CITERUS

Tactical Design and Clean Architecture

Ivan Paulovich

Stockholm Domain-Driven Design Meetup
January 30, 2020

@ivanpaulovich

https://paulovich.net

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

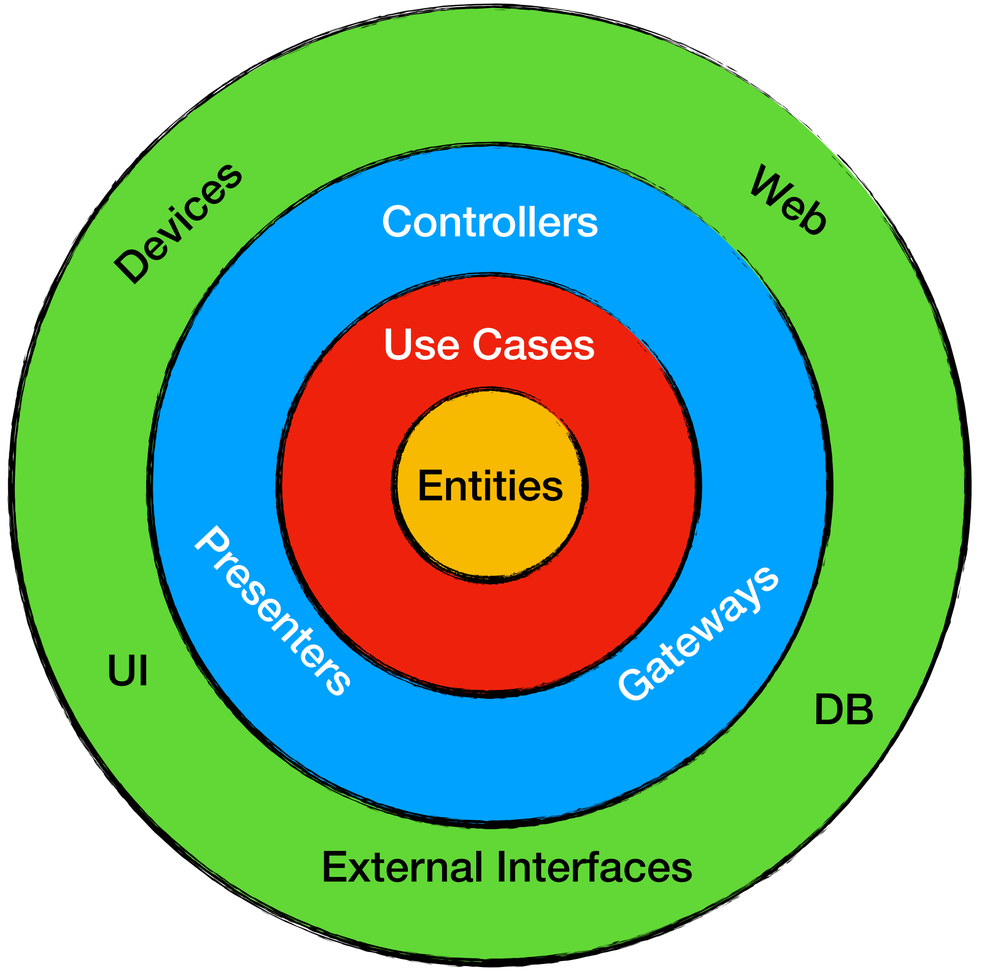
@ivanpaulovich

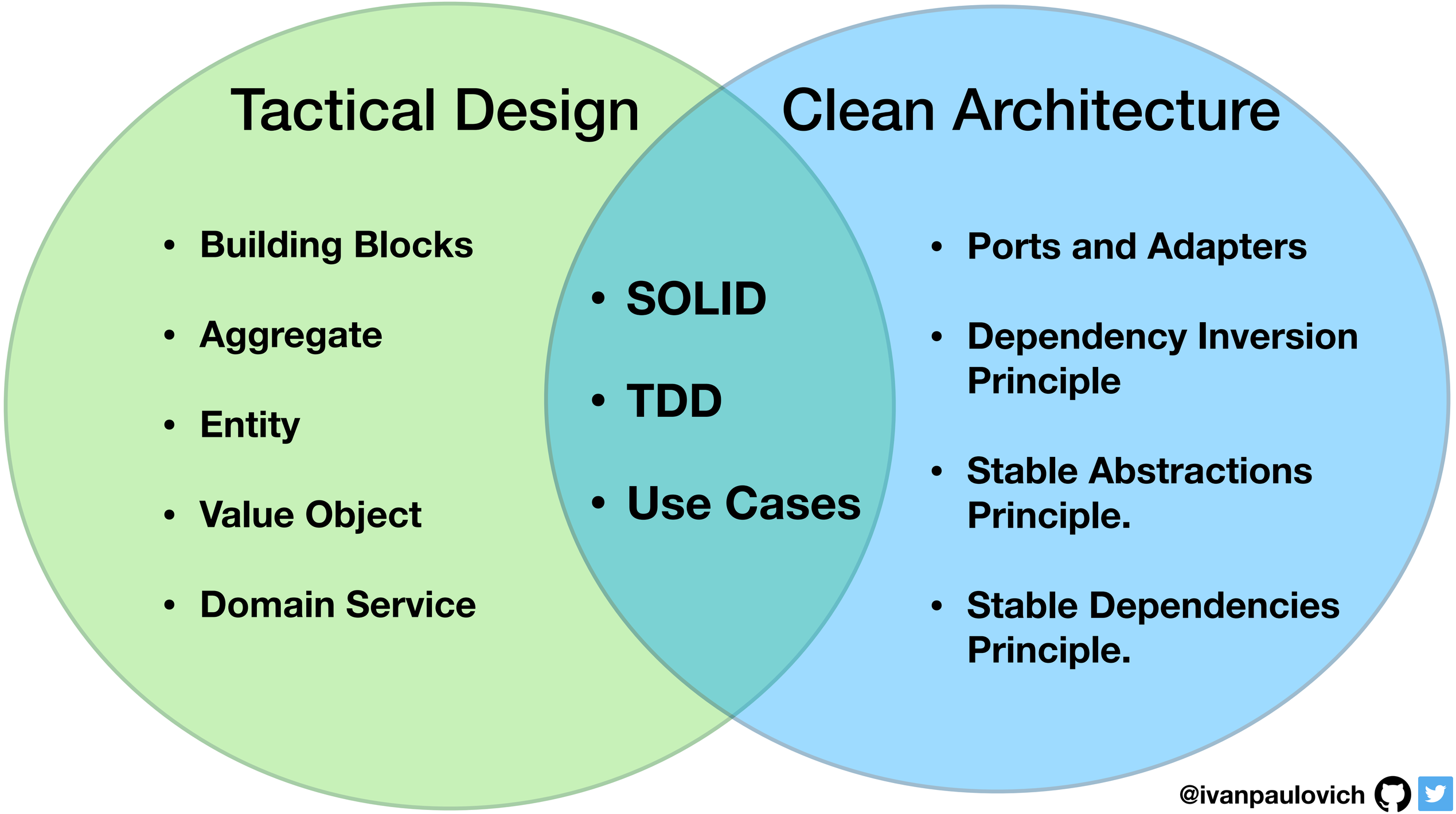
Tactical Design

- Design Patterns and Building Blocks applied within a Bounded Context.
- Used to enrich the Domain Model.
- It is hands-on (Classes, Modules)!
- It is the opposite of Anaemic Model!

Clean Architecture

- Patterns (not too many).
- Principles (lots of them).
- Practices (TDD?).
- Hands-on (Classes, Modules)

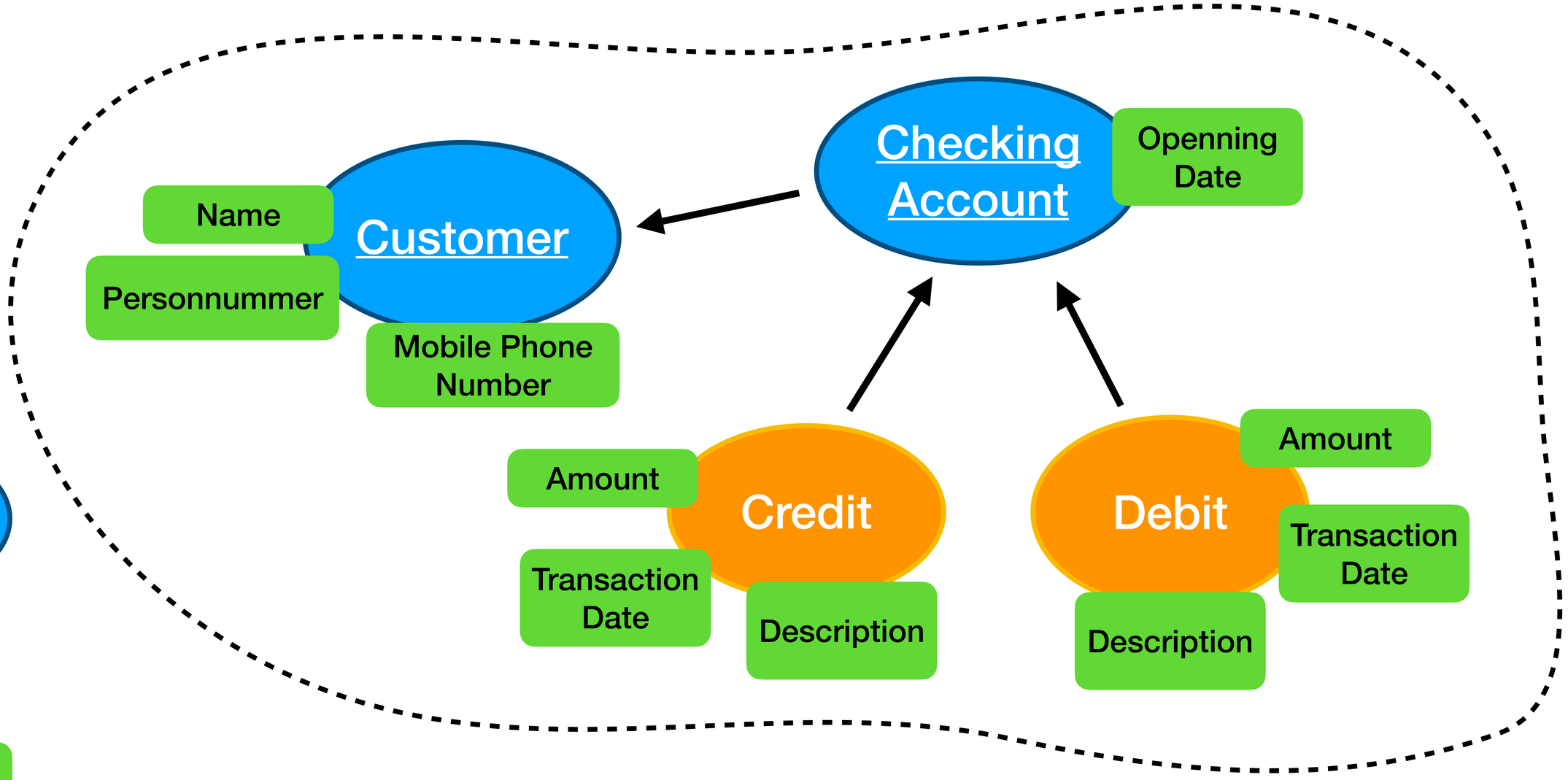



Tactical Design
Clean Architecture
• Building Blocks
• Aggregate
• Entity
• Value Object
• Domain Service
• SOLID
• TDD
• Use Cases
• Ports and Adapters
• Dependency Inversion Principle
• Stable Abstractions Principle.
• Stable Dependencies Principle.

Domain-Driven Design Patterns

- Bounded Context
- Value Object
- Entity
- Aggregate Root
- Repository
- Use Case
- Entity Factory
- Domain Service
- Application Service

Virtual Wallet



Aggregate Root

Entity

Value Object

@ivanpaulovich

Entity

1. Highly abstract.
2. Mutable object.
3. Unique identified by an ID (inside the aggregate).

```
public interface ICredit
{
    PositiveMoney Sum(PositiveMoney amount);
}
```

```
public abstract class Credit : ICredit
{
    public CreditId Id { get; protected set; }

    public PositiveMoney Amount { get; protected set; }

    public string Description
    {
        get { return "Credit"; }
    }

    public DateTime TransactionDate { get; protected set; }

    public PositiveMoney Sum(PositiveMoney amount)
    {
        return this.Amount.Add(amount);
    }
}
```

Value Object

1. Encapsulate tiny business rules.
2. Immutable object.
3. Unique identified by comparison of the properties.

```
public readonly struct Money
{
    private readonly decimal _money;

    public Money(decimal value)
    {
        this._money = value;
    }

    public decimal ToDecimal() => this._money;

    internal bool LessThan(PositiveMoney amount)
    {
        return this._money < amount.ToMoney()._money;
    }

    internal bool IsZero()
    {
        return this._money == 0;
    }

    internal PositiveMoney Add(Money value)
    {
        return new PositiveMoney(this._money + value.ToDecimal());
    }

    internal Money Subtract(Money value)
    {
        return new Money(this._money - value.ToDecimal());
    }
}
```

Aggregate Root

1. Owns entities object graph.
2. Ensures the children entities state are always consistent.
3. Defines the consistency boundary.
4. Highly Abstract.
5. Highly Stable.

```
public abstract class Account : IAccount
{
    protected Account()
    {
        this.Credits = new CreditsCollection();
        this.Debits = new DebitsCollection();
    }

    public AccountId Id { get; protected set; }

    public CreditsCollection Credits { get; protected set; }

    public DebitsCollection Debits { get; protected set; }

    public ICredit Deposit(IAccountFactory entityFactory, PositiveMoney amountToDeposit)
    {
        var credit = entityFactory.NewCredit(this, amountToDeposit, DateTime.UtcNow);
        this.Credits.Add(credit);
        return credit;
    }

    public IDebit Withdraw(IAccountFactory entityFactory, PositiveMoney amountToWithdraw)
    {
        if (this.GetCurrentBalance().LessThan(amountToWithdraw))
        {
            throw new MoneyShouldBePositiveException("Account has not enough funds.");
        }

        var debit = entityFactory.NewDebit(this, amountToWithdraw, DateTime.UtcNow);
        this.Debits.Add(debit);
        return debit;
    }

    public bool IsClosingAllowed()
    {
        return this.GetCurrentBalance().IsZero();
    }
```

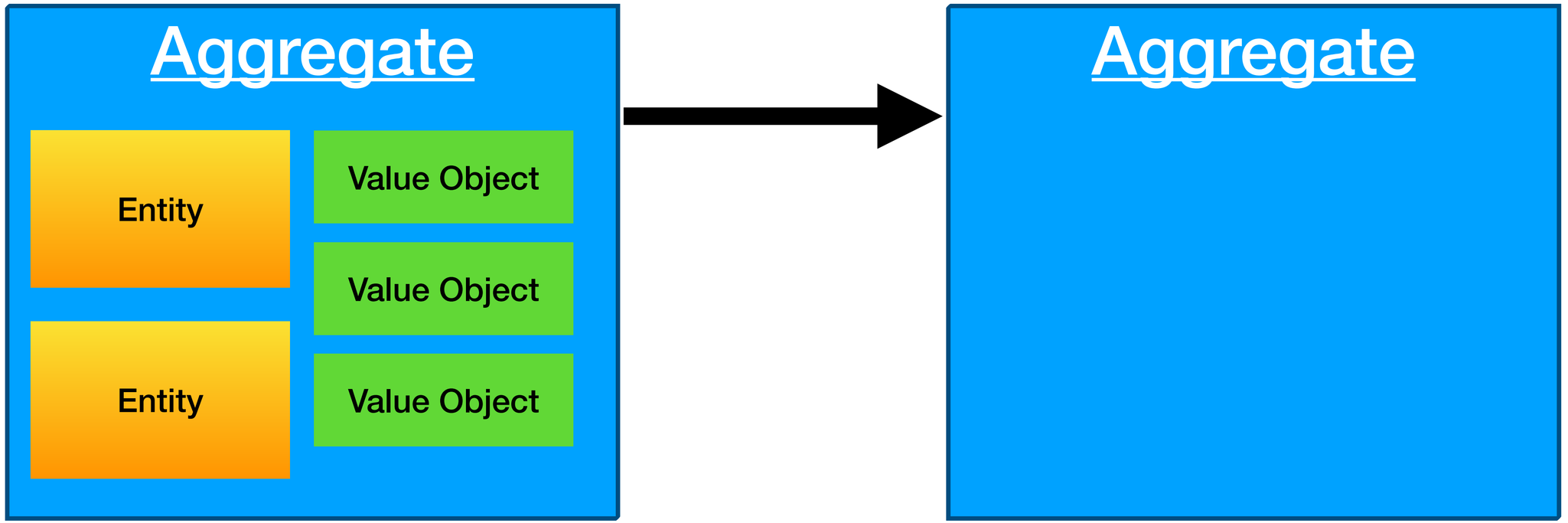
Aggregate Root

1. Owns entities object graph.
2. Ensures the children entities state are always consistent.
3. Defines the consistency boundary.
4. Highly Abstract.
5. Highly Stable.

```
public interface IAccount
{
    AccountId Id { get; }

    ICredit Deposit(IAccountFactory entityFactory, PositiveMoney amountToDeposit);

    IDebit Withdraw(IAccountFactory entityFactory, PositiveMoney amountToWithdraw);

    bool IsClosingAllowed();

    Money GetCurrentBalance();
}
public abstract class Account : IAccount
{
    protected Account()
    {
        this.Credits = new CreditsCollection();
        this.Debits = new DebitsCollection();
    }

    public AccountId Id { get; protected set; }

    public CreditsCollection Credits { get; protected set; }

    public DebitsCollection Debits { get; protected set; }

    public ICredit Deposit(IAccountFactory entityFactory, PositiveMoney amountToDeposit)
    {
        var credit = entityFactory.NewCredit(this, amountToDeposit, DateTime.UtcNow);
        this.Credits.Add(credit);
        return credit;
    }

    public IDebit Withdraw(IAccountFactory entityFactory, PositiveMoney amountToWithdraw)
    {
        if (this.GetCurrentBalance().LessThan(amountToWithdraw))
        {
            throw new MoneyShouldBePositiveException("Account has not enough funds.");
        }
```

@ivanpaulovich

Aggregates Together



- Aggregates know each other only by ID.
- Keep a low coupling between them.

Repository

1. Provides persistence capabilities to Aggregate Roots.

2. A Repository for every Entity is a code smell.

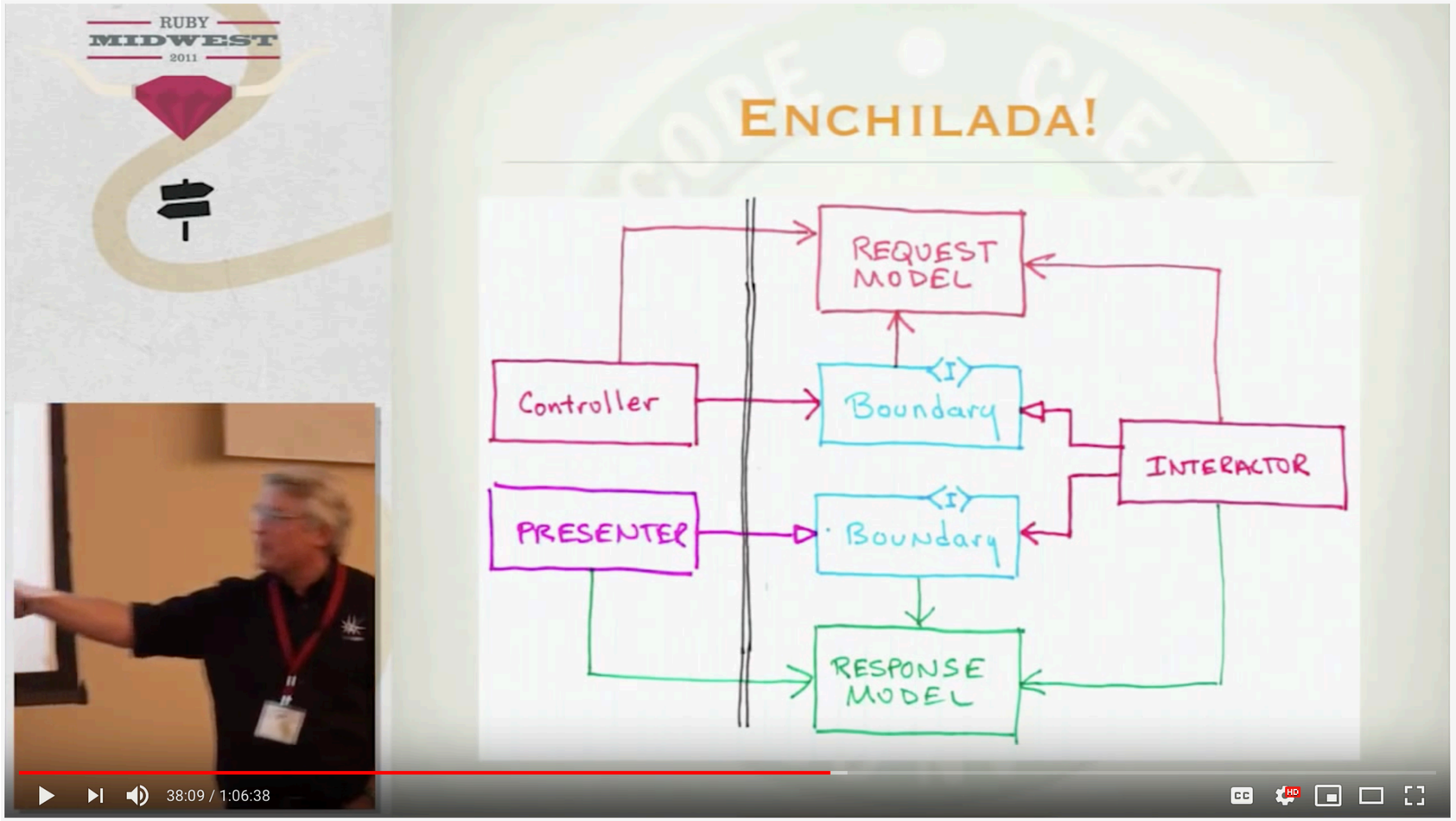
```
public interface IAccountRepository
{
    Task<IAccount> Get(AccountId id);

    Task Add(IAccount account, ICredit credit);

    Task Update(IAccount account, ICredit credit);

    Task Update(IAccount account, IDebit debit);

    Task Delete(IAccount account);
}
```

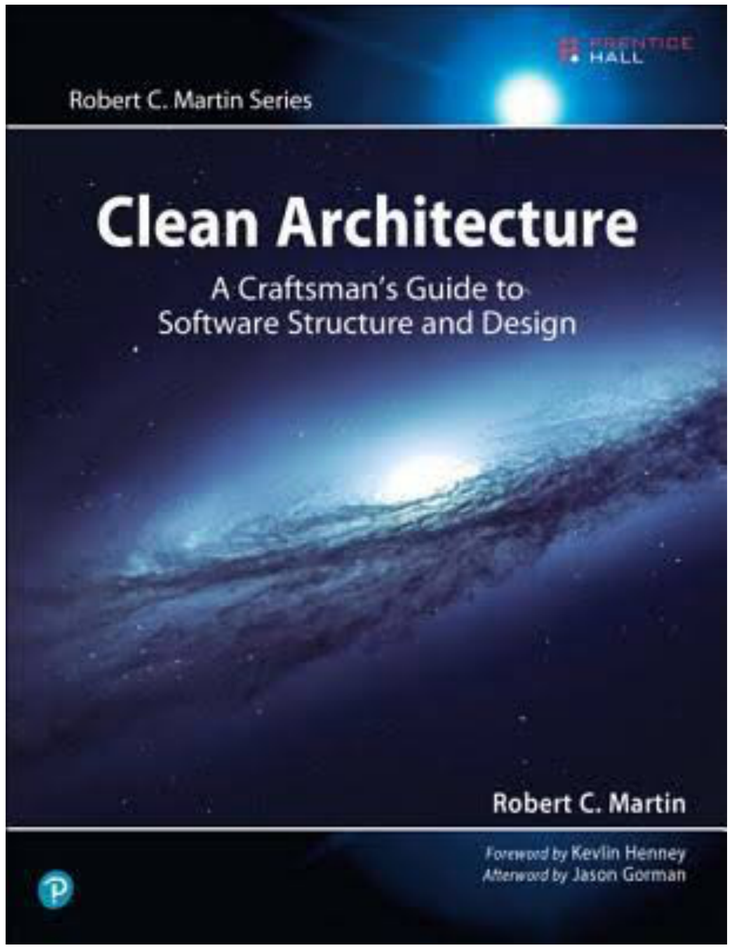
Domain Service

- Cross-entities functions.
- Without side effects outside the application memory.

```
public class AccountService
{
    private readonly IAccountFactory _accountFactory;
    private readonly IAccountRepository _accountRepository;

    public AccountService(
        IAccountFactory accountFactory,
        IAccountRepository accountRepository)
    {
        this._accountFactory = accountFactory;
        this._accountRepository = accountRepository;
    }

    public async Task<IAccount> OpenCheckingAccount(CustomerId customerId, PositiveMoney amount)
    {
        var account = this._accountFactory.NewAccount(customerId);
        var credit = account.Deposit(this._accountFactory, amount);
        await this._accountRepository.Add(account, credit);

        return account;
    }

    public async Task<IDebit> Withdraw(IAccount account, PositiveMoney amount)
    {
        var debit = account.Withdraw(this._accountFactory, amount);
        await this._accountRepository.Update(account, debit);

        return debit;
    }

    public async Task<ICredit> Deposit(IAccount account, PositiveMoney amount)
    {
        var credit = account.Deposit(this._accountFactory, amount);
        await this._accountRepository.Update(account, credit);

        return credit;
    }
}
```

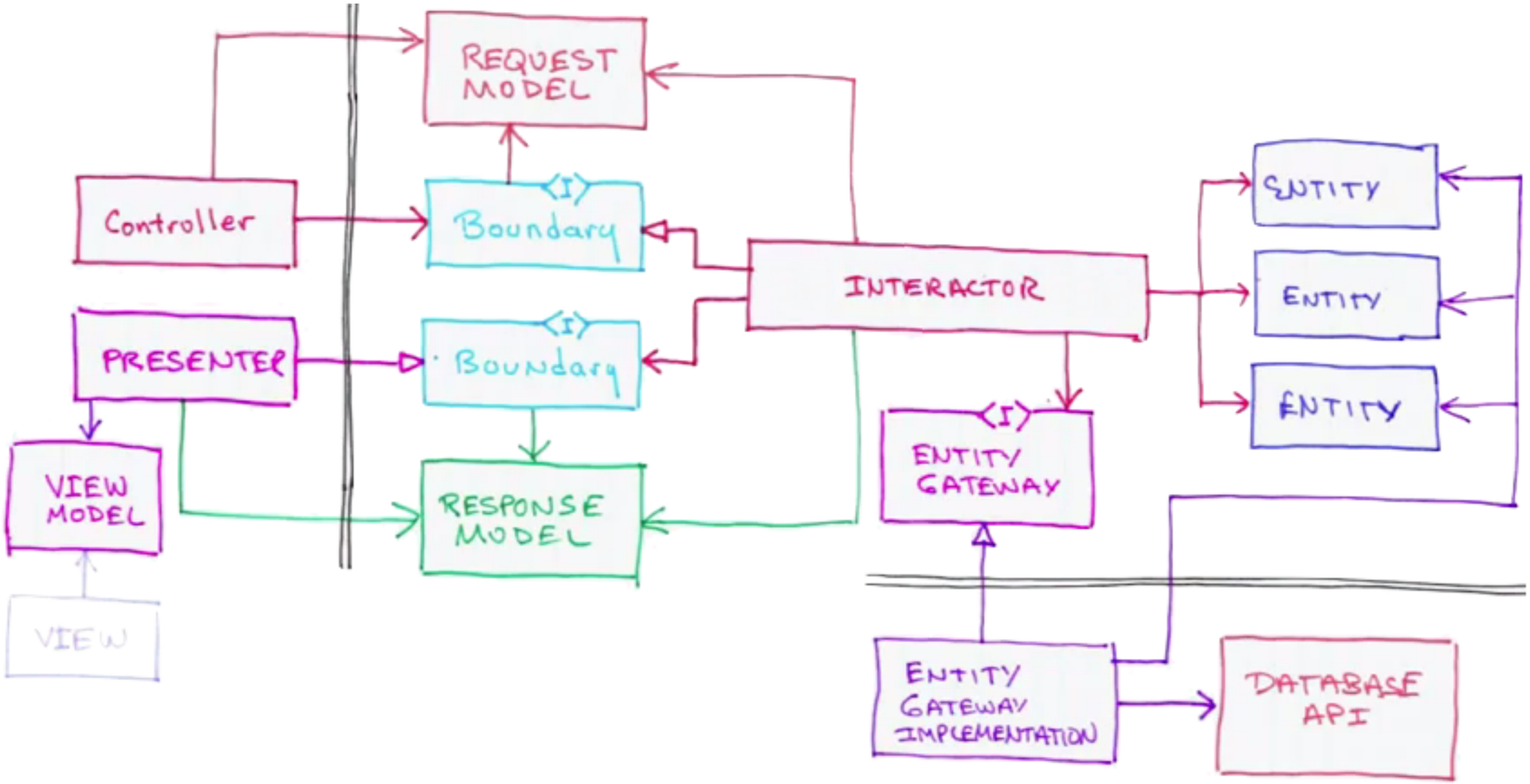
Architecture The Lost Years by Robert C. Martin





@ivanpaulovich

Architecture UML - Uncle Bob



The “Software Architecture”

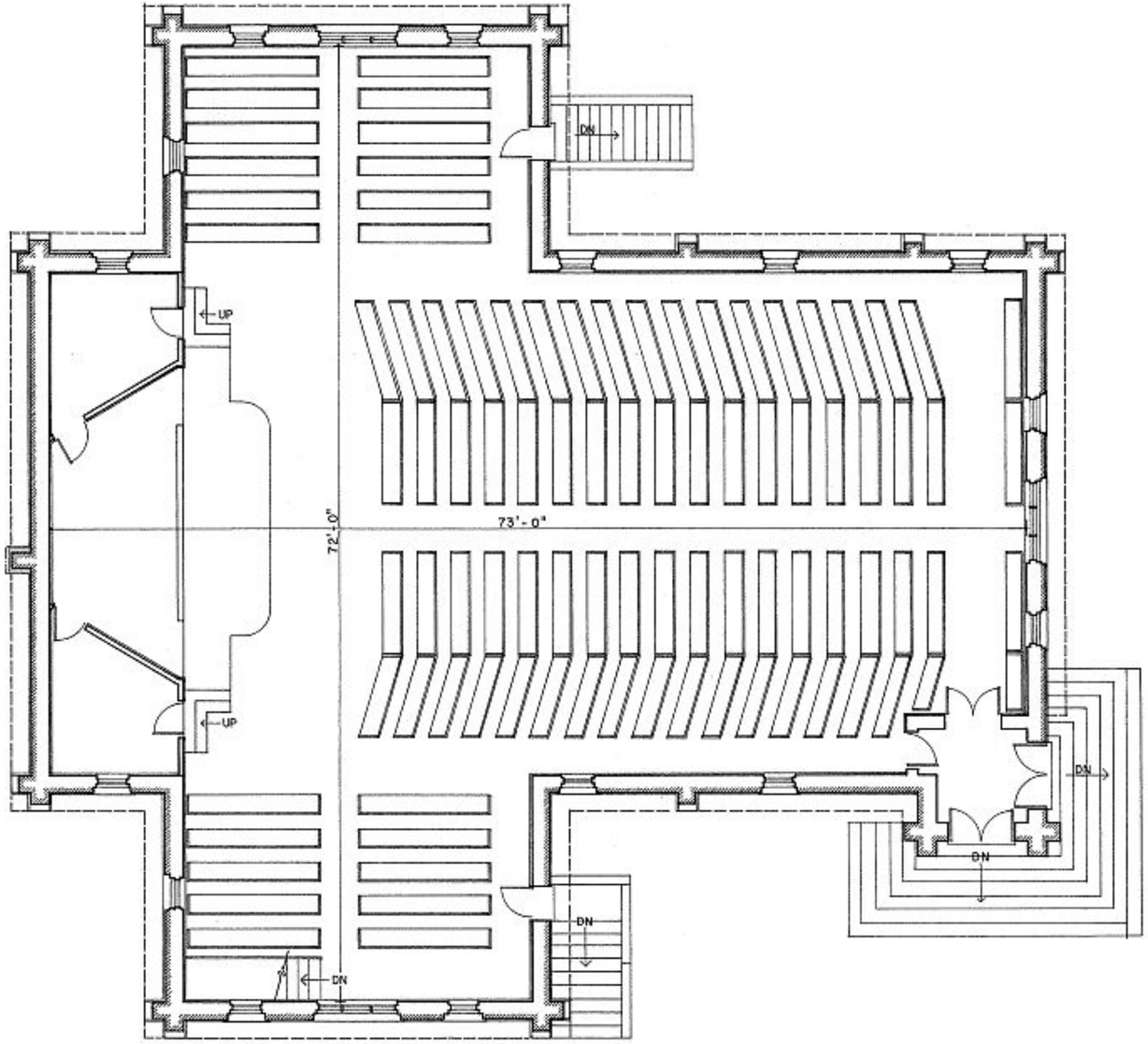
Presentation Layer	ASP.NET MVC and Autofac
Business Layer	MediatR and AutoMapper
Data Layer	Entity Framework and Dapper
Database and Messaging	SQL Server and RabbitMQ
IDE and Tools	Visual Studio, Resharper and .NET Framework

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!

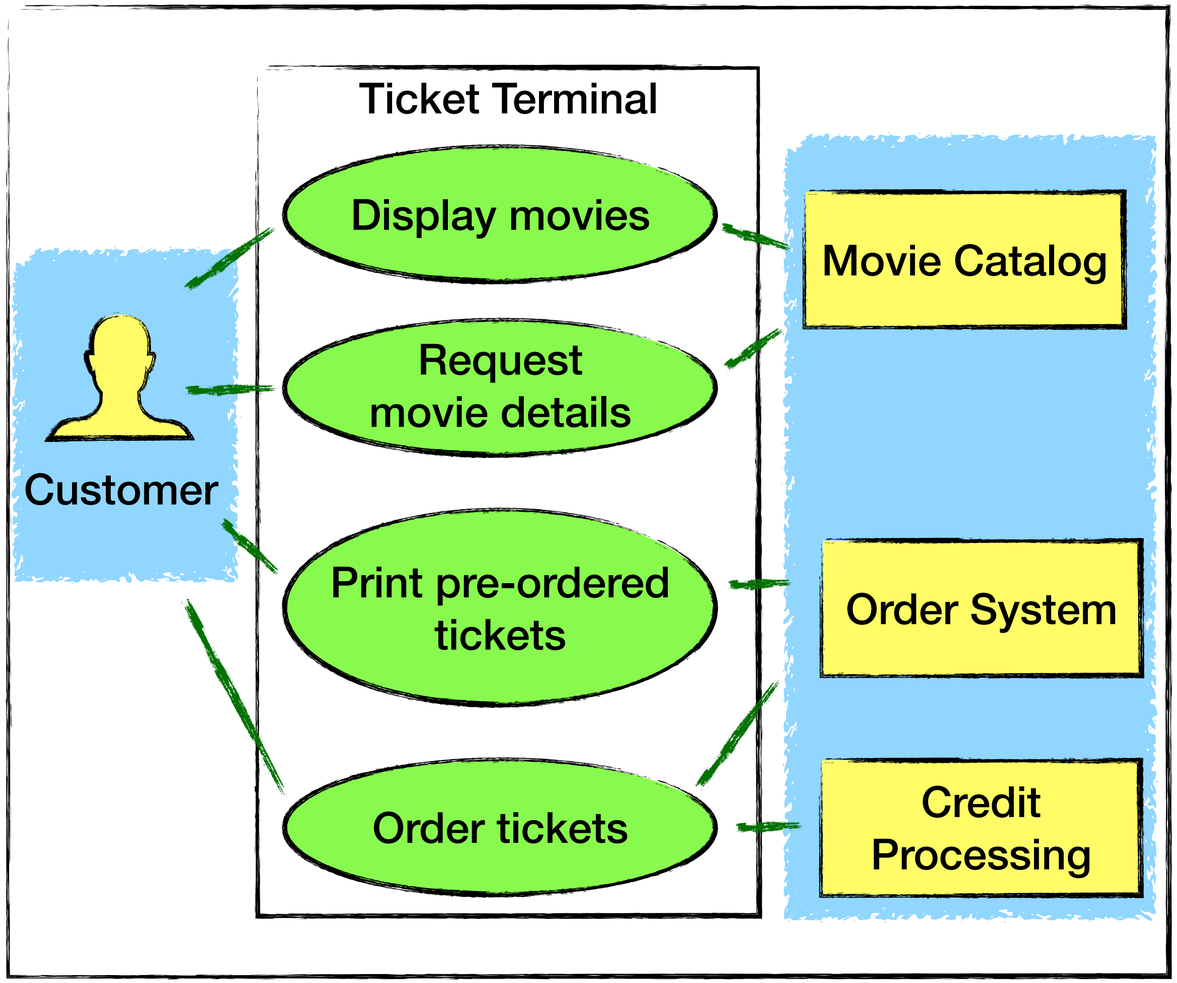
Architecture is About Usage



Clean Architecture

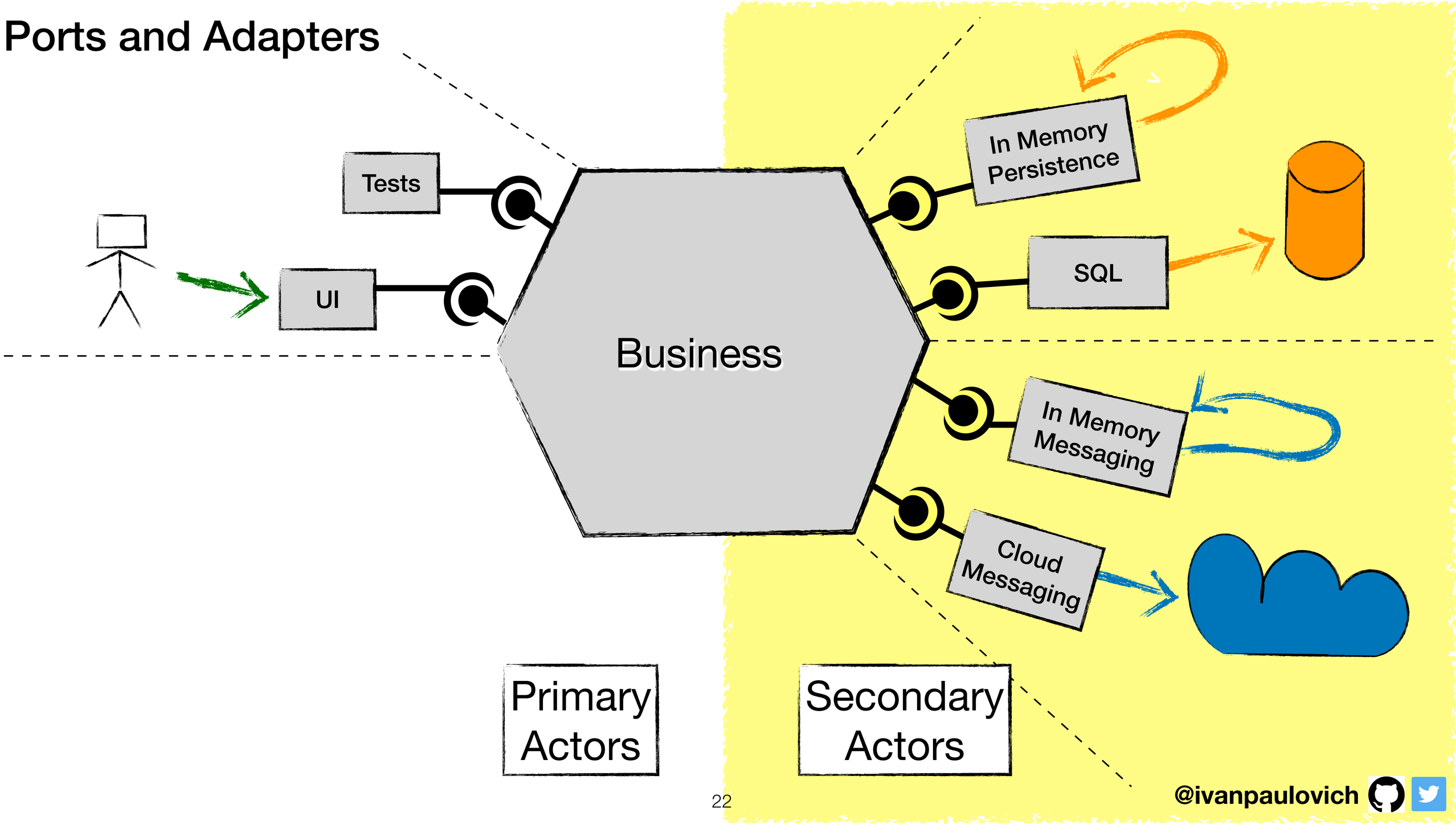
- **Use Cases** as central organising structure.
- Follows the **Ports and Adapters** pattern (Hexagonal Architecture).
  - The implementation is guided by tests.
  - It is decoupled from technology details.
- Follows lots of Principles (Stability, Abstractness, Dependencies, SOLID).
- Pluggable User Interface.

Use Cases

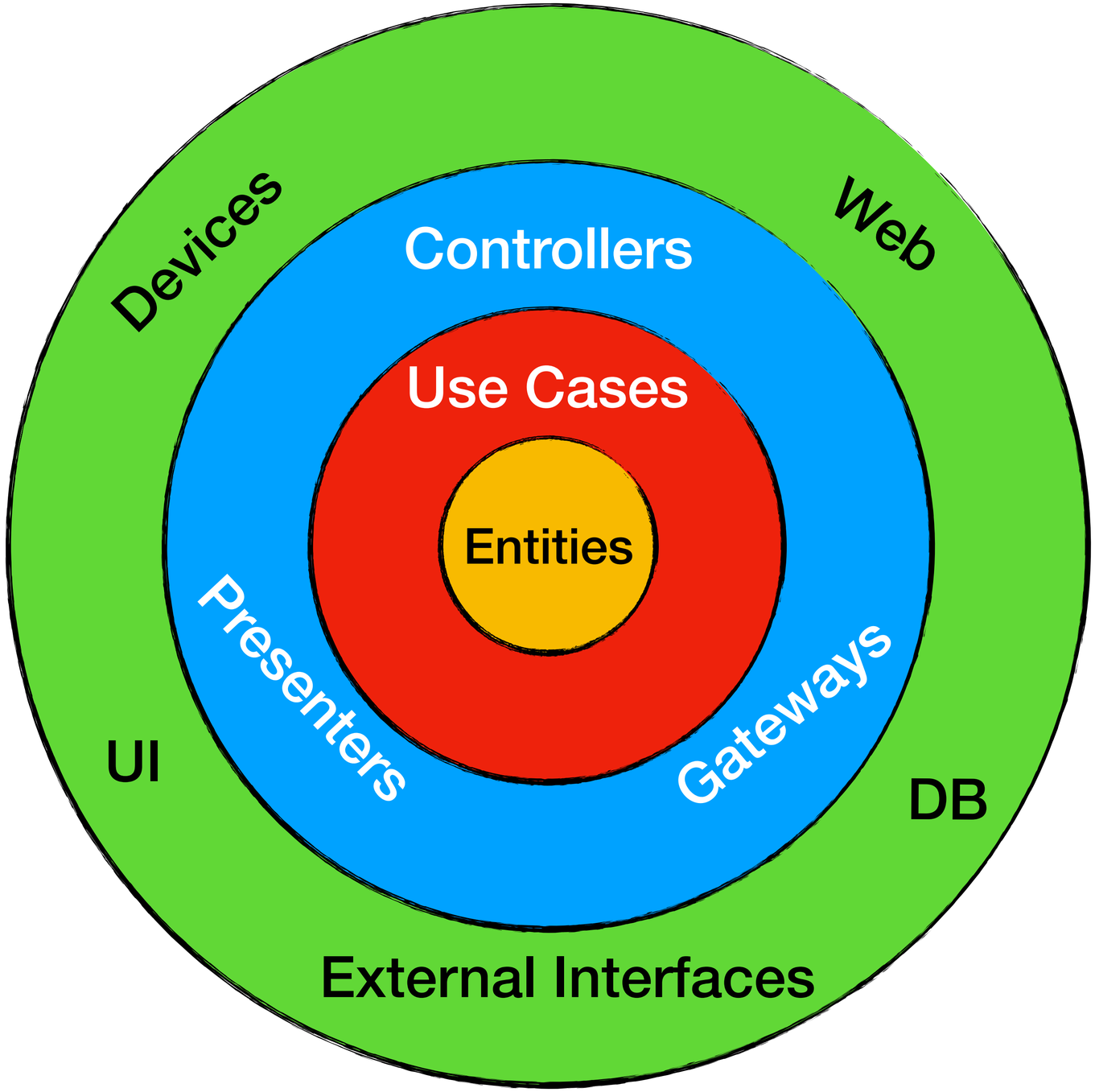


- Use Cases are **delivery independent.**
- Show the **intent** of a system.
- Use Cases are **algorithms** that interpret the input to generate the output data.
- Primary and secondary actors.

Ports and Adapters

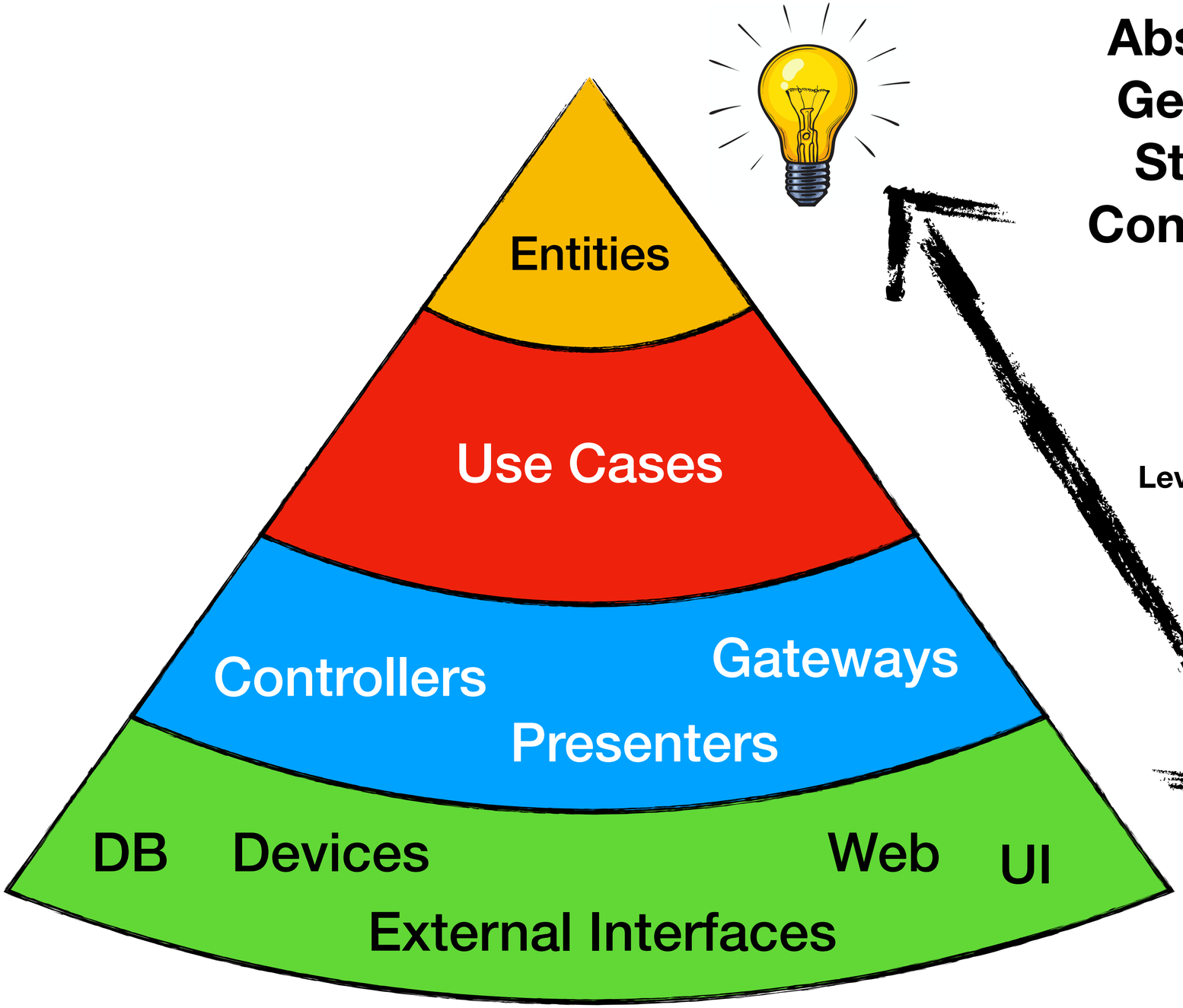


Clean Architecture



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

Clean Architecture



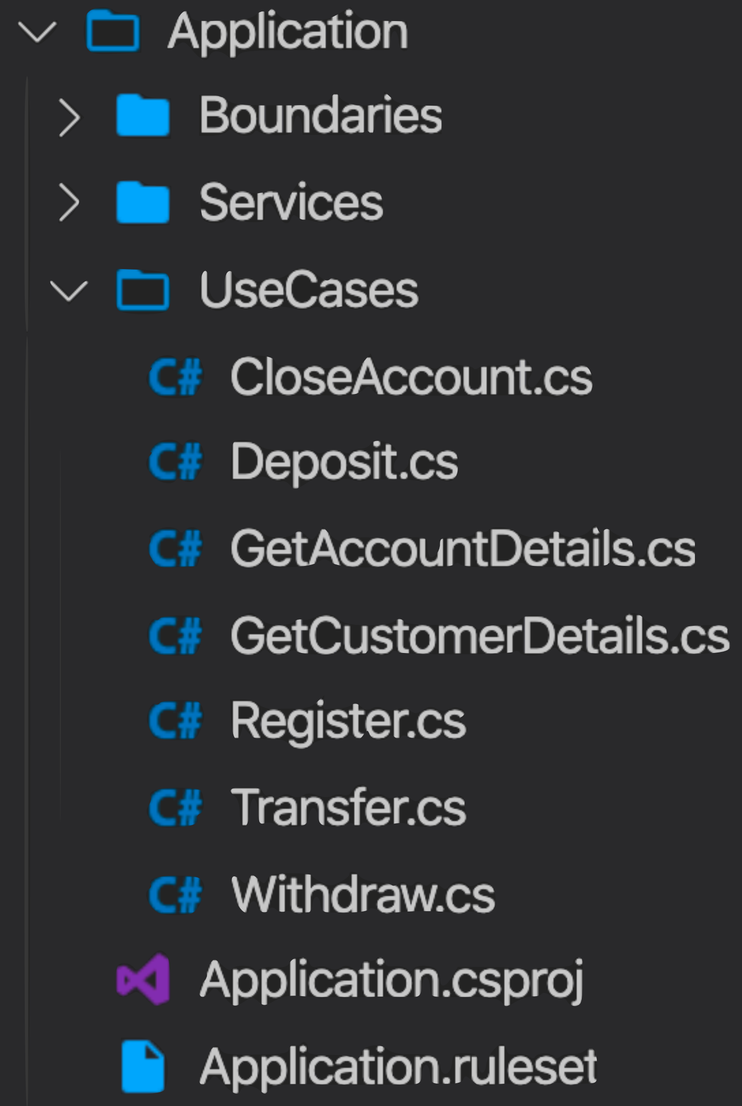
Abstract,
General,
Stable,
Consistent

Level

Concrete,
Specific,
Unstable,
Inconsistent

@ivanpaulovich

Use Case




```cs
public sealed class Register : IUseCase
{
    public async Task Execute(RegisterInput input)
    {
        if (this._userService.GetCustomerId() is CustomerId customerId)
        {
            if (await this._customerService.IsCustomerRegistered(customerId))
            {
                this._outputPort.CustomerAlreadyRegistered($"Customer already exists.");
                return;
            }
        }

        var customer = await this._customerService
            .CreateCustomer(input.SSN, this._userService.GetUserName());
        var account = await this._accountService
            .OpenCheckingAccount(customer.Id, input.InitialAmount);
        var user = await this._securityService
            .CreateUserCredentials(customer.Id, this._userService.GetExternalUserId());

        customer.Register(account.Id);

        await this._unitOfWork.Save();

        this.BuildOutput(this._userService
            .GetExternalUserId(), customer, account);
    }

    private void BuildOutput(
        ExternalUserId externalUserId,
        ICustomer customer,
        IAccount account)
    {
        var output = new RegisterOutput(
            externalUserId,
            customer,
            account);
        this._outputPort.Standard(output);
    }
```

@ivanpaulovich

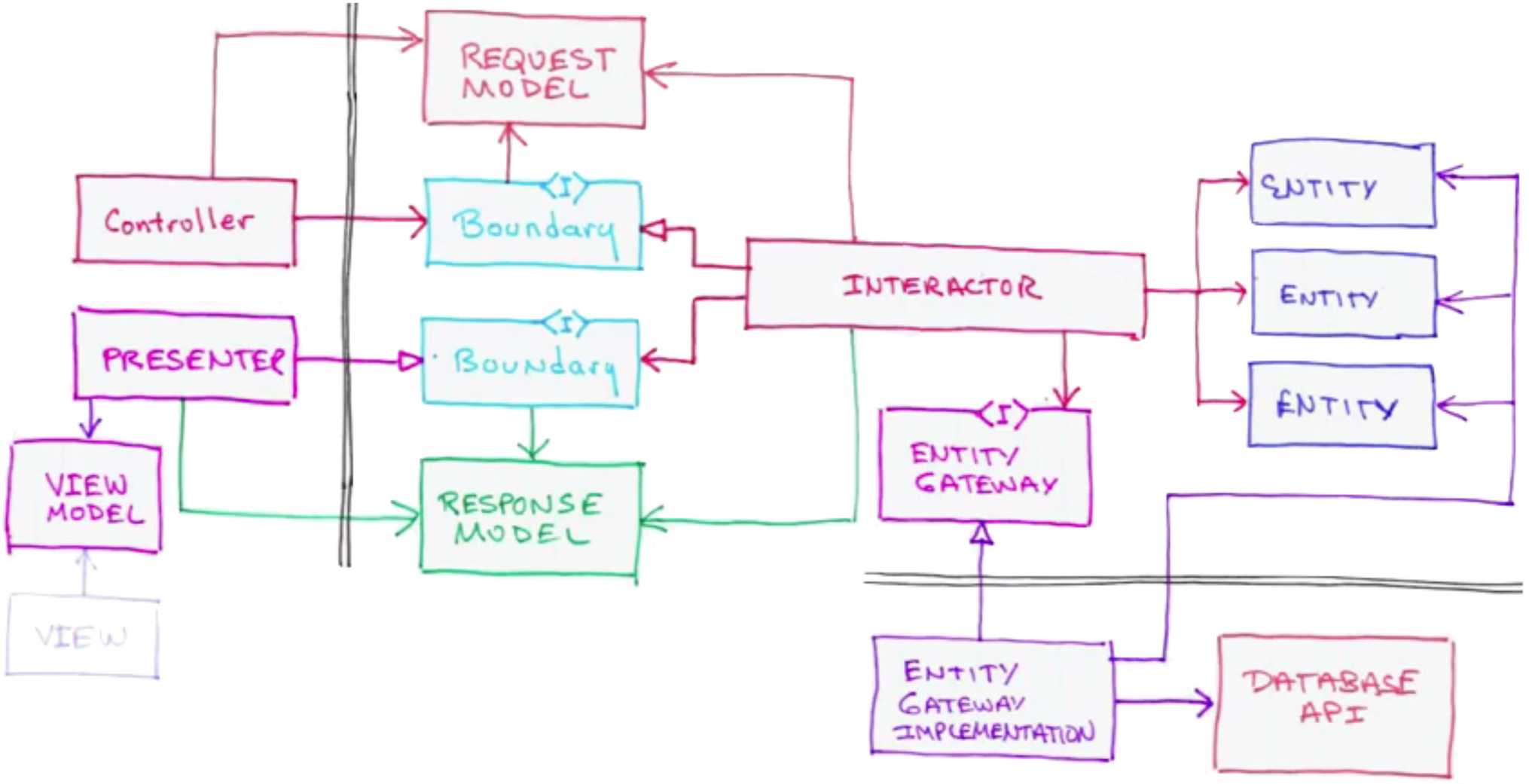
Use Case Input and Output Messages

1. Immutable.
2. Consistent-ish

```
public sealed class DepositInput : IUseCaseInput
{
    public DepositInput(
        AccountId accountId,
        PositiveMoney amount)
    {
        this.AccountId = accountId;
        this.Amount = amount;
    }

    public AccountId AccountId { get; }

    public PositiveMoney Amount { get; }
}
```

```
public sealed class DepositOutput : IUseCaseOutput
{
    public DepositOutput(
        ICredit credit,
        Money updatedBalance)
    {
        Credit creditEntity = (Credit)credit;

        this.Transaction = new Transaction(
            creditEntity.Description,
            creditEntity.Amount,
            creditEntity.TransactionDate);

        this.UpdatedBalance = updatedBalance;
    }

    public Transaction Transaction { get; }

    public Money UpdatedBalance { get; }
}
```

Architecture UML - Uncle Bob



Microservices?

Are you moving
from **Monolith** into **Microservices**?

Try moving
from **Monolith** into **Modular-Monolith** instead.

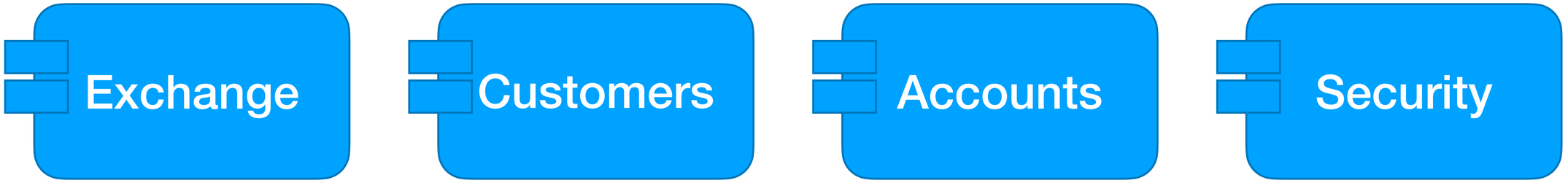
Microservices?

- Low coupling between aggregates.
- Adapters for dependencies.
- Defer extracting an Aggregate.

Bounded Context

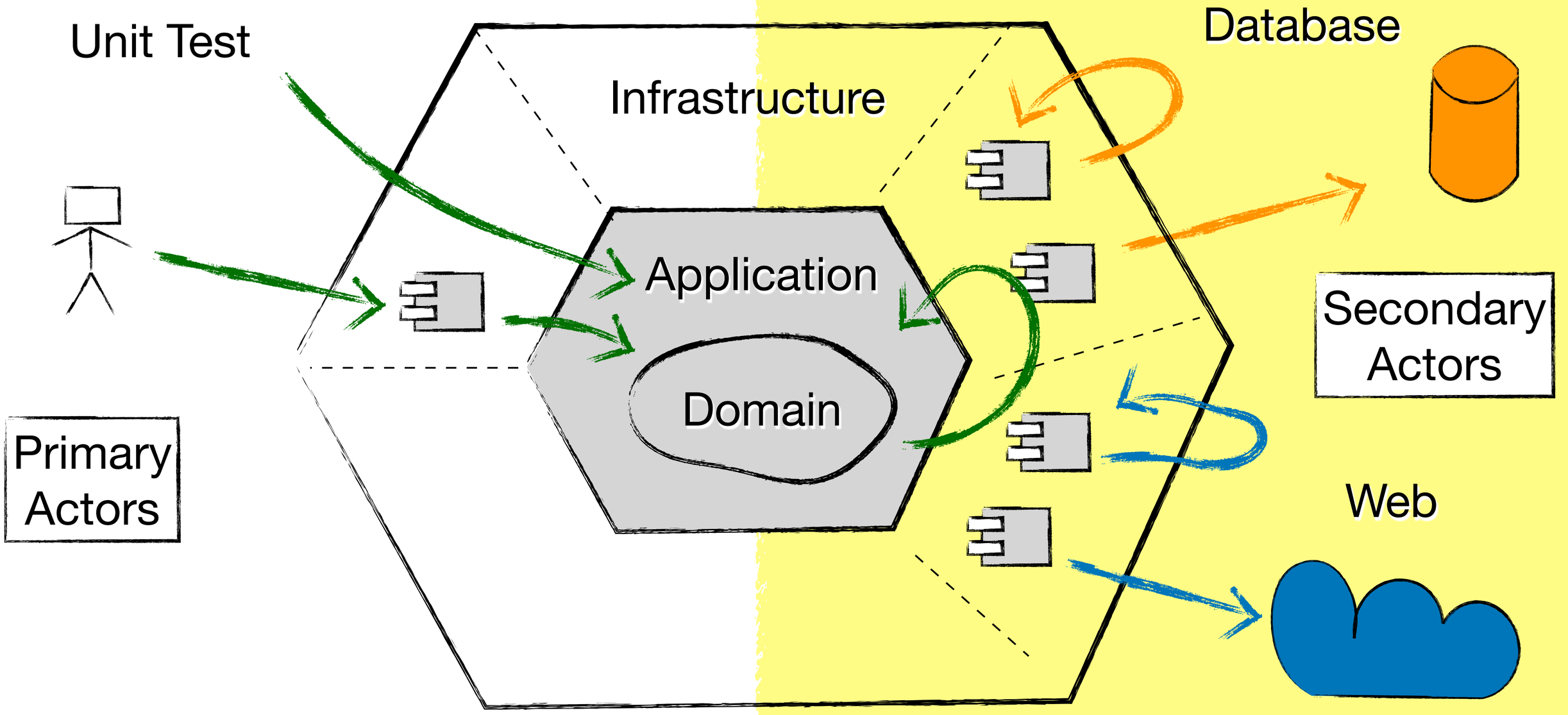
- Another way of saying a **Module**, **Logical Organisation** or **Component**.
- Developed and tested independently.
- Deployment separately is an option/decision.

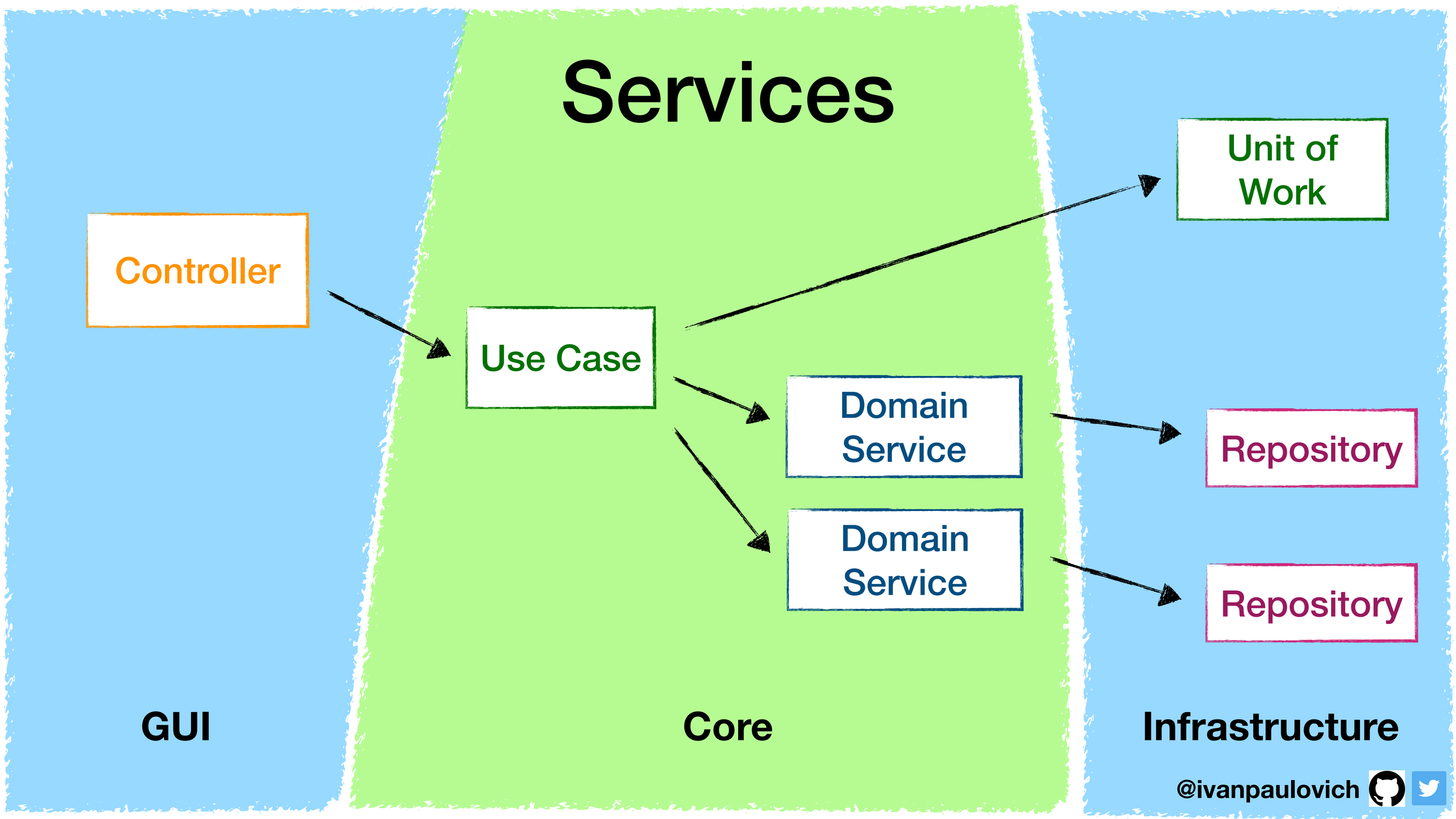
Bounded Context



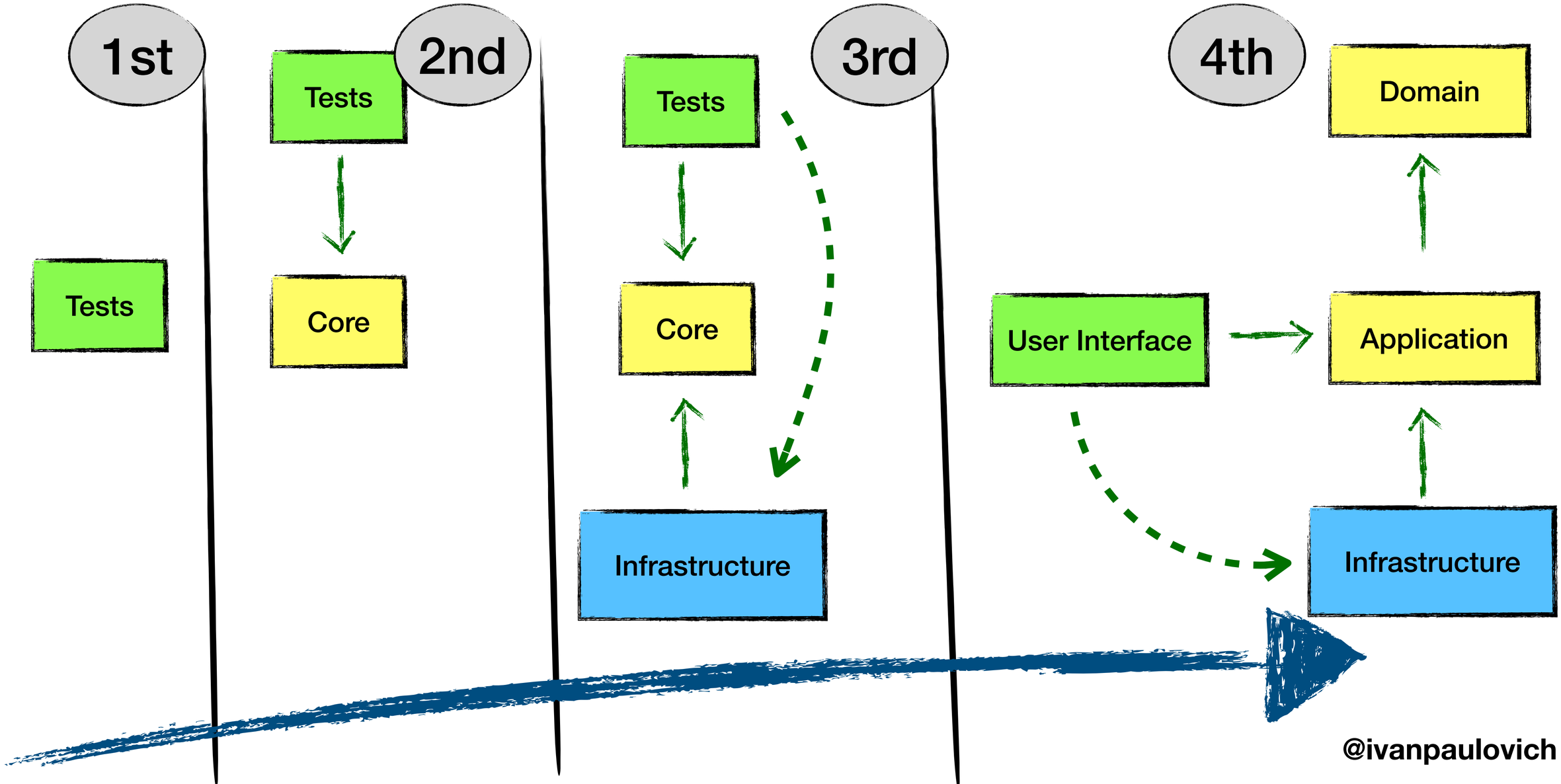
- **Context Map** sets the relationship between contexts.
  - Conformist.
  - Shared Kernel.
  - Customer / Supplier.
  - Anti-Corruption Layer.

Ports and Adapters




Services
Controller
Use Case
Unit of Work
Domain Service
Domain Service
Repository
Repository
GUI
Core
Infrastructure
@ivanpaulovich

Splitting packages on the software lifetime



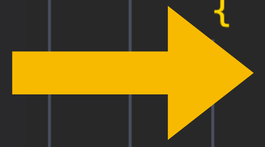
Walkthrough an Use Case

```
[ApiVersion("1.0")]
[Route("api/v{version:apiVersion}/[controller]")]
[ApiController]
public sealed class AccountsController : ControllerBase
{
    private readonly IMediator _mediator;
    private readonly DepositPresenter _presenter;

    public AccountsController(
        IMediator mediator,
        DepositPresenter presenter)
    {
        _mediator = mediator;
        _presenter = presenter;
    }

    [HttpPatch("Deposit")]
    [ProducesResponseType(StatusCodes.Status200OK, Type = typeof(DepositResponse))]
    [ProducesResponseType(StatusCodes.Status400BadRequest)]
    [ProducesResponseType(StatusCodes.Status500InternalServerError)]
    public async Task<IActionResult> Deposit([FromForm][Required] DepositRequest request)
    {
        var input = new DepositInput(
            new AccountId(request.AccountId),
            new PositiveMoney(request.Amount));

        await _mediator.PublishAsync(input);
        return _presenter.ViewModel;
    }
}
```

Controller

Walkthrough an Use Case

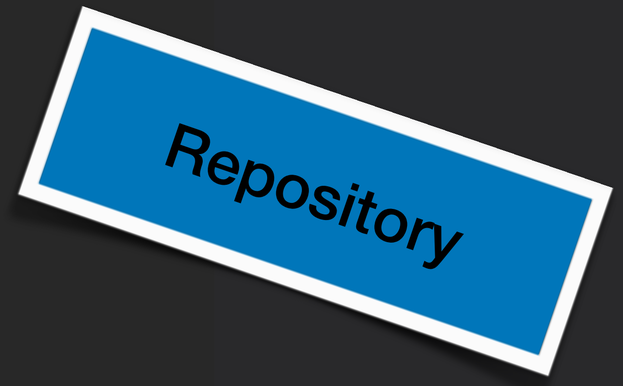
```
public sealed class Deposit : IUseCase
{
    public async Task Execute(DepositInput input)
    {
        try
        {
            var account = await this._accountRepository.Get(input.AccountId);
            var credit = await this._accountService.Deposit(account, input.Amount);
            await this._unitOfWork.Save();

            this.BuildOutput(credit, account);
        }
        catch (AccountNotFoundException ex)
        {
            this._outputPort.NotFound(ex.Message);
            return;
        }
    }

    private void BuildOutput(ICredit credit, IAccount account)
    {
        var output = new DepositOutput(
            credit,
            account.GetCurrentBalance());

        this._outputPort.Standard(output);
    }
```



@ivanpaulovich

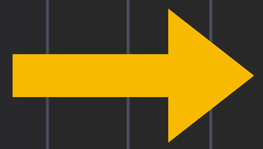
Walkthrough an Use Case

```
public sealed class AccountRepository : IAccountRepository
{
    public async Task<IAccount> Get(AccountId id)
    {
        var account = await _context
            .Accounts
            .Where(a => a.Id.Equals(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.Equals(id))
            .ToList();

        var debits = _context.Debits
            .Where(e => e.AccountId.Equals(id))
            .ToList();

        account.Load(credits, debits);

        return account;
    }
```



Walkthrough an Use Case

```
public sealed class Deposit : IUseCase
{
    public async Task Execute(DepositInput input)
    {
        try
        {
            var account = await this._accountRepository.Get(input.AccountId);
            var credit = await this._accountService.Deposit(account, input.Amount);
            await this._unitOfWork.Save();

            this.BuildOutput(credit, account);
        }
        catch (AccountNotFoundException ex)
        {
            this._outputPort.NotFound(ex.Message);
            return;
        }
    }

    private void BuildOutput(ICredit credit, IAccount account)
    {
        var output = new DepositOutput(
            credit,
            account.GetCurrentBalance());

        this._outputPort.Standard(output);
    }
```



Walkthrough an Use Case

Domain Service

```
public class AccountService
{
    public async Task<ICredit> Deposit(IAccount account, PositiveMoney amount)
    {
        var credit = account.Deposit(this._accountFactory, amount);
        await this._accountRepository.Update(account, credit);

        return credit;
    }
```

Walkthrough an Use Case

```
public sealed class AccountRepository : IAccountRepository
{
    public async Task<IAccount> Get(AccountId id)
    {
        var account = await _context
            .Accounts
            .Where(a => a.Id.Equals(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.Equals(id))
            .ToList();

        var debits = _context.Debits
            .Where(e => e.AccountId.Equals(id))
            .ToList();

        account.Load(credits, debits);

        return account;
    }

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

Repository

@ivanpaulovich

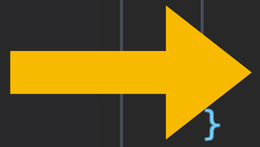
Walkthrough an Use Case

```
public sealed class UnitOfWork : IUnitOfWork, IDisposable
{
    private readonly MangaContext _context;
    private bool _disposed = false;

    public UnitOfWork(MangaContext context)
    {
        _context = context;
    }

    public async Task<int> Save()
    {
        int affectedRows = await _context.SaveChangesAsync();
        return affectedRows;
    }
```

Domain Service

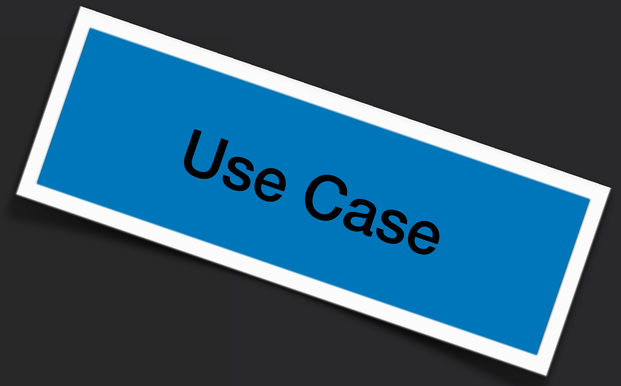
Walkthrough an Use Case

```
public sealed class Deposit : IUseCase
{
    public async Task Execute(DepositInput input)
    {
        try
        {
            var account = await this._accountRepository.Get(input.AccountId);
            var credit = await this._accountService.Deposit(account, input.Amount);
            await this._unitOfWork.Save();

            this.BuildOutput(credit, account);
        }
        catch (AccountNotFoundException ex)
        {
            this._outputPort.NotFound(ex.Message);
            return;
        }
    }

    private void BuildOutput(ICredit credit, IAccount account)
    {
        var output = new DepositOutput(
            credit,
            account.GetCurrentBalance());

        this._outputPort.Standard(output);
    }
```



Walkthrough an Use Case

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

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

    public void Standard(DepositOutput depositOutput)
    {
        var depositResponse = new DepositResponse(
            depositOutput.Transaction.Amount.ToMoney().ToDecimal(),
            depositOutput.Transaction.Description,
            depositOutput.Transaction.TransactionDate,
            depositOutput.UpdatedBalance.ToDecimal());
        ViewModel = new ObjectResult(depositResponse);
    }
}
```



Walkthrough an Use Case

```
[ApiVersion("1.0")]
[Route("api/v{version:apiVersion}/[controller]")]
[ApiController]
public sealed class AccountsController : ControllerBase
{
    private readonly IMediator _mediator;
    private readonly DepositPresenter _presenter;

    public AccountsController(
        IMediator mediator,
        DepositPresenter presenter)
    {
        _mediator = mediator;
        _presenter = presenter;
    }

    [HttpPatch("Deposit")]
    [ProducesResponseType(StatusCodes.Status200OK, Type = typeof(DepositResponse))]
    [ProducesResponseType(StatusCodes.Status400BadRequest)]
    [ProducesResponseType(StatusCodes.Status500InternalServerError)]
    public async Task<IActionResult> Deposit([FromForm][Required] DepositRequest request)
    {
        var input = new DepositInput(
            new AccountId(request.AccountId),
            new PositiveMoney(request.Amount));

        await _mediator.PublishAsync(input);
        return _presenter.ViewModel;
    }
}
```

Controller

@ivanpaulovich

Wrapping up Tactical Design

- It enriches the Domain Model.
- It applies Encapsulation, hides details.
- Focus on usage.
- Unambiguity.
- High consistency inside the domain.

Wrapping up Clean Architecture

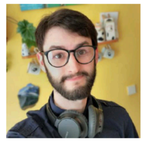
- Clean Architecture is about usage and the use cases are the central organizing principle.
- Use cases implementation are guided by tests.
- The User Interface and Persistence are designed to fulfil the core needs (not the opposite!).
- Defer decisions by implementing the simplest component first.

References

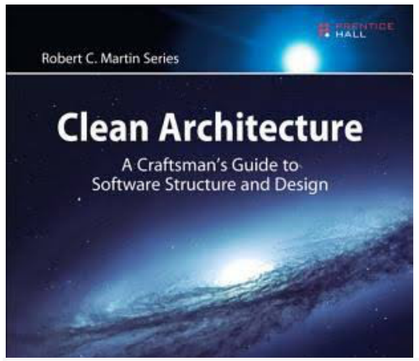
https://github.com/ivanpaulovich/clean-architecture-manga

Contributors ✨

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 🎨 💻
 Leandro Fagundes 💬	 Bart van Ommen 🤔 💻	 qpippop 🤔	 Cesar Pereira 💻	

https://cleancoders.com



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

@ivanpaulovich

always coding 🍄

Ivan Paulovich

ivanpaulovich

Sponsors dashboard

Edit profile

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

Nordax Bank

Stockholm, Sweden

ivan@paulovich.net

https://paulovich.net

Overview | Repositories 60 | Projects 0 | Packages 0 | Stars 287 | Followers 413 | Following 265

Fork me on GitHub

Pinned

Customize your pins

FluentMediator

FluentMediator is an unobtrusive library that allows developers to build custom pipelines for Commands, Queries and Events.

C# ★ 50 8

clean-architecture-manga

Clean Architecture with .NET Core 3.1 and C# 8. Use cases as central organizing structure, completely testable, decoupled from frameworks

C# ★ 1k 212

todo

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

C# ★ 84 12

dotnet-new-caju Template

Learn Clean Architecture with .NET Core 3.0 🔥

C# ★ 181 25

event-sourcing-jambo

An Hexagonal Architecture with DDD + Aggregates + Event Sourcing using .NET Core, Kafka e MongoDB (Blog Engine)

C# ★ 137 60

clean-architecture-webapi

❤️ The simplest Clean Architectu ...t a Web Api using .NET Core and Entity Fram...

C# ★ 72 16

Ask me two questions!

CITERUS

@ivanpaulovich