Projeto inspirado no repositório: <https://github.com/ivanpaulovich/clean-architecture-manga>  
  
**1 - Use Cases**

Os Casos de Uso são algoritmos que interpretam a entrada para gerar os dados de saída; sua implementação deve estar o mais próxima possível do vocabulário do negócio.

Quando falamos sobre um caso de uso, não importa se é uma aplicação web ou móvel ou desktop, os casos de uso são independentes da entrega. O mais importante sobre os casos de uso é como eles interagem com os atores.

Atores primários iniciam um caso de uso. Eles podem ser o Usuário Final, outro sistema etc. Atores secundários são afetados pelos casos de uso.

A arquitetura da aplicação está centrada no uso; uma boa arquitetura deve comunicar claramente os casos de uso do negócio ao desenvolvedor, enquanto as preocupações relacionadas ao framework se tornam detalhes secundários de implementação. No exemplo da Mottu, o Entregador pode registrar uma conta e, em seguida, realizar diversas ações, como alugar uma moto, atualizar a foto da CNH, devolver a moto e aceitar e entregar pedidos.

|  |  |
| --- | --- |
| Use Case (Exemplo) | Descrição |
| Registrar | Um entregador pode registrar uma conta usando seus dados pessoais. |
| Atualizar CNH | Um entregador pode atualizar a foto da sua CNH. |
| Alugar | Um entregador pode alugar uma moto. |
| Devolver | Um entregador pode devolver uma moto |
| Obter Informação | Um entregador pode acessar suas informações. |

**2 - Flow of Control**

**Fluxo de Controle de Registro do entregador**

Uma solicitação é recebida pelo PostDriverController, que invoca a ação Post. Essa ação cria uma mensagem e, em seguida, chama o caso de uso PostDriverUseCase.

O caso de uso PostDriverUseCase gera as entidades User e Driver. Repositórios são acionados, a mensagem de saída é construída e enviada ao Apresentador. O Apresentador, por sua vez, elabora a mensagem de resposta HTTP. Por fim, o PostDriverController solicita ao apresentador a mensagem de resposta atual.

**Fluxo de Controle para Obter Detalhes do Entregador**

Uma solicitação é recebida pelo GetDriverByIdController, que invoca a ação Get. Essa ação cria uma mensagem e, em seguida, chama o caso de uso GetDriverByIdUseCase.

O caso de uso GetDriverByIdUseCase consulta os repositórios para obter informações sobre o Driver e o User. Dependendo da disponibilidade dos dados, ele pode acionar a porta de saída NotFound ou a porta de saída padrão.

O Apresentador, por sua vez, elabora a mensagem de resposta HTTP. Por fim, o GetDriverByIdController solicita ao apresentador a resposta atual.

**3 - Architecture Styles**

Utilizamos conceitos de estilos arquitetônicos populares, sendo os Ports and Adapters o mais simples entre eles. Esses estilos se complementam e têm como objetivo desenvolver um software baseado em casos de uso que esteja desacoplado dos detalhes de implementação tecnológica.

**3.1 - Hexagonal Architecture Style**

A ideia central do estilo de arquitetura hexagonal é que as dependências (Adapters) necessárias para o funcionamento do software são acessadas por meio de uma interface (Port).

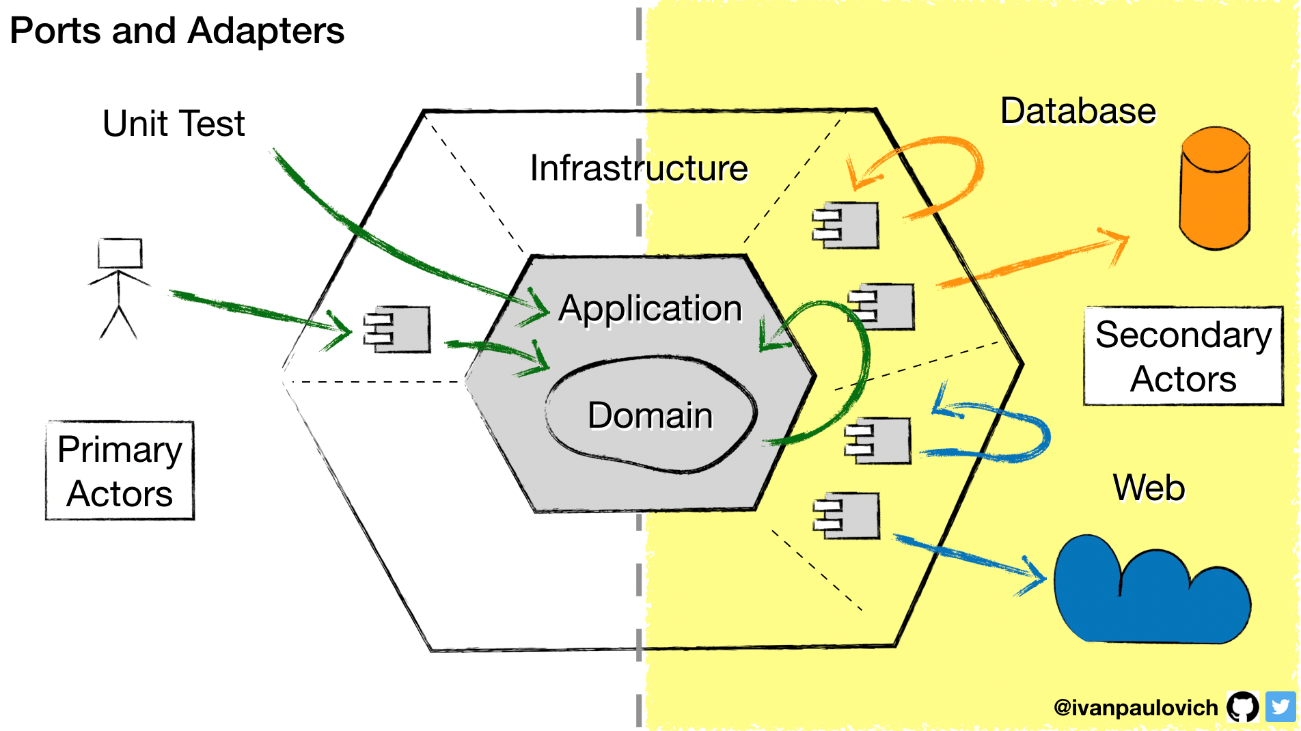
O software é dividido em Aplicação e Infraestrutura. Os adaptadores são componentes intercambiáveis, desenvolvidos e testados de forma isolada. A Aplicação é fracamente acoplada aos Adapters e seus detalhes de implementação.

**3.1.1 - Ports**

Interfaces como IDriverRepository, IOutputPort e IUnitOfWork são consideradas ports essenciais para a aplicação.

**3.1.2 - Adapters**

As implementações dessas interfaces são específicas a uma tecnologia e proporcionam capacidades externas. Por exemplo, o DriverRepository, localizado na pasta Infrastructure/DataAccess/Repositories, oferece recursos para interagir com um banco de dados.



Fonte: <https://github.com/ivanpaulovich/clean-architecture-manga>

**3.1.3 - The Left Side**

Os Atores Primários costumam ser a interface do usuário ou a suíte de testes.

**3.1.4 - The Right Side**

Os Atores Secundários, por sua vez, incluem bancos de dados, serviços em nuvem e outros sistemas.

**3.2 - Onion Architecture Style**

Semelhante ao padrão Ports and Adapters, neste estilo, os objetos de dados transitam entre limites como estruturas de dados simples. Por exemplo, quando o controlador executa um caso de uso, ele passa uma mensagem de entrada imutável. Quando o caso de uso invoca um apresentador, ele fornece uma mensagem de saída (também conhecidas como objetos de transferência de dados).

**3.3 - Clean Architecture Style**

O estilo de Arquitetura Limpa enfoca uma implementação fracamente acoplada de casos de uso, resumindo-se nos seguintes pontos:

**3.3.1 - É uma abordagem em que os Casos de Uso formam a estrutura central organizadora.**

**3.3.2 -** Segue o padrão Ports and Adapters.

**3.3.3 -** A implementação é orientada por testes (TDD Outside-In).

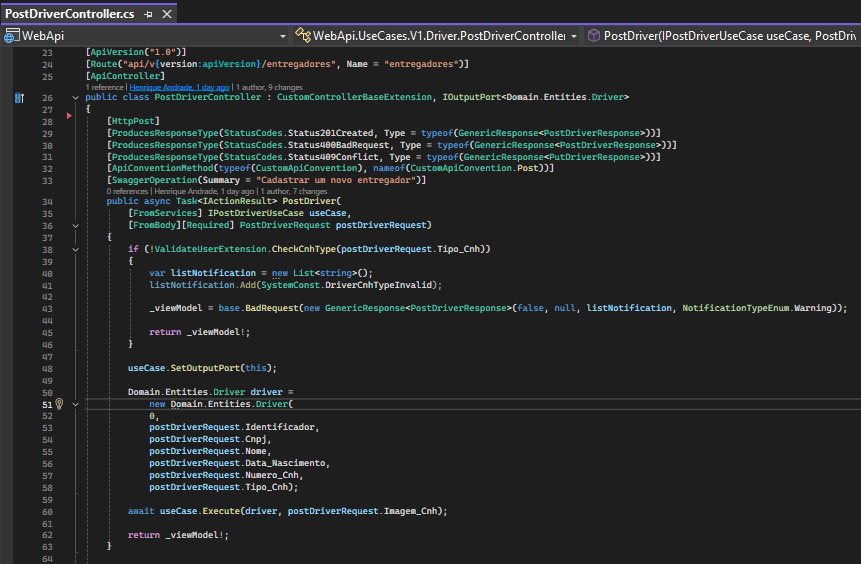
**3.3.4 -** Está desacoplada dos detalhes tecnológicos.

3.3.5 - Adota diversos princípios, como o Princípio das Abstrações Estáveis, o Princípio das Dependências Estáveis, SOLID, entre outros.

**4 - Design Patterns**

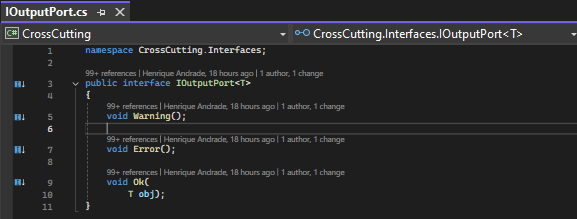
**4.1 - Controller**

Os controladores recebem as solicitações, constroem a mensagem de entrada e, em seguida, chamam o caso de uso. É importante notar que o controlador não constrói a resposta; em vez disso, essa responsabilidade é delegada ao objeto apresentador.



**4.2 - Presenter**

Para esta aplicação utilizamos um presenter genérico no qual podemos compartilhar com todas as respostas da aplicação.



É importante entender que, do ponto de vista da Aplicação, os casos de uso veem um OutputPort com métodos personalizados, dependendo da mensagem, enquanto, do ponto de vista da API Web, o Controlador vê apenas a propriedade Dto.

**4.3 - Dtos**

Os Dtos são objetos de transferência de dados que serão renderizados pelo framework, por isso precisamos seguir as diretrizes do framework.

**4.4 - UnitOfWork**

O padrão Unit of Work é uma técnica de design que gerencia as operações de banco de dados em uma única transação, garantindo que as alterações sejam aplicadas de forma atômica. É frequentemente utilizado com ORMs como o Entity Framework.

**4.4.1 - Gerenciamento de Transações:**   
Agrupa várias operações em uma única transação, garantindo integridade dos dados.

**4.4.2 - Rastreamento de Mudanças**:   
Monitora alterações em objetos durante uma transação.

**4.4.3 - Redução de Conexões**:   
Minimiza o número de conexões com o banco de dados.

**4.4.4 - Separação de Preocupações**:  
Melhora a manutenção ao separar lógica de negócios e de acesso a dados.

**4.4.5 - Facilidade de Testes**:   
Facilita a criação de testes unitários, permitindo simular a implementação.

**5 - Domain Driven Design Patterns**Domain Driven Design (DDD) é uma abordagem que enfatiza a modelagem do domínio de uma aplicação, permitindo que a lógica de negócios seja central para o design e a arquitetura do software. Abaixo, descrevo como Entity, Repository, Use Cases e Services se relacionam com os padrões de DDD.  
 **5.1 - Entity:**

Representa conceitos do domínio com identidade única e ciclo de vida, encapsulando dados e lógica.

**5.2 - Repository:**

Abstrai o acesso a entidades, organizando a interação com a fonte de dados.

**5.3 - Use Cases:**

Orquestram interações do sistema, traduzindo necessidades do negócio em ações concretas, muitas vezes implementados como serviços de domínio.

**5.4 - Services:**

Encapsulam lógica de negócios reutilizável, facilitando a colaboração entre entidades e repositórios.

**6 - Separation of Concerns**

**6.1 - Domain**

O pacote que contém os Módulos de Alto Nível que descrevem o Domínio por meio de Raízes Agregadas, Entidades e Objetos de Valor. Por design, este projeto é Altamente Abstrato e Estável; em outras palavras, contém uma quantidade considerável de interfaces e não deve depender de bibliotecas e frameworks externos. Idealmente, deve ser fracamente acoplado, até mesmo ao .NET Framework.

**6.2 - Application**

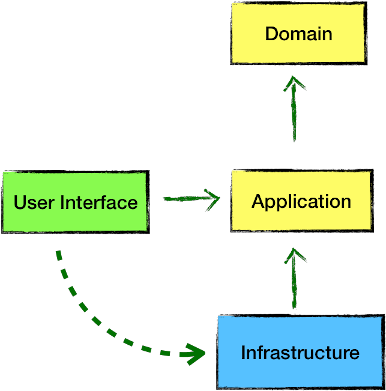
Um projeto que contém os Casos de Uso da Aplicação, os quais orquestram as regras de negócio de alto nível. Por design, a orquestração dependerá de abstrações de serviços externos (por exemplo, Repositórios). O pacote expõe Interfaces de Limite (em outras palavras, Contratos ou Ports) que são utilizadas pela interface do usuário.

**6.3 - Infrastructure**

A camada de infraestrutura é responsável por implementar os Adapters para os Atores Secundários. Por exemplo, um Banco de Dados PostgreSql é um ator secundário que é afetado pelos casos de uso da aplicação; toda a implementação e as dependências necessárias para consumir o Postgresql são criadas na infraestrutura. Por design, a infraestrutura depende da camada de aplicação.

**6.4 - User Interface**

Responsável por renderizar a Interface Gráfica do Usuário (GUI) para interagir com o Usuário ou outros sistemas. Composta por Controladores que recebem Requisições HTTP e Apresentadores que convertem as saídas da aplicação em Dtos que são renderizadas como Respostas HTTP. No ASP.NET, é o ponto de entrada da aplicação; no nosso caso, o ponto de entrada da aplicação conhece o Container de Injeção de Dependência e todas as suas dependências.



Fonte: <https://github.com/ivanpaulovich/clean-architecture-manga>

**7 - Modelagem**