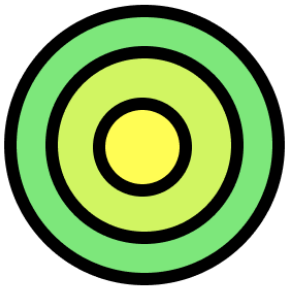
Comparison of Domain-Driven Design and Clean Architecture Concepts

Software Design

Last updated Invalid date

Eric Evans' "Domain-Driven Design" and Uncle Bob's "Clean Architecture" are books that have introduced tactical approaches towards building complex enterprise applications. Given the fact that we implement ideas from both, in this article, I'll aim to clear up any confusion between parallel topics introduced from both.

domain-driven design clean architecture



Before I got into software design and architecture, my code was hurting 🤕.

I spent a lot of time doing rework, writing untestable code, trying to invent my own (bad) abstractions, and putting all my business logic into anemic services.

Eventually, I ended up reading **Clean Architecture** by Uncle Bob and then **Domain-Driven Design** by Eric Evans.

Domain-Driven Design, initially written in 2003 by Eric Evans, introduced new approaches towards designing software by using a layered architecture with a rich domain model in the

center.

Uncle Bob wrote Clean Architecture in 2017 and summarized his research on what constitutes a clean architecture, *also* using a layered architecture with a domain layer in the center.

Even through there's some overlap between the concepts that both of these books introduced, there's a little bit of confusion on the definitions of the constructs.

That's what I'd like to clear up.

Over the past year or so, I've realized that in software development,

> There's a construct for everything.

For *validation logic*, we have Value Objects. For abstracting the challenges of retrieving and persisting data, we have repositories. A *generally cohesive group of code* can be called a **component**.

For just about every scenario, and at every layer of the layered architecture, there exists a construct or design pattern we can use to solve our problems.

The second thing I've learned is,

> Every developer has a different name for these constructs

Depending on who you ask, a Use Case might only be known to someone as an **application service**.

Some developers see **domain services** and **application services** as the same thing.

Ironic to the essence of DDD, it's the lack of a shared understanding towards the tools that we use in our *domain* (software development, that is) that makes it difficult for us to have conversations about software design and architecture.

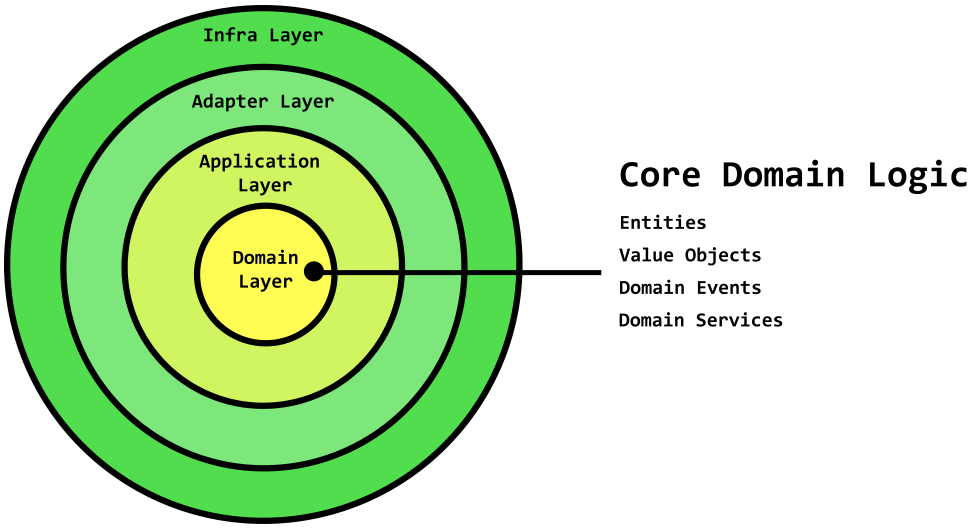
Because we so regularly involve topics from both DDD and Clean Architecture on this blog (I'll refer to Clean Architecture as *CA* from now on), I think it would be a good idea to attempt to identify the mapping between the two.

By the end of this article, should:

- Understand the similarities and differences between concepts introduced by DDD and CA towards implementing a layered architecture.

Starting from the center of the layered architecture, we have the concept of **entities**.

Entities



Comparison between books

Book	Known as	Layer
Domain-Driven Design (DDD)	Entity (and sometimes Aggregate)	Domain
Clean Architecture (CA)	Entity	Domain

Observations

From **Clean Architecture**, Uncle Bob said:

> "An Entity is an object within our computer system that embodies a small set of critical business rules operating on Critical Business Data."

I like this definition! The critical business data is comparable to *domain logic/business rules* in DDD.

It also sounds a lot like we're talking about aggregates (a very specific type of entity) because in aggregate design, we put a lot of effort into identifying the exact aggregate boundaries in order to keep it small.

> "The interface of the Entity consists of the functions that implement the Critical Business Rules that operate on that data"

This right here sounds like we're talking about designing intention revealing interfaces.

That means the API for entity (the methods that it exposes) should only be operations that are valid and make sense to our domain.

In a `Job` aggregate, if the class invariant (business rule) said that the `Job` 's `paymentType` can't be changed after people applied to it, we shouldn't even expose a setter for `paymentType` , because doing that would introduce a surface area for the class invariants to be circumvented.

domain/job.ts

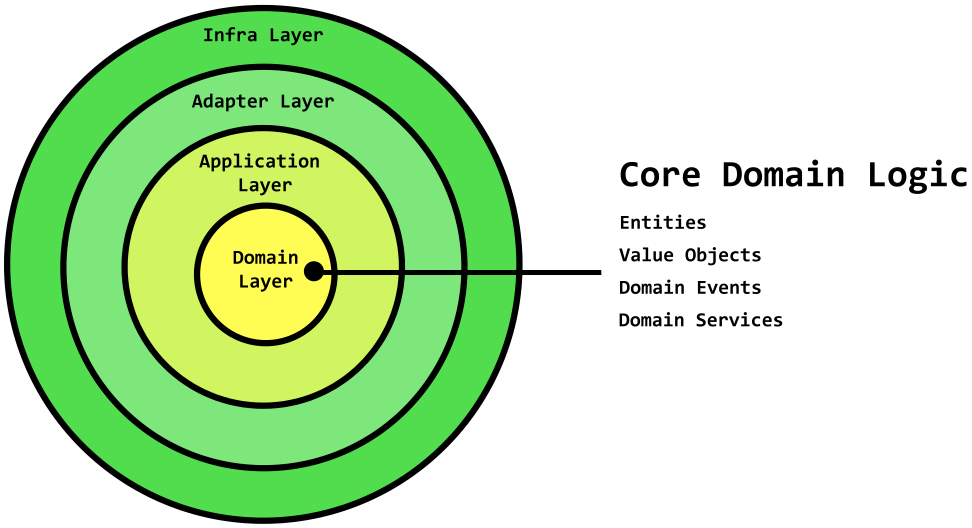
```
export class Job extends AggregateRoot<JobProps> {
  ...
  get paymentType (): PaymentType {
    return this.props.paymentType;
  }

  // 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>();
    }
  }
}
```

To learn how to use entities in your Node.js / TypeScript projects to encapsulate *critical business data*, read "Understanding Domain Entities [with Examples] - DDD w/ TypeScript".

To learn how to design aggregates, read "How to Design and Persist Aggregates - DDD w/ TypeScript".

Domain Services



Book	Known as	Layer
Domain-Driven Design (DDD)	Domain Service	Domain
Clean Architecture (CA)	N/A	Domain

In DDD, a **Domain Service** is a specific type of domain layer class that we use when we want to put some domain logic that relies on two or more entities.

We use Domain Services when putting the logic on a particular entity would break encapsulation and require the entity to know about things it really shouldn't be concerned with.

For example, in DDDForum.com, when we want to `upvotePost(member: Member, post: Post, existingVotes: PostVote[])`, which entity's class should that method belong to?

Is it the `Member` entity?

The `Post`?

The `PostVote`?

None, because it doesn't belong to one, really.

I probably would have tried to use `Post` but since `Member` isn't within the `Post` aggregate boundary, it's apparent that we need something else.

This is when we use Domain Services: when we have some business logic that involves multiple entities and putting it into an arbitrary selection of one of them would break domain model encapsulation.

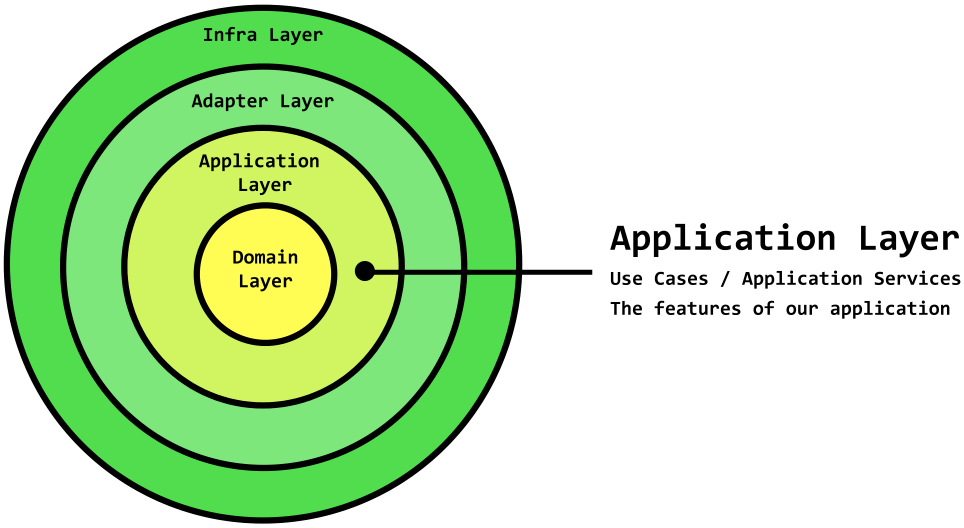
domain/services/postService.ts

```
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>());
  }
  ...
}
```

This is a really specific tactical domain modeling tool, so I'm not shocked to see that it wasn't mentioned in CA.

Application Services



Book	Known as	Layer
Domain-Driven Design (DDD)	Application Service	Application
Clean Architecture (CA)	Use Case	Application

In **Clean Architecture**, Uncle Bob describes use cases as the main **features** of the application.

These are all the things our application can do.

And in a previous article, we discovered that use cases were either commands or queries.

Use Cases (a Clean Architecture term) are similar to **Application Services** in DDD. At least their *relative positioning* is.

In DDD, **Application Services** (application layer concerns, obviously) represent commands or queries (like `createComment` - COMMAND or `getCommentById` - QUERY) that:

- Contain no domain-specific business logic.
- Are used in order to fetch domain entities (and anything else) from persistence and the outside world.

- Either passes of control to an Aggregate to execute domain logic by using a method of the Aggregate, or passes off several entities to a Domain Service to facilitate their interaction.
- Have low-levels of Cyclomatic Complexity.

For example, the application service for the `UpvotePost` command would:

1. Get the member, post, and post votes
2. Pass them to the domain service
3. Save the transaction if it was successful.

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));
    }
  }
}
```

That's Application Services.

In Uncle Bob-land, Use Cases *do* allow for business logic, but there's a differentiation between what consistutes *application layer* business logic and what constitutes *domain business logic*. I haven't seen the Clean Coder videos myself to dive deeper on this differentiation, so maybe someone else can provide some more clarification, but it does sound quite similar to the way application and domain layer logic is organized in DDD.

I think these are the primary building blocks and areas of confusion between implementing the concepts from Clean Architecture and Domain-Driven Design.

It just goes to show how you can argue back and forth with someone about what the correct tool for the job is, before identifying the fact that you were talking about the same thing, semantically.

Good to know!