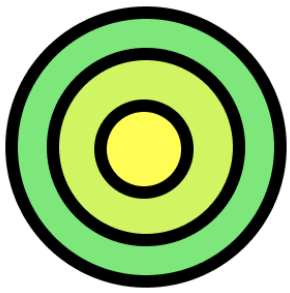
# Comparação de Design Orientado a Domínio e Conceitos de Arquitetura Limpa

Design de software

Data inválida da última atualização

"Domain-Driven Design" de Eric Evans e "Clean Architecture" de Uncle Bob são livros que introduziram abordagens táticas para a construção de aplicativos corporativos complexos. Dado o fato de que implementamos ideias de ambos, neste artigo, tentarei esclarecer qualquer confusão entre tópicos paralelos introduzidos por ambos.

design orientado por domínio arquitetura limpa



Antes de entrar no design e arquitetura de software, meu código estava doendo 🤕.

Passei muito tempo fazendo retrabalho, escrevendo código não testável, tentando inventar minhas próprias abstrações (ruins) e colocando toda a minha lógica de negócios em serviços anêmicos .

Eventualmente, acabei lendo **Clean Architecture** de Uncle Bob e depois **Domain-Driven Design** de Eric Evans.

O Domain-Driven Design, escrito inicialmente em 2003 por Eric Evans, introduziu novas abordagens para projetar software usando uma arquitetura em camadas com um modelo de domínio rico no centro.

Uncle Bob escreveu Clean Architecture em 2017 e resumiu sua pesquisa sobre o que constitui uma arquitetura limpa, *também* usando uma arquitetura em camadas com uma camada de domínio no centro.

Mesmo havendo alguma sobreposição entre os conceitos que ambos os livros introduziram, há um pouco de confusão nas definições das construções.

É isso que eu gostaria de esclarecer.

---

Ao longo do último ano, percebi que no desenvolvimento de software,

> Há uma construção para tudo.

Para *lógica de validação* , temos Value Objects . Para abstrair os desafios de recuperar e persistir dados, temos repositórios . Um *grupo de código geralmente coeso* pode ser chamado de **componente** .

Para praticamente todos os cenários e em todas as camadas da arquitetura em camadas, existe uma construção ou padrão de design que podemos usar para resolver nossos problemas.

A segunda coisa que aprendi é,

> Cada desenvolvedor tem um nome diferente para essas construções

Dependendo de quem você pergunta, um Caso de Uso pode ser conhecido apenas como um **serviço de aplicativo** .

Alguns desenvolvedores veem os serviços de **domínio** e **os serviços de aplicativo** como a mesma coisa.

Irônico para a essência do DDD, é a falta de um entendimento compartilhado em relação às ferramentas que usamos em nosso *domínio* (desenvolvimento de software, isto é) que torna difícil para nós ter conversas sobre design e arquitetura de software.

Como envolvemos regularmente tópicos de DDD e Clean Architecture neste blog (vou me referir à Clean Architecture como *CA* de agora em diante), acho que seria uma boa ideia tentar identificar o mapeamento entre os dois.

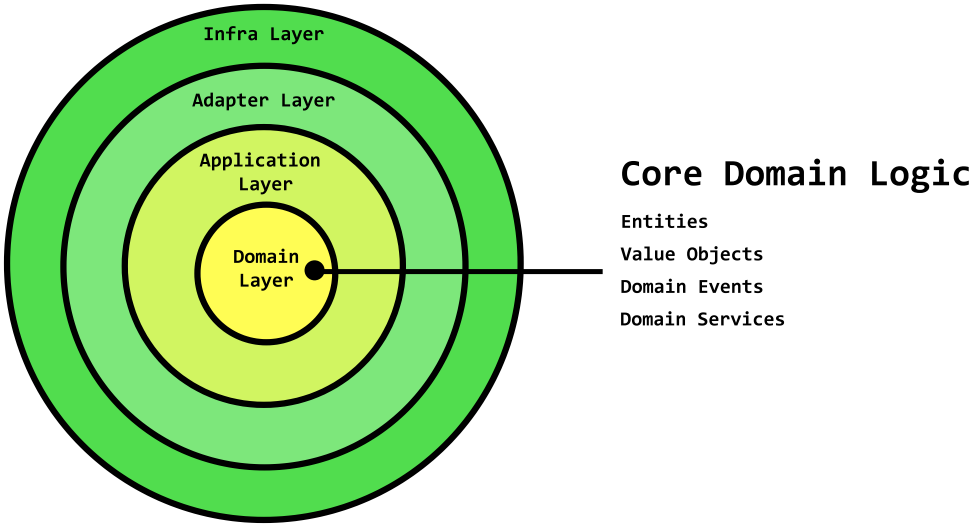
Ao final deste artigo, deverá:

- Entenda as semelhanças e diferenças entre os conceitos introduzidos pelo DDD e CA para implementar uma arquitetura em camadas.

---

Partindo do centro da arquitetura em camadas, temos o conceito de **entidades** .

## Entidades




### Comparação entre livros

| Livro | Conhecido como | Camada |
|---|---|---|
| Design Orientado por Domínio (DDD) | Entidade (e às vezes Agregado) | Domínio |
| Arquitetura Limpa (CA) | Entidade | Domínio |

### Observações

De **Clean Architecture** , o tio Bob disse:

> "Uma Entidade é um objeto dentro do nosso sistema de computador que incorpora um pequeno conjunto de regras críticas de negócios que operam em Dados Críticos de Negócios ."

Eu gosto dessa definição! Os dados de negócios críticos são comparáveis à *lógica de domínio/regras de negócios* no DDD.

Também soa muito como se estivéssemos falando de agregados (um tipo muito específico de entidade) porque no design agregado, nos esforçamos muito para identificar os limites agregados exatos para mantê-

lo pequeno.

> "A interface da Entidade consiste nas funções que implementam as Regras Críticas de Negócios que operam nesses dados"

Isso aqui soa como se estivéssemos falando sobre projetar interfaces reveladoras de intenção .

Isso significa que a API para entidade (os métodos que ela expõe) deve ser apenas operações válidas e que façam sentido para nosso domínio.

Em um `Job` agregado, se a invariante de classe (regra de negócios) disser que os `Job` 's `paymentType` não podem ser alterados depois que as pessoas aplicarem a ela, não deveríamos nem expor um setter para `paymentType` , porque isso introduziria uma área de superfície para as invariantes de classe ser contornado.

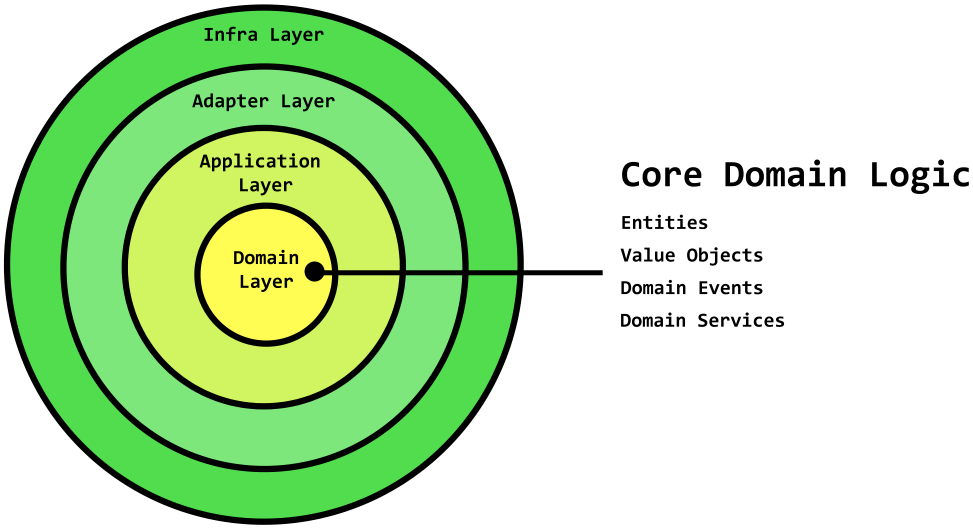
domínio/trabalho.ts

```
export class Job extends AggregateRoot<JobProps> {
  ...
  get paymentType (): PaymentType {
    return this.props.paymentType;
  }

  // No setter exposed

  public updatePaymentType (paymentType: PaymentType): Result<void> {
    if (this.hasApplicants()) {
      return Result.fail<void>(
        `Can't update payment type
        on a job that has applicants`
      )
    } else {
      this.props.paymentType = paymentType;
      return Result.ok<void>();
    }
  }
}
```

Para aprender a usar entidades em seus projetos Node.js/TypeScript para encapsular *dados críticos de negócios* , leia "Entendendo entidades de domínio [com exemplos] - DDD w/ TypeScript" .

Para aprender a projetar agregados, leia "Como projetar e persistir agregados - DDD com TypeScript" .

## Serviços de domínio




| Livro | Conhecido como | Camada |
|---|---|---|
| Design Orientado por Domínio (DDD) | Serviço de domínio | Domínio |
| Arquitetura Limpa (CA) | N / D | Domínio |

Em DDD, um **Domain Service** é um tipo específico de classe de camada de domínio que usamos quando queremos colocar alguma lógica de domínio que dependa de duas ou mais entidades.

Usamos os Serviços de Domínio quando colocar a lógica em uma entidade específica quebraria o encapsulamento e exigiria que a entidade soubesse sobre coisas com as quais realmente não deveria se preocupar.

Por exemplo, em DDDForum.com , quando queremos `upvotePost(member: Member, post: Post, existingVotes: PostVote[])` , a qual classe de entidade esse método deve pertencer?

É a `Member` entidade?

O `Post` ?

O `PostVote` ?

Nenhum, porque não pertence a um, realmente.

Eu provavelmente teria tentado usar, `Post` mas como `Member` não está dentro do `Post` limite agregado, é evidente que precisamos de outra coisa.

É quando usamos Domain Services: quando temos alguma lógica de negócios que envolve várias entidades e colocá-la em uma seleção arbitrária de uma delas quebraria o encapsulamento do modelo de domínio.

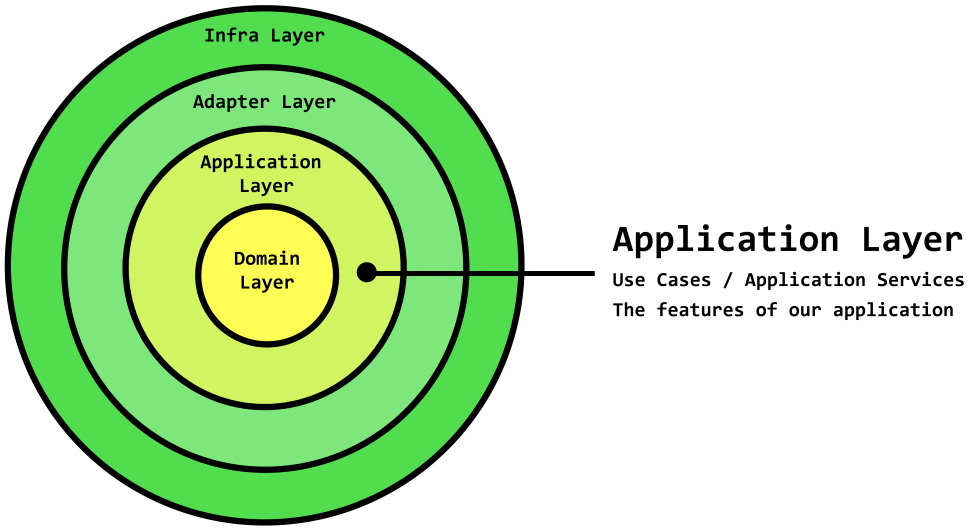
domain/services/postService.ts

```
class PostService {

  ...

  /**
   * Demonstrating the use of a domain service to
   * encapsulate domain logic that involves several
   * entities.
   */

  public upvotePost (
    post: Post,
    member: Member,
    existingVotesOnPostByMember: PostVote[]
  ): UpvotePostResponse {

    const existingUpvote: PostVote = existingVotesOnPostByMember
      .find((v) => v.isUpvote());

    // If already upvoted, do nothing
    const upvoteAlreadyExists = !!existingUpvote;

    if (upvoteAlreadyExists) {
      return right(Result.ok<void>());
    }

    // If downvoted, remove the downvote
    const existingDownvote: PostVote = existingVotesOnPostByMember
    .find((v) => v.isDownvote());

    const downvoteAlreadyExists = !!existingDownvote;
```

```
    if (downvoteAlreadyExists) {
      post.removeVote(existingDownvote);
      return right(Result.ok<void>());
    }

    // Otherwise, add upvote
    const upvoteOrError = PostVote
      .createUpvote(member.memberId, post.postId);

    if (upvoteOrError.isFailure) {
      return left(upvoteOrError);
    }

    const upvote: PostVote = upvoteOrError.getValue();
    post.addVote(upvote);

    return right(Result.ok<void>());
  }
  ...
}
```

Esta é uma ferramenta de modelagem de domínio tático realmente específica, então não estou chocado ao ver que ela não foi mencionada na CA.

## Serviços de aplicativos




| Livro | Conhecido como | Camada |
|---|---|---|

| Livro | Conhecido como | Camada |
|---|---|---|
| Design Orientado por Domínio (DDD) | Serviço de Aplicativo | Inscrição |
| Arquitetura Limpa (CA) | Caso de uso | Inscrição |

Em **Clean Architecture** , o tio Bob descreve os casos de uso como os principais **recursos** do aplicativo.

Estas são todas as coisas que nosso aplicativo pode fazer.

E em um artigo anterior, descobrimos que os casos de uso eram comandos ou consultas .

Casos de uso (um termo de Arquitetura Limpa) são semelhantes aos **Serviços de Aplicativos** em DDD. Pelo menos o seu *posicionamento relativo* é.

No DDD, os **Serviços de Aplicação** (obviamente dizem respeito à camada de aplicação) representam comandos ou consultas (como `createComment` - COMMAND ou `getCommentById` - QUERY) que:

- Não contém lógica de negócios específica do domínio.
- São usados para buscar entidades de domínio (e qualquer outra coisa) da persistência e do mundo exterior.
- Ou passa o controle para um Agregado para executar a lógica de domínio usando um método do Agregado ou passa várias entidades para um Serviço de Domínio para facilitar sua interação.
- Têm baixos níveis de complexidade ciclomática .

Por exemplo, o serviço de aplicativo para o `UpvotePost` comando seria:

1. Obter o membro, postar e postar votos
2. Passe-os para o serviço de domínio
3. Salve a transação se ela foi bem sucedida.

useCases/upvotePost/UpvotePost.ts

```
import { UseCase } from "../../../../../shared/core/UseCase";
import { UpvotePostDTO } from "./UpvotePostDTO";
import { IMemberRepo } from "../../../repos/memberRepo";
import { IPostRepo } from "../../../repos/postRepo";
import { left, right, Result } from "../../../../../shared/core/Result";
import { AppError } from "../../../../../shared/core/AppError";
import { UpvotePostErrors } from "./UpvotePostErrors";
```

```
import { Member } from "../../../domain/member";
import { Post } from "../../../domain/post";
import { IPostVotesRepo } from "../../../repos/postVotesRepo";
import { PostVote } from "../../../domain/postVote";
import { PostService } from "../../../domain/services/postService";
import { UpvotePostResponse } from "./UpvotePostResponse";

export class UpvotePost implements UseCase<UpvotePostDTO, Promise<UpvotePostResponse>> {
  private memberRepo: IMemberRepo;
  private postRepo: IPostRepo;
  private postVotesRepo: IPostVotesRepo;
  private postService: PostService;

  constructor (
    memberRepo: IMemberRepo,
    postRepo: IPostRepo,
    postVotesRepo: IPostVotesRepo,
    postService: PostService
  ) {
    this.memberRepo = memberRepo;
    this.postRepo = postRepo;
    this.postVotesRepo = postVotesRepo
    this.postService = postService;
  }

  public async execute (req: UpvotePostDTO): Promise<UpvotePostResponse> {
    let member: Member;
    let post: Post;
    let existingVotesOnPostByMember: PostVote[];

    try {

      // Get the member
      try {
        member = await this.memberRepo.getMemberByUserId(req.userId);
      } catch (err) {
        return left(new UpvotePostErrors.MemberNotFoundError())
      }

      // Get the post
      try {
        post = await this.postRepo.getPostBySlug(req.slug);
      } catch (err) {
        return left(new UpvotePostErrors.PostNotFoundError(req.slug));
      }

      // Get the existing post votes
      existingVotesOnPostByMember = await this.postVotesRepo
        .getVotesForPostByMemberId(post.postId,  member.memberId);
```

```
      // Allow the domain service to exec
      const upvotePostResult = this.postService
        .upvotePost(post, member, existingVotesOnPostByMember);

      // If failed, return a failure result
      if (upvotePostResult.isLeft()) {
        return left(upvotePostResult.value);
      }

      // Otherwise, commit the transaction
      await this.postRepo.save(post);

      // Success!
      return right(Result.ok<void>())

    } catch (err) {
      return left(new AppError.UnexpectedError(err));
    }
  }
}
```

Isso é Serviços de Aplicativos.

No Uncle Bobland, os Casos de Uso *permitem* a lógica de negócios, mas há uma diferenciação entre o que consiste na lógica de negócios da *camada de aplicativo* e o que constitui *a lógica de negócios do domínio* . Eu mesmo não vi os vídeos do Clean Coder para aprofundar essa diferenciação, então talvez outra pessoa possa fornecer mais esclarecimentos, mas parece bastante semelhante à maneira como a lógica da camada de aplicativo e domínio é organizada no DDD.

---

Acho que esses são os principais blocos de construção e áreas de confusão entre a implementação dos conceitos de Arquitetura Limpa e Design Orientado por Domínio.

Isso só mostra como você pode discutir com alguém sobre qual é a ferramenta correta para o trabalho, antes de identificar o fato de que você estava falando sobre a mesma coisa, semanticamente.

Bom saber!

---

## Discussão