EIU
TRƯỜNG ĐẠI HỌC
QUỐC TẾ
MIỀN ĐÔNG
EASTERN
INTERNATIONAL
UNIVERSITY

Coding Practice

Chapter 1: Clean Code

EIU

What is clean code and best practices?

What is Clean Code?

Code that is easy to read, understand, and maintain. Reflects care and craftsmanship of the developer.

Key Principles:

- Simplicity.
- Readability.
- Minimal dependencies.
- Maintenance.

Characteristics of Clean Code

- Readable: Like a well-written story.
- Simple: Avoids unnecessary complexity.
- Tested: Ensures reliability through unit and acceptance tests.
- Focused: Each piece does one thing well.
- Minimal Dependencies: Explicit and kept to a minimum.

Why Clean Code?

Clean code is essential for the long-term success and maintainability of any software project

Maintainability

- **Easier to understand**: Clean code is easier for other developers (and your future self) to read and comprehend.
- **Quicker updates**: Developers can identify the purpose of each component quickly, making updates or modifications less error-prone.

Debugging and Testing

- **Fewer bugs**: Cleaner, more organized code reduces logical errors and unexpected behaviors.
- **Simplified testing**: Well-structured code with clear functions and responsibilities is easier to write unit tests for.

Collaboration

- **Teamwork-friendly**: Clean code is easier to work on collaboratively since everyone can understand it without requiring extensive explanations.
- **Consistent style**: Adhering to clean coding principles ensures consistency, which helps team members quickly adapt to the codebase.

Why Clean Code?

Scalability

- **Handles complexity better**: Clean code adheres to good design principles (e.g., SOLID principles), ensuring the project scales gracefully as new features are added.

Cost Efficiency

- **Reduces technical debt**: Clean code minimizes the cost of future refactoring.
- **Speeds up onboarding**: New developers can get up to speed faster when the code is easy to follow.

Professionalism

- **Reflects care and expertise**: Writing clean code demonstrates professionalism and attention to detail, fostering trust among peers and stakeholders.

Best Practices for Clean Code

Meaningful Names

- **Variables, Functions, and Classes:** Use descriptive names that convey purpose. Avoid ambiguous or overly abbreviated names.
- **Naming Conventions:** Follow consistent naming conventions (e.g., camelCase for variables, PascalCase for classes).

Bad Example:

```
// 'd' does not clearly describe the meaning of the variable
int d;
```

Good Example:

```
// Variable names are clear and easy to understand
int daysSinceLastLogin;
```

Best Practices for Clean Code

Keep Functions Small

- Functions should perform a single task and be as small as possible. If a function does more than one thing, it should be split.

Bad

```
public void ProcessUserData(User user)
{
    // Check user
    if (user == null) throw new ArgumentNullException();

    // Perform complex processing logic
    // Check logic
    // Data processing logic
    // Notification sending logic
    // Logging
}
```

Good

```
public void ProcessUserData(User user)
    {
        ValidateUser(user);
        ProcessUserLogic(user);
        SendUserNotification(user);
        LogAction(user);
    }

private void ValidateUser(User user) { /* validation logic */ }
private void ProcessUserLogic(User user) { /* processing logic */ }
private void SendUserNotification(User user) { /* notification sending logic */ }
private void LogAction(User user) { /* logging logic */ }
```

Best Practices for Clean Code

Write Readable Code

- **Indentation:** Properly indent the code for better readability.
- **Whitespace:** Use blank lines to separate code logically (e.g., between different sections of a function or class).
- **Comments:** Use comments to explain *why* something is done, not *what* is done, unless the code is complex or non-obvious.
- **Avoid unnecessary comments:** Let the code be self-explanatory where possible.

Bad

```
int a = 1000; // '1000' has no clear meaning
if (a == 1000) { /* do something */ }
```

Good

```
const int MaxRetries = 3; // Clearly named constant
if (retries == MaxRetries) { /* do something */ }
```

Best Practices for Clean Code

Avoid Duplication (DRY Principle)

- Don’t Repeat Yourself. If the same logic or code is used in multiple places, extract it into a function or method.
- Use functions, methods, or classes to abstract repetitive logic.

Bad

```
public void SaveUser(User user)
    {
        if (user.Name == null) throw new ArgumentNullException();
        if (user.Email == null) throw new ArgumentNullException();
        // Logic for saving users
    }

public void SaveOrder(Order order)
    {
        if (order.Name == null) throw new ArgumentNullException();
        if (order.Email == null) throw new ArgumentNullException();
        // Logic to save orders
    }
```

Good

```
public void SaveEntity(Entity entity)
    {
        ValidateEntity(entity);
        // Logic to save entities
    }

private void ValidateEntity(Entity entity)
    {
        if (entity.Name == null) throw new ArgumentNullException();
        if (entity.Email == null) throw new ArgumentNullException();
    }
```

Best Practices for Clean Code

Error Handling

- Properly handle exceptions or errors using try-catch blocks, and ensure you provide meaningful error messages.
- Avoid silent failure (e.g., swallowing exceptions without logging them).

Bad

```
try
    {
        // Handling logic
    }
catch (Exception)
    {
        // Handle the error without logging or reporting anything
    }
```

Good

```
try
    {
        // Handling logic
    }
catch (Exception)
    {
        // Handle the error without logging or reporting anything
    }


try
    {
        // Processing logic
    }
catch (Exception ex)
    {
        LogError(ex);
        throw new CustomException("An error occurred while processing your request", ex);
    }

private void LogError(Exception ex)
    {
       // Log error details for easy debugging
    }
```

Best Practices for Clean Code

Single Responsibility Principle

- A class or function should have one responsibility. If you find yourself adding multiple concerns to a single class, refactor it.

Bad

```
0 references
public class UserManager
    {
        0 references
        public void RegisterUser(User user) { /* User registration logic */ }
        0 references
        public void SendWelcomeEmail(User user) { /* Welcome email sending logic */ }
    }
```

Good

```
1 reference
public class UserManager
    {
        1 reference
        private readonly EmailService _emailService;

        0 references
        public UserManager(EmailService emailService)
            {
                _emailService = emailService;
            }

        0 references
        public void RegisterUser(User user) { /* User registration logic */ }
    }

2 references
public class EmailService
    {
        0 references
        public void SendWelcomeEmail(User user) { /* Logic for sending welcome email */ }
    }
```

Best Practices for Clean Code

Keep It Simple (KISS Principle)

- Strive for simplicity in your code. Avoid over-complicating logic or solutions that could be solved in simpler ways.

Bad

```
public bool IsUserEligibleForDiscount(User user)
{
    if (user.Age > 18 && user.Age < 65 && user.MembershipType == "Gold" || user.MembershipType == "Platinum" || user.MembershipType == "Silver")
    {
        return true;
    }
    return false;
}
```

Good

```
public bool IsUserEligibleForDiscount(User user)
{
    var eligibleMemberships = new[] { "Gold", "Platinum", "Silver" };
    return user.Age > 18 && user.Age < 65 && eligibleMemberships.Contains(user.MembershipType);
}
```

Best Practices for Clean Code

Use Version Control Properly

- Commit code often with meaningful commit messages. This helps to track changes, revert if needed, and collaborate with others.

Bad

```
Commit message: "Fixed stuff"
```

Good

```
Commit message: "Fix issue #123: Resolve null reference exception in UserService"
Commit message: "Add validation for email format in registration form"
Commit message: "Update README.md with setup instructions"
```

Best Practices for Clean Code

9. Refactor Regularly

- Refactor your code frequently to improve readability, performance, and maintainability.
- Use automated tests to ensure refactoring does not introduce bugs.

10. Unit Testing

- Write unit tests for your functions and methods. This ensures your code works as expected and helps to identify issues early.
- Keep tests independent, isolated, and focused on one unit of work.

Best Practices for Clean Code

11. Avoid Magic Numbers and Strings

- Replace magic numbers (literal constants) or hardcoded strings with named constants or enums.

```
Bad Example:
public double CalculatePrice(double price)
{
    return price * 1.25; // What is 1.25? Magic number
}


Good Example:
public double CalculatePrice(double price)
{
    const double TaxRate = 1.25;
    return price * TaxRate;
}
```

12. Consistent Formatting

- Use consistent code formatting tools like linters or formatters to ensure code consistency across your project.

EIU

Naming conventions and styles in C# Programming Language

Naming conventions

C# Coding Standards and Naming Conventions

Object Name	Notation	Length	Plural	Prefix	Suffix	Abbreviation	Char Mask	Underscores
Namespace name	PascalCase	128	Yes	Yes	No	No	[A-z][0-9]	No
Class name	PascalCase	128	No	No	Yes	No	[A-z][0-9]	No
Constructor name	PascalCase	128	No	No	Yes	No	[A-z][0-9]	No
Method name	PascalCase	128	Yes	No	No	No	[A-z][0-9]	No
Method arguments	camelCase	128	Yes	No	No	Yes	[A-z][0-9]	No
Local variables	camelCase	50	Yes	No	No	Yes	[A-z][0-9]	No
Constants name	PascalCase	50	No	No	No	No	[A-z][0-9]	No
Field name Public	PascalCase	50	Yes	No	No	Yes	[A-z][0-9]	No
Field name Private	_camelCase	50	Yes	No	No	Yes	_[A-z][0-9]	Yes
Properties name	PascalCase	50	Yes	No	No	Yes	[A-z][0-9]	No
Delegate name	PascalCase	128	No	No	Yes	Yes	[A-z]	No
Enum type name	PascalCase	128	Yes	No	No	No	[A-z]	No

Naming conventions

Do use PascalCasing for class names and method names:

```
0 references
public class ClientActivity
{
    0 references
    public void ClearStatistics()
    {
        //...
    }
    0 references
    public void CalculateStatistics()
    {
        //...
    }
}
```

Naming conventions

Do use camelCasing for method arguments and local variables:

```
public class UserLog
{
    0 references
    public void Add(LogEvent logEvent)
    {
        int itemCount = logEvent.Items.Count;
        // ...
    }
}
```

Naming conventions

Do not use Hungarian notation or any other type identification in identifiers

```
// Correct
int counter;
string name;

// Avoid
int iCounter;
string strName;
```

Naming conventions

Do not use Screaming Caps for constants or readonly variables:

```
// Correct
0 references
public const string ShippingType = "DropShip";
// Avoid
0 references
public const string SHIPPINGTYPE = "DropShip";
```

Naming conventions

Use meaningful names for variables. The following example uses binhduongCustomers for customers who are located in Binh Duong:

```
var binhduongCustomers = from customer in customers
  where customer.City == "Bình Dương"
  select customer.Name;
```

Naming conventions

Avoid using Abbreviations. Exceptions: abbreviations commonly used as names, such as Id, Xml, Ftp, Uri.

```
// Correct
UserGroup userGroup;
Assignment employeeAssignment;
// Avoid
UserGroup usrGrp;
Assignment empAssignment;
// Exceptions
CustomerId customerId;
XmlDocument xmlDocument;
FtpHelper ftpHelper;
UriPart uriPart;
```

Naming conventions

Do use PascalCasing or camelCasing (Depending on the identifier type) for abbreviations 3 characters or more (2 chars are both uppercase when PascalCasing is appropriate or inside the identifier).:

```
HtmlHelper htmlHelper;
FtpTransfer ftpTransfer, fastFtpTransfer;
UIControl uiControl, nextUIControl;
```

Naming conventions

Do not use Underscores in identifiers.
Exception: you can prefix private fields with an underscore:

```
1   // Correct
    0 references
2   public DateTime clientAppointment;
    0 references
3   public TimeSpan timeLeft;
4   // Avoid
    0 references
5   public DateTime client_Appointment;
    0 references
6   public TimeSpan time_Left;
7   // Exception (Class field)
    0 references
8   private DateTime _registrationDate;
```

Naming conventions

Do use predefined type names (C# aliases) like int, float, string for local, parameter and member declarations. Do use .NET Framework names like Int32, Single, String when accessing the type's static members like Int32.TryParse or String.Join.

```
1   // Correct
2   string firstName;
3   int lastIndex;
4   bool isSaved;
5   string commaSeparatedNames = String.Join(", ", names);
6   int index = Int32.Parse(input);
7   // Avoid
8   String firstName;
9   Int32 lastIndex;
10  Boolean isSaved;
11  string commaSeparatedNames = string.Join(", ", names);
12  int index = int.Parse(input);
13
```

Naming conventions

Do use implicit type var for local variable declarations. Exception: primitive types (int, string, double, etc) use predefined names.

```
1    var stream = File.Create(path);
2    var customers = new Dictionary();
3    // Exceptions
4    int index = 100;
5    string timeSheet;
6    bool isCompleted;
```

Naming conventions

Do use noun or noun phrases to name a class.

```
    0 references
1   public class Employee
2   {
3   }
    0 references
4   public class BusinessLocation
5   {
6   }
    0 references
7   public class DocumentCollection
8   {
9   }
```

Naming conventions

Do name source files according to their main classes. Exception: file names with partial classes reflect their source or purpose, e.g. designer, generated, etc.

```
1   // Located in Task.cs
    1 reference
2   public partial class Task
3   {
4   }
5   // Located in Task.generated.cs
    1 reference
6   public partial class Task
7   {
8   }
```

Naming conventions

Do organize namespaces with a clearly defined structure:

```
1   // Examples
2   namespace Company.Technology.Feature.Subnamespace
3   {
4   }
5   namespace Company.Product.Module.SubModule
6   {
7   }
8   namespace Product.Module.Component
9   {
10  }
11  namespace Product.Layer.Module.Group
12  {
13  }
```

Naming conventions

Do vertically align curly brackets:

```
1   // Correct
    0 references
2   class Program
3   {
        0 references
4       static void Main(string[] args)
5       {
6           //...
7       }
8   }
9
```

Naming conventions

Do declare all member variables at the top of a class, with static variables at the very top.

```
// Correct
public class Account
{
  public static string BankName;
  public static decimal Reserves;
  public string Number { get; set; }
  public DateTime DateOpened { get; set; }
  public DateTime DateClosed { get; set; }
  public decimal Balance { get; set; }
  // Constructor
  public Account()
  {
    // ...
  }
}
```

Naming conventions

Do use singular names for enums. Exception: bit field enums.

```
// Correct
public enum Color
{
   Red,
   Green,
   Blue,
   Yellow,
   Magenta,
   Cyan
}
```

```
// Exception
[Flags]
public enum Dockings
{
   None = 0,
   Top = 1,
   Right = 2,
   Bottom = 4,
   Left = 8
}
```

Naming conventions

Do not explicitly specify a type of an enum or values of enums (except bit fields):

```
// Don't
public enum Direction : long
{
  North = 1,
  East = 2,
  South = 3,
  West = 4
}
```

```
// Correct
public enum Direction
{
  North,
  East,
  South,
  West
}
```

```
[Flags]
public enum FileAccess
{
    Read = 1,        // 0001
    Write = 2,       // 0010
    Execute = 4,     // 0100
    ReadWrite = Read | Write
}
```

Khi nào nên chỉ định giá trị cụ thể?
Trường hợp phổ biến khi cần gán giá trị cụ thể là khi sử dụng bit field (các cờ trạng thái). Lúc này, việc gán giá trị là cần thiết để đảm bảo các phép toán bitwise hoạt động đúng.

Naming conventions

Do not use an "Enum" suffix in enum type names:

```
// Don't
public enum CoinEnum
{
  Penny,
  Nickel,
  Dime,
  Quarter,
  Dollar
}
```

```
// Correct
public enum Coin
{
  Penny,
  Nickel,
  Dime,
  Quarter,
  Dollar
}
```

Naming conventions

Do not use "Flag" or "Flags" suffixes in enum type names:

```
// Don't
[Flags]
public enum DockingsFlags
{
  None = 0,
  Top = 1,
  Right = 2,
  Bottom = 4,
  Left = 8
}
```

```
// Correct
[Flags]
public enum Dockings
{
  None = 0,
  Top = 1,
  Right = 2,
  Bottom = 4,
  Left = 8
}
```

Naming conventions

Do use suffix EventArgs at creation of the new classes comprising the information on event:

```
// Correct
public class BarcodeReadEventArgs : System.EventArgs
{
}
```

- The .NET Framework and .NET Core follow a consistent naming convention for event argument classes, using the suffix EventArgs. This makes it easier for developers to understand that the class is associated with an event and contains data for that event. It provides a clear and predictable structure for event-related classes.
- By using the EventArgs suffix, the purpose of the class is immediately clear. Anyone reading the code can easily identify that the class is meant to hold event-related data. This enhances the readability of the code and reduces confusion.

Naming conventions

Do name event handlers (delegates used as types of events) with the "EventHandler" suffix, as shown in the following example:

```
public delegate void ReadBarcodeEventHandler(object sender, ReadBarcodeEventArgs e);
```

Naming conventions

Do not create names of parameters in methods (or constructors) which differ only by the register:

```
// Avoid
private void MyFunction(string name, string Name)
{
  //...
}
```

Naming conventions

Do use two parameters named sender and e in event handlers. The sender parameter represents the object that raised the event. The sender parameter is typically of type object, even if it is possible to employ a more specific type.

```
1 public void ReadBarcodeEventHandler(object sender, ReadBarcodeEventArgs e)
2 {
3     //...
4 }
```

Naming conventions

Do use suffix Exception at creation of the new classes comprising the information on exception:

```
1 // Correct
2 public class BarcodeReadException : System.Exception
3 {
4
5 }
```

Naming conventions

Do use prefix Any, Is, Have or similar keywords for boolean identifier:

```
1 // Correct
2 public static bool IsNullOrEmpty(string value)
3 {
4     return (value == null || value.Length == 0);
5 }
```

Naming conventions

Use Named Arguments in method calls:
When calling a method, arguments are passed with the parameter name followed by a colon and a value.

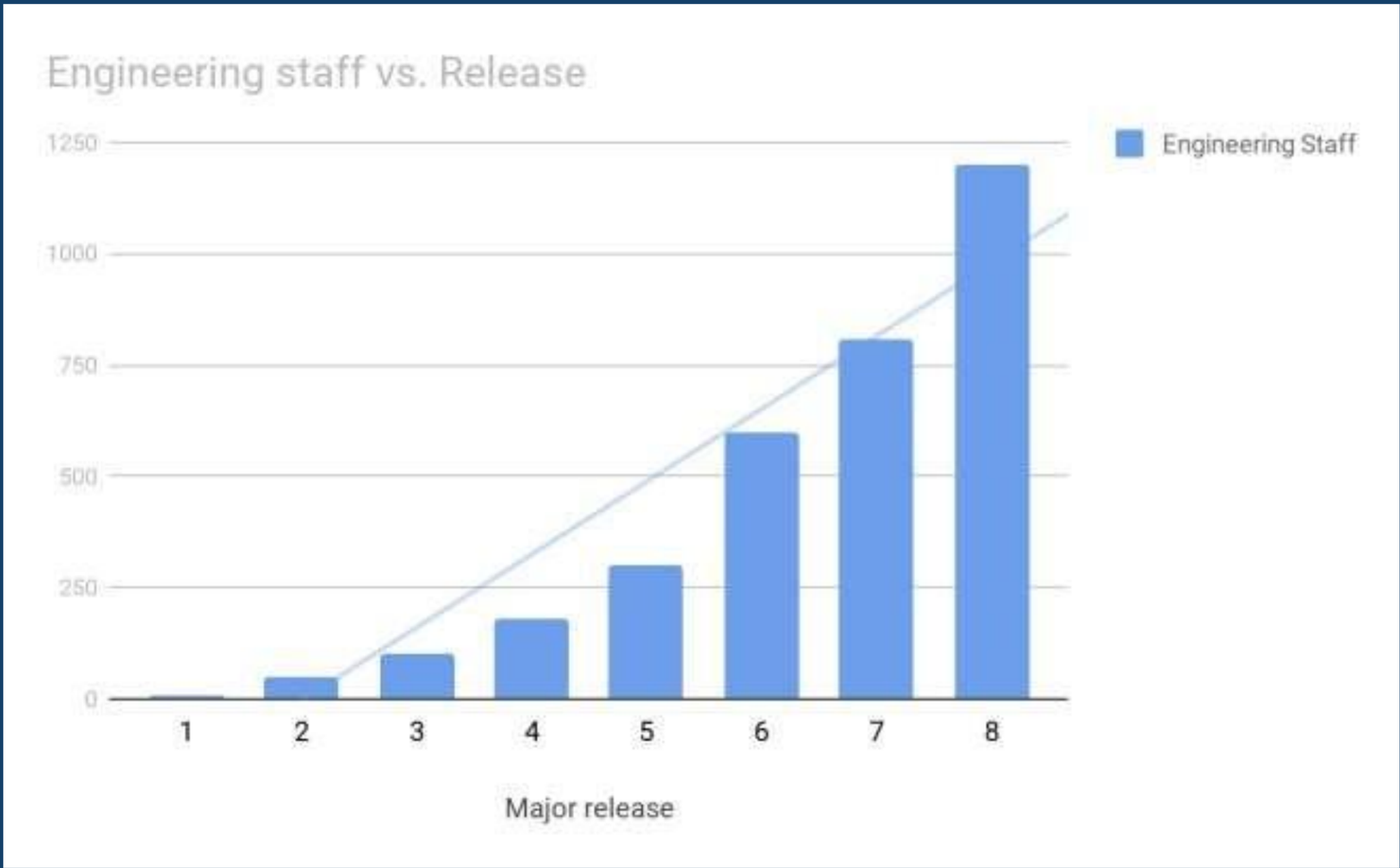
```
// Method
public void DoSomething(string foo, int bar)
{
...
}

// Avoid
DoSomething("someString", 1);
// Correct
DoSomething(foo: "someString", bar: 1);
```

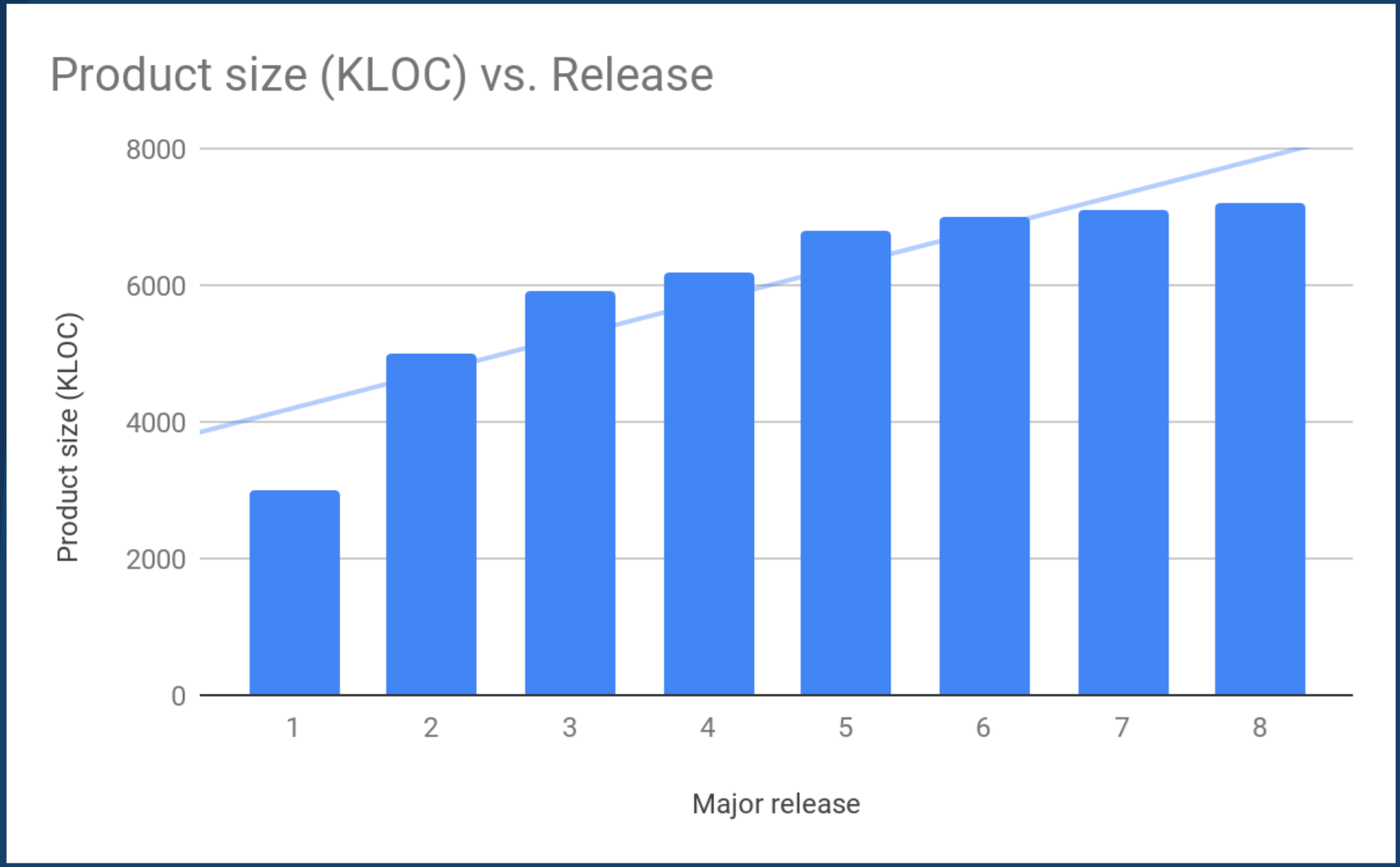
Coding structure

Coding structure

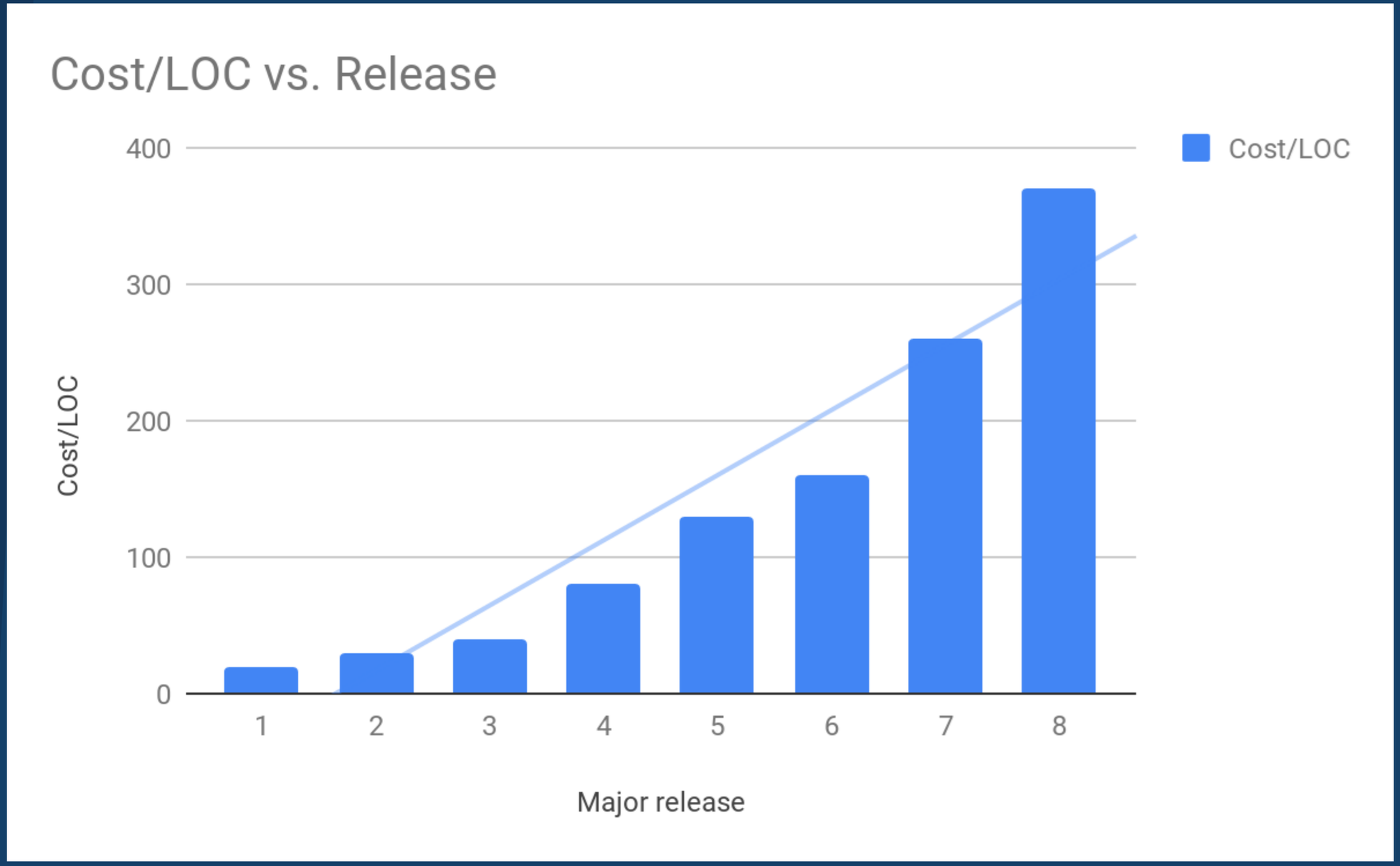
Case study for market leading software



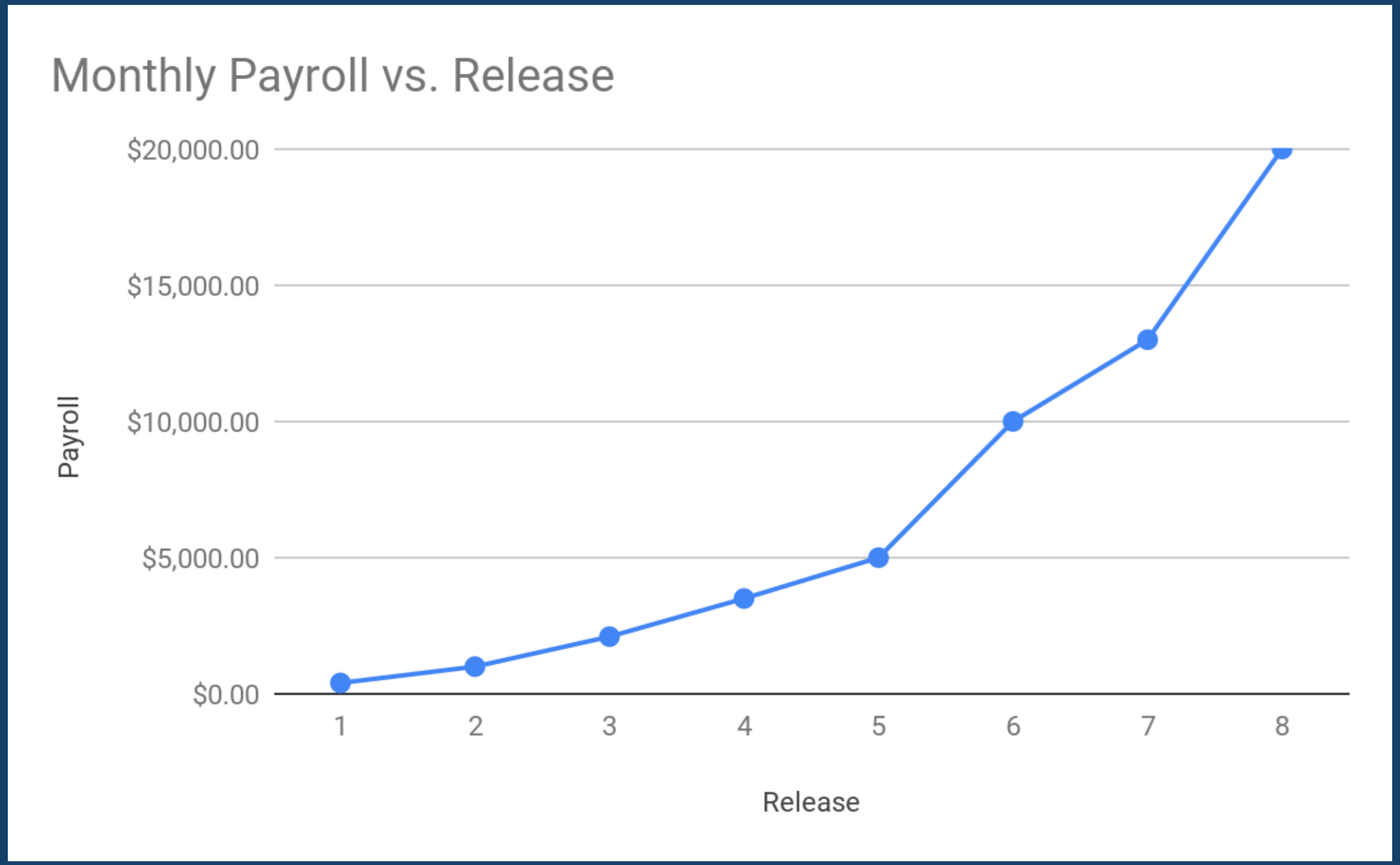
Coding structure



Coding structure



Coding structure



Coding structure

Why?

Accumulated technical debt

Bad software design

Coding structure

Problem:
So many bad architecture projects in the wild

Coding structure

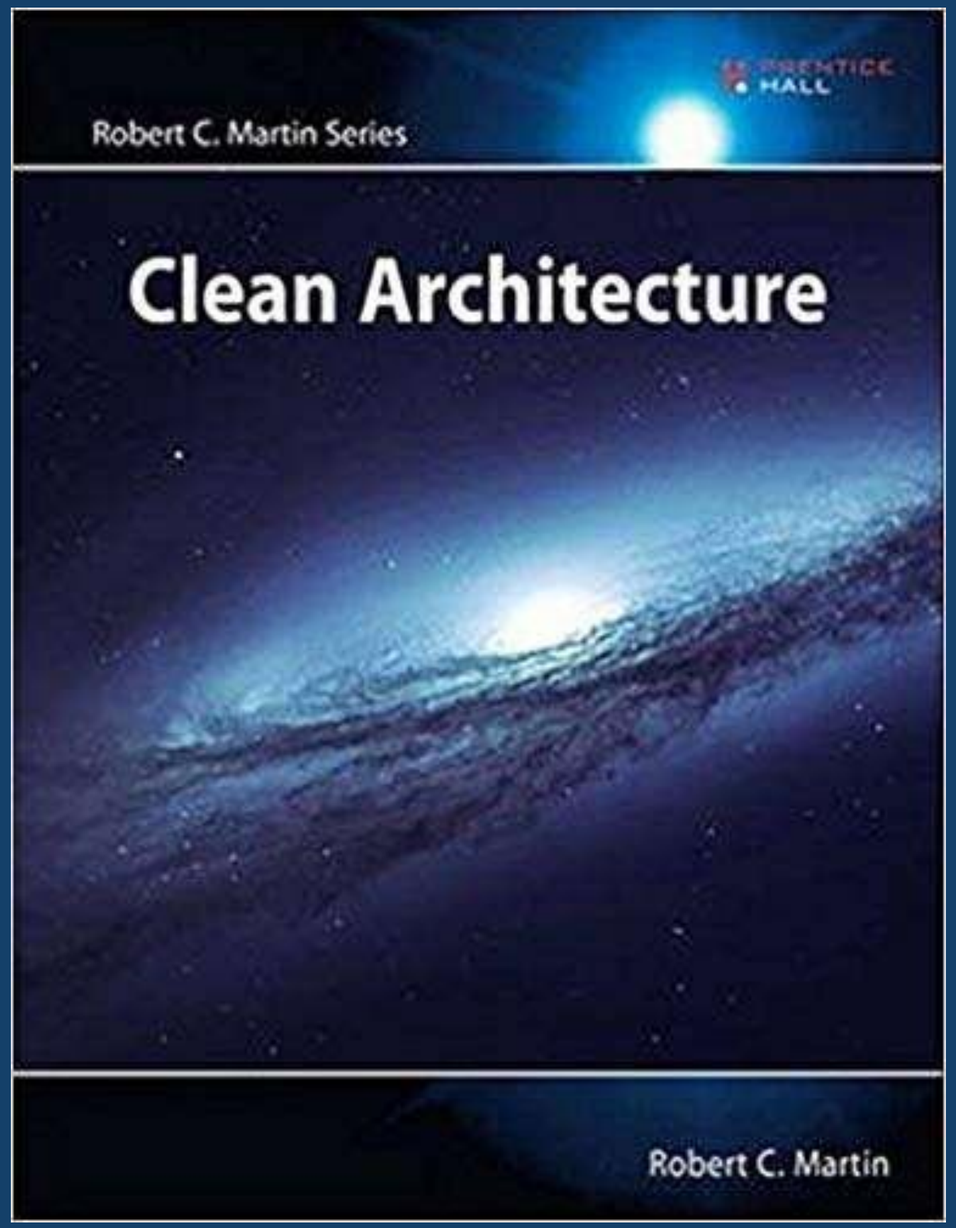
The goal of software architecture is to minimize the human effort required to build and maintain the required system

Coding structure

What is good software architecture ?

- Good architecture must be time-proof
- Architecture must change because of new business requirements, not because of new technologies
- Good architecture must clearly communicate code purpose
- Being flexible in wrong places is bad example of software architecture
- Good architecture let's defer major decisions

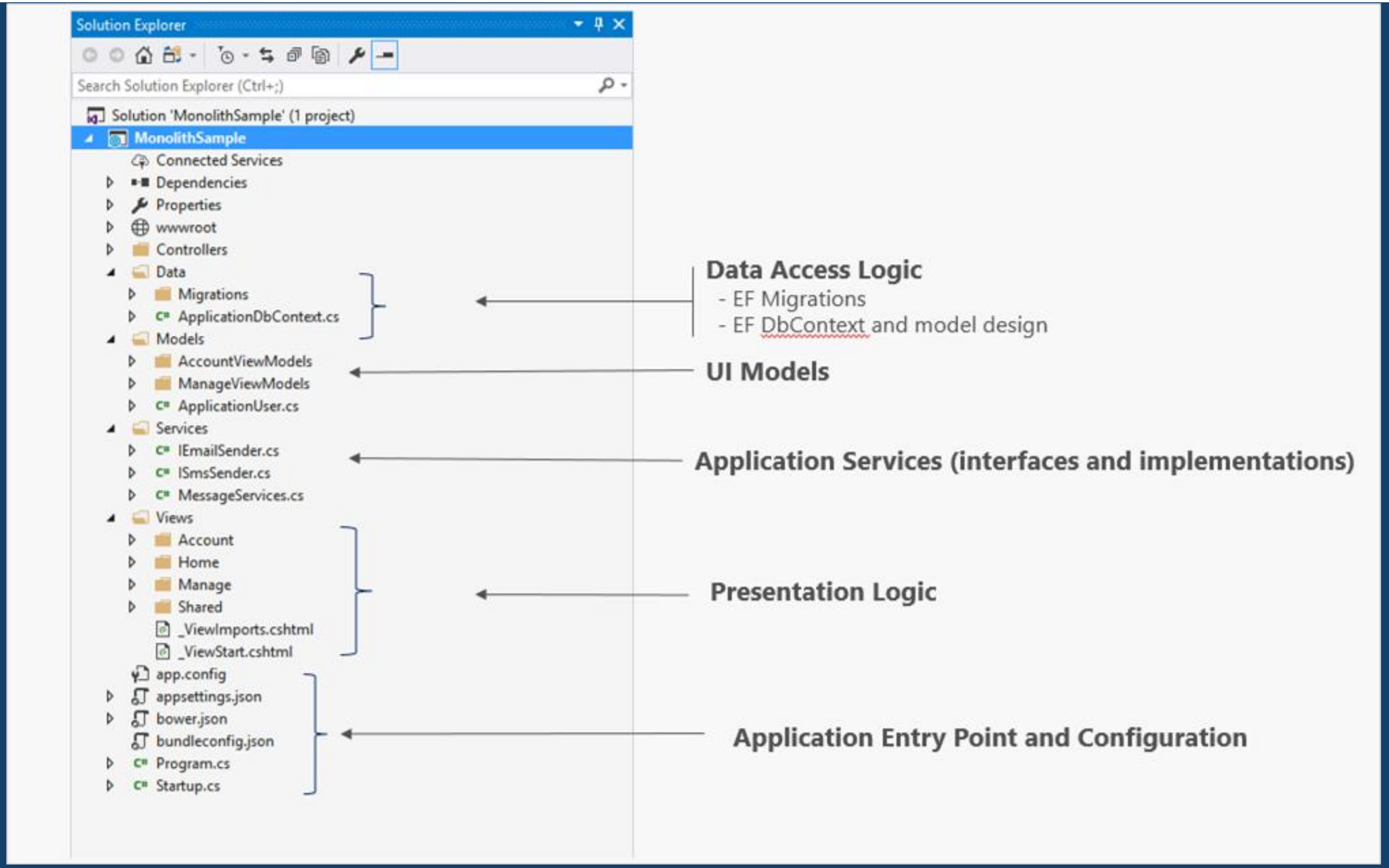
Coding structure



Robert C. Martin

Coding structure

All-in-one applications - Example



Coding structure

Traditional Project Structure

- Simple
- As the project's size and complexity grows, the number of files and folders will continue to grow as well.
- User interface (UI) concerns (models, views, controllers) reside in multiple folders
- Business logic is scattered between the Models and Services folders, and there's no clear indication of which classes in which folders should depend on which others

Coding structure

Multiple-layers architecture applications

- Break up the application according to its responsibilities or concerns

- By organizing code into layers, common low-level functionality can be reused throughout the application.

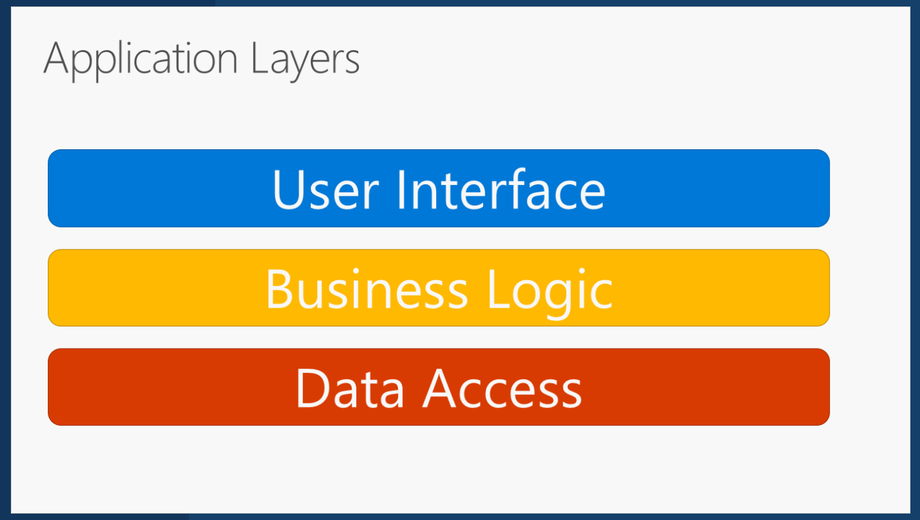
    - Less code needs to be written

    - Follow the don't repeat yourself (DRY) principle.

- When a layer is changed or replaced, only those layers that work with it should be impacted. By limiting which layers depend on which other layers, the impact of changes can be mitigated so that a single change doesn't impact the entire application

Coding structure

Traditional "N-Layer" architecture applications

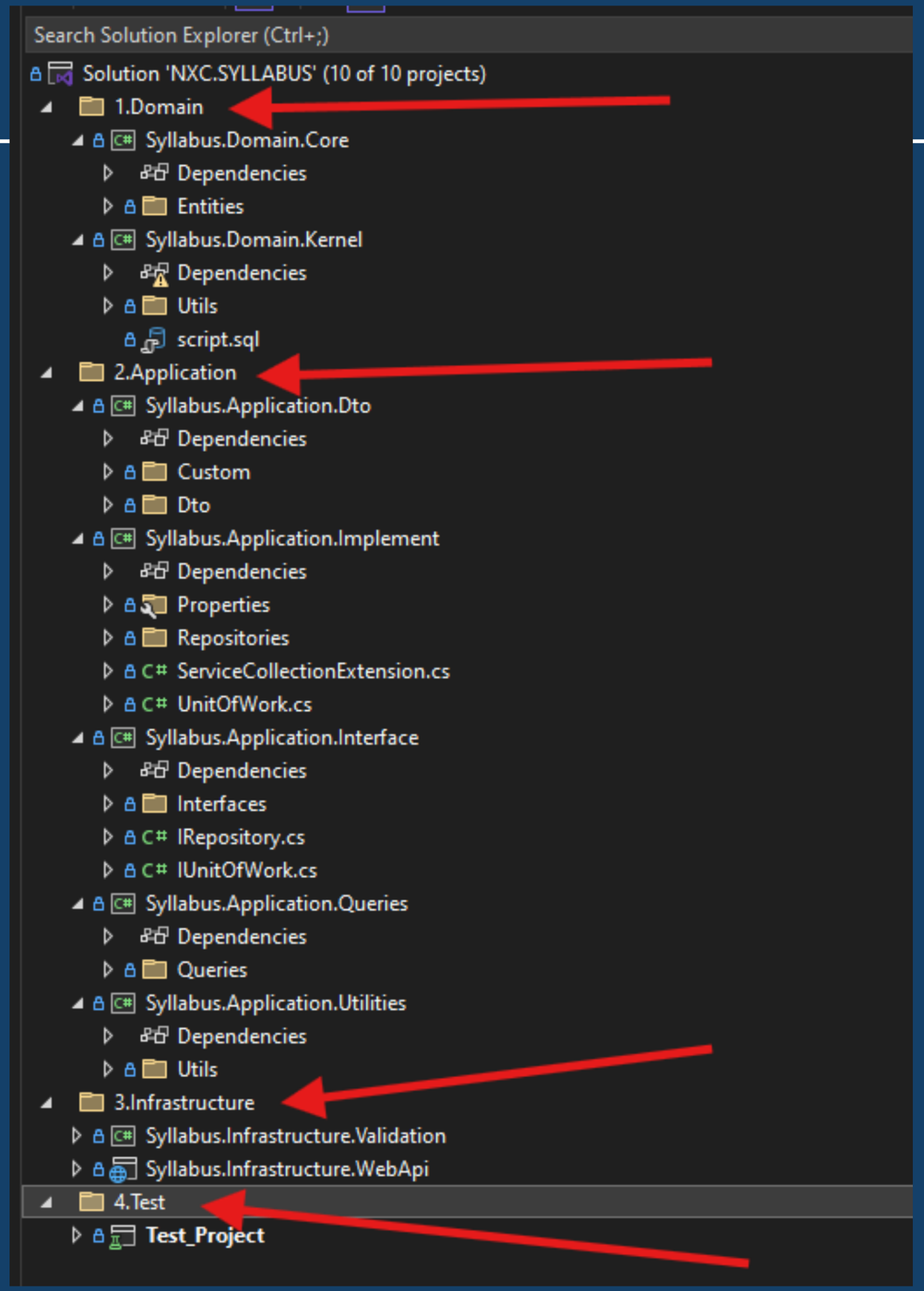


- BLL, which usually holds the most important logic in the application, is dependent on data access implementation details (and often on the existence of a database).

- Testing business logic in such an architecture is often difficult, requiring a test database. The dependency inversion principle can be used to address this issue, as you'll see in the next section.

Coding structure

Traditional "N-Layer" architecture applications



Coding structure

Clean architecture

- Architectural style suitable for all platforms, languages and domains
- Abstract, more like “mindset” instead of concrete rules / frameworks
- Easily adoptable

Coding structure

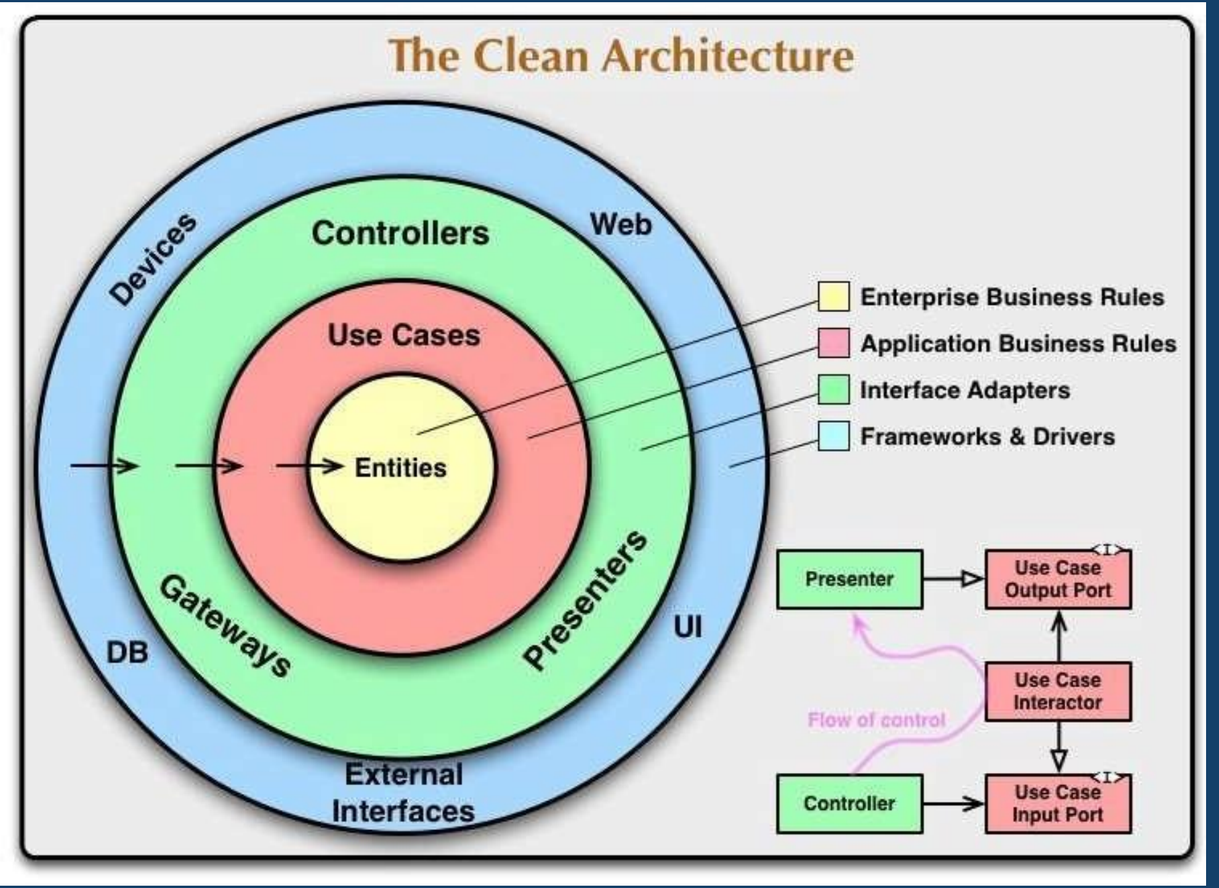
Clean architecture

- Easily testable
- Independent from database
- Independent from frameworks
- Always ready to deploy
- High isolation of modules

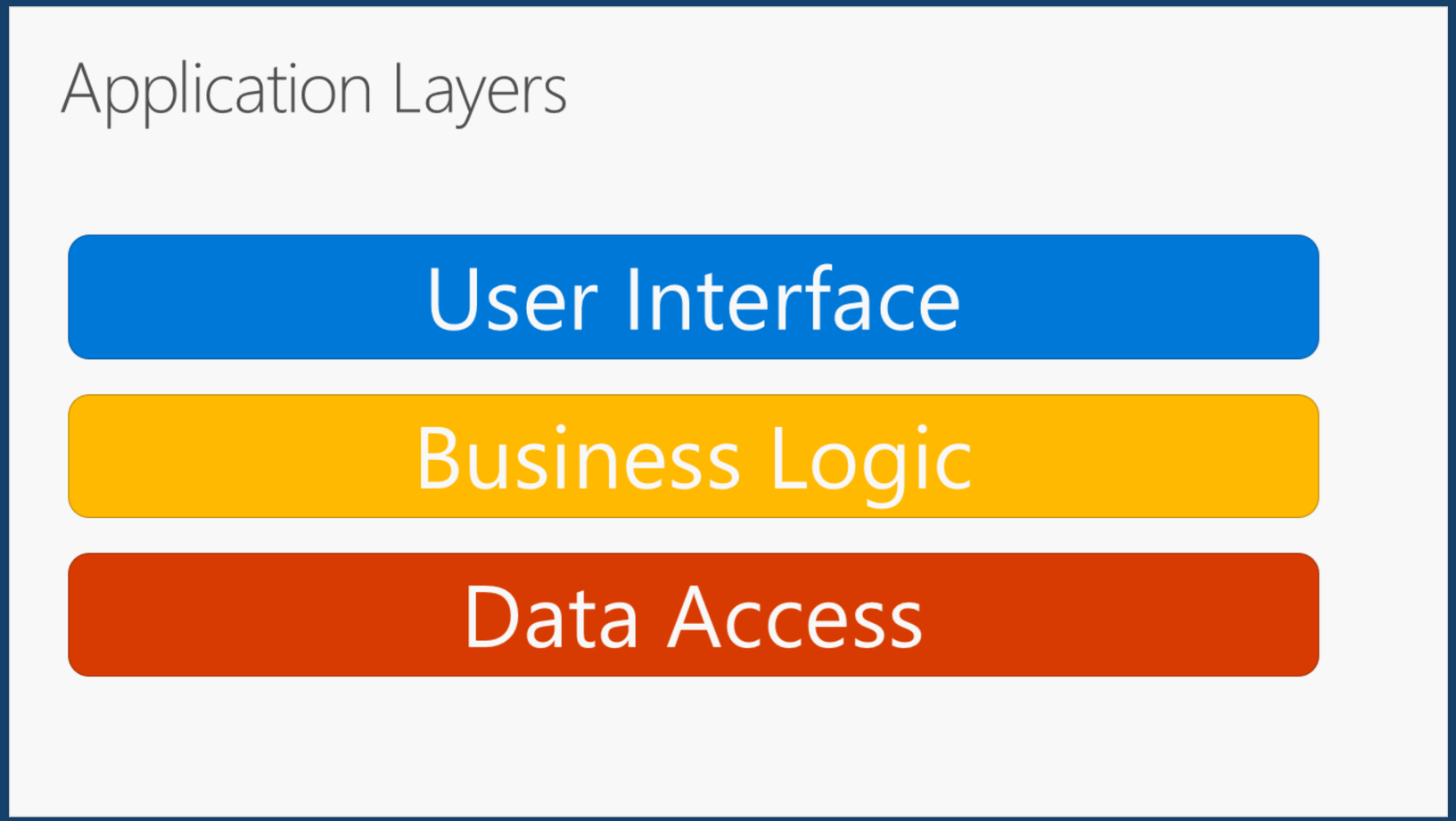
Coding structure

Clean architecture is not always right for small projects

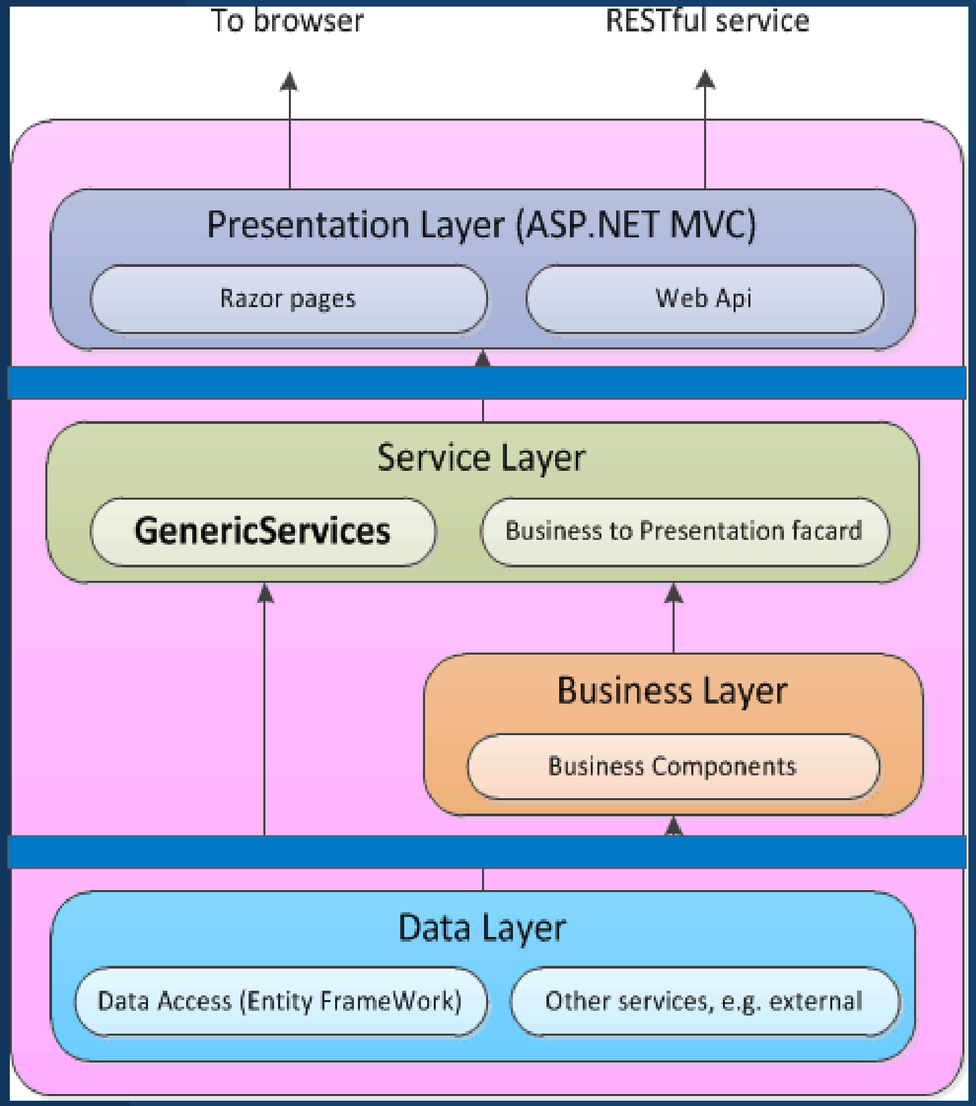
Coding structure

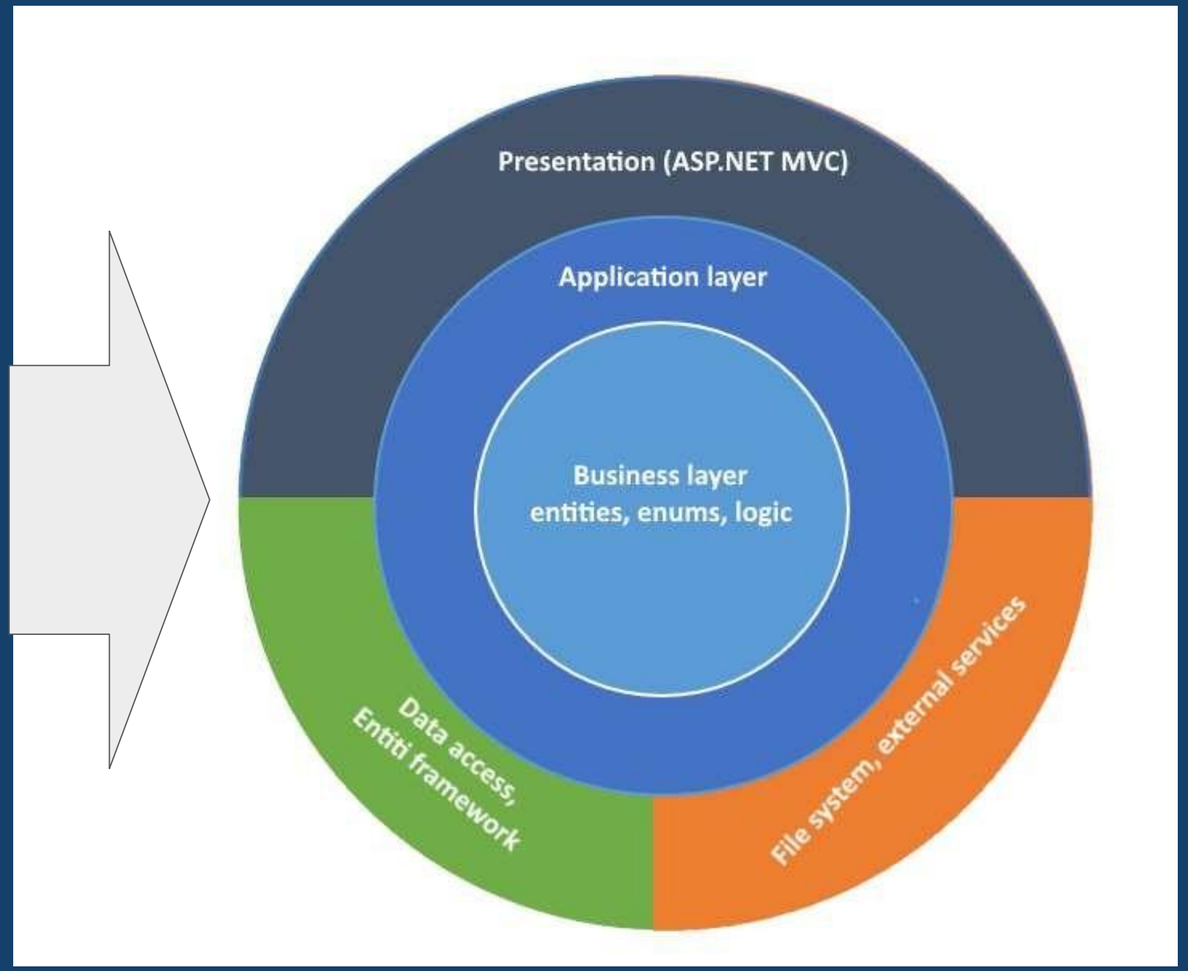


Coding structure

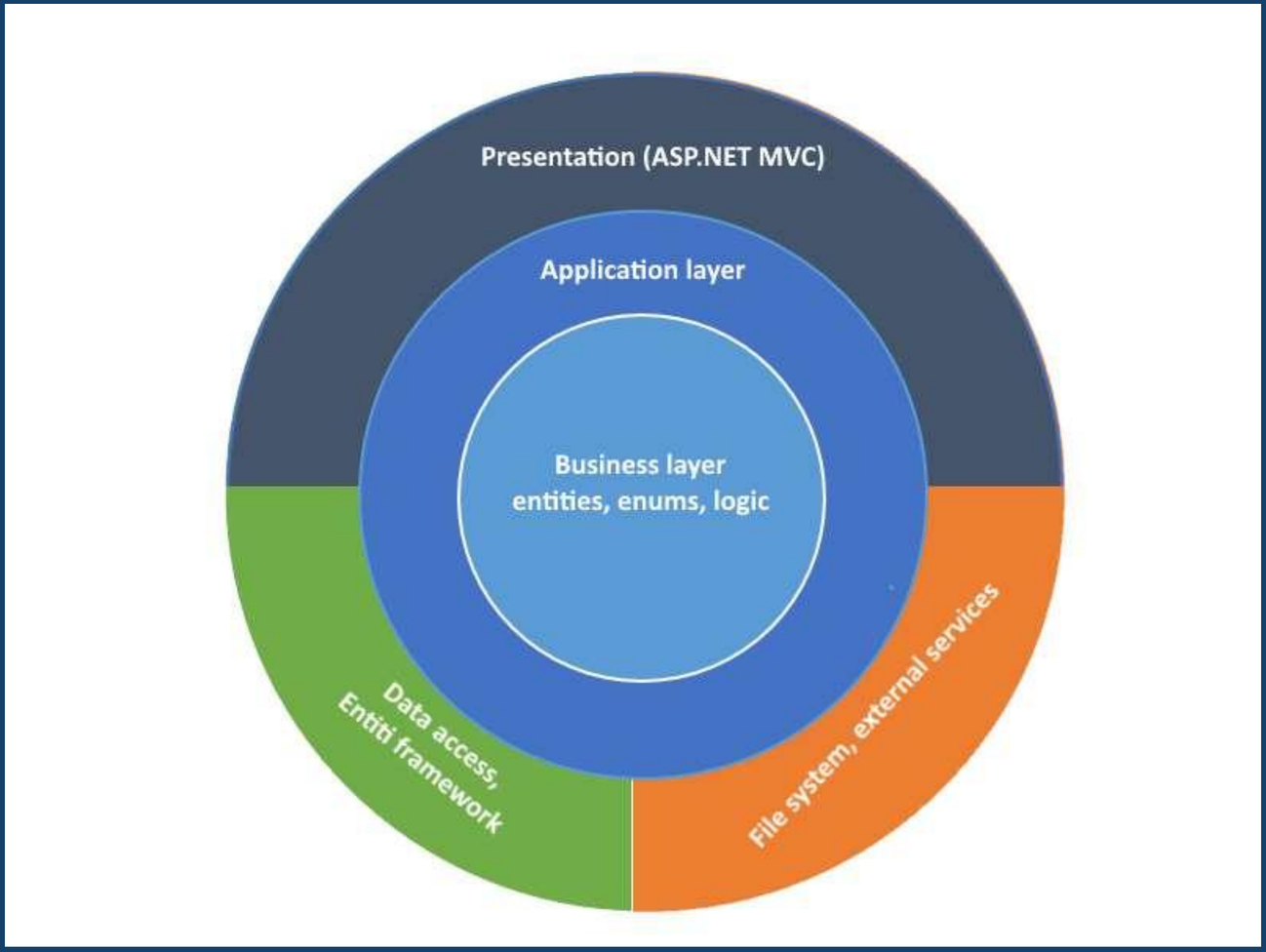


Coding structure





Coding structure



Coding structure

The center of your application is not the database. Nor is it one or more of the frameworks you may be using. The center of your application is the use cases of your application

Coding structure

What is use-case ?

Wikipedia:

a use case is a list of actions or event steps typically defining the interactions between a role (known in the Unified Modeling Language as an actor) and a system to achieve a goal.

Clean Architecture book:

These use cases orchestrate the flow of data to and from the entities, and direct those entities to use their Critical Business Rules to achieve the goals of the use case.

Coding structure

Two cases of business rules

Business specific business rules
- Fundamental business rules

Application specific business rules
- For example communication with user, validation

It’s important to separate those rules in application architecture

Coding structure

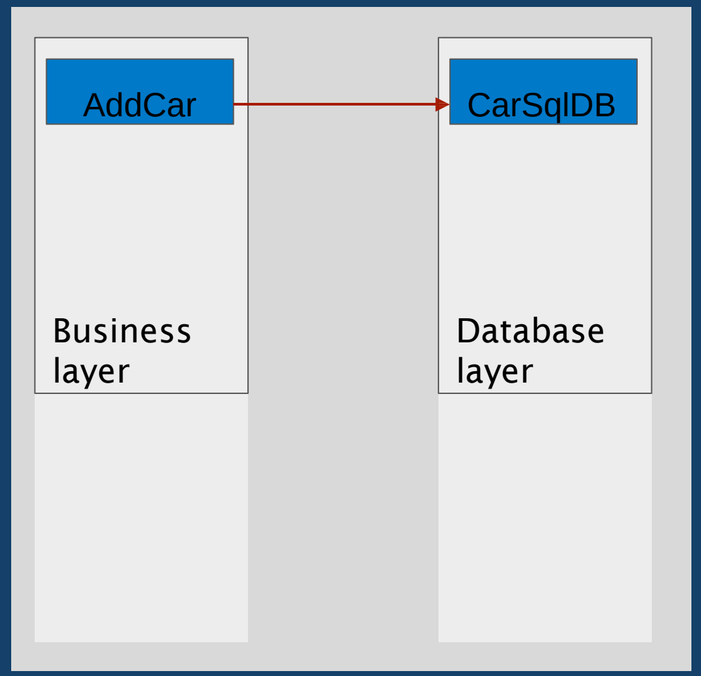
Software dependency

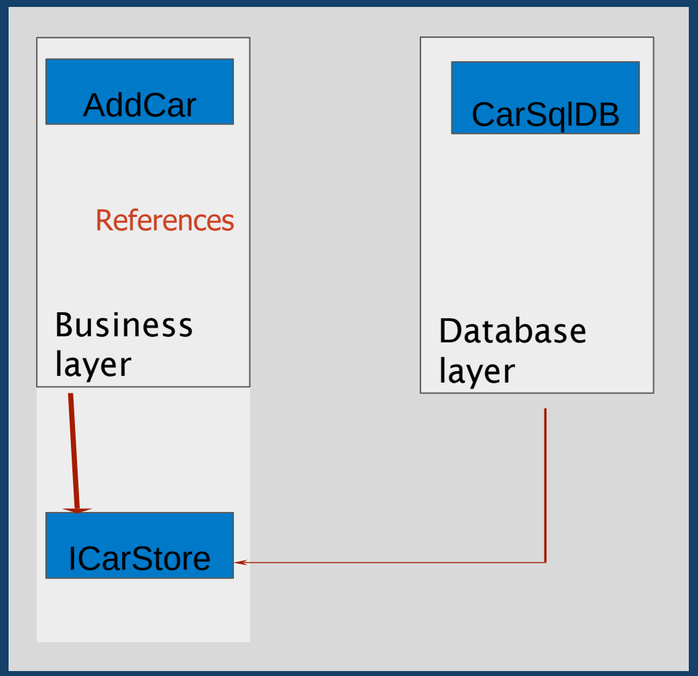
In clean architecture we INVERT software dependencies for

- Database
- Framework
- UI

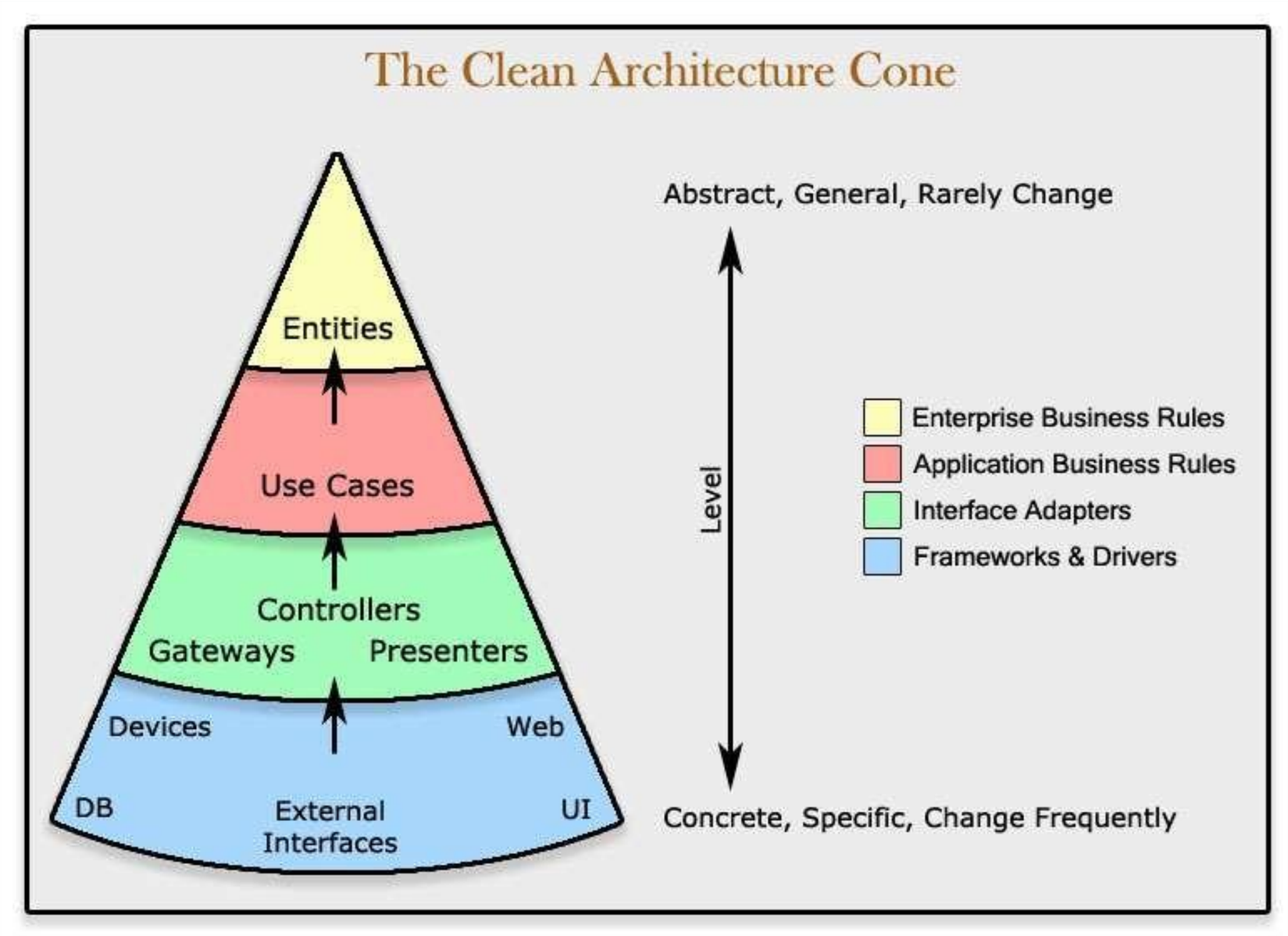
Coding structure

How to invert dependency





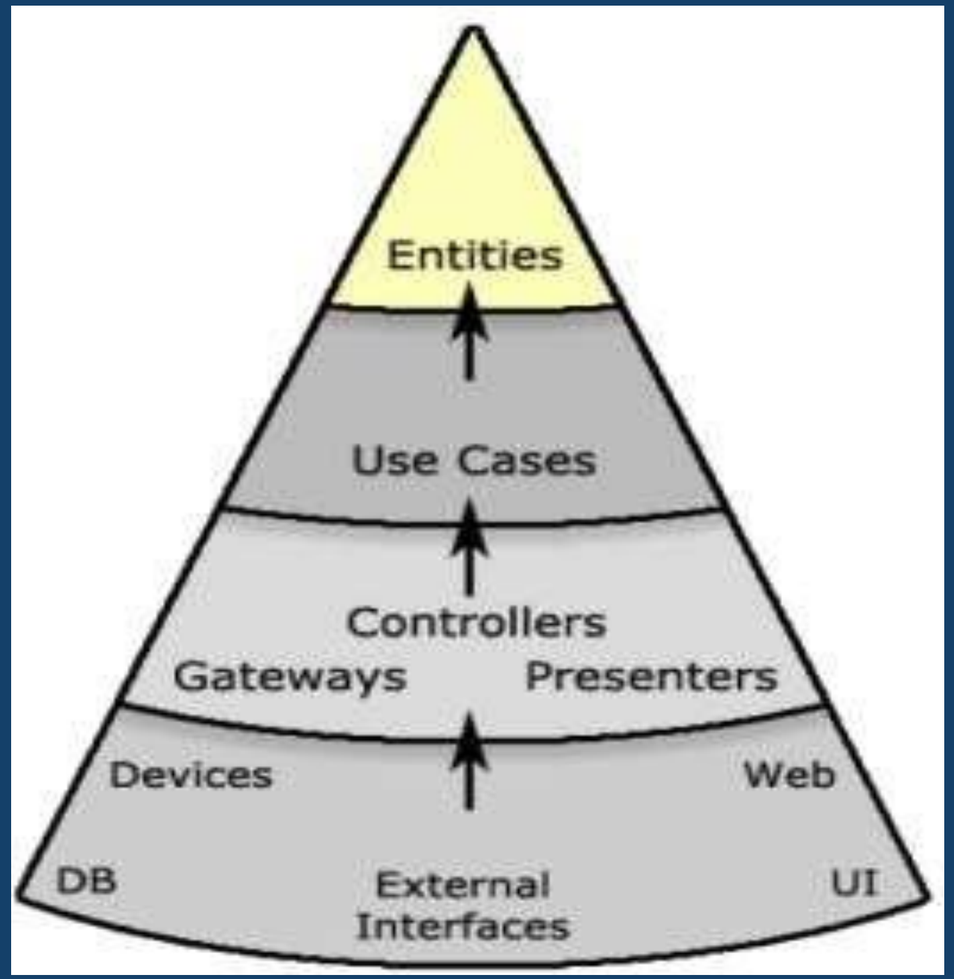
Coding structure



Coding structure

Entities

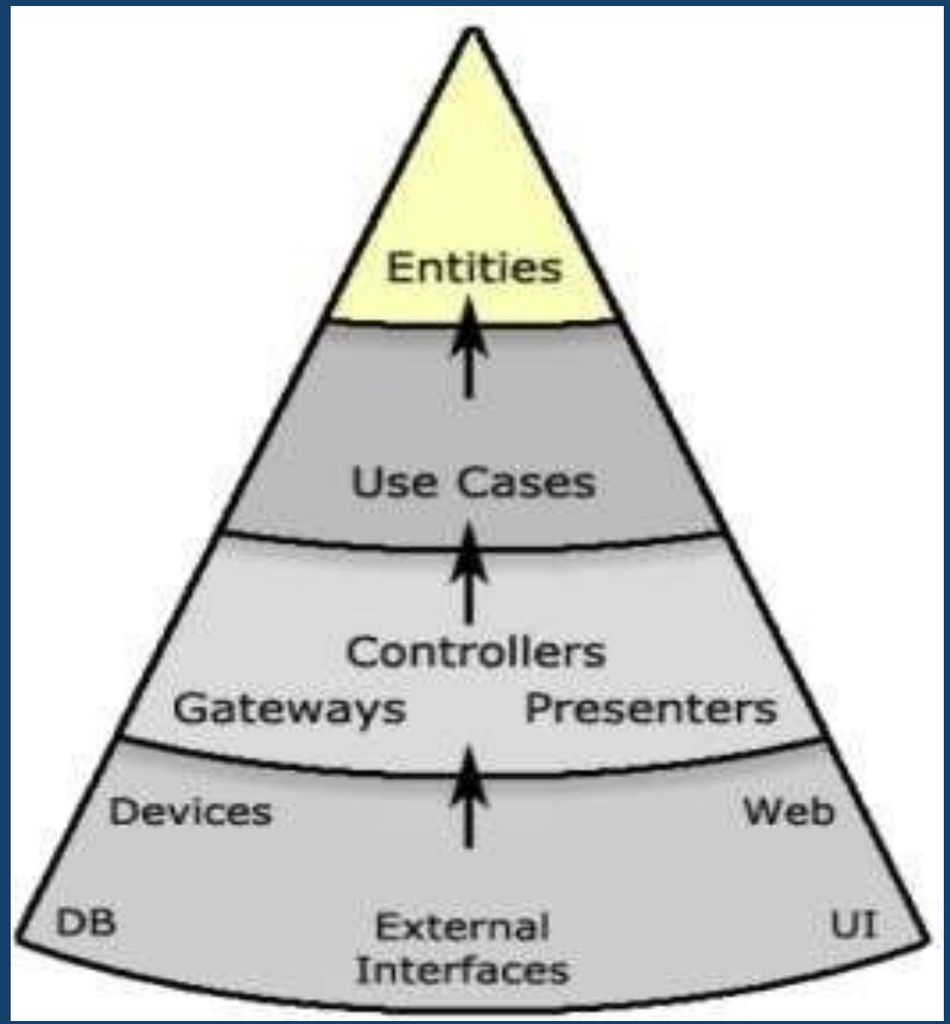
- Represent your domain model
- Entities should be reusable in many applications
- They encapsulate the most general/high-level rules.



Coding structure

Entities

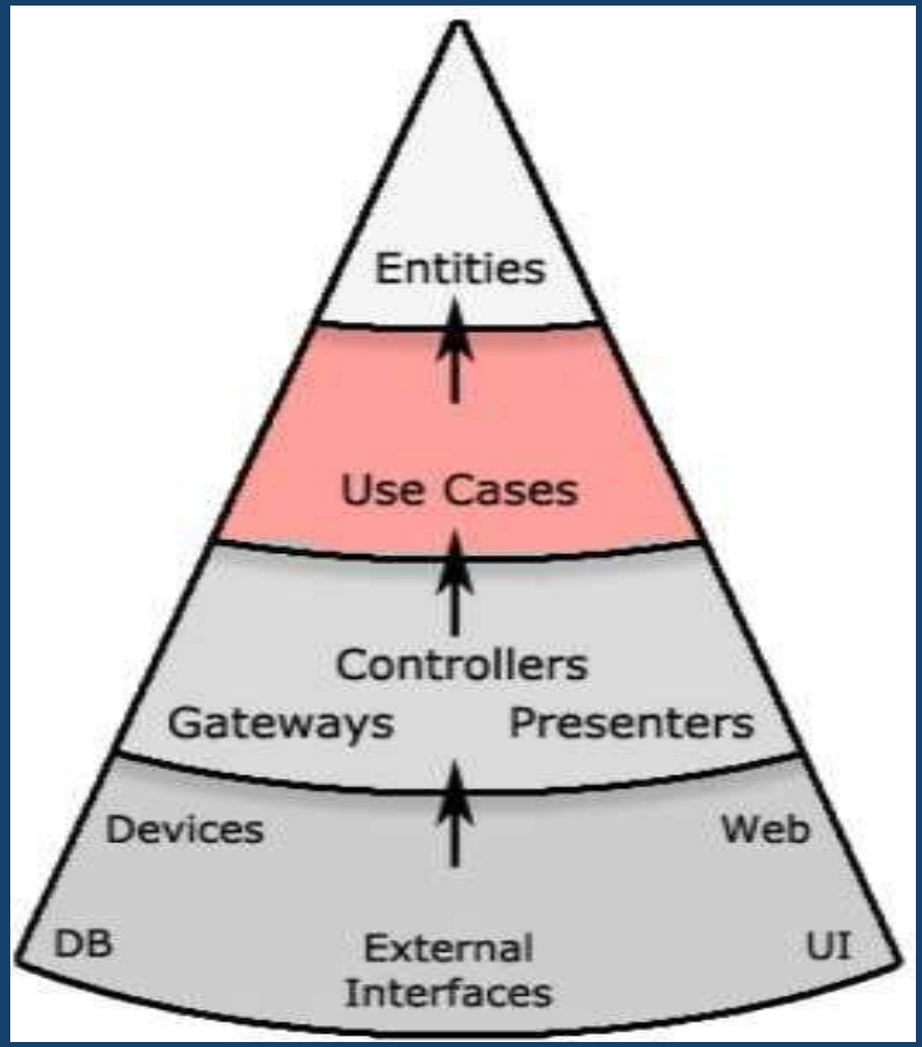
- Entity layer must contain only POCO (Plain old CLR object) objects
  - Easy to migrate between different .NET versions
  - Easy serialization and caching
  - Easy testable



Coding structure

Use Cases

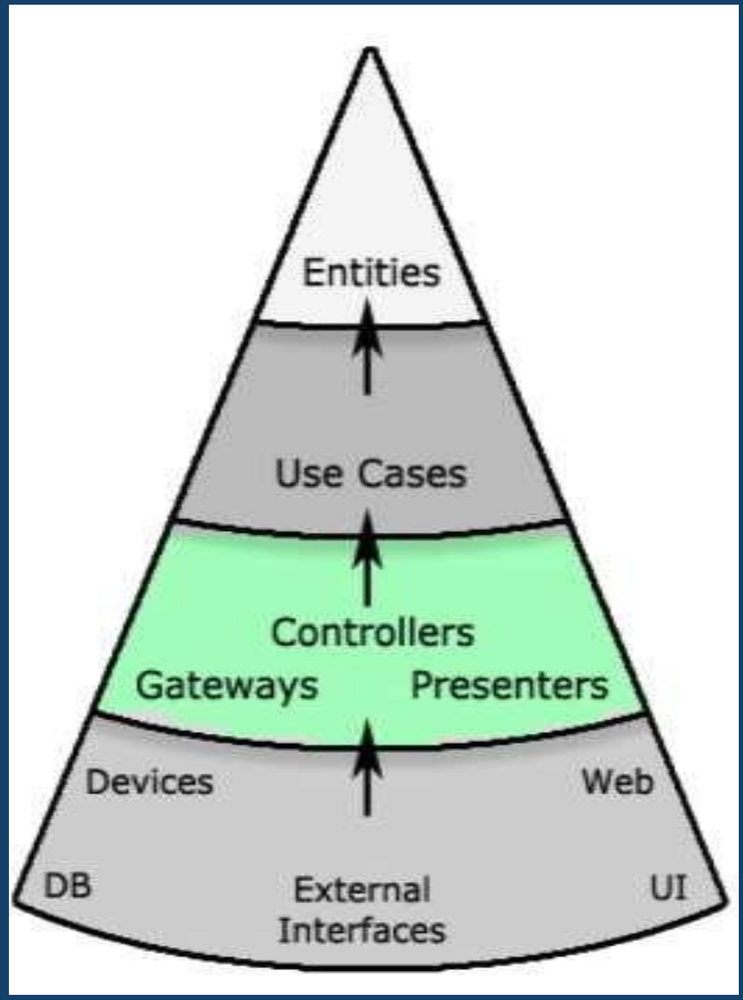
- Use cases are application specific business rules
  - Changes should not impact the Entities
  - Changes should not be impacted by infrastructure such as a database
- Orchestrate data flow from and to Entities
- Validators, exceptions, commands, queries and interfaces



Coding structure

Controllers / Gateways / Presenters

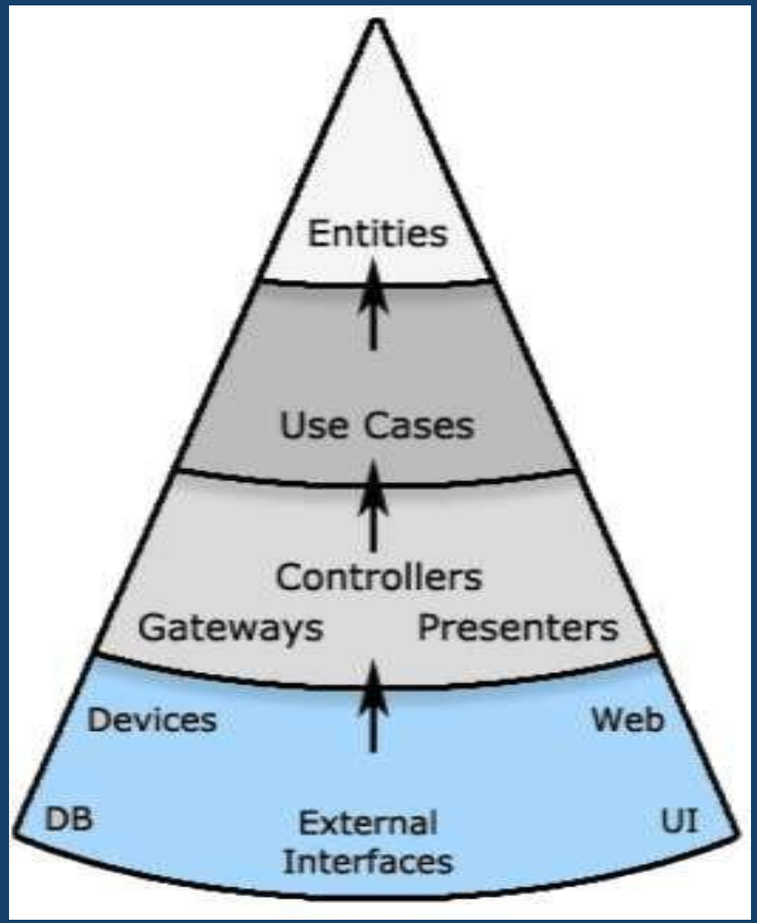
- Layer where 3rd party code can be used
    - MVC, API controllers, presenters, all belong here
    - Entity Framework: DbContext, Migrations, Configurations
- Implements interfaces defined in Use-Cases level
- No code further in (use cases, entities) should
    - have any knowledge of the db



Coding structure

DB / Devices / WEB / UI

- OS functions, api clients, file system, GPIO
- Web Server, SQL Server, Redis cache
- You're not writing much of this code, i.e. we use SQL Server, but we don't write it.
- These are the glue that hook the various layers up (IoC container XML config)



Coding structure

Only 4 circles?

- No, the circles are schematic. You may find that you need more than just these four.
- BUT: Dependency Rule always applies. Source code dependencies always point inwards.
- As you move inwards the software grows more abstract, and encapsulates higher level policies. The innermost circle is the most general and most stable

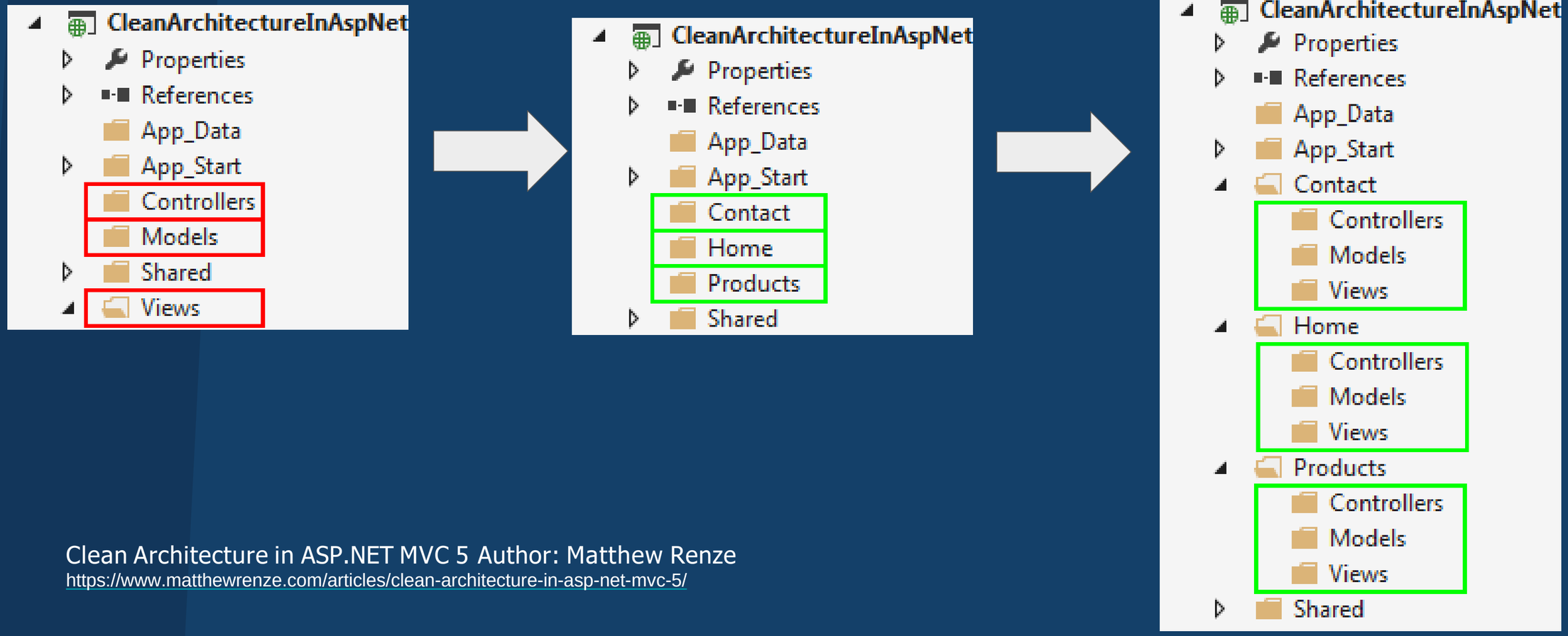
Layers can not skip layers

Coding structure

Crossing boundaries

- Data that can cross boundaries are only simple data structures
- They are called DTOs, ViewModels
- You can not pass database rows or whole entities
- Passed data structure must be most convenient for inner circle

Coding structure



Clean Architecture in ASP.NET MVC 5 Author: Matthew Renze
https://www.matthewrenze.com/articles/clean-architecture-in-asp-net-mvc-5/

Coding structure

What about .NET Framework?

Coding structure

What about MVC?

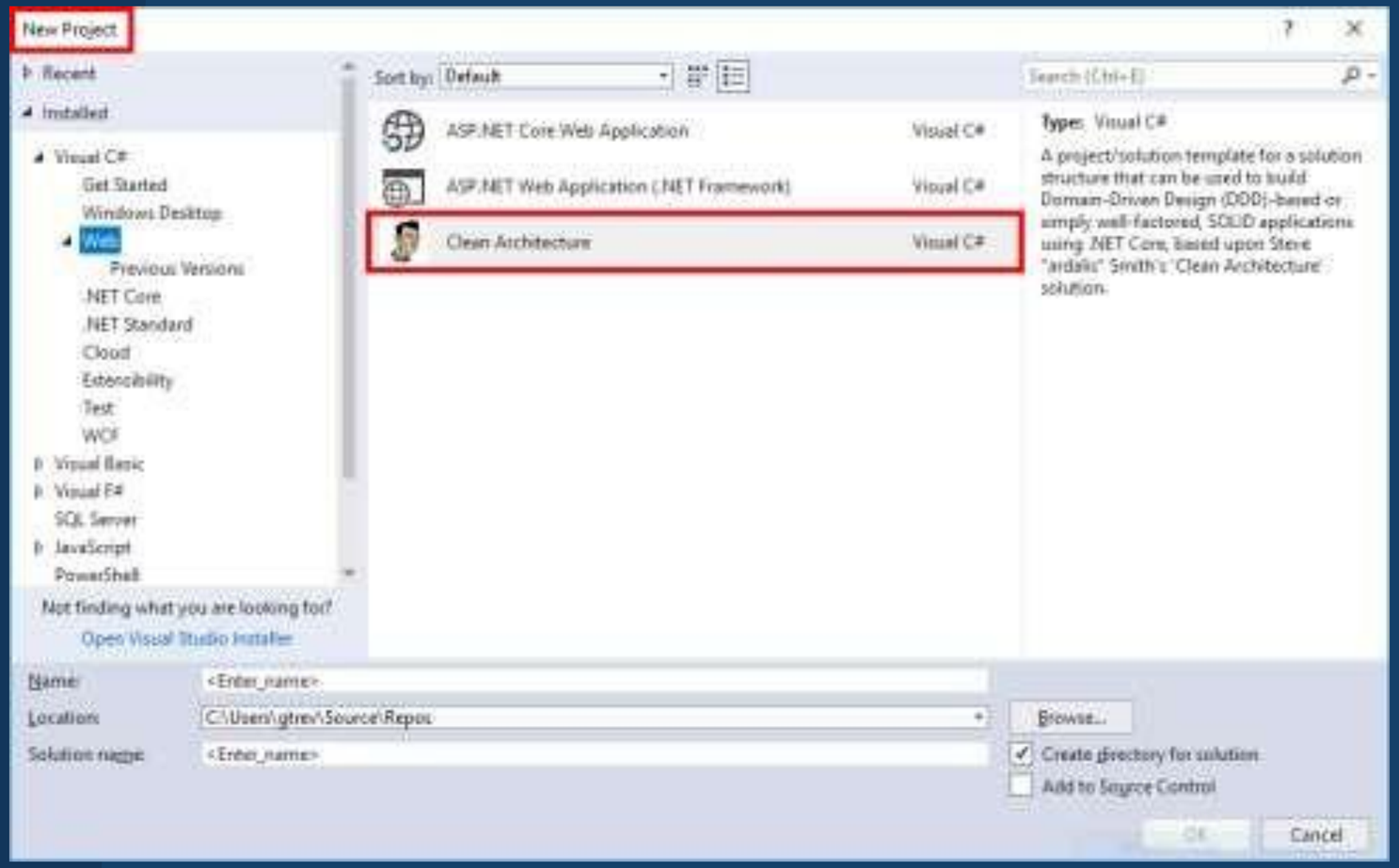
- MVC is not an architecture
  - MVC was introduced in 1970s into Smalltalk
  - Intention was to use for small UI object (button, text input, circle)
  - MVC is a **delivery** design pattern
  - Should be in Presentation layer

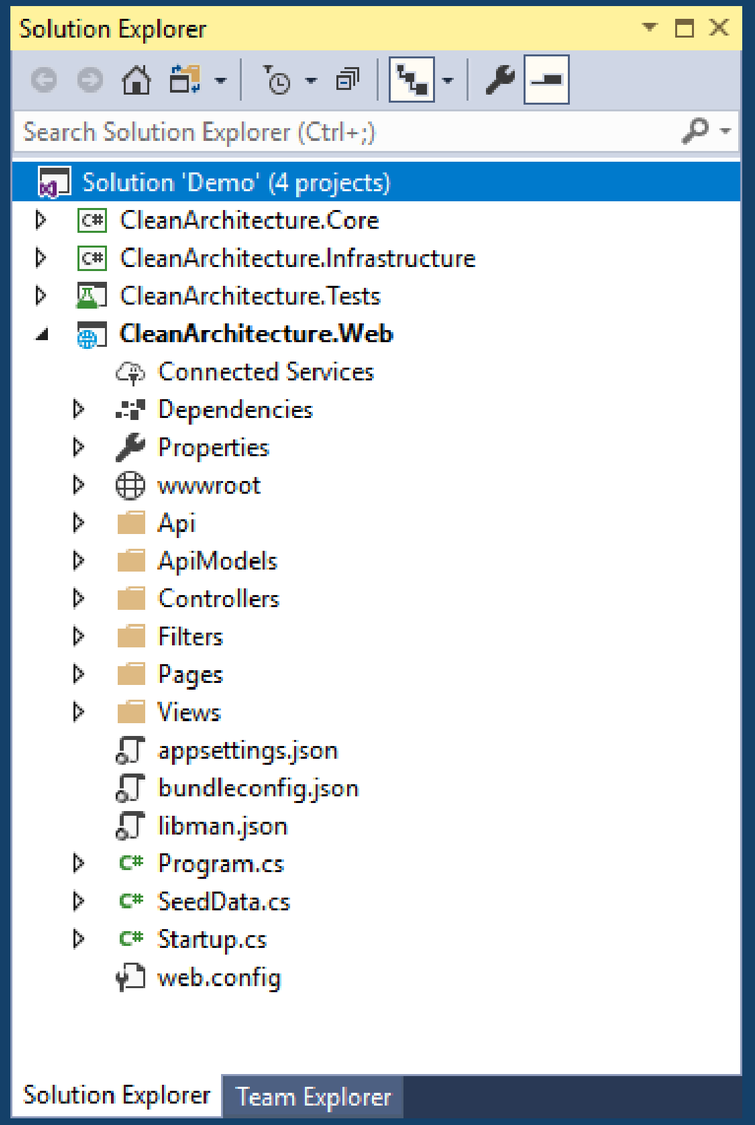
ASP.NET Examples

Coding structure

ASP.NET project template

https://marketplace.visualstudio.com/items?itemName= GregTrevellick.CleanArchitecture



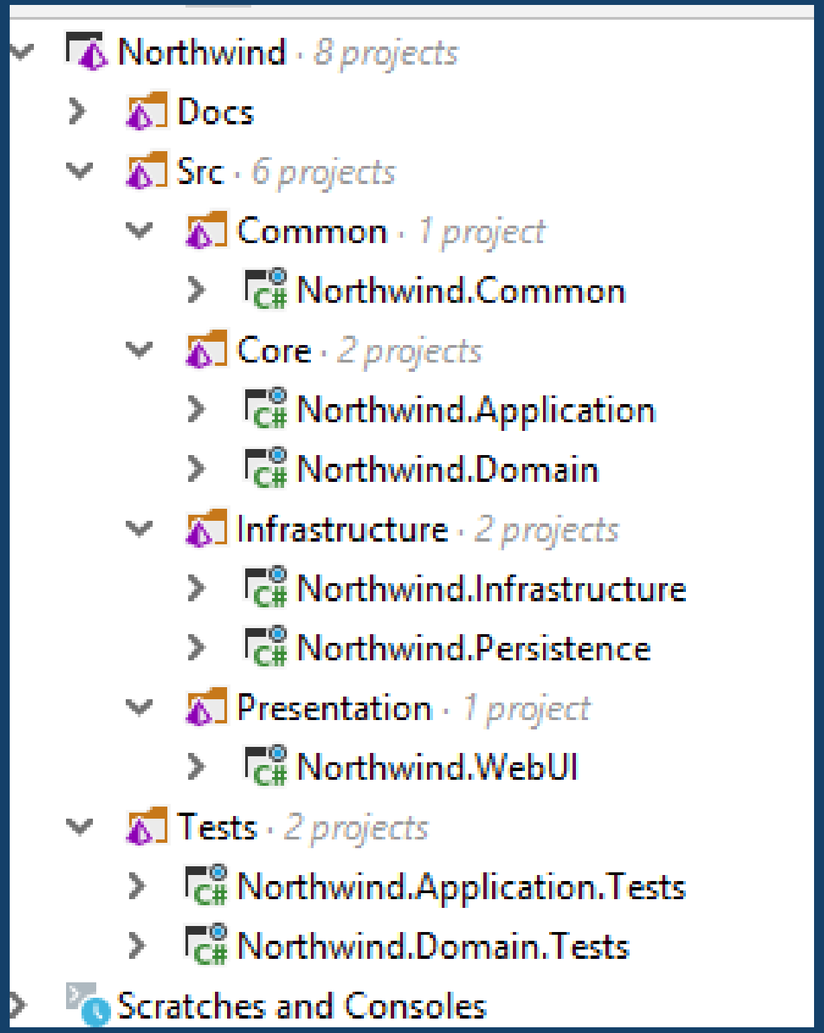


Coding structure

Northwind Traders in ASP .NET Core

https://github.com/JasonGT/NorthwindTraders

Clean Architecture with ASP.NET Core
https://www.youtube.com/watch?v=_lwCVE_XgqI



Coding structure

Clean architecture

♡

KISS

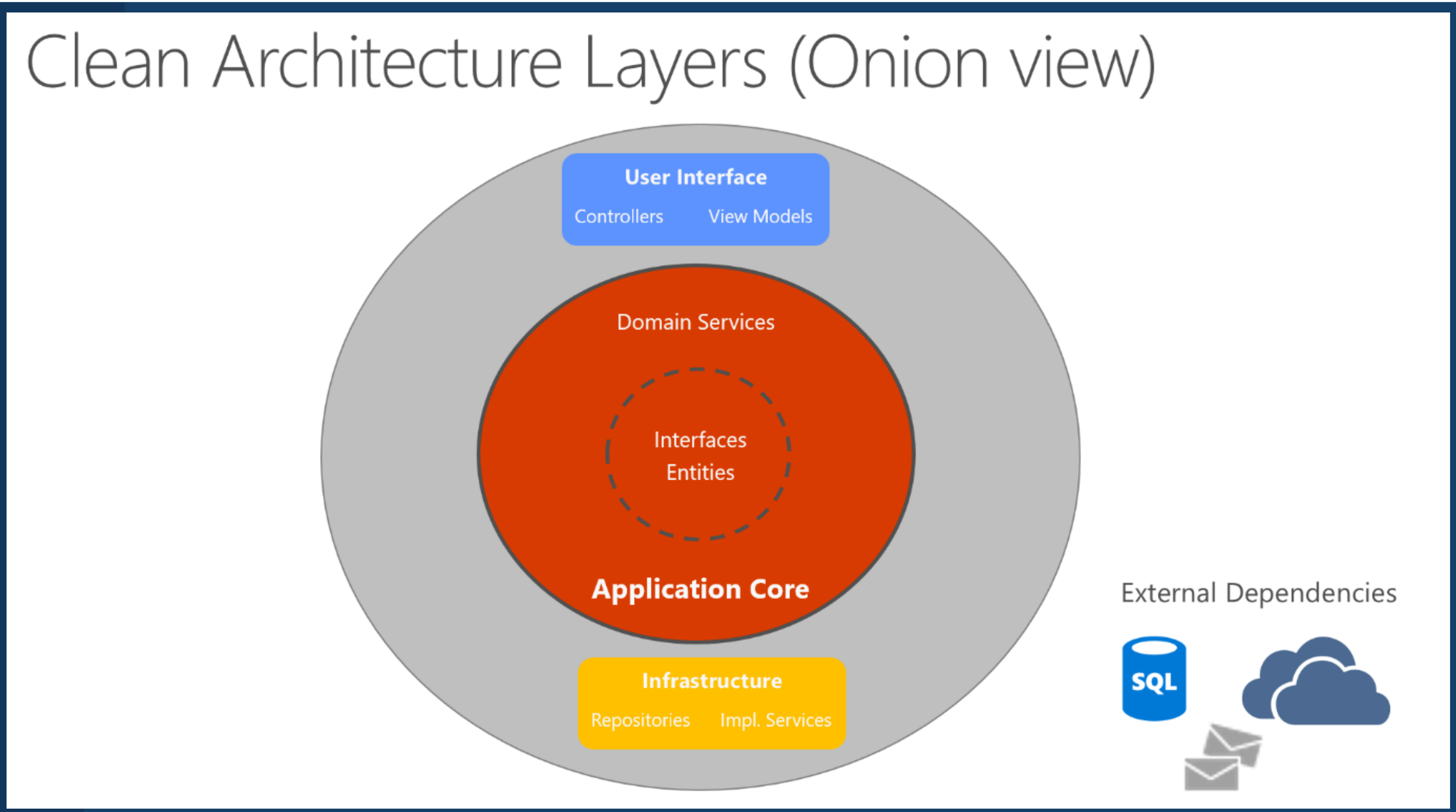
Summary

- **Business operations (*use cases*)** are the most important thing! Respect them and don’t pollute them with unnecessary things
- Database is only a **Detail**
- Frameworks are only the **Details**
- UI (Presentation) is a **Detail**

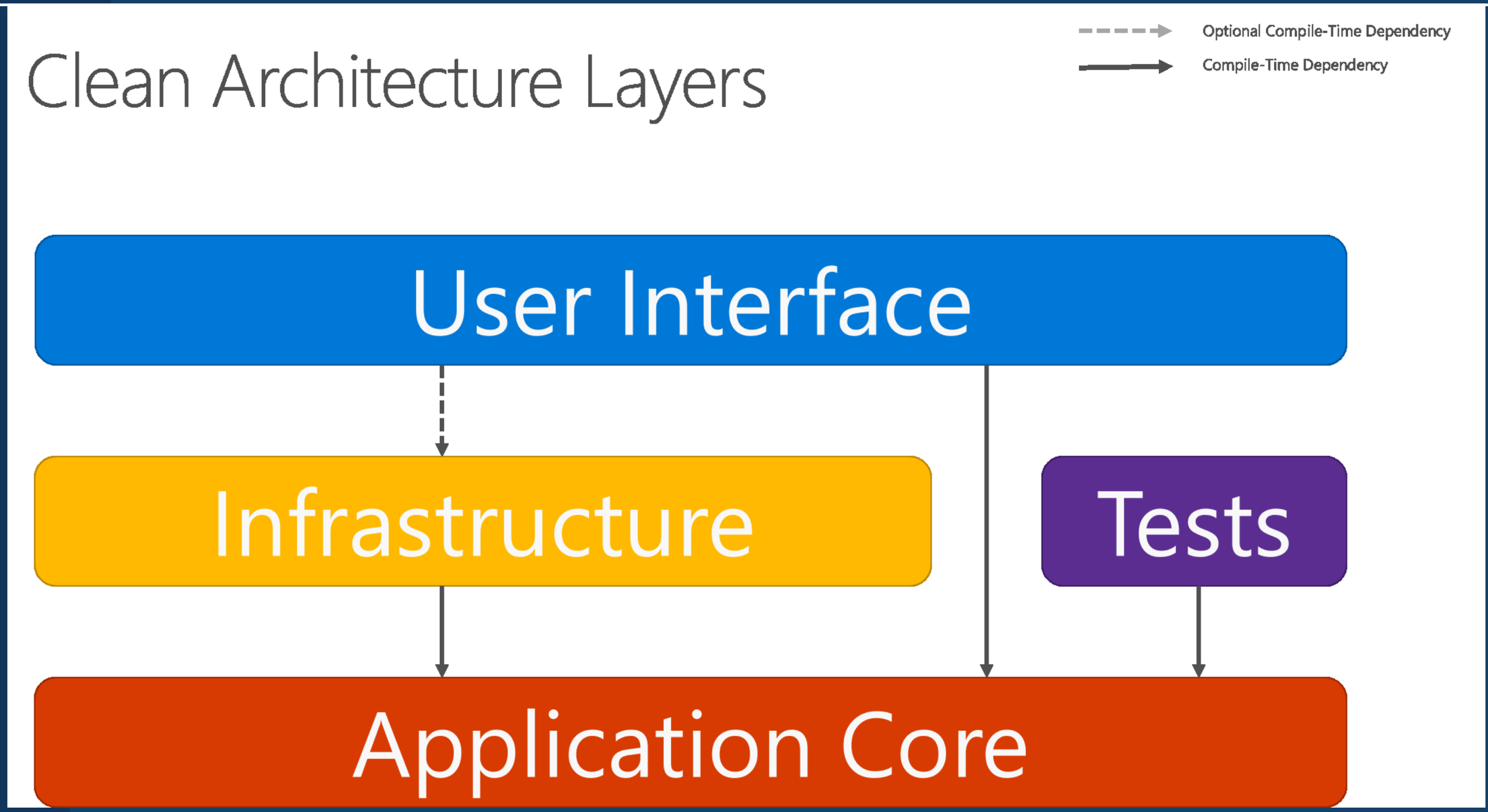
Clean architecture

- Follow the Dependency Inversion Principle as well as the Domain-Driven Design (DDD)
- Arrive at a similar architecture to Traditional "N-Layer" architecture
- Puts the business logic and application model at the center of the application.
- Instead of having business logic depend on data access or other infrastructure concerns, this dependency is inverted: infrastructure and implementation details depend on the Application Core
- This functionality is achieved by defining abstractions, or interfaces, in the Application Core, which are then implemented by types defined in the Infrastructure layer

Coding structure



Coding structure



Application Core

- Holds the business model, which includes entities, services, and interfaces.
- These interfaces include abstractions for operations that will be performed using Infrastructure, such as data access, file system access, network calls, etc.
- Sometimes services or interfaces defined at this layer will need to work with non-entity types that have no dependencies on UI or Infrastructure. These can be defined as simple Data Transfer Objects (DTOs).

Application Core types

- Entities (business model classes that are persisted)
- Aggregates (groups of entities)
- Interfaces
- Domain Services
- Specifications
- Custom Exceptions and Guard Clauses
- Domain Events and Handlers

Coding structure

Infrastructure

- The Infrastructure project typically includes data access implementations.

In a typical ASP.NET Core web application, these implementations include the Entity Framework (EF) DbContext, any EF Core Migration objects that have been defined, and data access implementation classes.

- Types:

EF Core types (DbContext, Migration)

Data access implementation types (Repositories)

Infrastructure-specific services (for example, FileLogger or SmtpNotifier)

UI Layer

- The user interface layer in an ASP.NET Core MVC application is the entry point for the application
- This project should reference the Application Core project, and its types should interact with infrastructure strictly through interfaces defined in Application Core.
- No direct instantiation of or static calls to the Infrastructure layer types should be allowed in the UI layer.

References

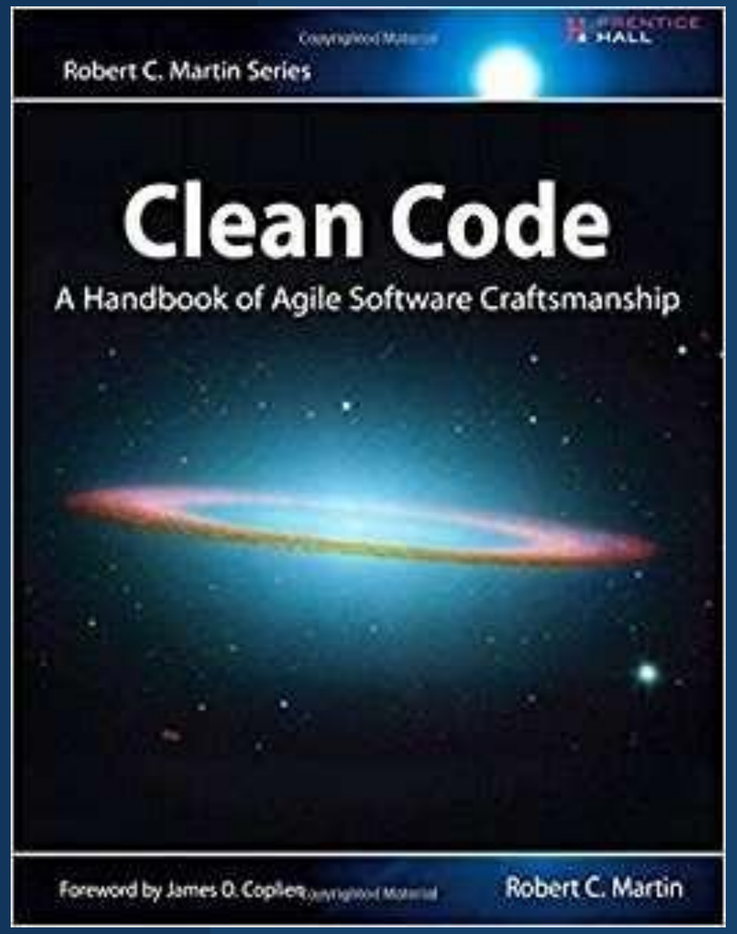
1. Common web application architectures, https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures

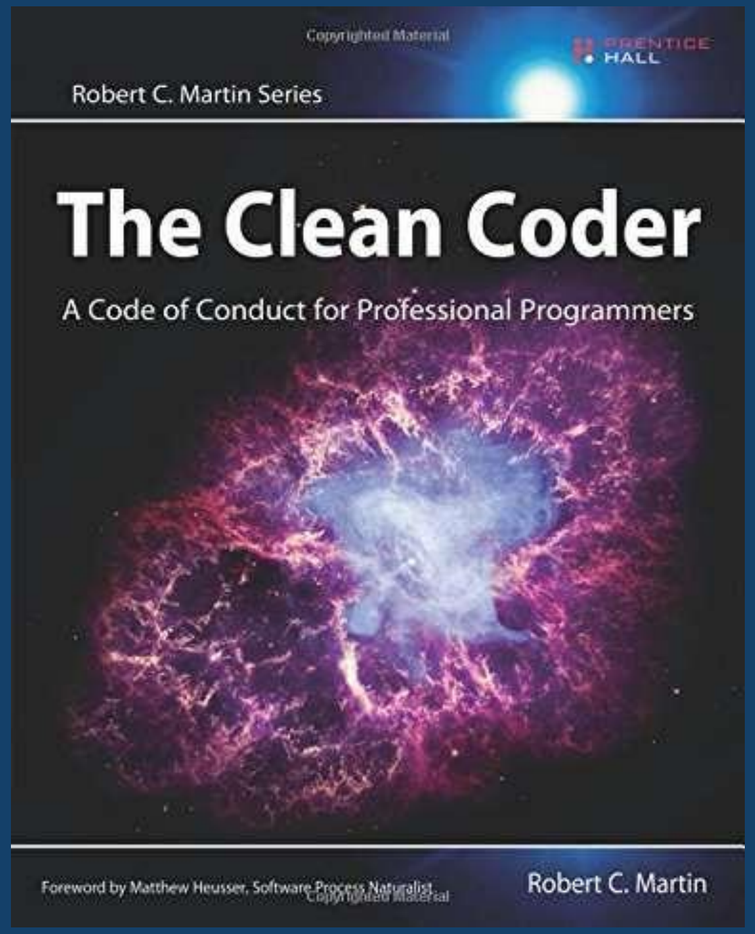
2. Solution templates:

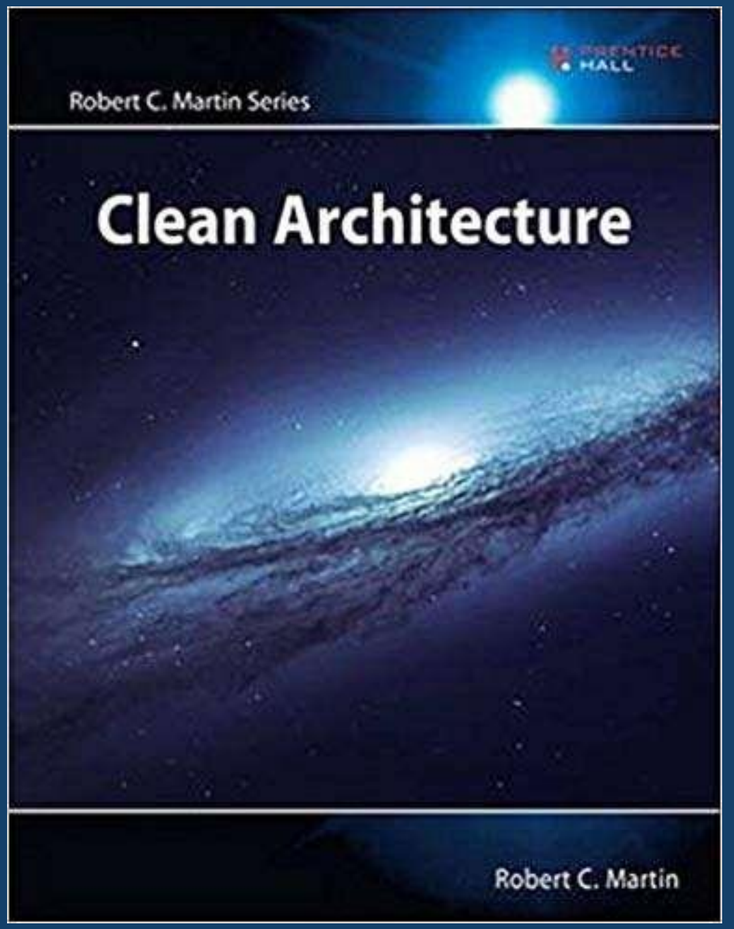
https://github.com/jasontaylordev/CleanArchitecture

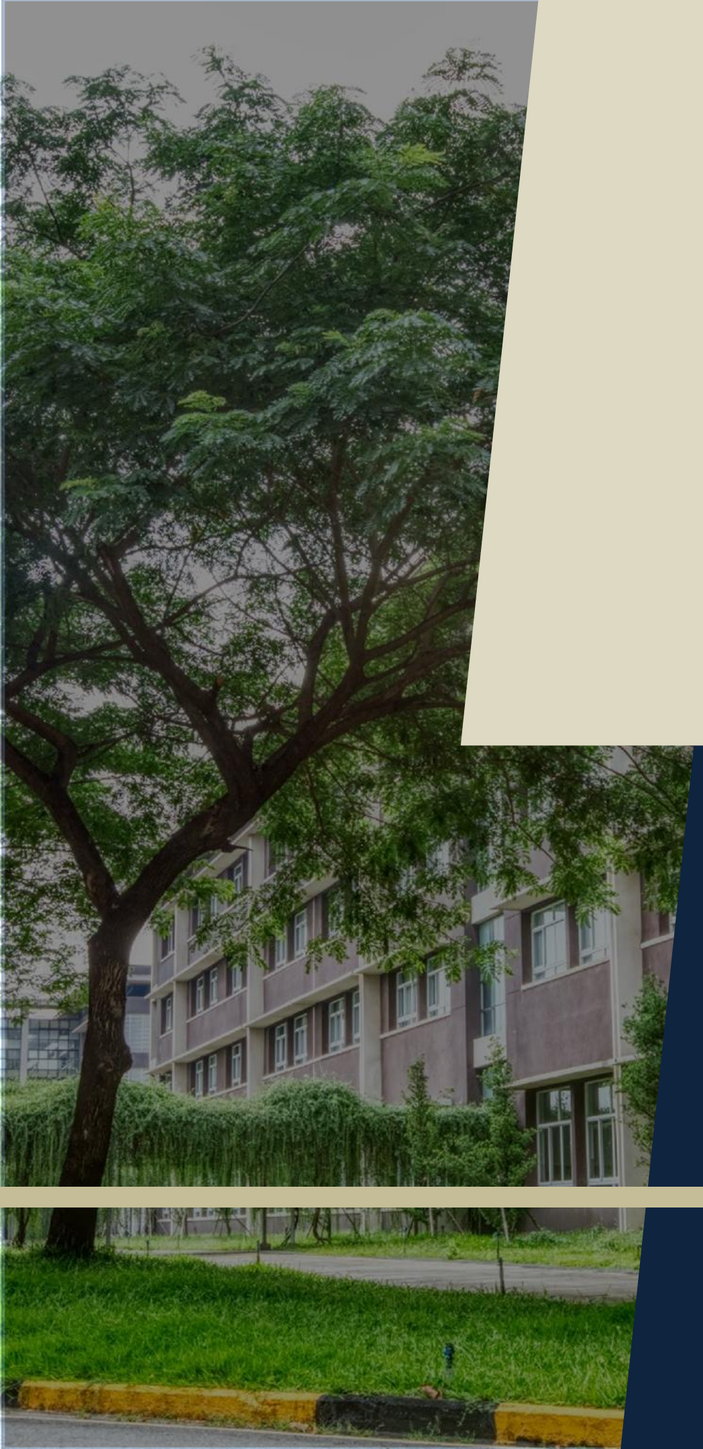
https://github.com/ardalis/cleanarchitecture

References


Robert C. Martin Series
Clean Code
A Handbook of Agile Software Craftsmanship
Foreword by James O. Coplien
Robert C. Martin


Robert C. Martin Series
The Clean Coder
A Code of Conduct for Professional Programmers
Foreword by Matthew Heusser, Software Process Naturalist
Robert C. Martin

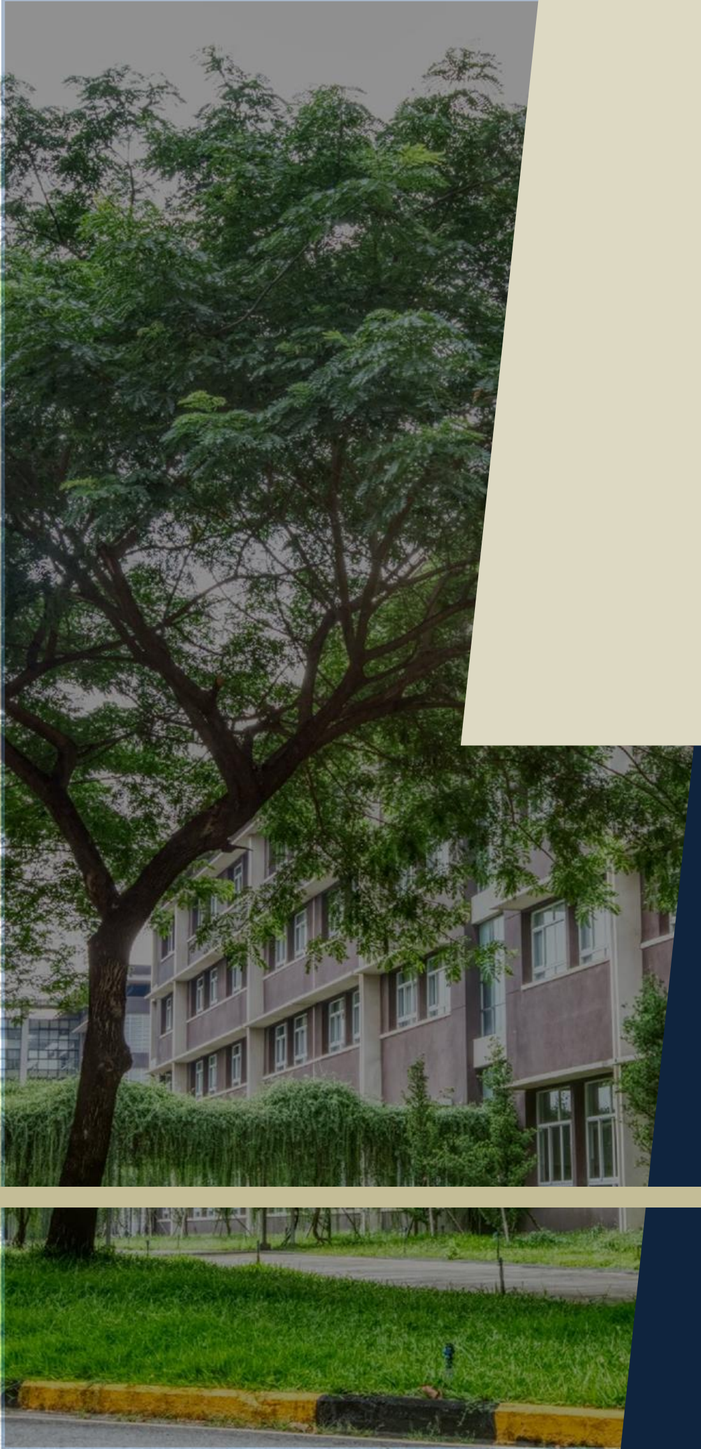

Robert C. Martin Series
Clean Architecture
Robert C. Martin



EIU
TRƯỜNG ĐẠI HỌC
QUỐC TẾ
MIỀN ĐÔNG
EASTERN
INTERNATIONAL
UNIVERSITY

Start your future at EIU

Q&A



EIU
TRƯỜNG ĐẠI HỌC
QUỐC TẾ
MIỀN ĐÔNG
EASTERN
INTERNATIONAL
UNIVERSITY

Start your future at EIU

THANK YOU