- **Abstractions/interfaces must be defined somewhere accessible by**:
  - Low level implementation services
  - High level business services
  - User interface entry points

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?

Business/domain classes shouldn't depend directly on infrastructure classes

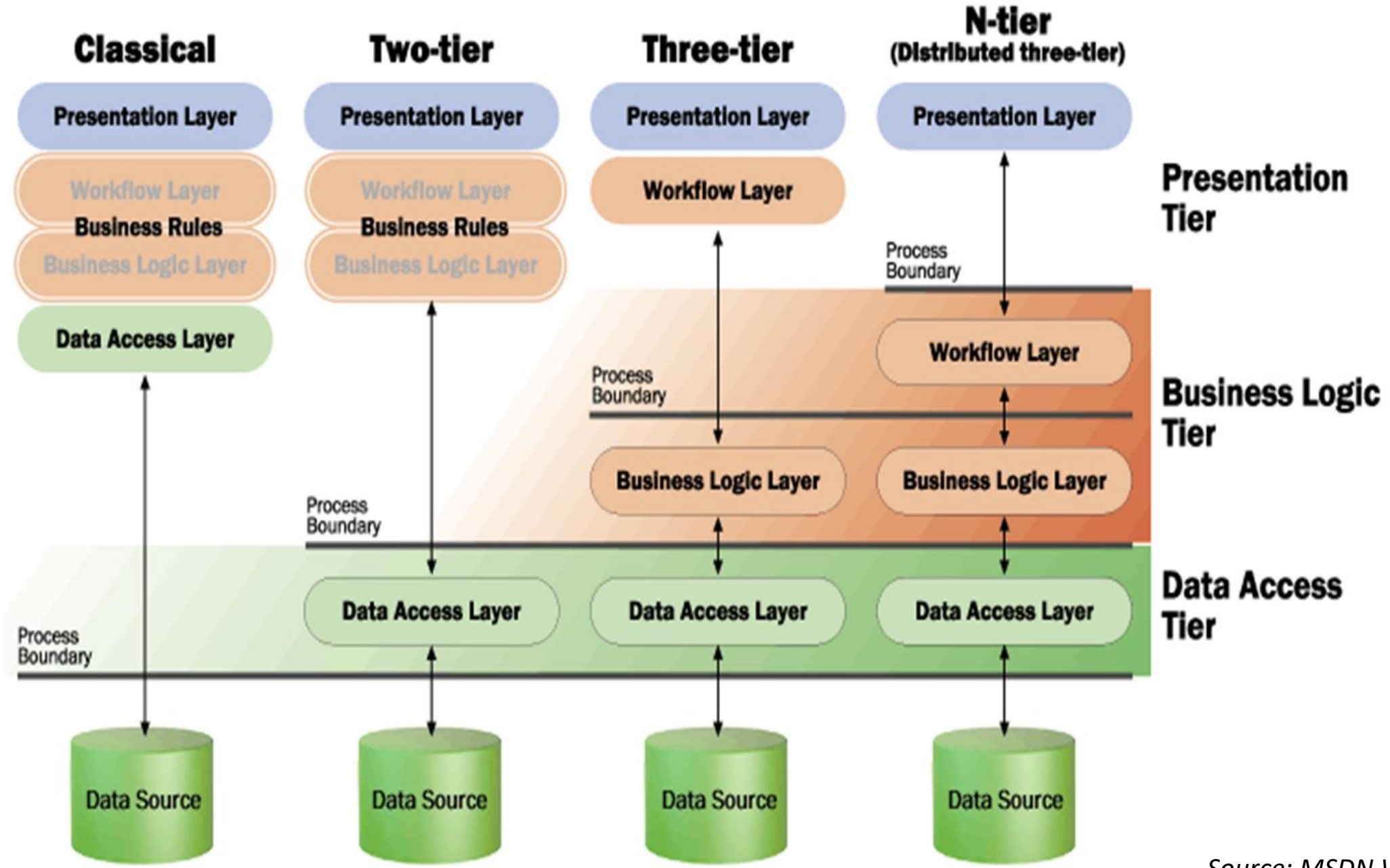
- How can our solution design help?

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

N-Tier Benefits

Code Reuse

Team Segmentation

Better Maintainability

(slightly) Looser Coupling

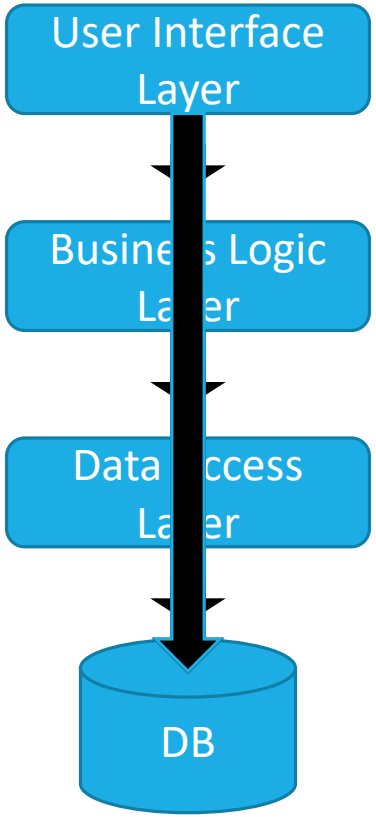
N-Tier Drawbacks

Transitive Dependencies

(some) Complexity

(still) Tight Coupling

Transitive Dependencies



Everything

Depends on the *database*

Domain-Centric Design

AND THE CLEAN ARCHITECTURE

Domain Model

Not just “business logic”

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

Interfaces define contracts for working with domain objects

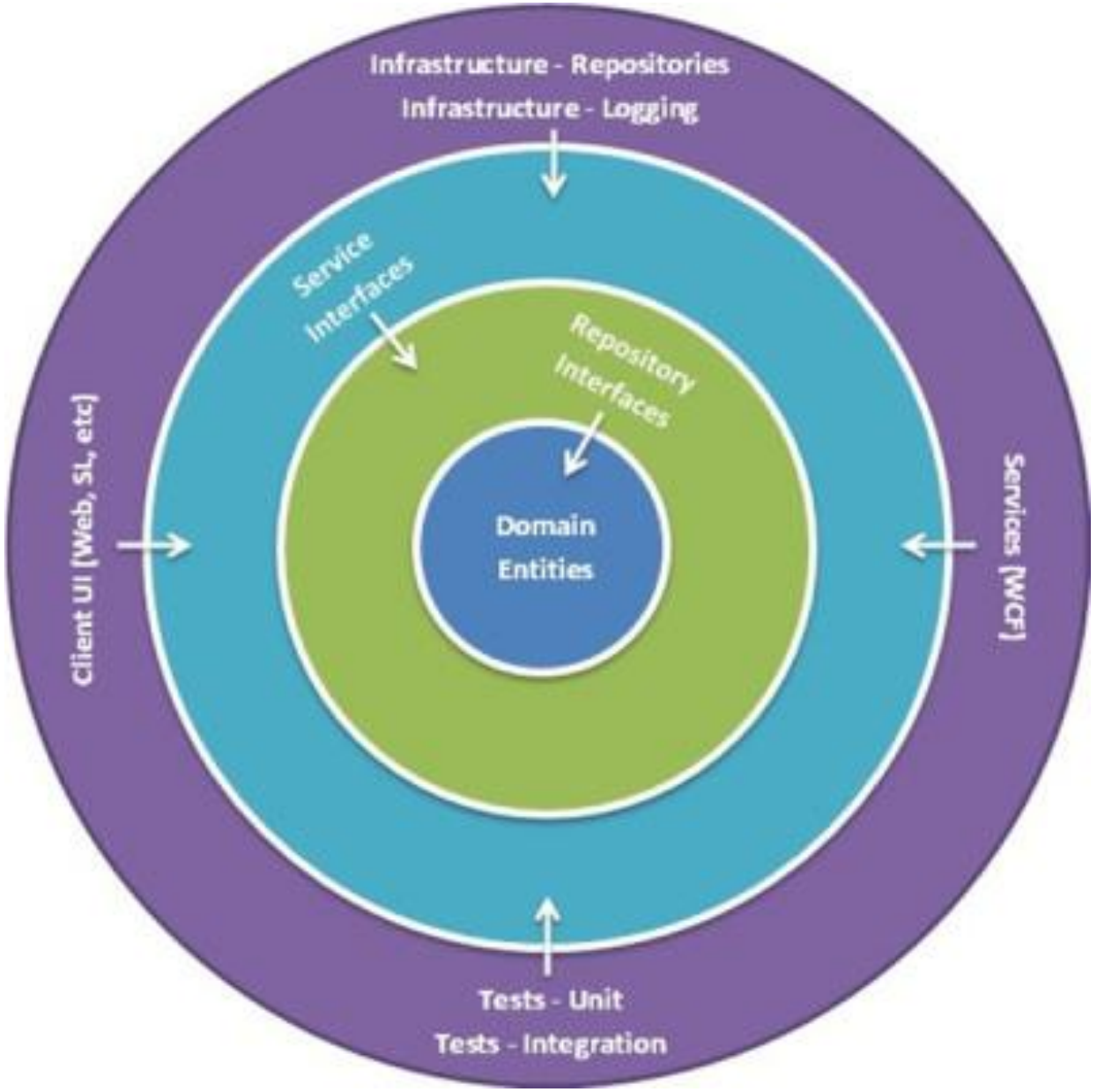
Everything in the application (including infrastructure and data access) depends on these interfaces and domain objects

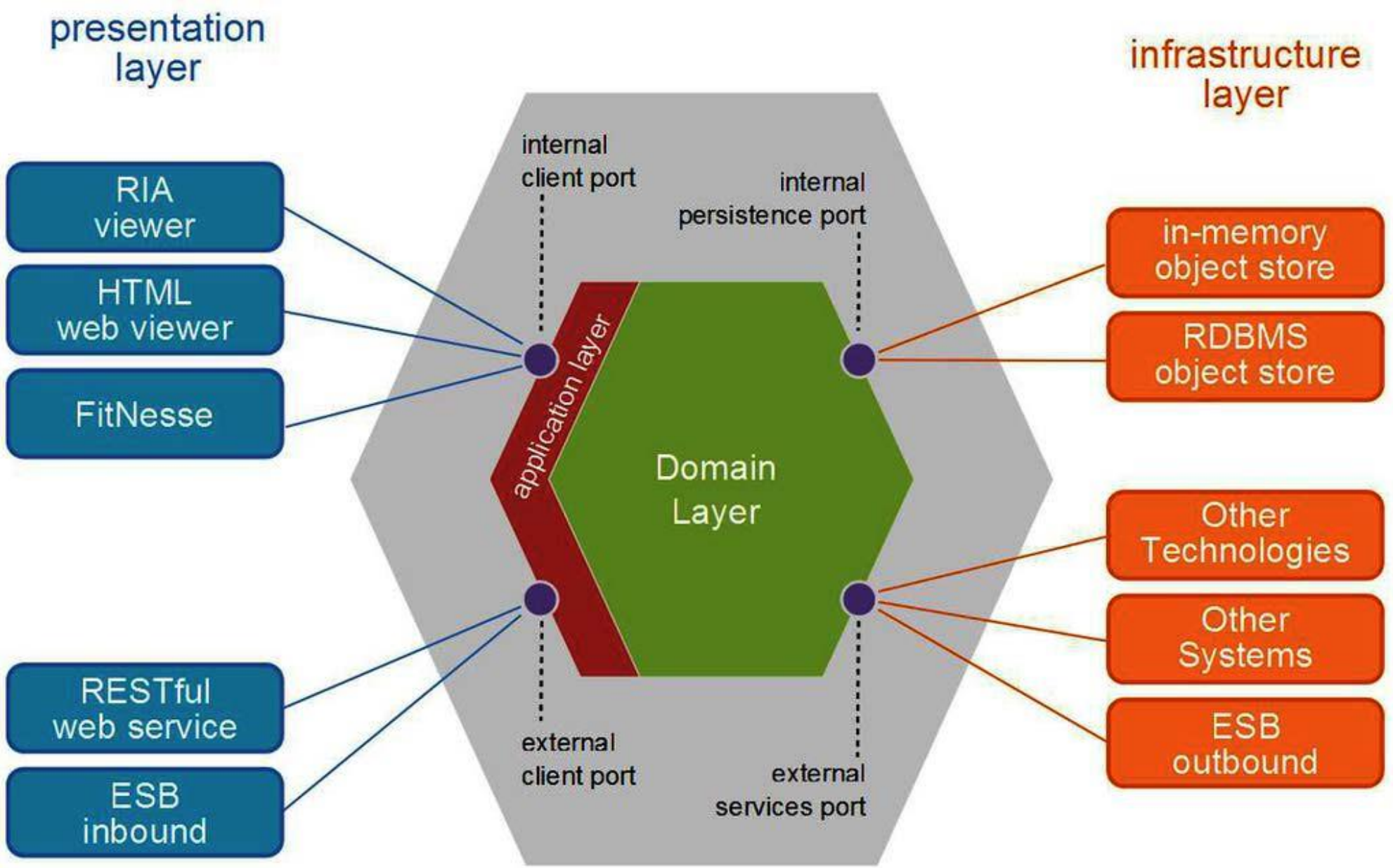
Clean Architecture

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
external client port
external services port
application layer
Domain Layer
in-memory object store
RDBMS object store
Other Technologies
Other Systems
ESB outbound

Clean Architecture “Rules”

The Application Core contains the Domain Model

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

Inner projects define interfaces; outer projects implement them

Avoid direct dependency on Infrastructure project (except from Integration tests)

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.

Database Independent

- The vast majority of the code has no knowledge of what database, if any, might be used by the application. Often, this knowledge will exist in a single class, in a single project that no other project references.

UI Independent

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

Testable

- Apps built using this approach, and especially the core domain model and its business rules, are extremely testable.

Refactoring to a Clean Architecture

Best to start from a properly organized solution

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

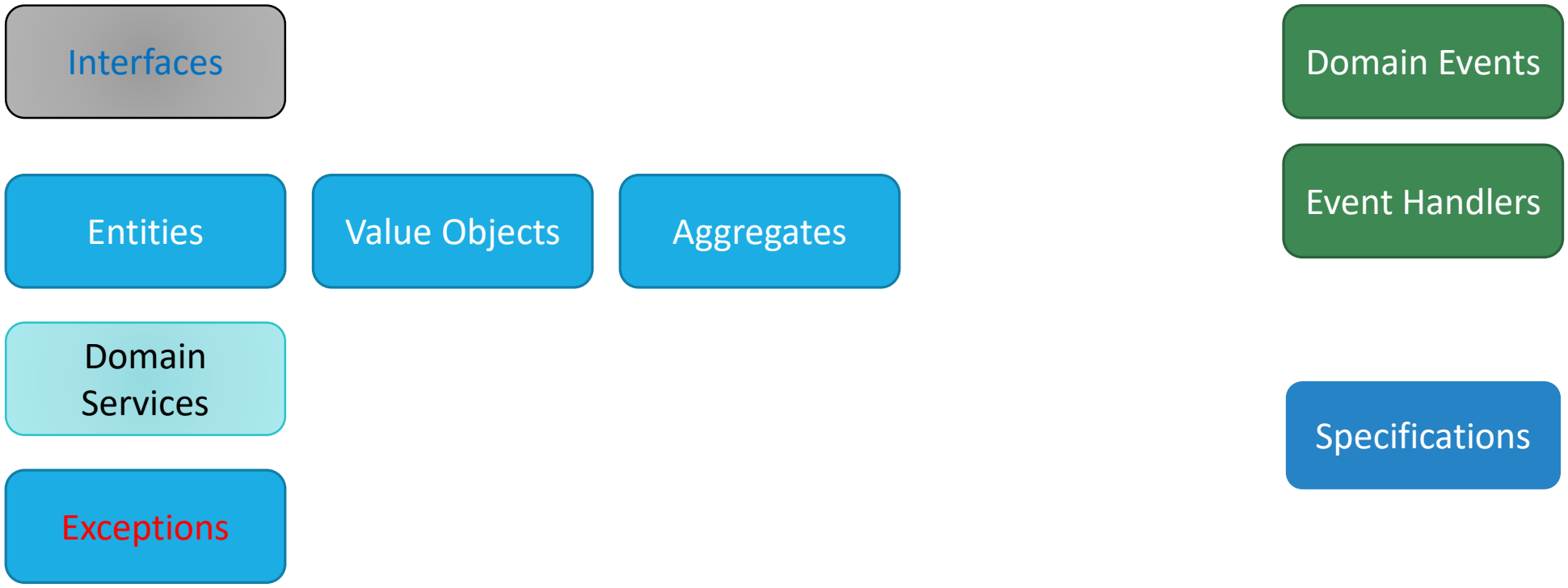
Next-best: Start from a single project

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

The Core Project

Minimal dependencies – none on *Infrastructure*.

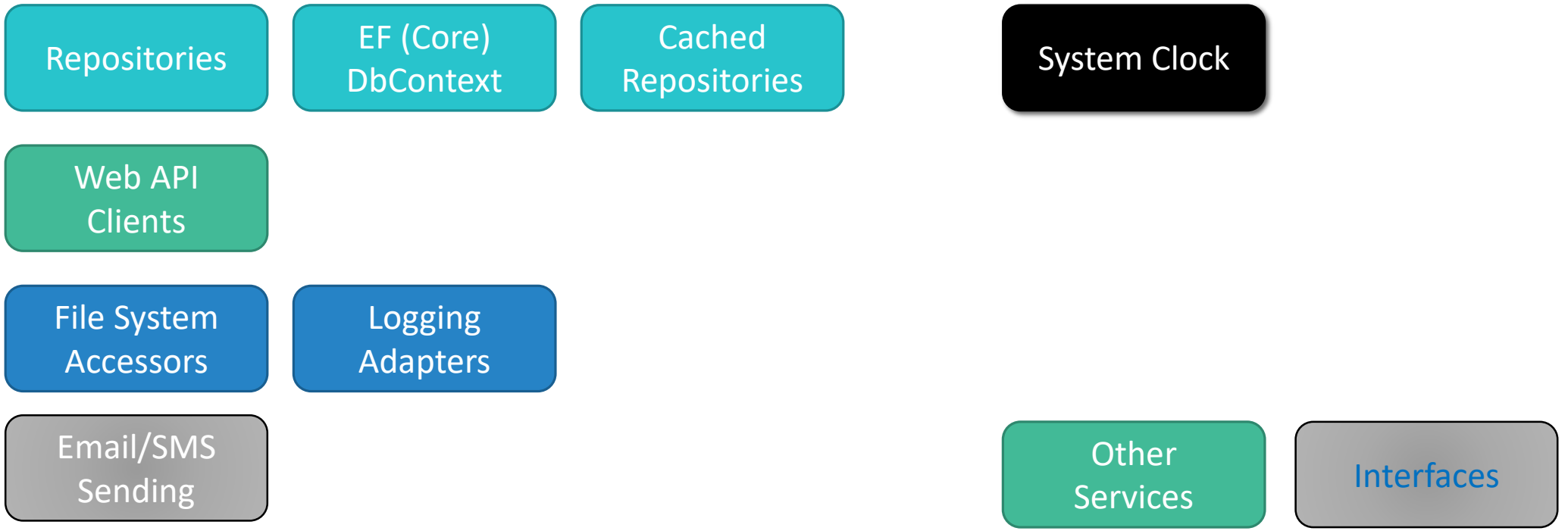
What Goes in Core:



The Infrastructure Project

All dependencies on out-of-process resources.

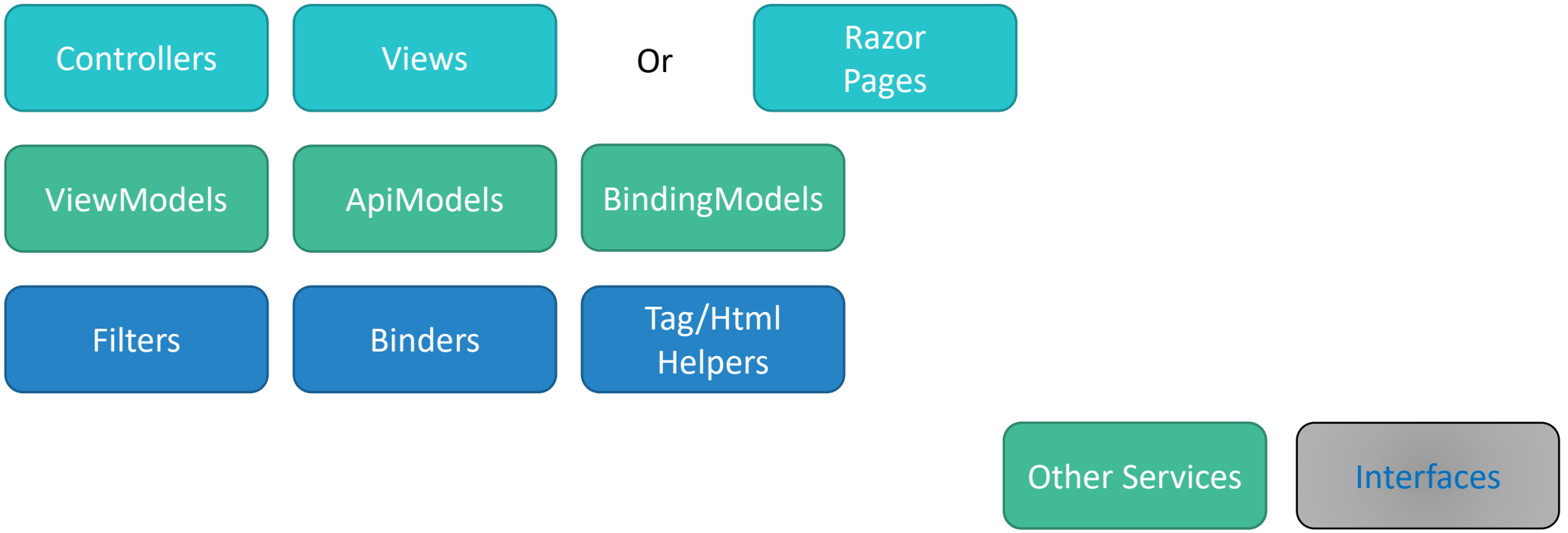
What Goes in Infrastructure:



The Web Project

All dependencies on out-of-process resources.

What Goes in Web:

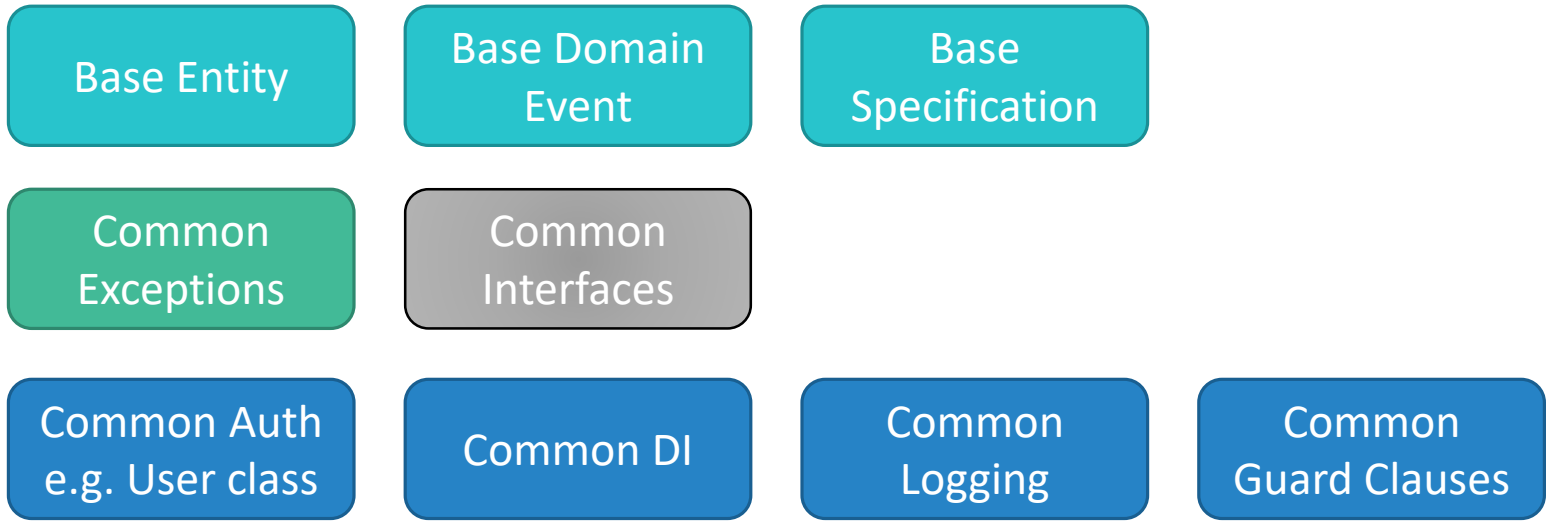


Sharing Between Solutions: Shared Kernel

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

Ideally distributed as **Nuget Packages**.

What Goes in Shared Kernel:



Guard Clauses?

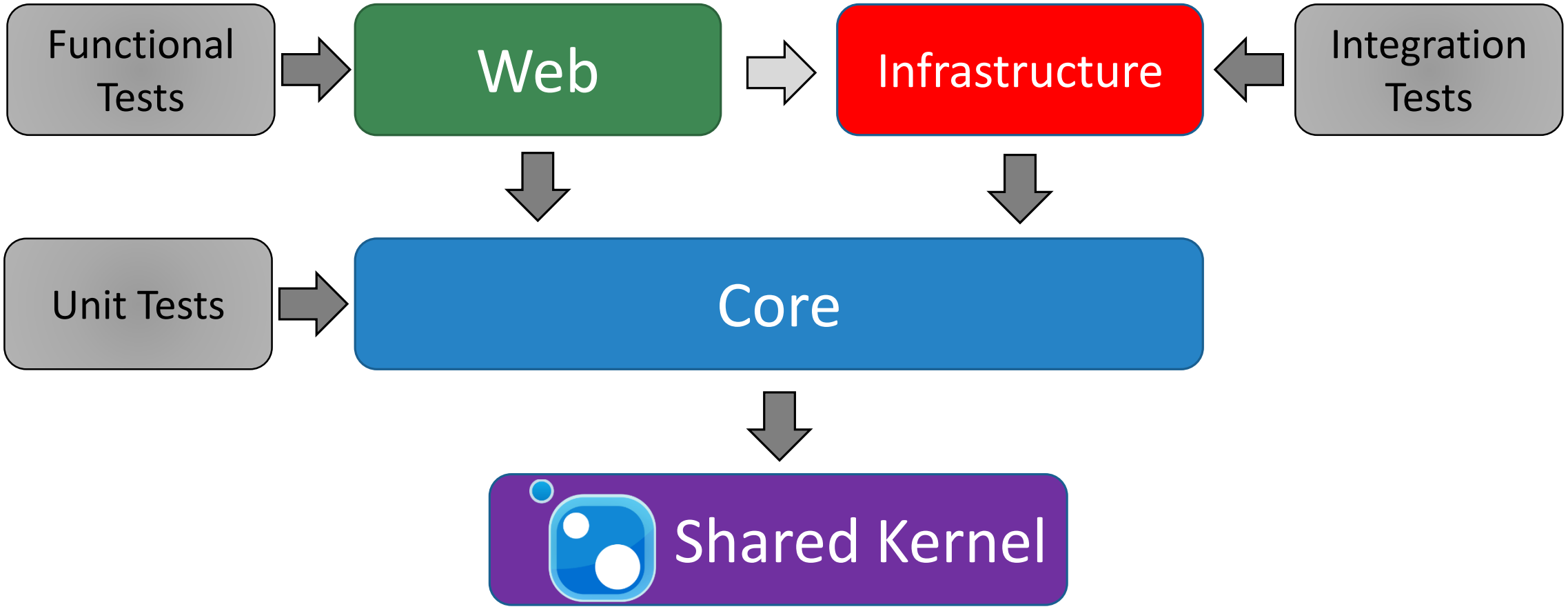
Simple checks for input that use common rules and exceptions.

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

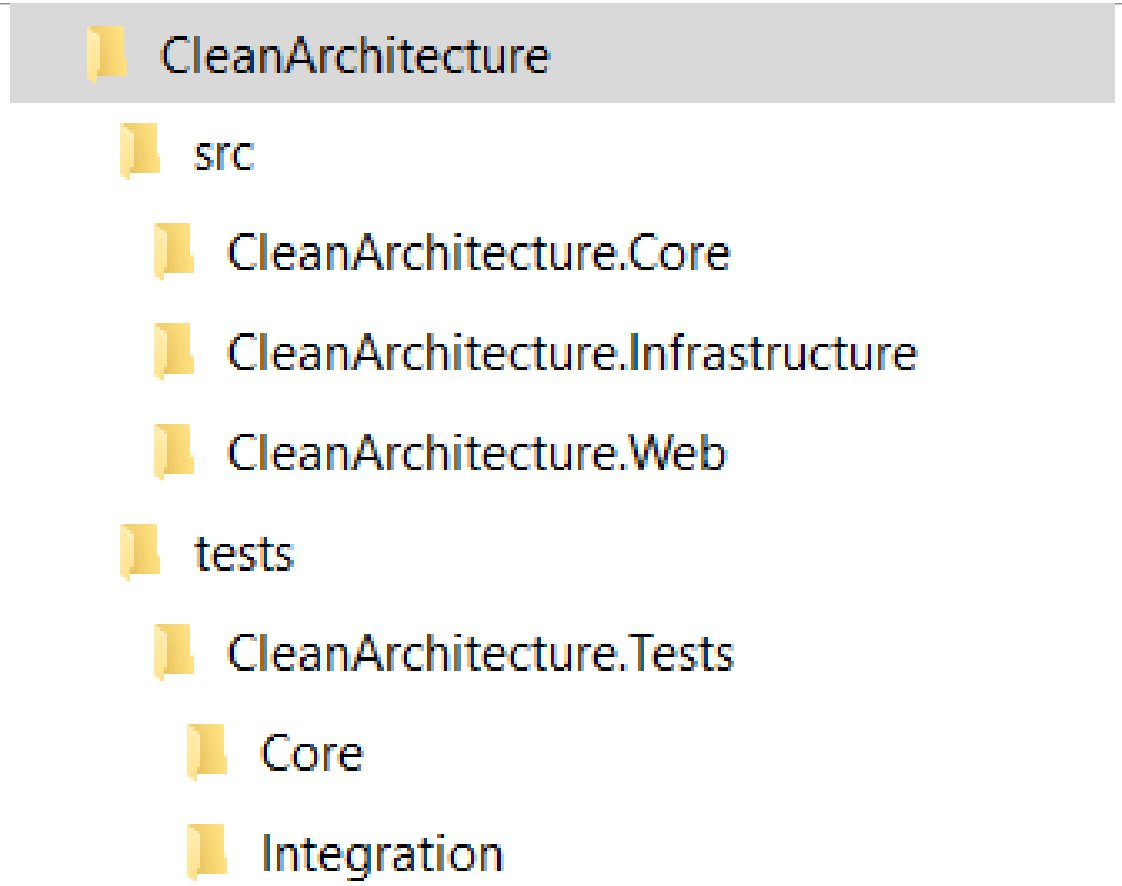
Example:

```
public void ProcessOrder(Order order)
{
    Guard.Against.Null(order, nameof(order));
    // process order here
}
```

Solution Structure – Clean Architecture



Typical (Basic) Folder Structure



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 web models and domain model.

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.

```
[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();
}
```

```
[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

Resources

Online Courses (Pluralsight and DevIQ)

- SOLID Principles of OO Design — http://bit.ly/SOLID-OOP
- N-Tier Architecture in C# — http://bit.ly/PS-NTier1 and http://bit.ly/PS-NTier2
- DDD Fundamentals — http://bit.ly/ddd-fundamentals
- ASP.NET Core Quick Start — http://aspnetcorequickstart.com/ **DEVINTFALL17** 20% OFF!

- **Weekly Dev Tips Podcast** — http://www.weeklydevtips.com/
- Microsoft Architecture eBook/sample — http://aka.ms/WebAppArchitecture