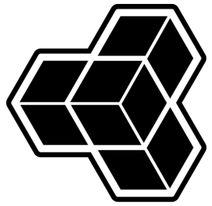


DERIVCO

SPORTS

proudly powering betway

Clean Architecture Essentials

Ivan Paulovich

Stockholm Software Craftsmanship
November 13, 2019

@ivanpaulovich

https://paulovich.net

Agenda

- The “Software Architecture”
- Introduction to Clean Architecture
- Plugin Argument
- Use Cases Implementation Guided By Tests
- Ports and Adapters Explained
- The User Interface is a Detail
- Frameworks are Details
- Sample Application
- Wrapping up
- References

Ivan Paulovich

Agile Software Developer, Tech Lead, 20+ GitHub projects about Clean Architecture, SOLID, DDD and TDD. Speaker/Streamer.

MVP Microsoft® Most Valuable Professional Reconnect

Nordax Bank

@ivanpaulovich

The “Software Architecture”

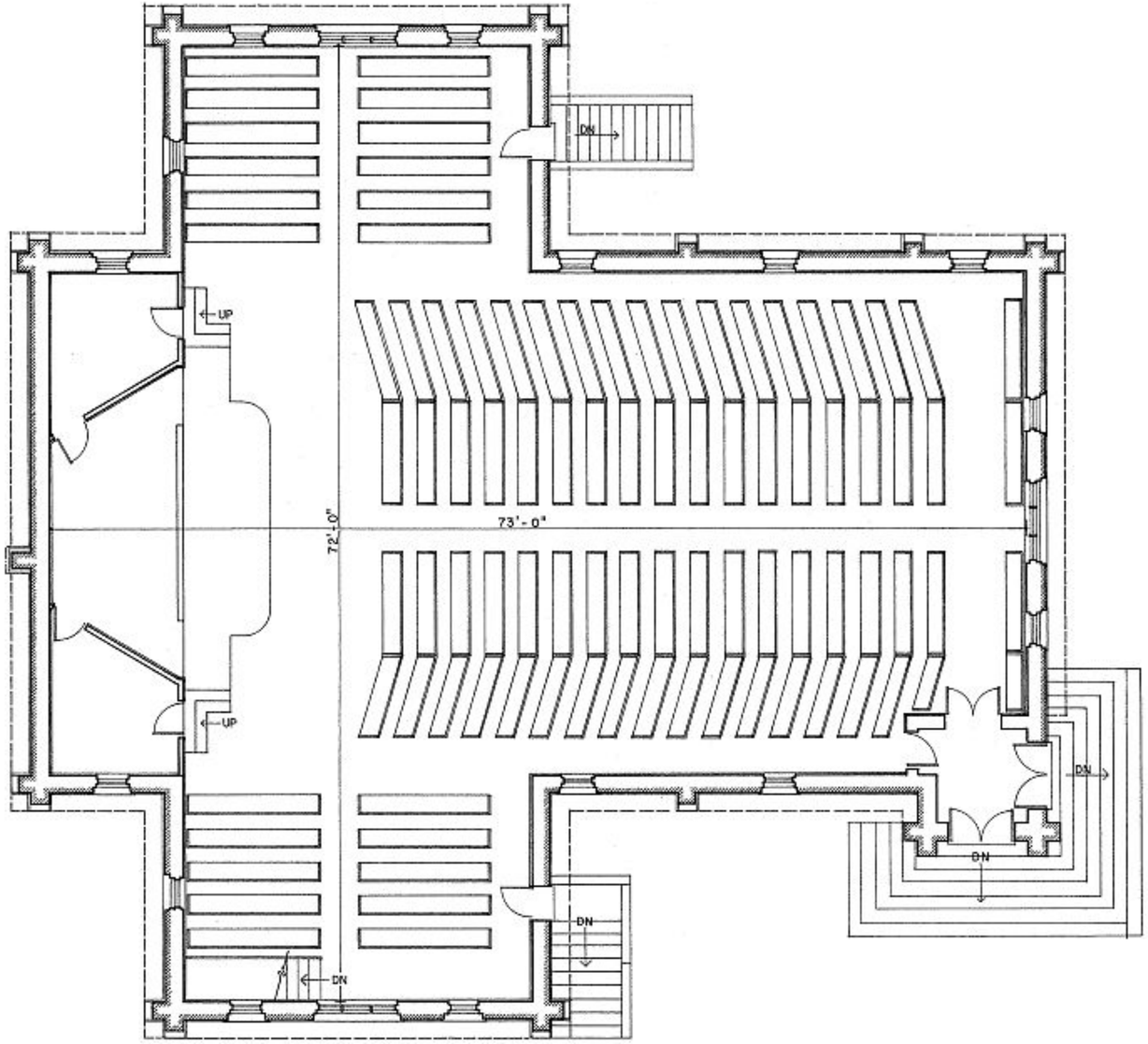
Presentation Layer	ASP.NET MVC and Autofac
Business Layer	MediatR and AutoMapper
Data Layer	Entity Framework and Dapper
Database and Messaging	SQL Server and RabbitMQ
IDE and Tools	Visual Studio, Resharper and .NET Framework

The “Software Architecture”

Presentation Layer	ASP.NET MVC and Autofac
Business Layer	MediatR and AutoMapper
Data Layer	Entity Framework and Dapper
Database and Messaging	SQL Server and RabbitMQ
IDE and Tools	Visual Studio, Resharper and .NET Framework

Not a Software Architecture!

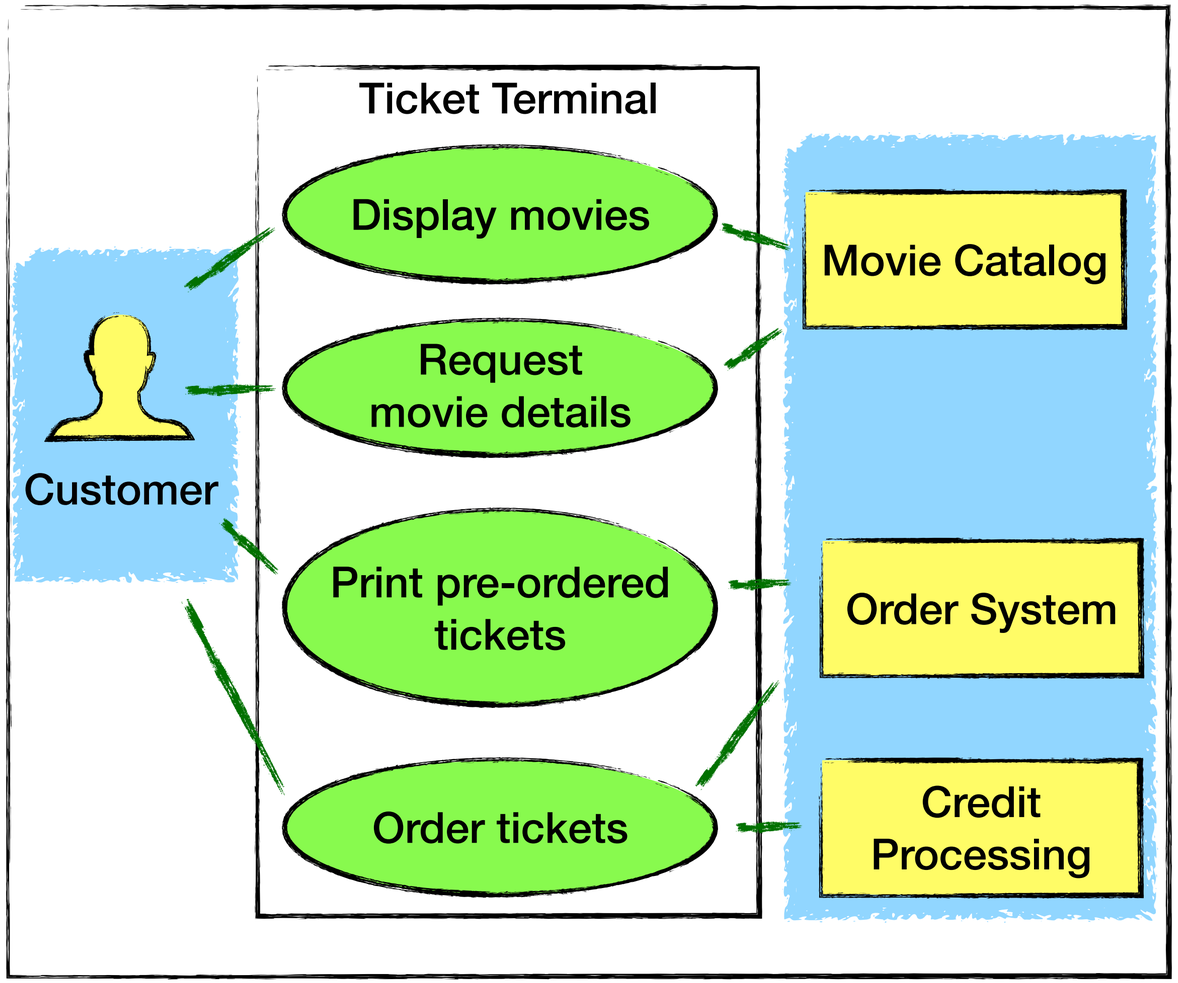
Architecture is About Usage



Clean Architecture

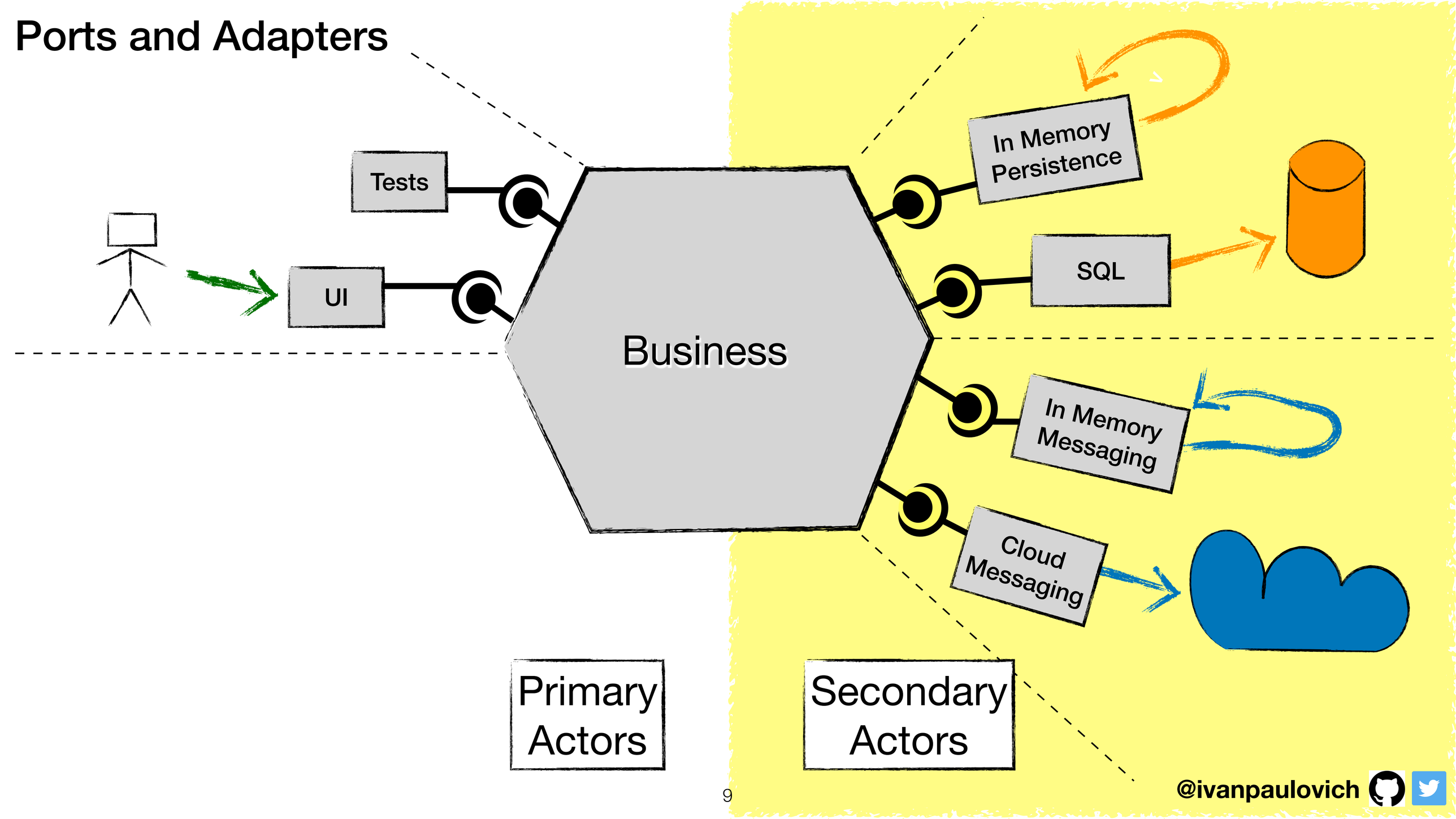
- **Use Cases** as central organising structure.
- Follows the **Ports and Adapters** pattern (Hexagonal Architecture).
  - Implementation is guided by tests.
  - Decoupled from technology details.
- Lots of Principles (SAP, SDP, SOLID..)
- Pluggable User Interface

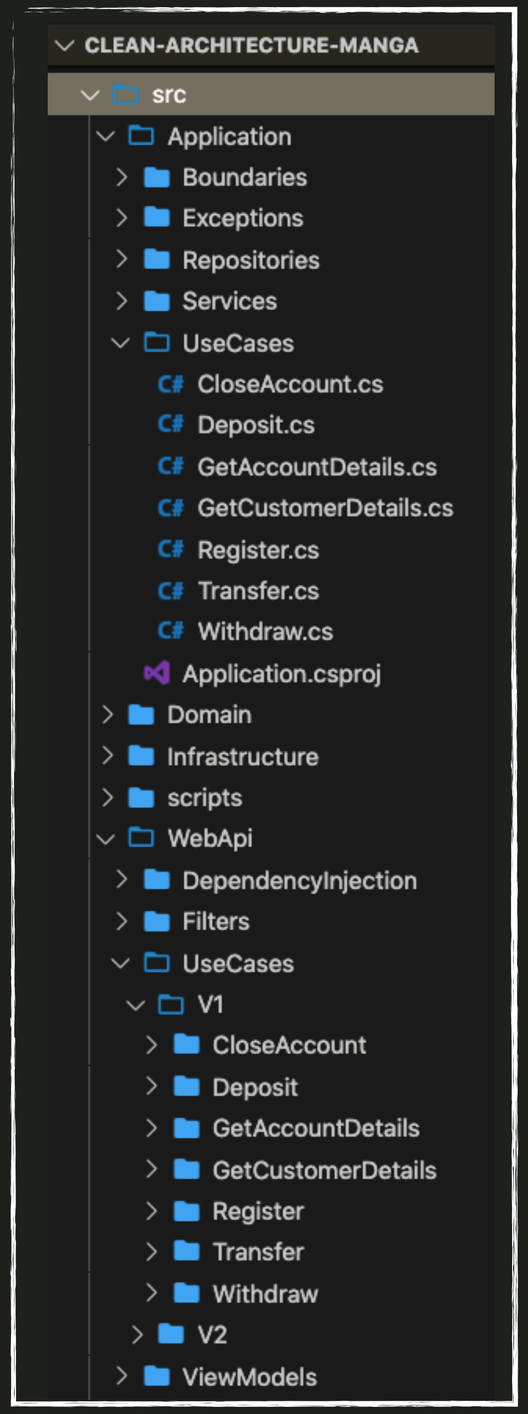
Use Cases



- Use Cases are **delivery independent.**
- Show the **intent** of a system.
- Use Cases are **algorithms** that interpret the input to generate the output data.
- Primary and secondary actors.

Ports and Adapters





Use Cases

```
public sealed class Register : IUseCase
{
    public async Task Execute(RegisterInput input)
    {
        var customer = _entityFactory.NewCustomer(input.SSN, input.Name);
        var account = _entityFactory.NewAccount(customer);

        var credit = account.Deposit(_entityFactory, input.InitialAmount);
        customer.Register(account);

        await _customerRepository.Add(customer);
        await _accountRepository.Add(account, credit);
        await _unitOfWork.Save();

        var output = new RegisterOutput(customer, account);
        _outputPort.Standard(output);
    }
```

Abstractions and Stability Principles

```csharp
public interface IAccountRepository
{
    Task<IAccount> Get(Guid id);
    Task Add(IAccount account, ICredit credit);
    Task Update(IAccount account, ICredit credit);
    Task Update(IAccount account, IDebit debit);
    Task Delete(IAccount account);
}
```

Abstract
General
Stable

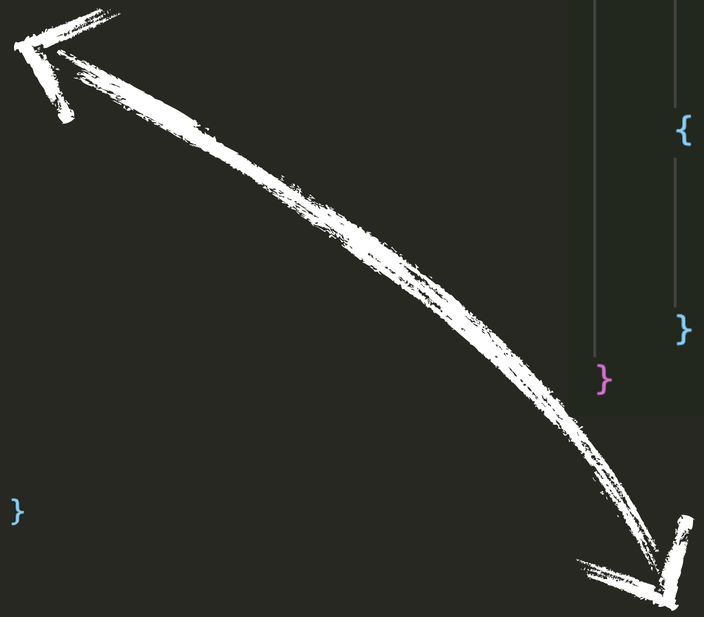
Concrete
Specific
Unstable

```csharp
public sealed class AccountRepository : IAccountRepository
{
    private readonly MangaContext _context;

    public AccountRepository(MangaContext context)
    {
        _context = context ??
            throw new ArgumentNullException(nameof(context));
    }

    public async Task Add(IAccount account, ICredit credit)
    {
        await _context.Accounts.AddAsync((EntityFrameworkDataAccess.Account) account);
        await _context.Credits.AddAsync((EntityFrameworkDataAccess.Credit) credit);
    }

    public async Task Delete(IAccount account)
    {
        string deleteSQL =
            @"DELETE FROM Credit WHERE AccountId = @Id;
                  DELETE FROM Debit WHERE AccountId = @Id;
                  DELETE FROM Account WHERE Id = @Id;";

        var id = new SqlParameter("@Id", account.Id);

        int affectedRows = await _context.Database.ExecuteSqlRawAsync(
            deleteSQL, id);
    }

    public async Task<IAccount> Get(Guid id)
    {
        Infrastructure.EntityFrameworkDataAccess.Account account = await _context
            .Accounts
            .Where(a => a.Id == id)
            .SingleOrDefaultAsync();

        if (account is null)
            throw new AccountNotFoundException($"The account {id} does not exist or is not processed yet.");

        var credits = _context.Credits
            .Where(e => e.AccountId == id)
            .ToList();

        var debits = _context.Debits
```

@ivanpaulovich

```
/// <summary>
/// Registration Request
/// </summary>
public sealed class RegisterRequest
{
    /// <summary>
    /// SSN
    /// </summary>
    [Required]
    public string SSN { get; set; }

    /// <summary>
    /// Name
    /// </summary>
    [Required]
    public string Name { get; set; }

    /// <summary>
    /// Initial Amount
    /// </summary>
    [Required]
    public decimal InitialAmount { get; set; }
}
```

Concrete ⭐⭐
Inconsistent ⭐⭐
Specific ⭐⭐



```
public sealed class RegisterInput : IUseCaseInput
{
    public SSN SSN { get; }
    public Name Name { get; }
    public PositiveMoney InitialAmount { get; }

    public RegisterInput(
        SSN ssn,
        Name name,
        PositiveMoney initialAmount)
    {
        SSN = ssn;
        Name = name;
        InitialAmount = initialAmount;
    }
}
```

Concrete ⭐⭐
Consistent ⭐
Specific ⭐

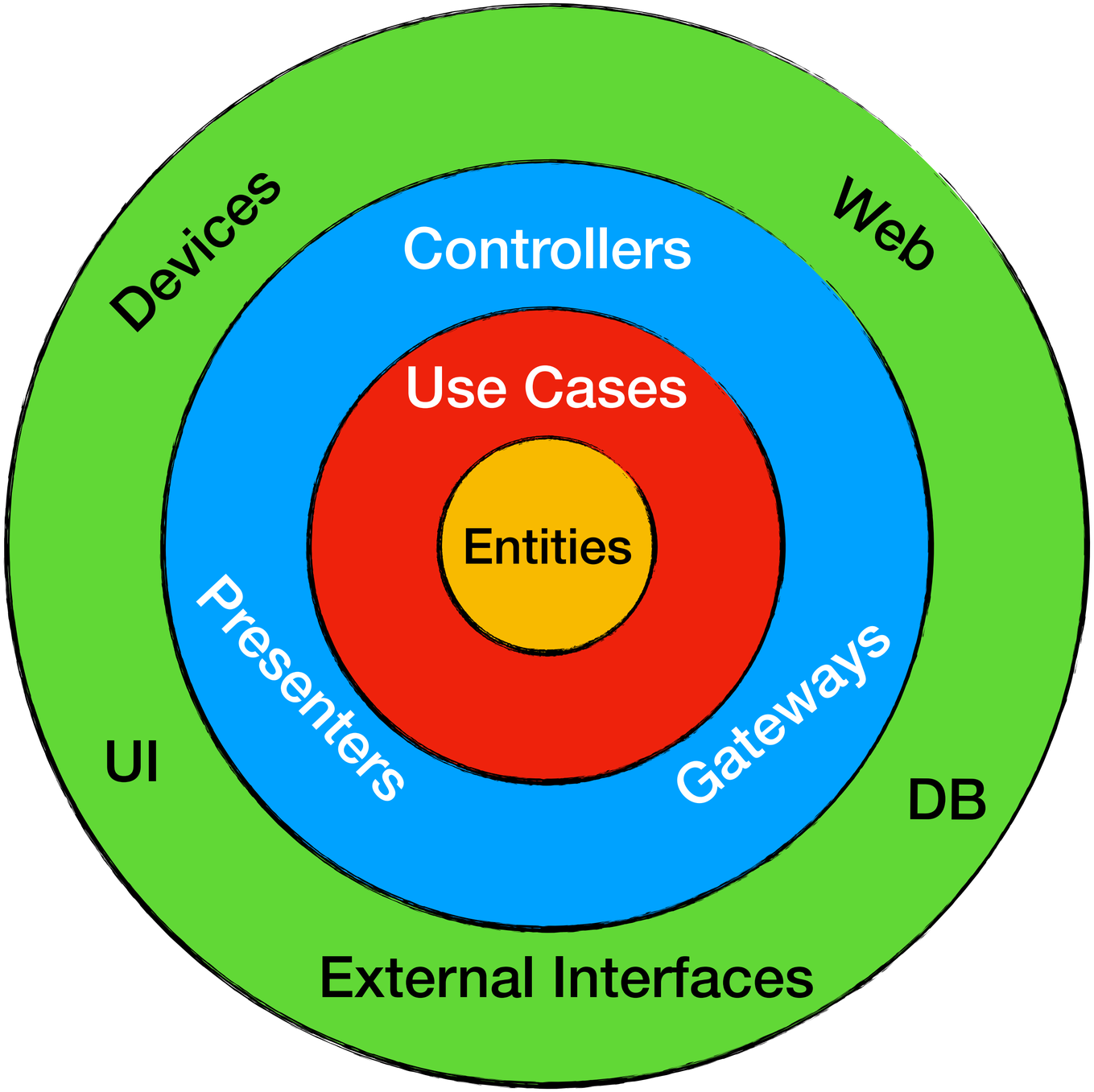
Abstract ⭐⭐
General ⭐⭐
Stable ⭐⭐

```
public interface IAccount : IAggregateRoot
{
    ICredit Deposit(IEntityFactory entityFactory, PositiveMoney amountToDeposit);
    IDebit Withdraw(IEntityFactory entityFactory, PositiveMoney amountToWithdraw);
    bool IsClosingAllowed();
    Money GetCurrentBalance();
}
```

Abstractions and Stability Principles

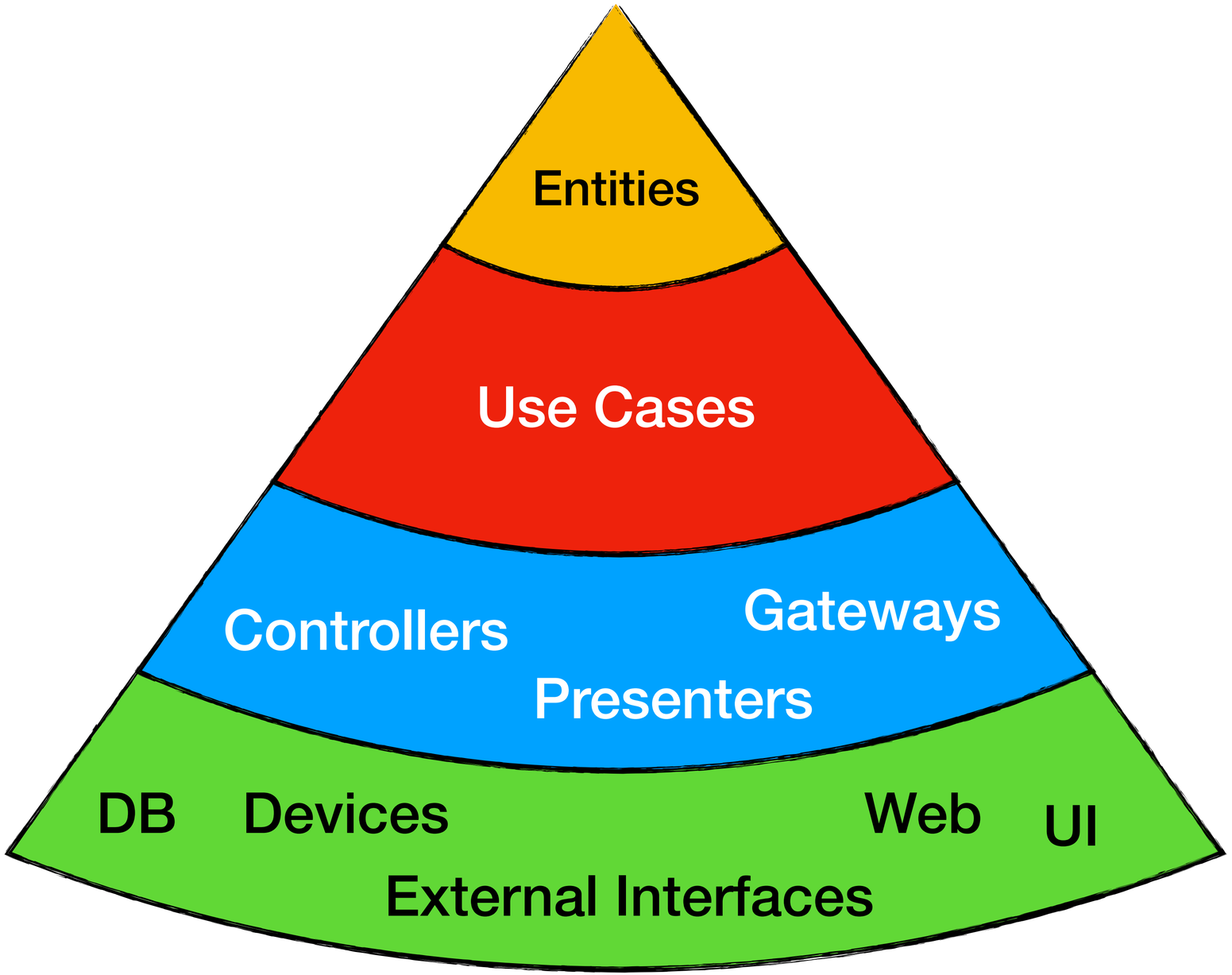
@ivanpaulovich

Clean Architecture



- Abstractness increases with stability.
- Depend in the direction of stability.
- Classes that change together are packaged together.

Clean Architecture

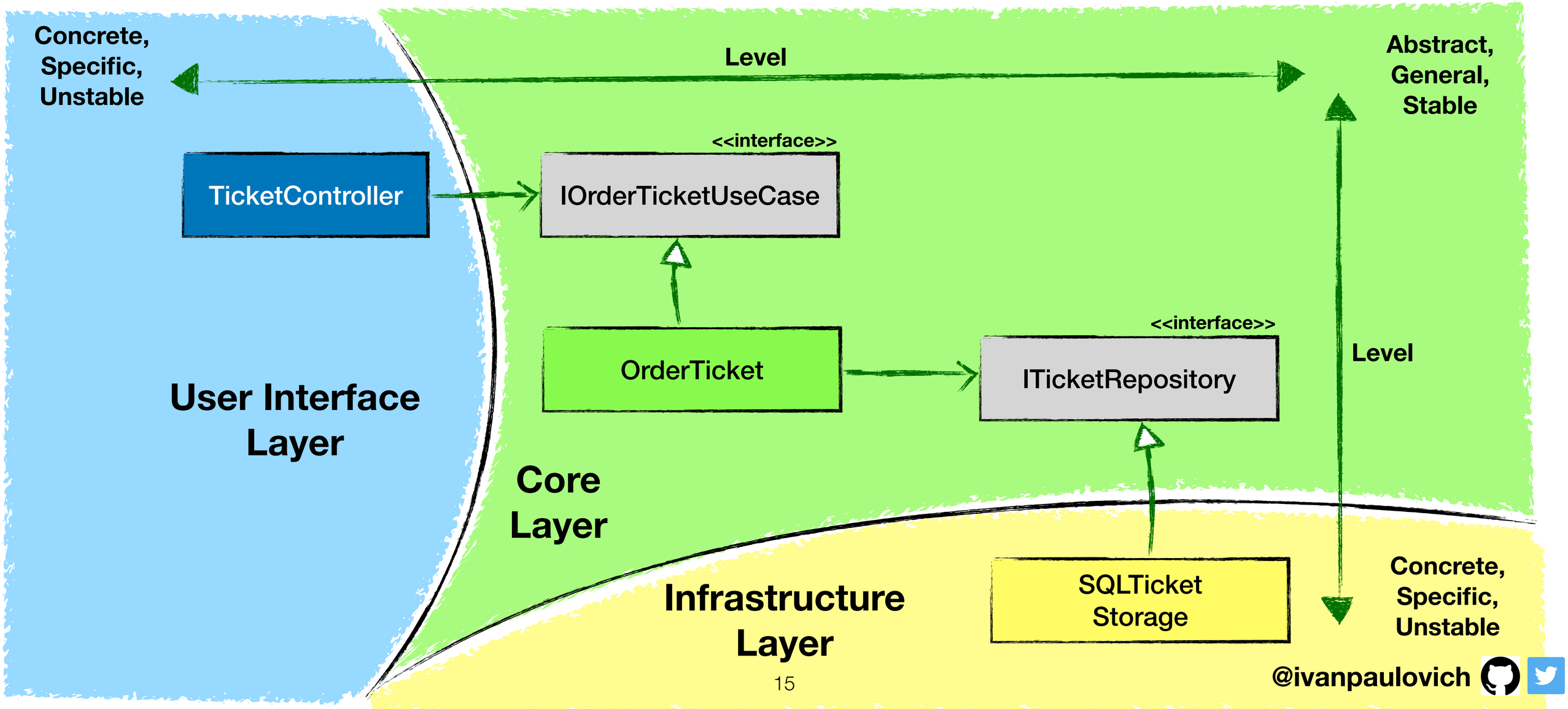


Abstract, General, Stable, Consistent

Concrete, Specific, Unstable, Inconsistent

@ivanpaulovich

Plugin Architecture



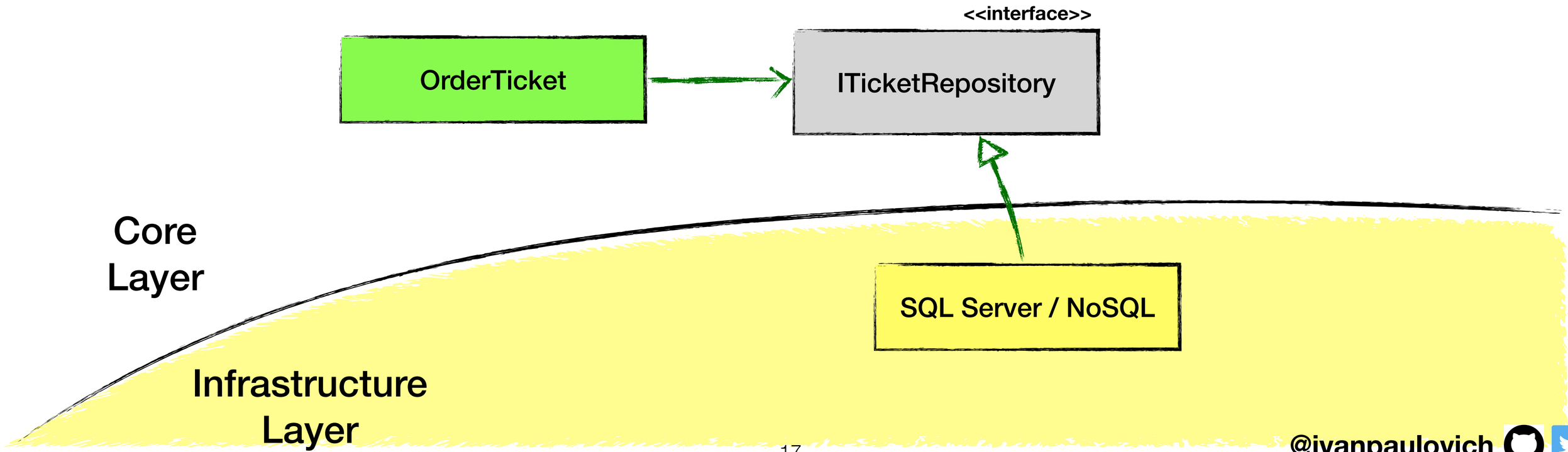
Software Architecture

“A good architecture **allows major decisions to be deferred.**
A good architect maximizes the number of **decisions not made.**

Uncle Bob.

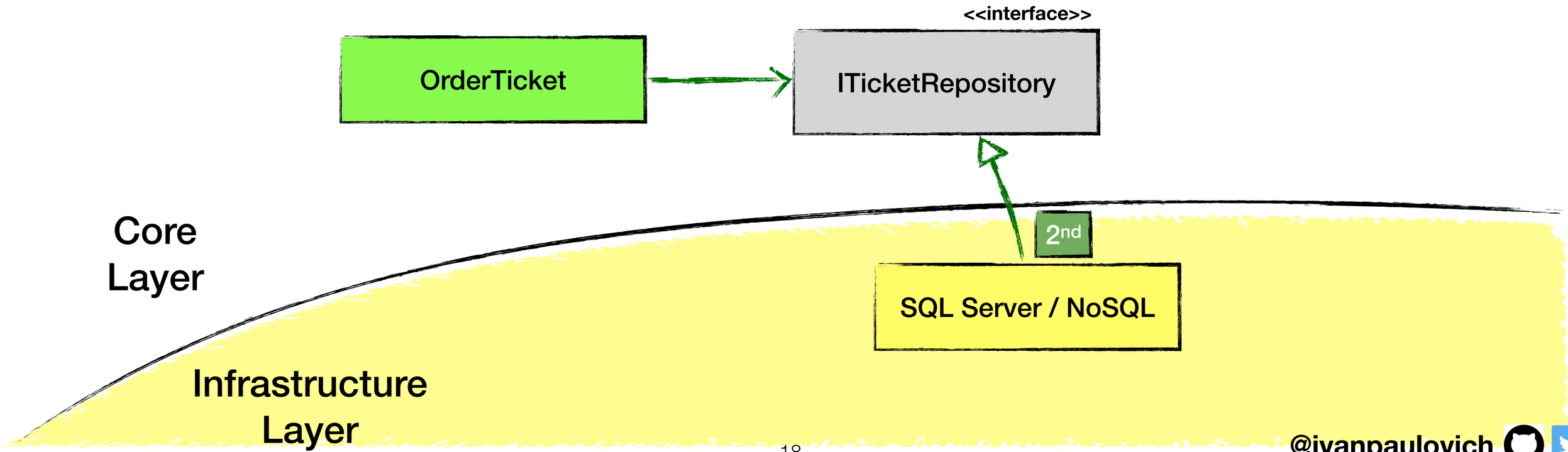
Plugin Architecture

- The dependencies point in the direction of the abstract components.



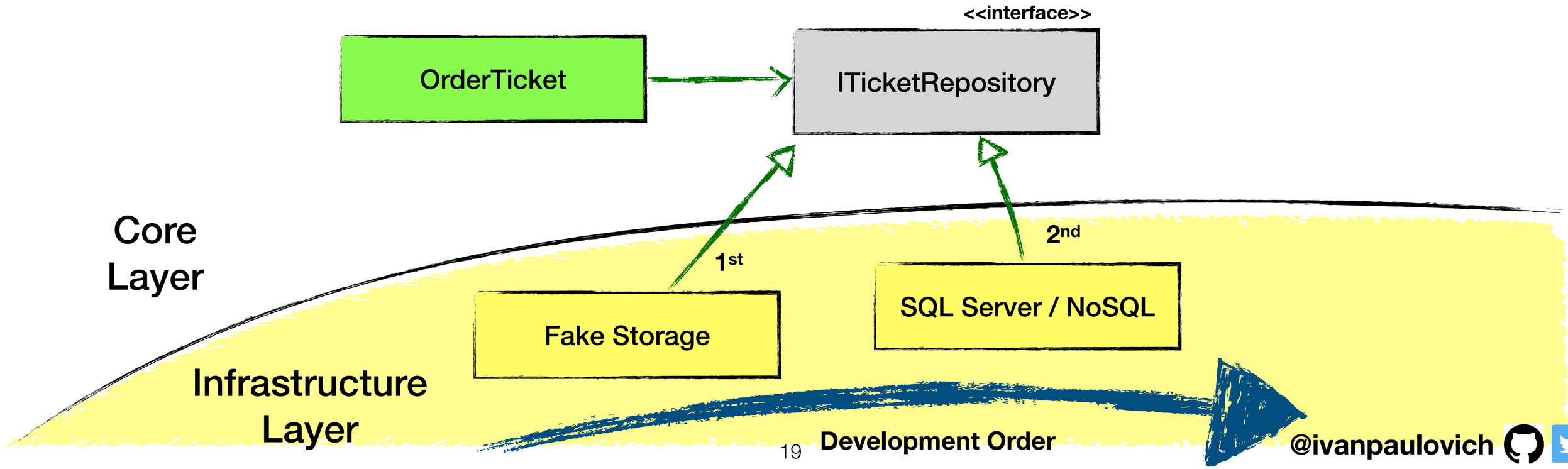
Plugin Architecture

- The dependencies point in the direction of the abstract components.



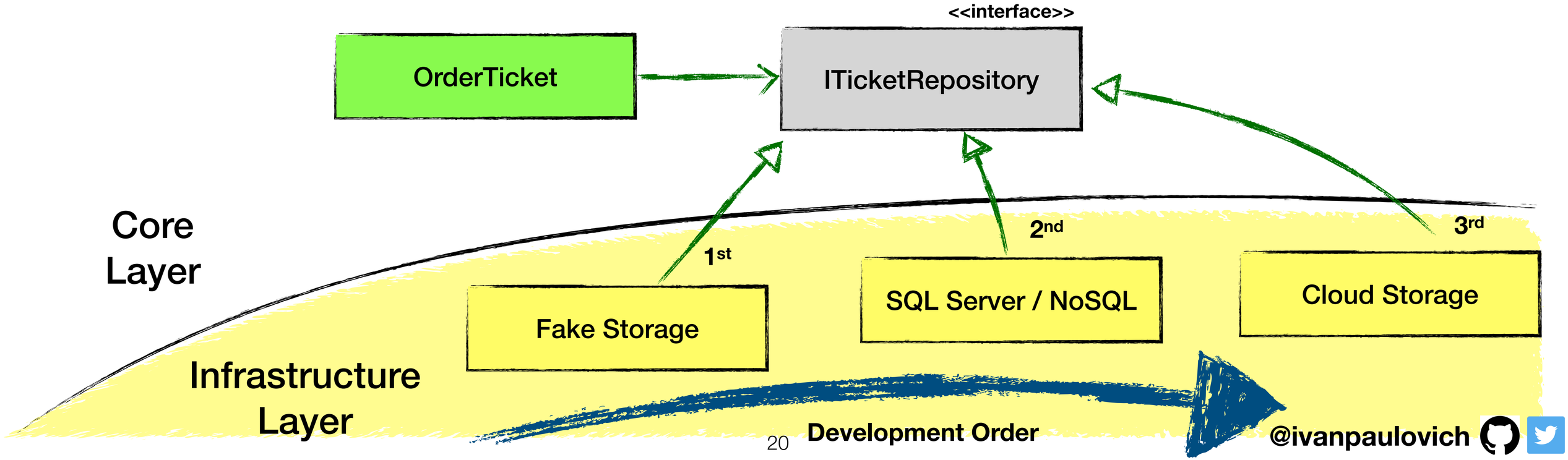
Plugin Architecture

- The dependencies point in the direction of the abstract components.
- The simplest component version are implemented first.



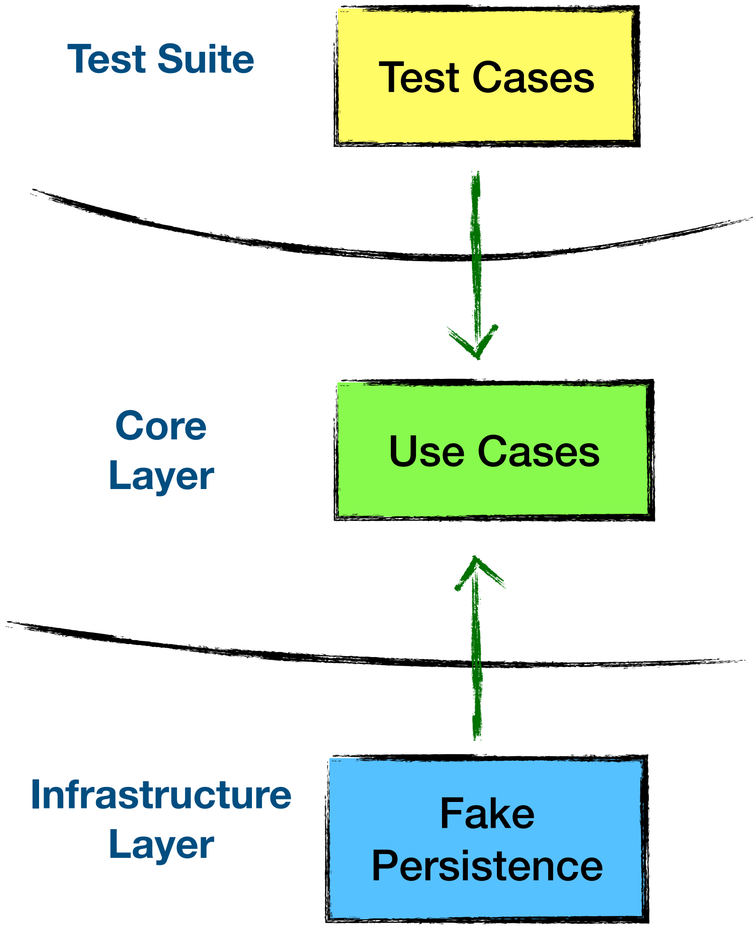
Plugin Architecture

- The dependencies point in the direction of the abstract components.
- The simplest component version are implemented first.
- Defer decisions. Keep the options open as long as possible!



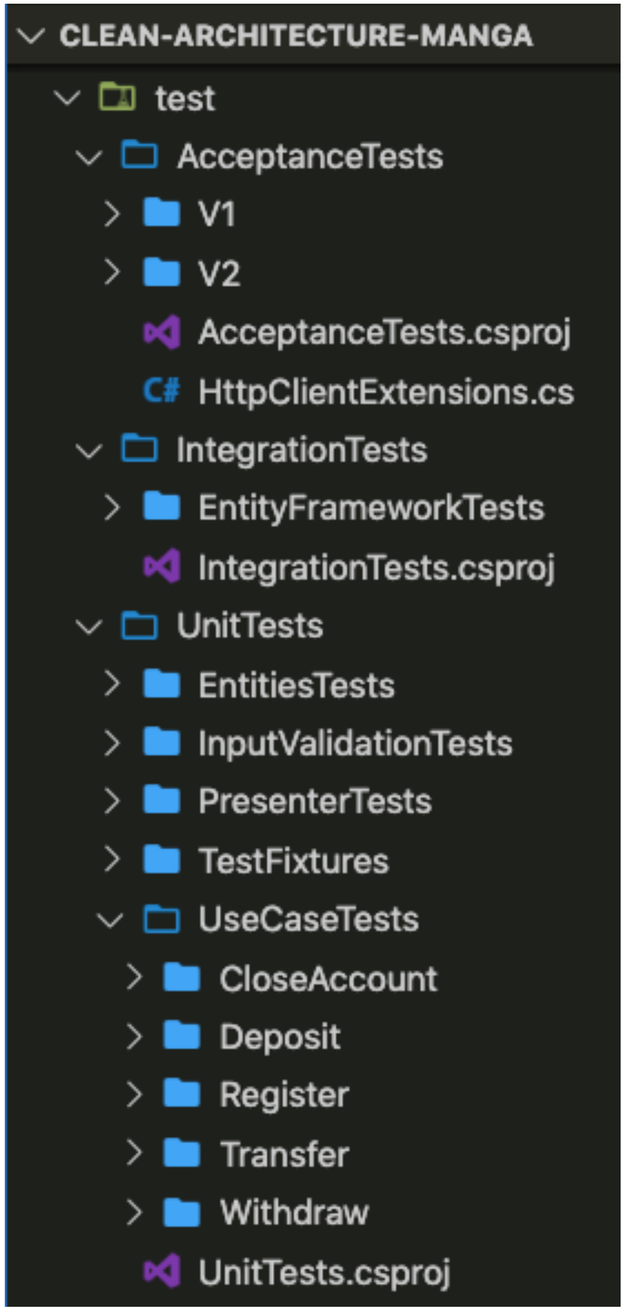
Use Cases Implementation Guided by Tests

- Test Cases are the first consumers implemented.
- Design the Use Cases with Single Responsibility Principle in mind.
- Fake implementations are the first delivery mechanisms implemented.
- Test Cases help you design fine grained Use Cases (Interface Segregation Principle).



Use Cases Implementation Guided by Tests

- Test Cases are the first consumers implemented.
- Design the Use Cases with Single Responsibility Principle in mind.
- Fake implementations are the first delivery mechanisms implemented.
- Test Cases help you design fine grained Use Cases (Interface Segregation Principle).



@ivanpaulovich

```
public sealed class MangaContext
{
    public Collection<Customer> Customers { get; set; }
    public Collection<Account> Accounts { get; set; }
    public Collection<Credit> Credits { get; set; }
    public Collection<Debit> Debits { get; set; }
    public MangaContext()
    {
        Customers = new Collection<Customer>();
        Accounts = new Collection<Account>();
        Credits = new Collection<Credit>();
        Debits = new Collection<Debit>();
    }
}
```

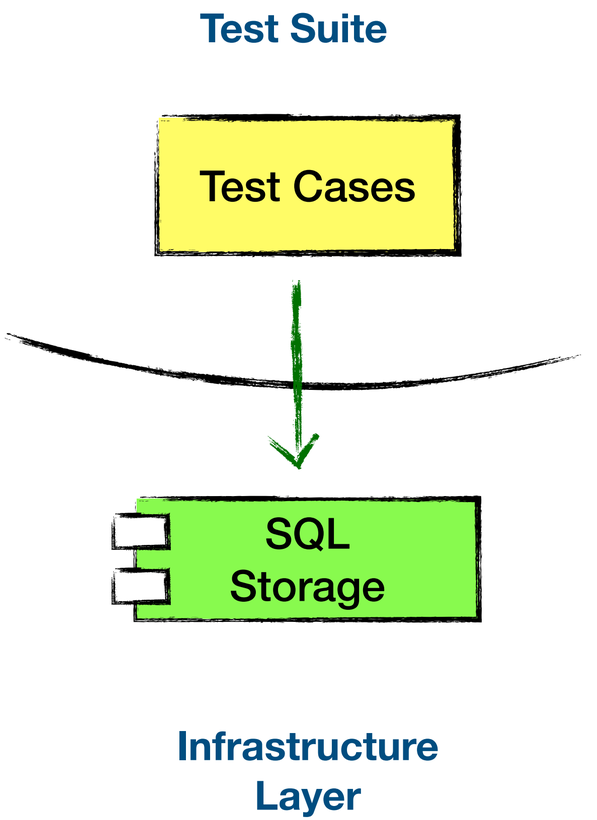
Whats is a Fake (Test Double)?

- Runs in memory (No I/O).
- Same capability as the real one.

```
public sealed class CustomerRepository : ICustomerRepository
{
    private readonly MangaContext _context;
    public CustomerRepository(MangaContext context)
    {
        _context = context;
    }
    public async Task Add(ICustomer customer)
    {
        _context.Customers.Add((InMemoryDataAccess.Customer) customer);
        await Task.CompletedTask;
    }
    public async Task<ICustomer> Get(Guid id)
    {
        Customer customer = _context.Customers
            .Where(e => e.Id == id)
            .SingleOrDefault();
        if (customer == null)
            throw new CustomerNotFoundException(
                $@"The customer {id} does not exist or
                it was not processed yet.");
        return await Task.FromResult<Customer>(customer);
    }
    public async Task Update(ICustomer customer)
    {
        Customer customerOld = _context.Customers
            .Where(e => e.Id == customer.Id)
            .SingleOrDefault();
        customerOld = (Customer) customer;
        await Task.CompletedTask;
    }
```

Components Overview

1. Components are designed to fulfill the Core needs and are implemented and tested in isolation.
2. Components could be replaced, upgraded or decommissioned with minimum Core business impact.
3. Good components are loosely coupled.



ASPNETCORE_ENVIRONMENT=Development

Run the host using **Fakes**

```
public void ConfigureDevelopmentServices(IServiceCollection services)
{
    services.AddControllers().AddControllersAsServices();
    services.AddBusinessExceptionFilter();
    services.AddFeatureFlags(Configuration);
    services.AddVersioning();
    services.AddSwagger();
    services.AddUseCases();
    services.AddInMemoryPersistence();
    services.AddPresentersV1();
    services.AddPresentersV2();
}
```

ASPNETCORE_ENVIRONMENT=Production

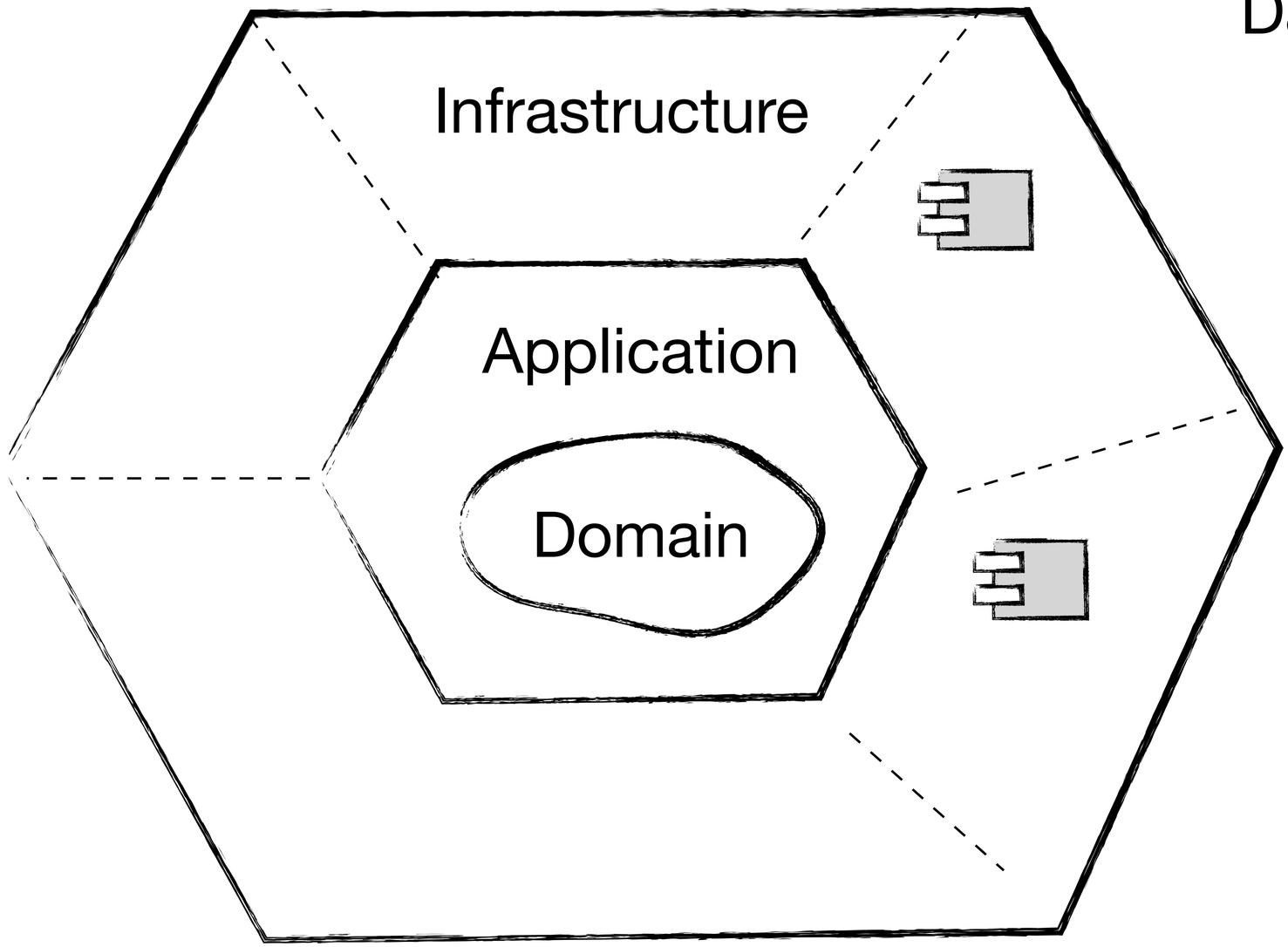
Run the host using the **Real Implementations**

```
public void ConfigureProductionServices(IServiceCollection services)
{
    services.AddControllers().AddControllersAsServices();
    services.AddBusinessExceptionFilter();
    services.AddFeatureFlags(Configuration);
    services.AddVersioning();
    services.AddSwagger();
    services.AddUseCases();
    services.AddSQLServerPersistence(Configuration);
    services.AddPresentersV1();
    services.AddPresentersV2();
}
```

@ivanpaulovich

Ports and Adapters

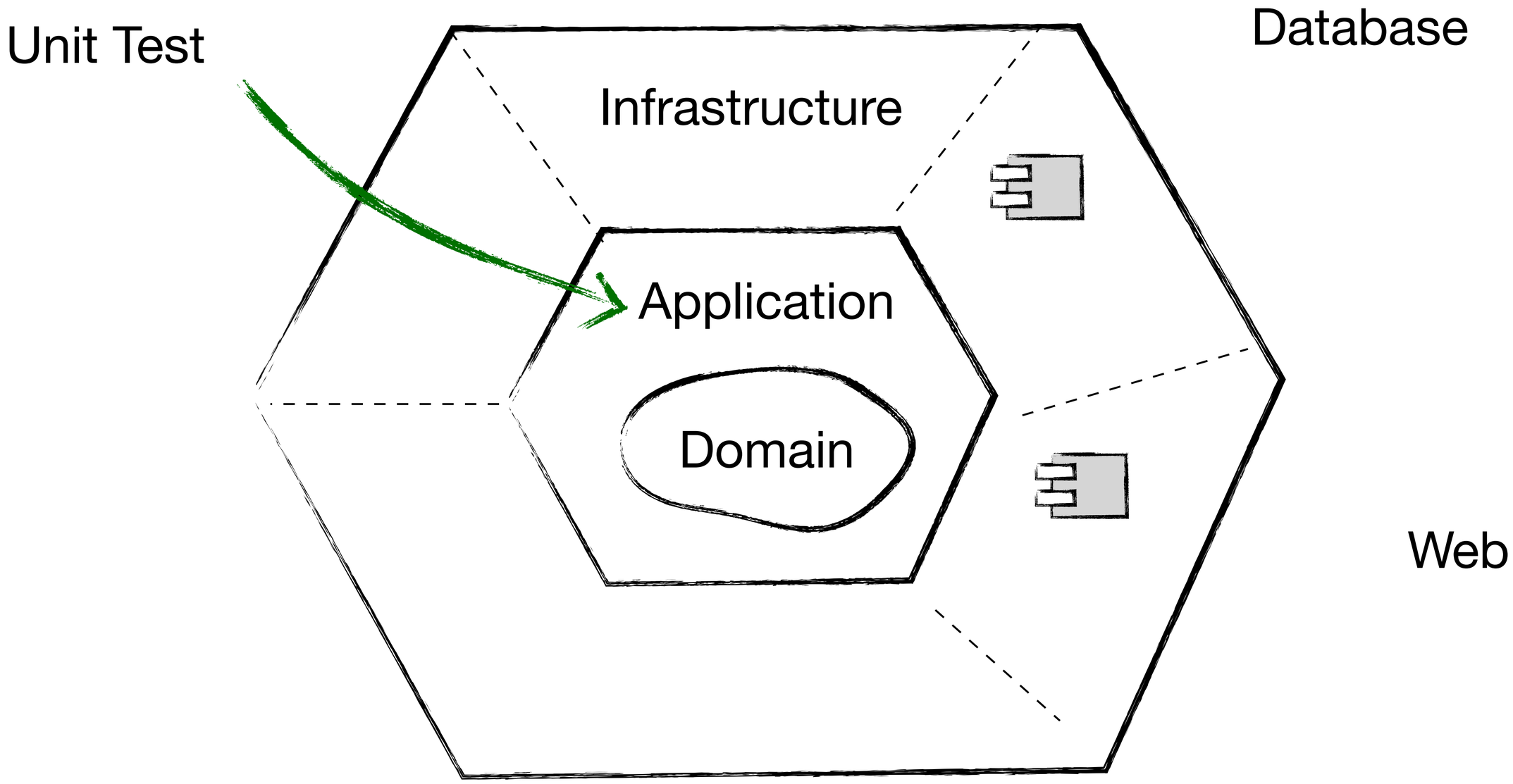
Unit Test



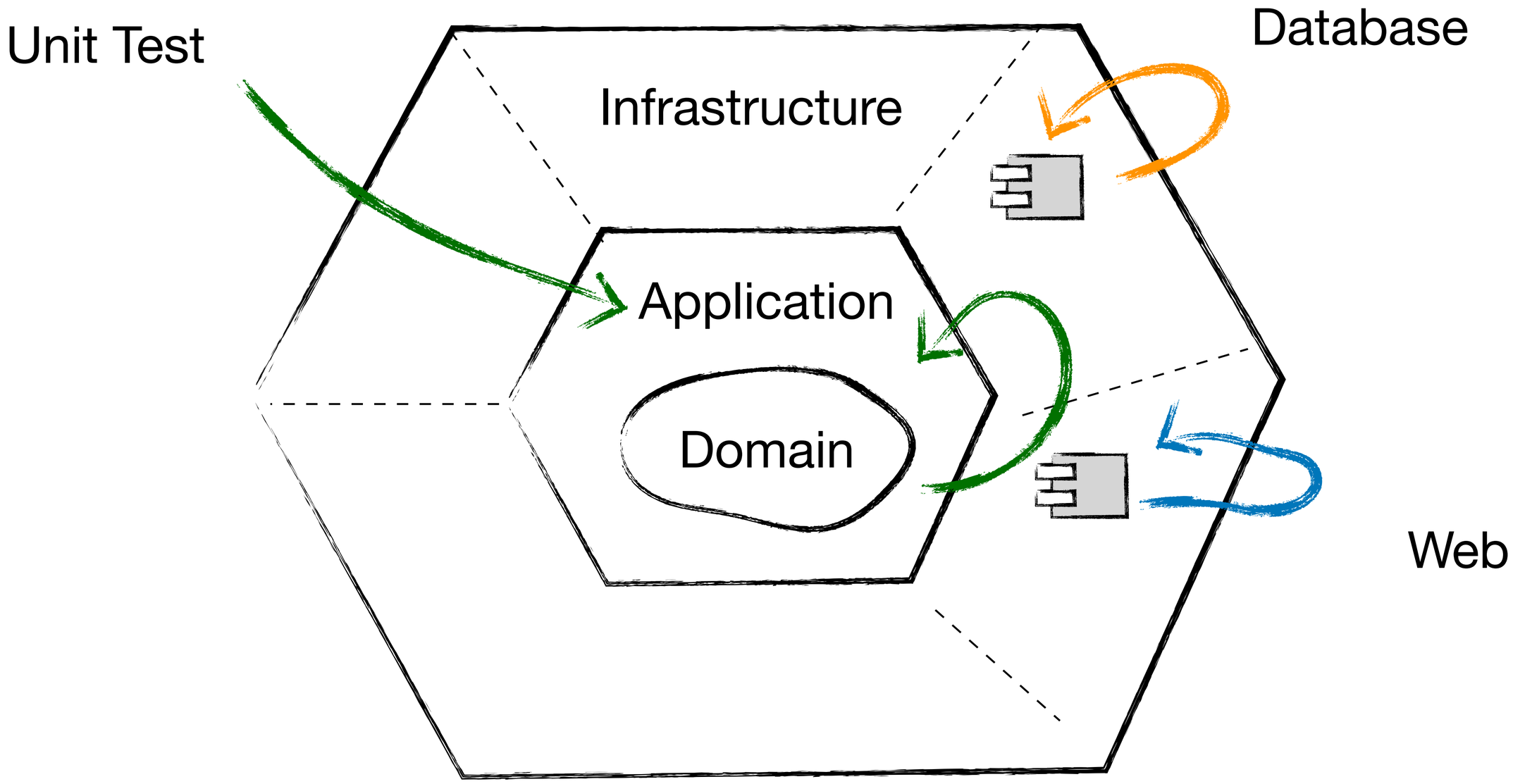
Database

Web

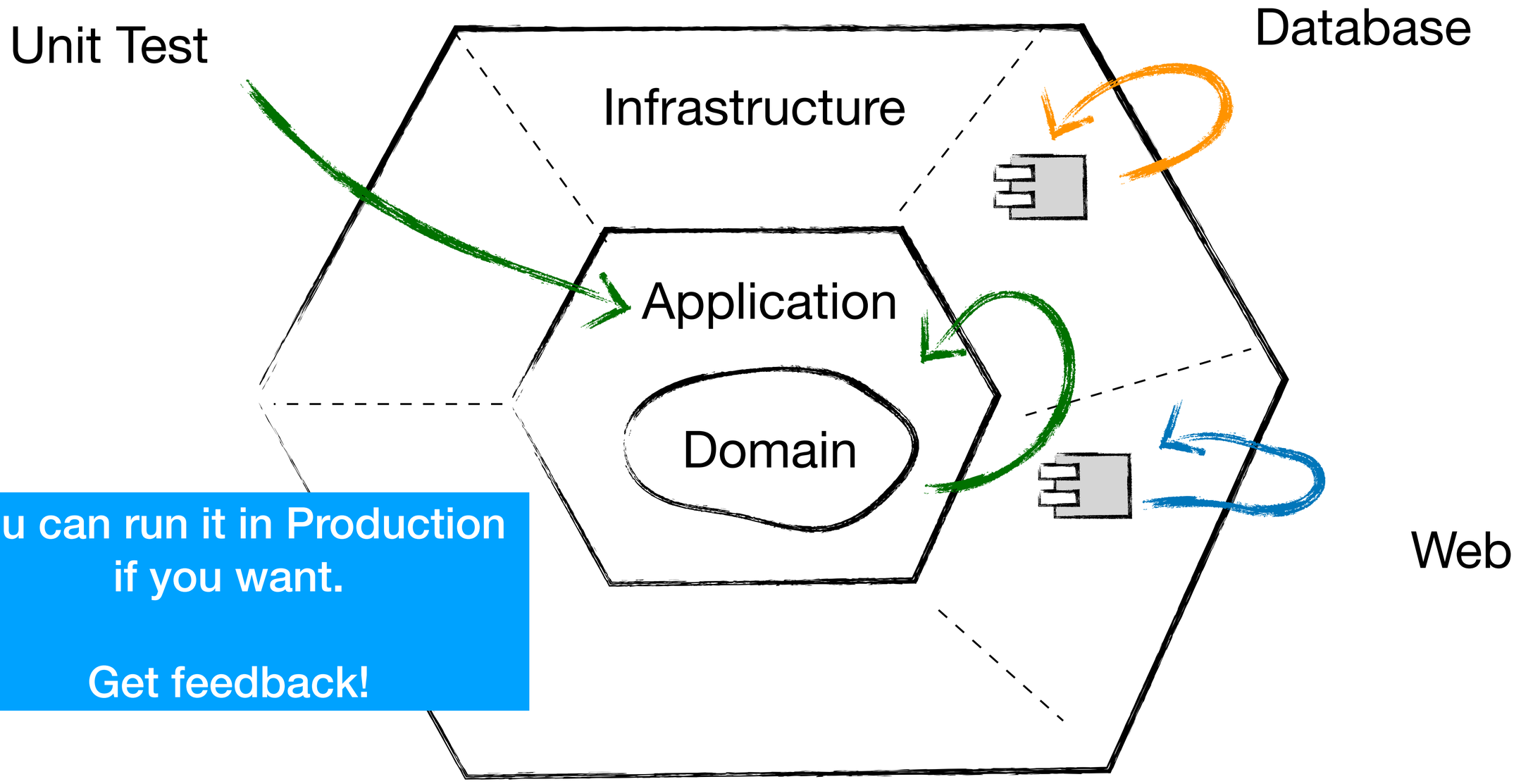
Ports and Adapters



Ports and Adapters



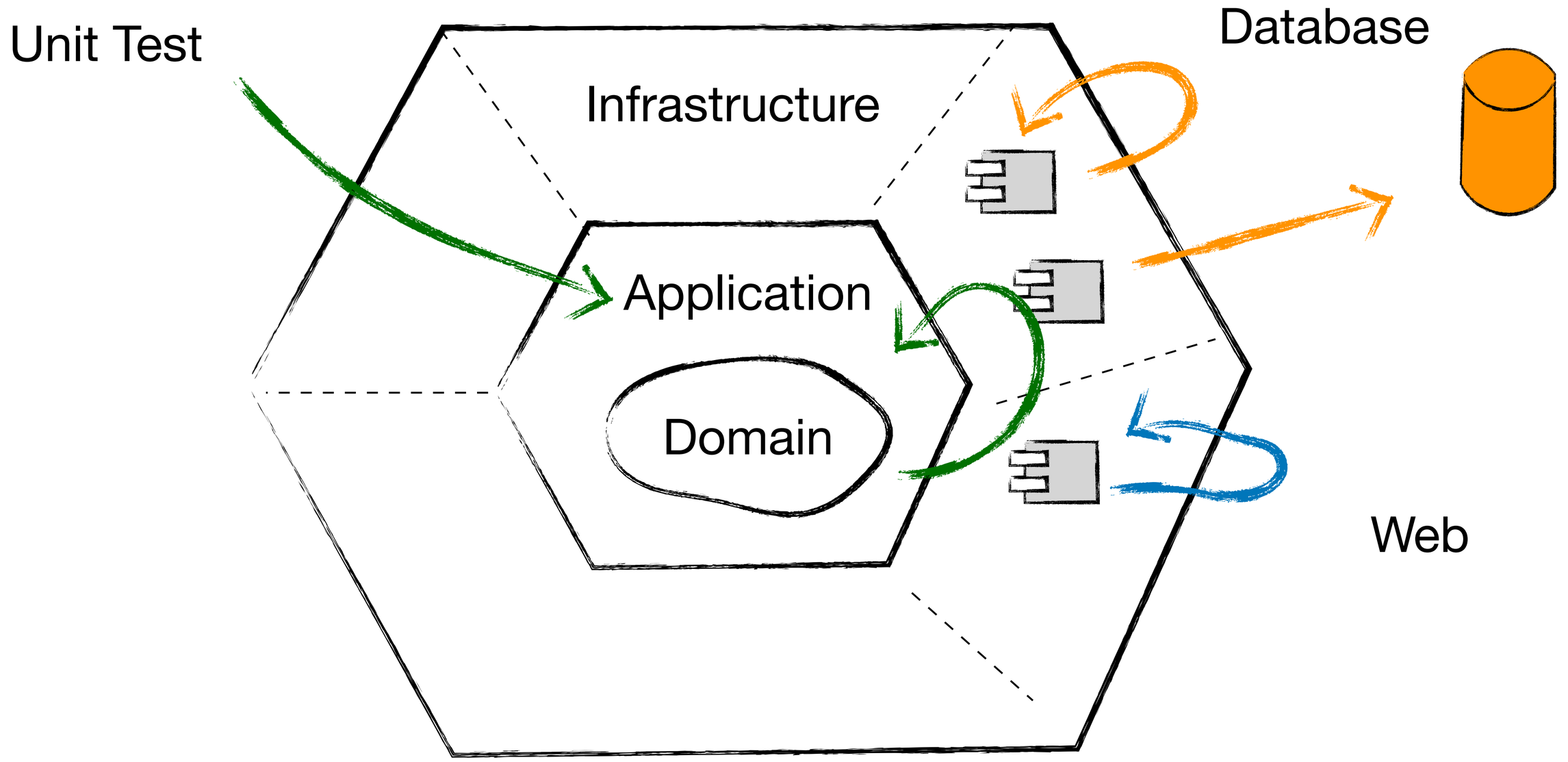
Ports and Adapters



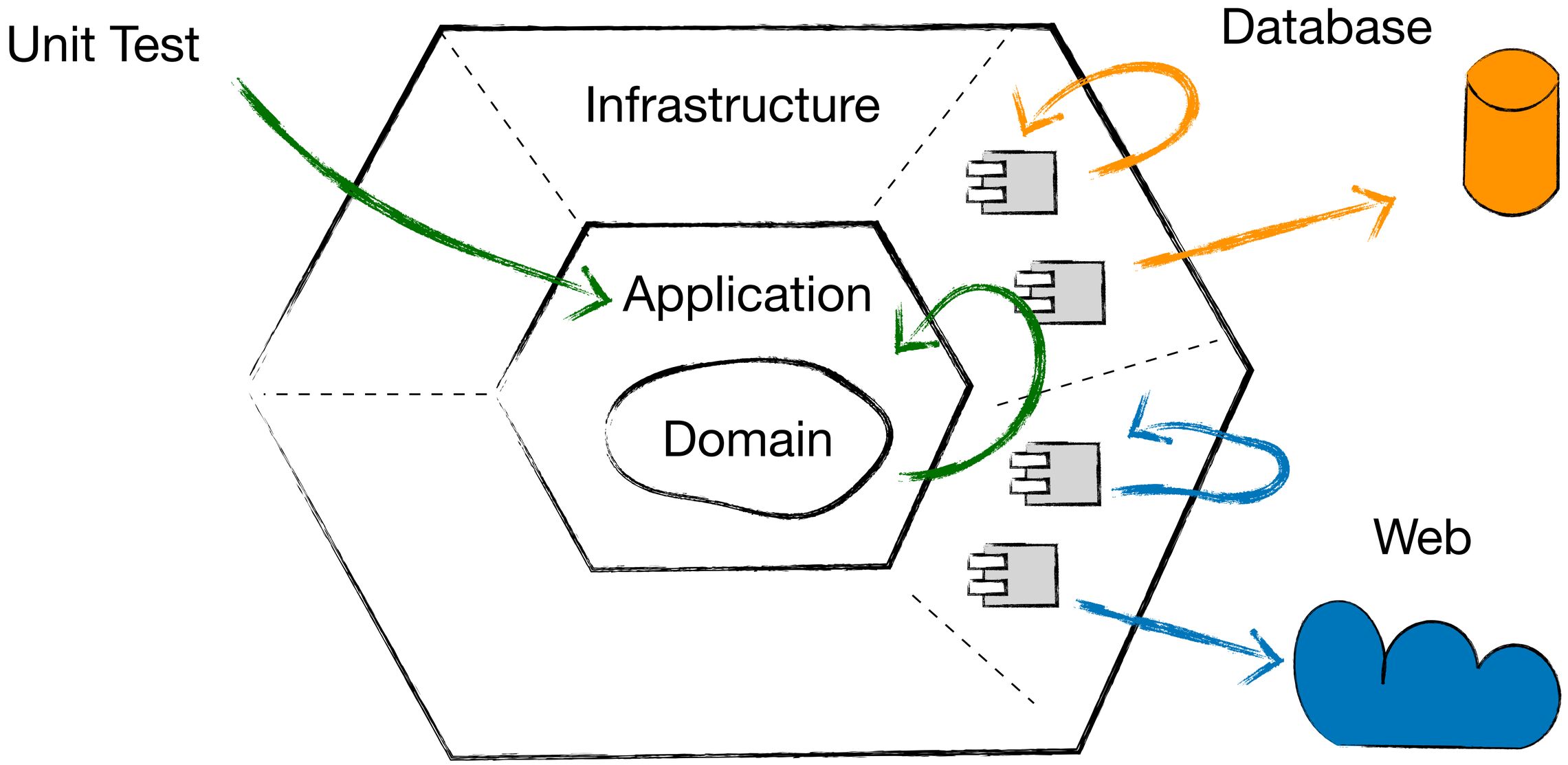
You can run it in Production
if you want.

Get feedback!

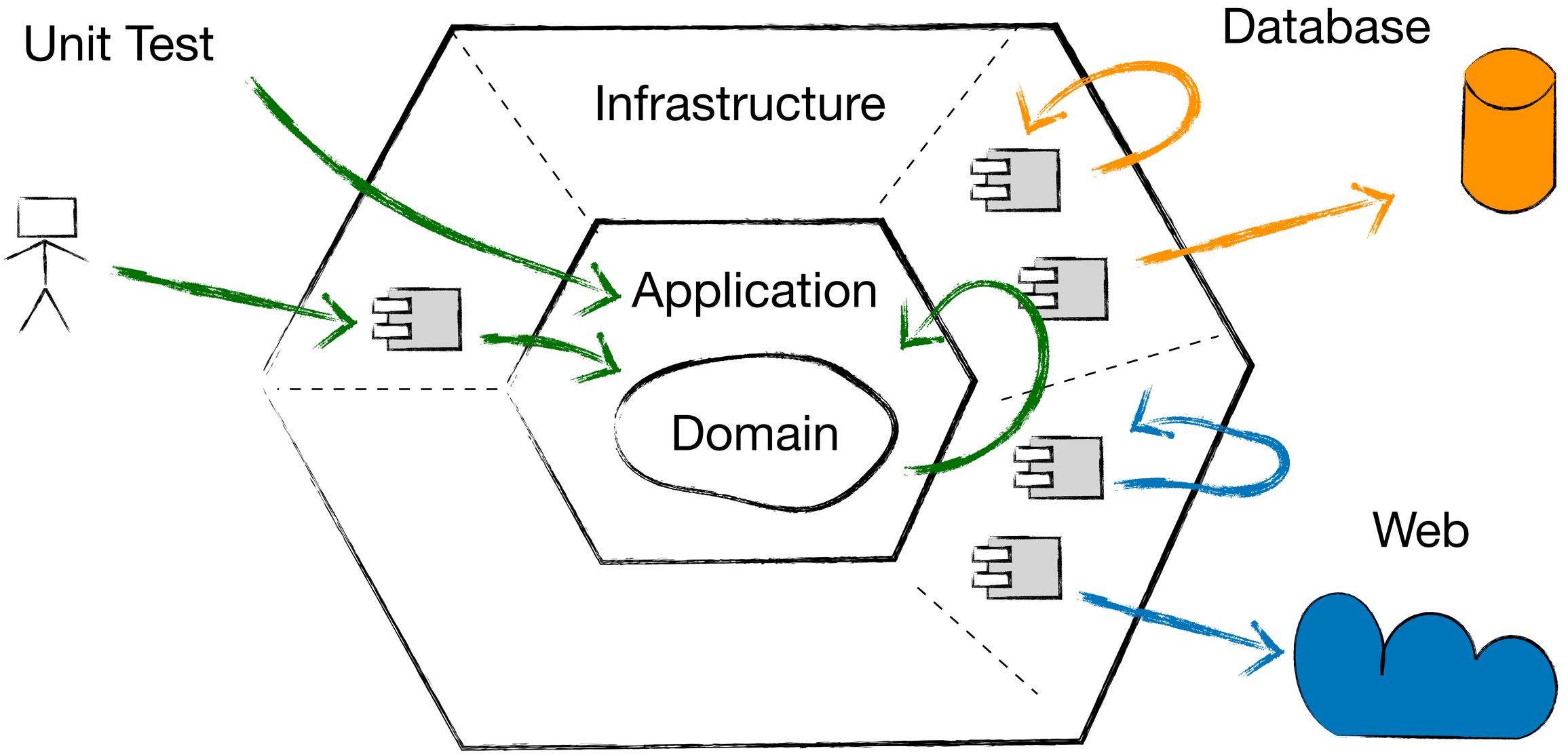
Ports and Adapters



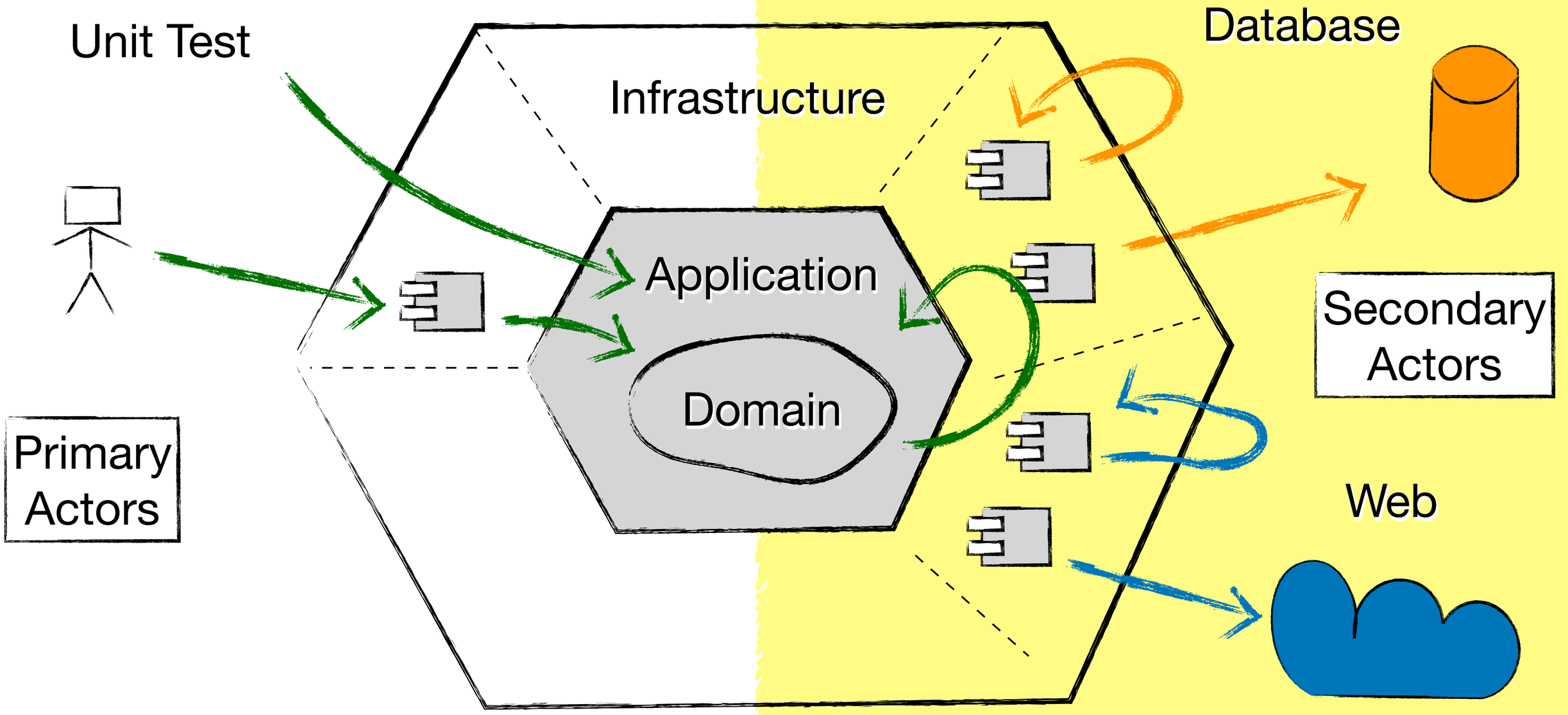
Ports and Adapters



Ports and Adapters



Ports and Adapters



Frameworks are *Details*

- Application code called by frameworks should be under control.
- Hide 3rd party libraries, use them behind core interfaces.
- The .NET Framework is a detail.
- Keep outside the Business Layer:
  Reflection, Linq, Entity Framework, MVC, Data Annotations, WCF.

```
public sealed class CreditsCollection
{
    private readonly IList<ICredit> _credits;

    public CreditsCollection()
    {
        _credits = new List<ICredit>();
    }

    public void Add<T>(IEnumerable<T> credits) where T : ICredit
    {
        foreach (var credit in credits)
        {
            Add(credit);
        }
    }

    public void Add(ICredit credit) => _credits.Add(credit);

    public IReadOnlyCollection<ICredit> GetTransactions()
    {
        var transactions = new ReadOnlyCollection<ICredit>(_credits);
        return transactions;
    }

    public PositiveMoney GetTotal()
    {
        PositiveMoney total = new PositiveMoney(0);

        foreach (ICredit credit in _credits)
        {
            total = credit.Sum(total);
        }

        return total;
    }
```

- First-Class Collection.
- Encapsulation.

The User Interface is a *detail*

```
public async Task<IActionResult> Get([FromRoute][Required] GetAccountDetailsRequestV2 request)
{
    var input = new GetAccountDetailsInput(request.AccountId);
    var output = _getAccountDetailsUseCase.Execute(input);

    if (input == null)
    {
        // return NotFound()
    }
    else
    {
        // return OkObjectResult(output);
    }
```

The User Interface is a *detail*

```
/// <summary>
/// Get an account details
/// </summary>
[HttpGet("{AccountId}", Name = "GetAccount")]
[ProducesResponseType(StatusCodes.Status200OK, Type = typeof(GetAccountDetailsResponse))]
[ProducesResponseType(StatusCodes.Status404NotFound)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
[ProducesResponseType(StatusCodes.Status500InternalServerError)]
public async Task<IActionResult> Get([FromRoute][Required] GetAccountDetailsRequest request)
{
    var input = new GetAccountDetailsInput(request.AccountId);
    await _mediator.PublishAsync(input);
    return _presenter.ViewModel;
}
```

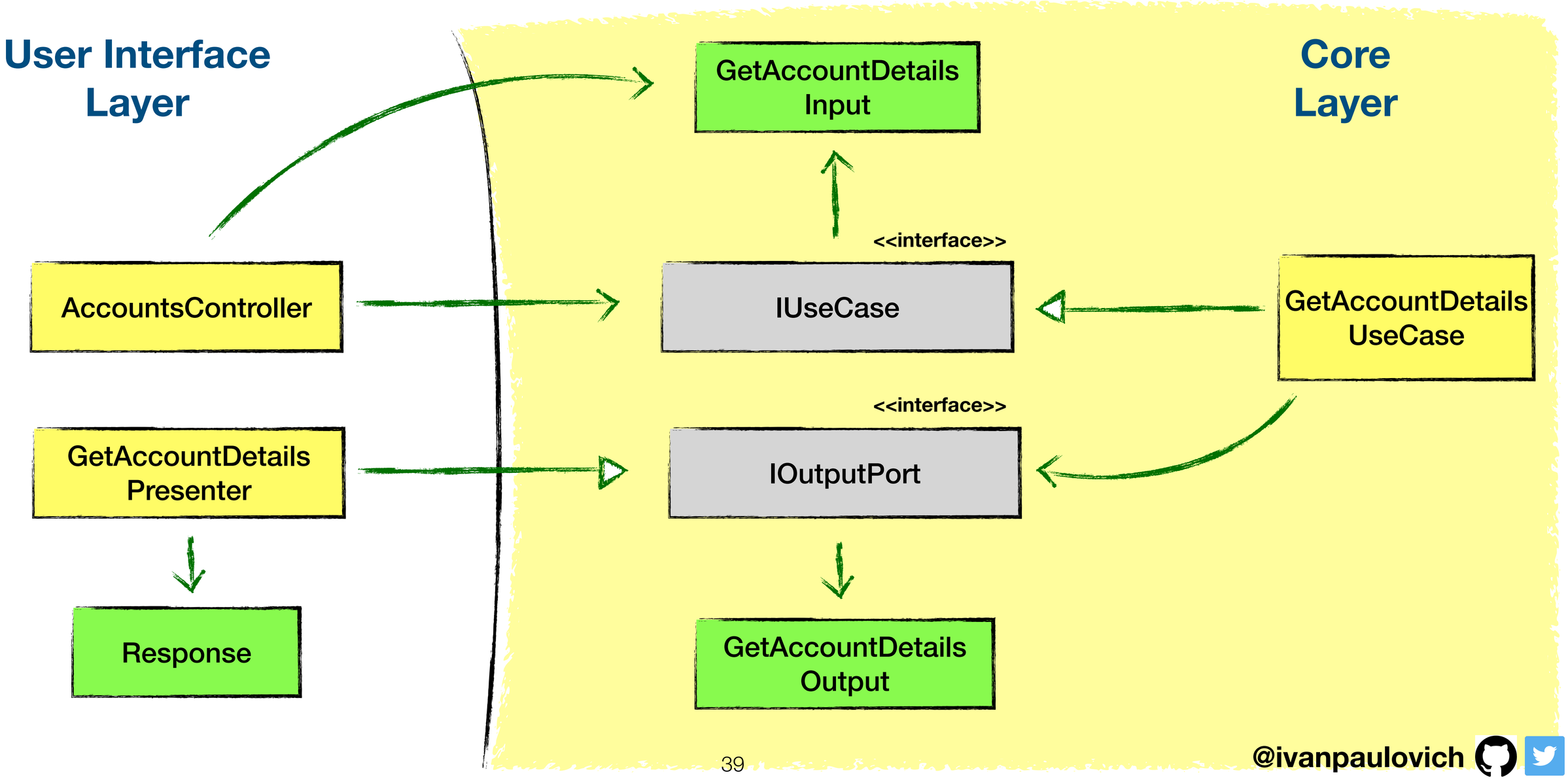
```
public interface IOutputPort
{
    void NotFound(string message);
    void Standard(GetAccountDetailsOutput output);
}
```

```
public sealed class GetAccountDetails : IUseCase, IUseCaseV2
{
    private readonly IOutputPort _outputPort;
    private readonly IAccountRepository _accountRepository;

    public GetAccountDetails(
        IOutputPort outputPort,
        IAccountRepository accountRepository)
    {
        _outputPort = outputPort;
        _accountRepository = accountRepository;
    }

    public async Task Execute(GetAccountDetailsInput input)
    {
        IAccount account;

        try
        {
            account = await _accountRepository.Get(input.AccountId);
        }
        catch (AccountNotFoundException ex)
        {
            _outputPort.NotFound(ex.Message);
            return;
        }

        var output = new GetAccountDetailsOutput(account);
        _outputPort.Standard(output);
    }
```

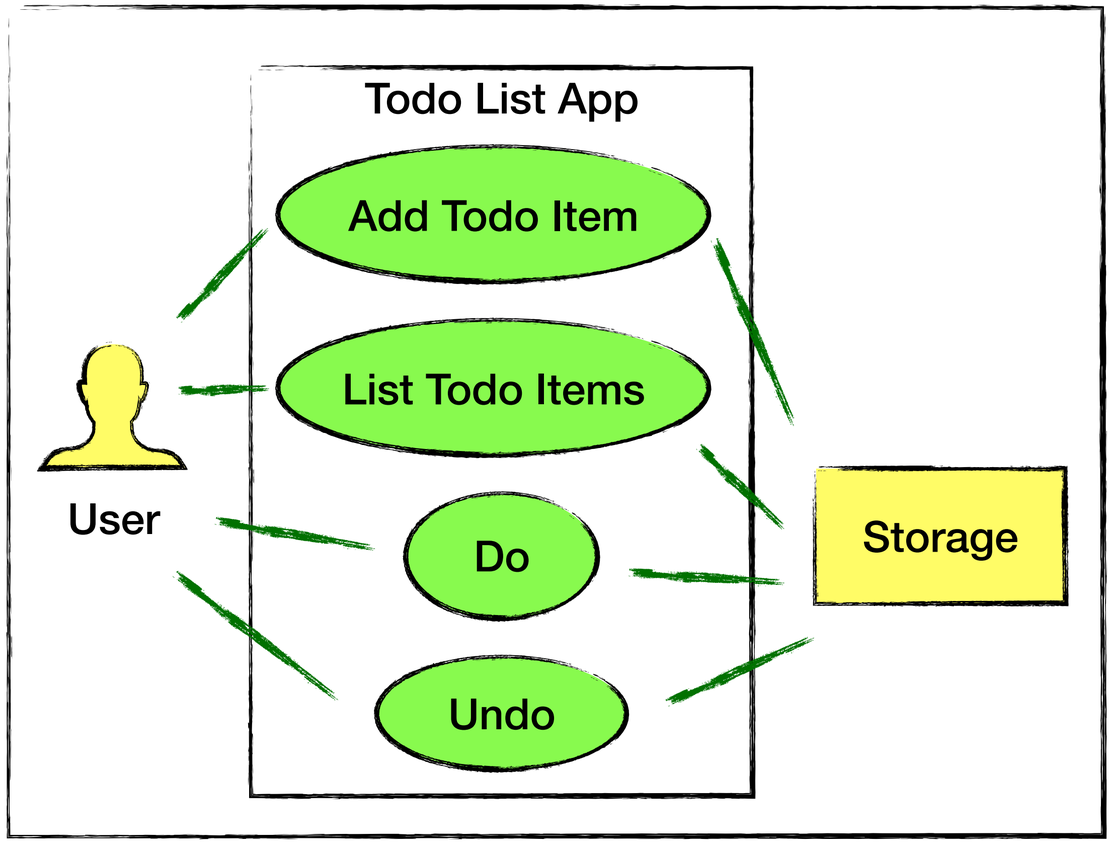
The User Interface is a *detail*

```
public sealed class GetAccountDetailsPresenter : IOutputPort
{
    public IActionResult ViewModel { get; private set; }

    public void NotFound(string message)
    {
        ViewModel = new NotFoundObjectResult(message);
    }

    public void Standard(GetAccountDetailsOutput getAccountDetailsOutput)
    {
        List<TransactionModel> transactions = new List<TransactionModel>();

        foreach (var item in getAccountDetailsOutput.Transactions)
        {
            var transaction = new TransactionModel(
                item.Amount.ToMoney().ToDecimal(),
                item.Description,
                item.TransactionDate);

            transactions.Add(transaction);
        }

        var getAccountDetailsResponse = new GetAccountDetailsResponse(
            getAccountDetailsOutput.AccountId,
            getAccountDetailsOutput.CurrentBalance.ToDecimal(),
            transactions);

        ViewModel = new OkObjectResult(getAccountDetailsResponse);
    }
}
```

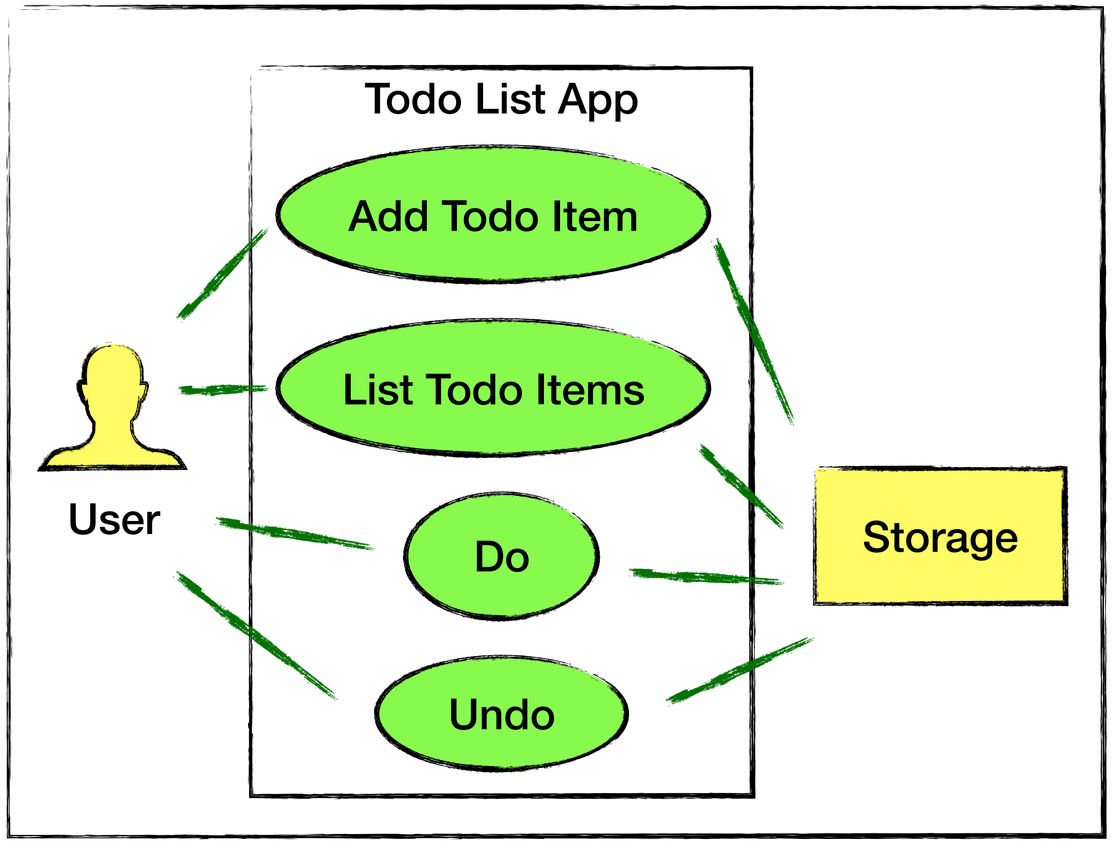
The User Interface is a *detail*

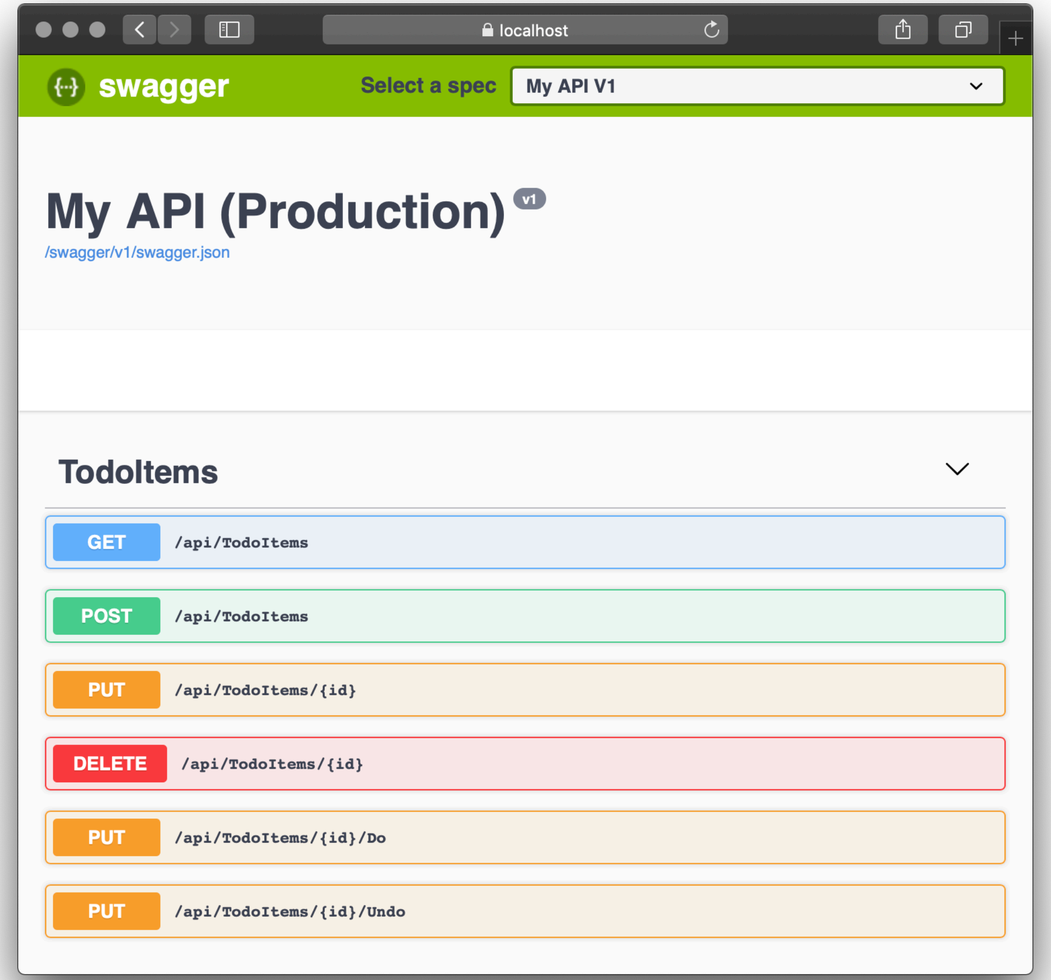


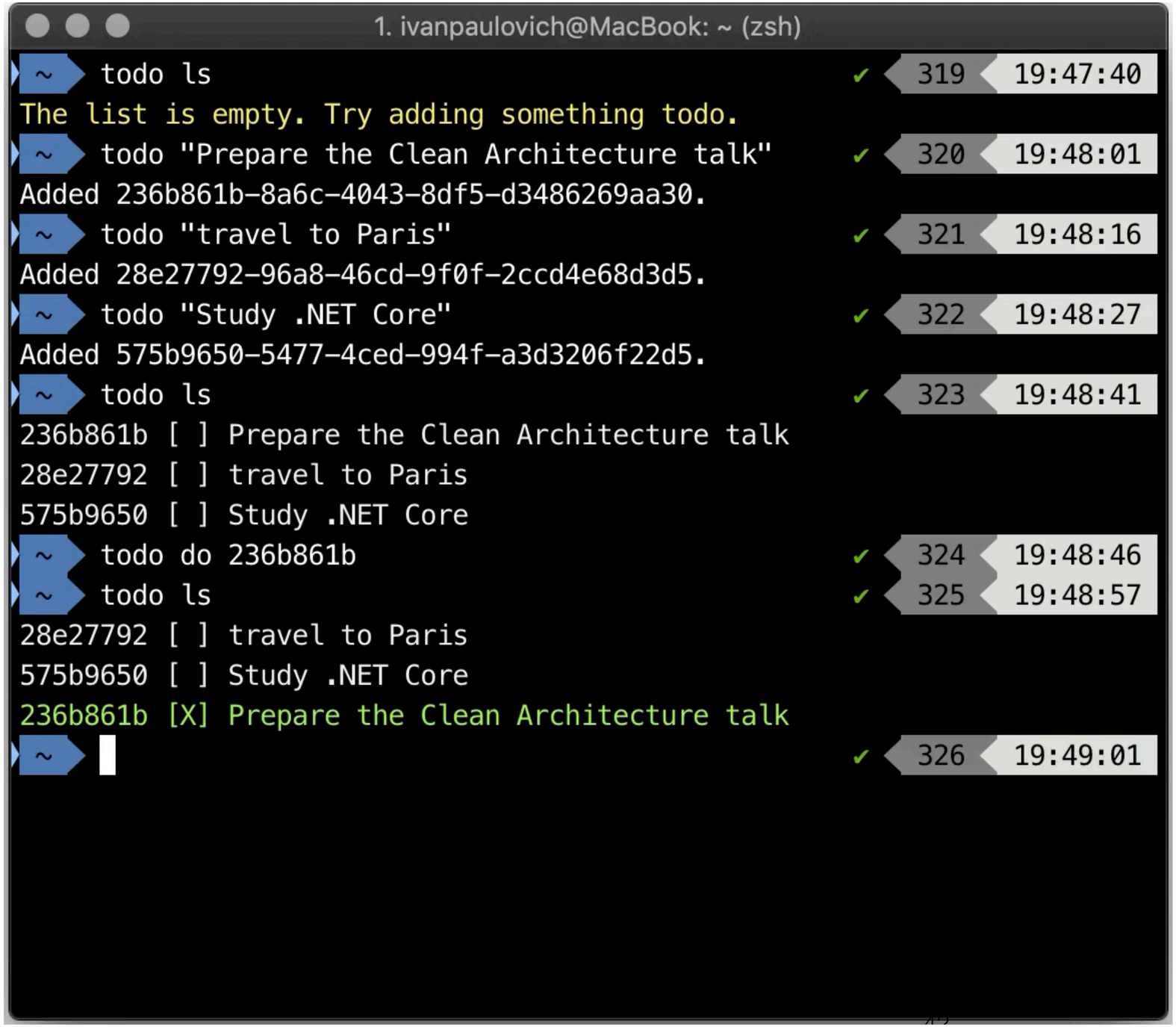
Sample Application



Sample Application







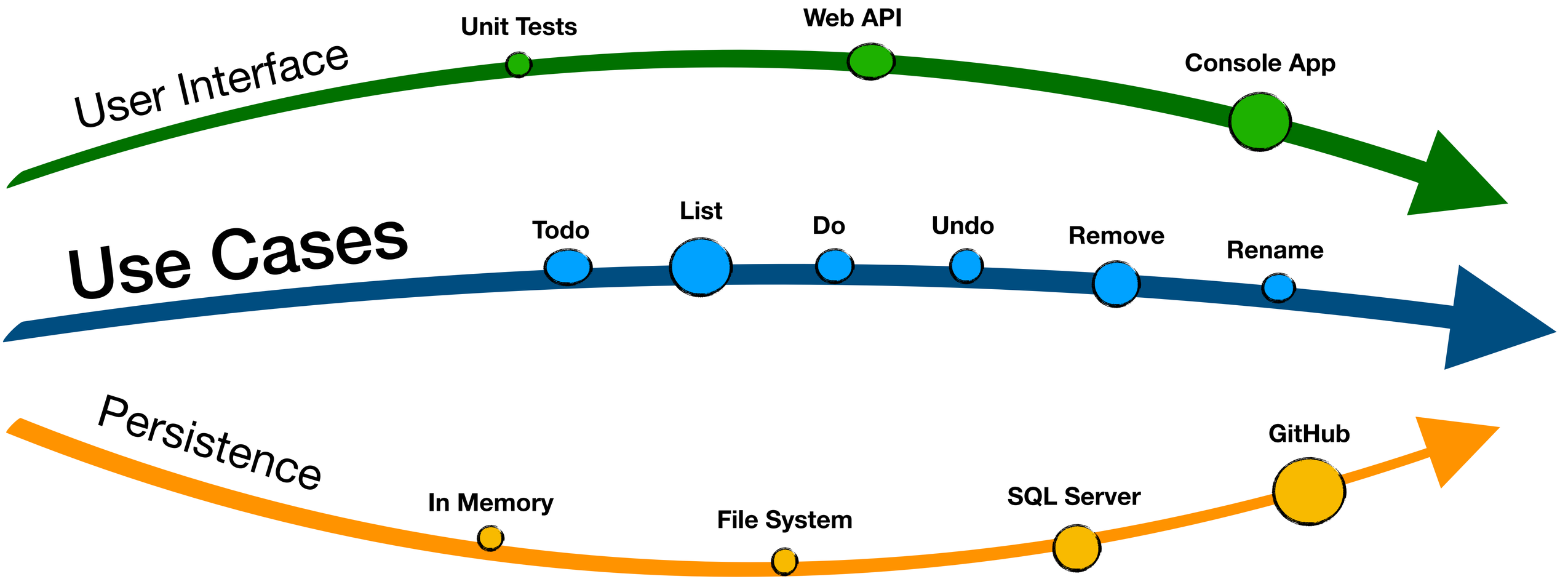
✅ Commad-Line Task management with storage on your GitHub 🔥

```
$ dotnet tool install -g todo
```

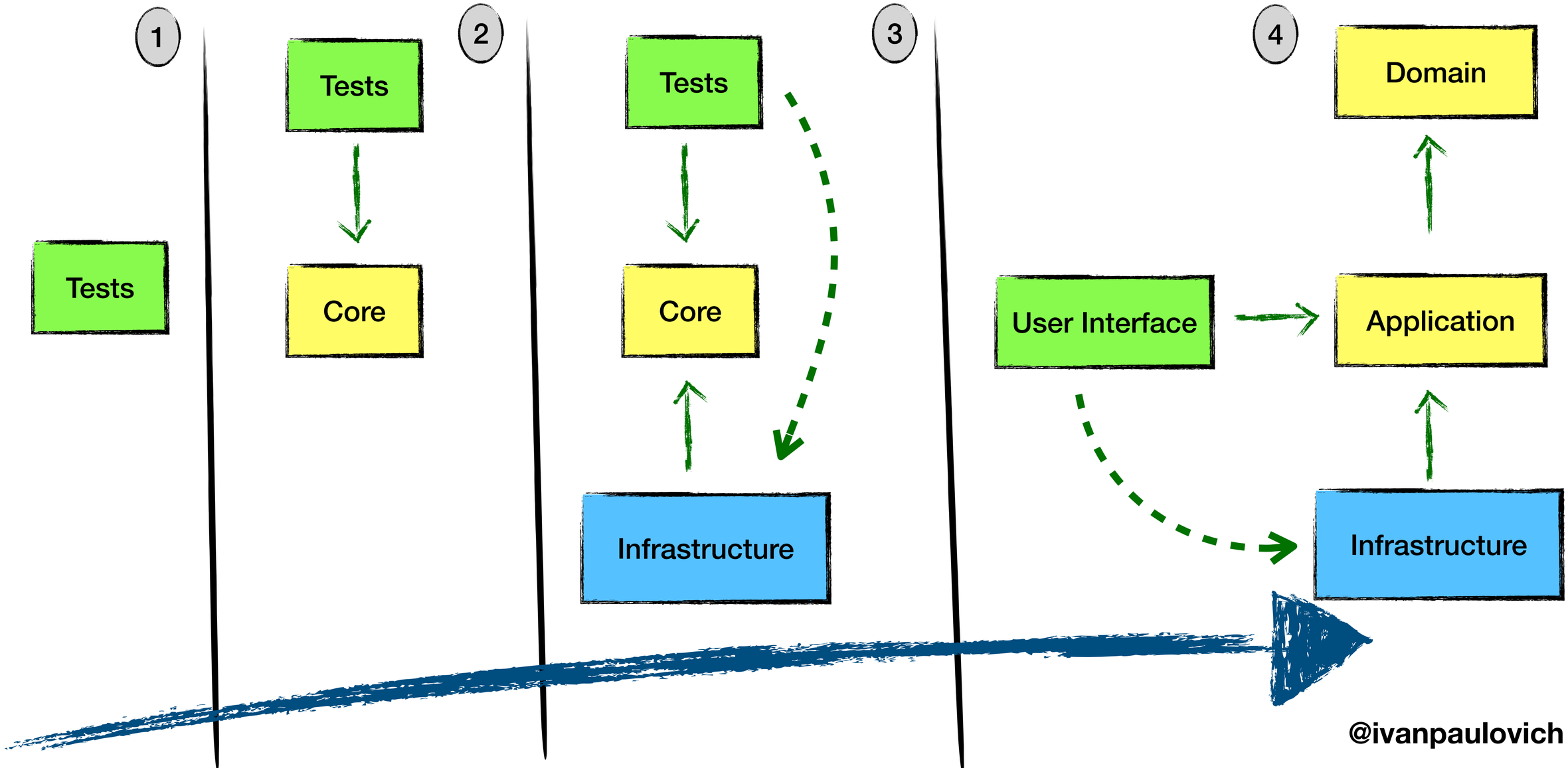
https://github.com/ivanpaulovich/todo

@ivanpaulovich

Evolutionary Architecture In Action



Splitting packages on the software lifetime



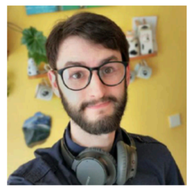
Wrapping up Clean Architecture

- Clean Architecture is about usage and the use cases are the central organising principle.
- Use cases implementation are guided by tests.
- The User Interface and Persistence are designed to fulfil the core needs (not the opposite!).
- Defer decisions by implementing the simplest component first.

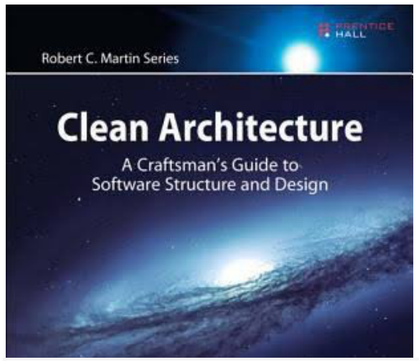
References

https://github.com/ivanpaulovich/clean-architecture-manga

Thanks goes to these wonderful people (emoji key):

 Ivan Paulovich	 Petr Sedláček	Gus	 arulconsultant	 Guilherme Silva
 Ondřej Štorc	Marlon Miranda da Silva	NicoCG	 Filipe Augusto Lima de Souza	sshaw-sml
 Matheus Neder	 димитрий матиенко	 morphlogic	 Felipe Lambert	 Philippe Matray

https://cleancoders.com



Clean Code: Component Design
Clean Code: SOLID Principles
Clean Code: Fundamentals

Ask me two questions!

always coding 🍄

Ivan Paulovich
ivanpaulovich

Edit profile

Agile Software Developer, Tech Lead, 20+ GitHub projects about Clean Architecture, SOLID, DDD and TDD. Speaker/Streamer. rMVP.

Overview | Repositories 59 | Projects 0 | Packages 0 | Stars 278 | Followers 360 | Follow...

Pinned

FluentMediator
FluentMediator is an unobtrusive library that allows developers to build custom pipelines for Commands, Queries and Events.
C# ★ 24 5

clean-architecture-manga
Clean Architecture sample with .NET Core 3.0 and C# 8. Use cases as central organising structure, completely testable, decoupled from frameworks
C# ★ 720 165

todo
✅ Commad-Line Task management with storage on your GitHub 🔥
C# ★ 78 11

dotnet-new-caju Template
Learn Clean Architecture with .NET Core 3.0 🔥
C# ★ 169 25

event-sourcing-jambo
An Hexagonal Architecture with DDD + Aggregates + Event Sourcing using .NET Core, Kafka e MongoDB (Blog Engine)
C# ★ 132 53

clean-architecture-webapi-ef-core
❤️ The simplest Clean Architecture demo on how to implement a Web Api using .NET Core and Entity Framework
C# ★ 66 15

Fork me on GitHub

DERIVCO SPORTS
proudly powering betway

Nordax Bank

@ivanpaulovich