Domain-Driven Design: Everything You Always Wanted to Know About it, But Were Afraid to Ask

Pablo Martinez Follow

May 15, 2020 · 10 min read

Photo by Jezael Melgoza on Unsplash

As one's codebases grow, it is inevitable that their complexity will increase. As this

happens, it tends to get more difficult to keep code organized and structured as originally intended, this is known as Software Entropy. Over numerous iterations, maintaining good separation of concerns and properly decoupling classes and modules becomes more challenging if no strict architectural guidelines are enforced.

In the traditional Model-View-Controller (MVC) architecture, the "M" layer would hold all the business logic, but would not provide clear rules on how to maintain proper responsibility boundaries. Several patterns came up to mitigate this problem, but still there was always risk of logic and responsibility leakage between components, making maintainability and stability trickier as the model evolved.

On the other hand, communication with business experts, requirements gathering, and consensus between technical and non-technical teams to properly design and implement a system that solves a business problem is a constant iterative process where things can easily get misinterpreted, and ultimately derail the project from its original goals.

Naming things, for example, has always been one of the most difficult challenges Software Developers face. We should be clear enough for other developers to understand our intentions in the code, while using appropriate naming choices that can facilitate a conversation with business stakeholders.

Domain-Driven Design (DDD) attempts to solve these challenges, by reconciling the technical and non-technical forces that collide in a software project, and proposing a set of practices and patterns that facilitate building a successful system.

So what is Domain-Driven Design?

Let's start by defining what the word domain means in this context. I like to define it as

> *"A specific sphere of activity or knowledge that defines a set of common requirements, terminology, and functionality on which the application logic works to solve a problem."*

Domain-Driven Design is an approach to software design that glues the system's implementation to a constantly evolving model, leaving aside irrelevant details like programming languages, infrastructure technologies, etc…

It focuses mainly on a business problem and how to strictly organize the logic that solves it. This approach was first described by Eric Evans in his book *Domain-Driven Design*

Tackling Complexity in the Heart of Software.

Now that we know the definition of DDD and what its goals are, let's dive into the three main pillars of this methodology.

Strategic Design: Splitting your design so as to not lose your mind

As the implementation evolves through many iterations, and the system's complexity grows progressively, it can be daunting to maintain control over it. Therefore, a rigorous strategy to comprehend and control large systems is fundamental. Breaking down the model into Bounded Contexts that interact with each other — which themselves have their own unified model both in concept and in code — is an effective way to avoid complexity pitfalls.

Bounded Context

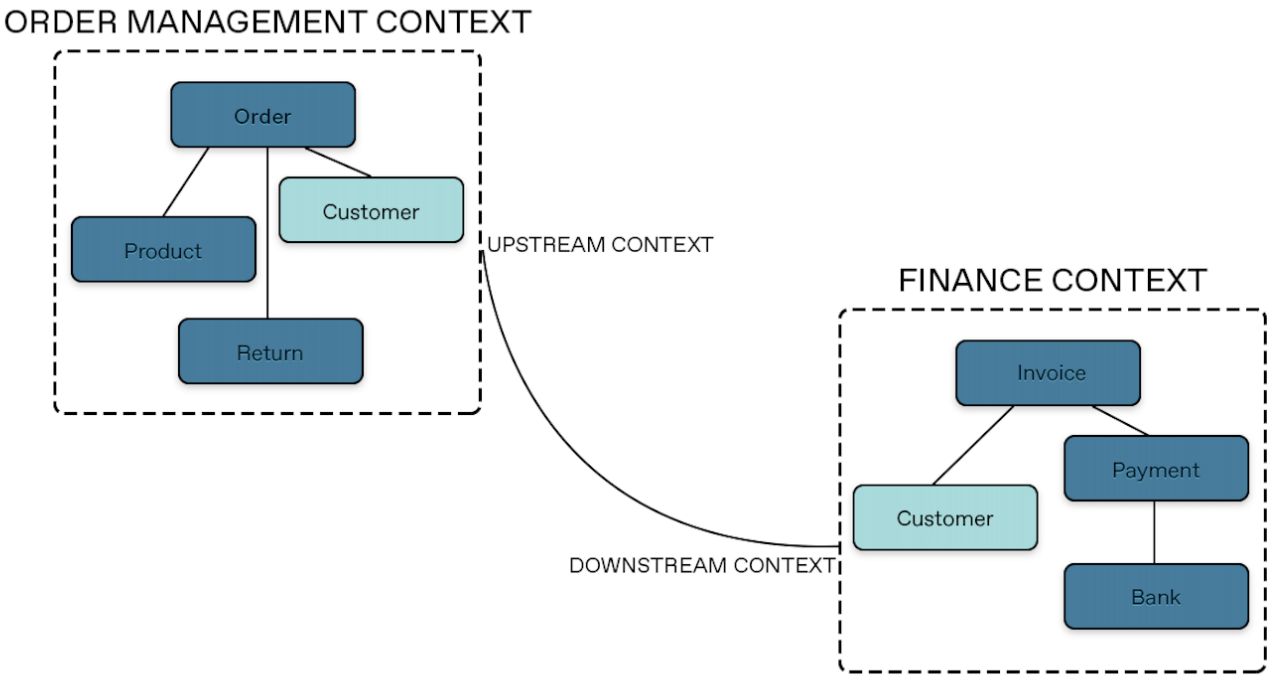
A Bounded Context is a conceptual boundary around parts of the application and/or the project in terms of business domain, teams, and code. It groups related components and concepts and avoids ambiguity as some of these could have similar meanings without a clear context.

For example, outside of a Bounded Context, a "letter" could mean two very different things: either a character or a message written on paper. By defining a boundary and context, you can determine its meaning

In many projects, teams are split by Bounded Contexts, each of them specializing on its own domain expertise and logic.

Context Mapping

Identifying and graphically documenting each Bounded Context in the project is called Context Mapping. Context Maps help better understand how Bounded Contexts and teams relate and communicate with each other. They give a clear idea of the actual boundaries and help teams visually describe the conceptual subdivisions of the system's design.



Relationships between Bounded Contexts can vary, depending on design requirements and other project-specific constraints, some relationships will be omitted from this article with the exception of the following four:

Anti-corruption Layer:

The downstream bounded context implements a layer that translates data or objects coming from the upstream context, ensuring that it supports the internal model.

Conformist:

Downstream bounded contexts conform and adapt to upstream contexts, having to change if needed. In this case the upstream context has no interest in meeting requirements downstream.

Customer/Supplier:

Upstream supplies downstream with a service, and downstream context acts as a customer, determining requirements and requesting changes upstream to meet their needs.

Shared Kernel:

Sometimes it's inevitable that two (or more) contexts overlap, and end up sharing resources or components. This relationship requires both contexts to be in continuous synchronization when changes are needed, therefore it should be avoided if possible.

Collaborative Modeling: Rich communication and effective collaboration

DDD proposes to model the domain effectively by doing a collaborative approach, involving all parties that hold not only technical, but business knowledge as well. As Evans describes it, the Domain Model "is not only the knowledge in a domain expert's head; it is a rigorously organized and selective abstraction of that knowledge."

Developers collaborate with domain experts with the intention of constantly refining the Domain Model, forcing them to learn important details and principles of the business problem they are trying to solve, instead of just producing code mechanically.

To enable this collaboration between business and technical teams, the Domain Model should use a language that joins business with technical jargon, and finds a middle ground that all team members can understand and agree on, this is called a Ubiquitous Language. Using a well-defined Ubiquitous Language will improve every interaction between tech and business teams, making them less ambiguous and more effective.

Ultimately this Ubiquitous Language will be embedded in the code.

Tactical Design: The Nuts and Bolts of DDD

Implementing associations between domain objects and describing their functionality feels easy at a glance, but proper distinction of their meaning and reason of existence should be done in a clear and intuitive way. DDD proposes a set of constructs and patterns to achieve it.

Entities

Objects that have a unique identity and possess a thread of continuity are called Entities, they are not defined solely by their attributes, but more by *who they are*. Their attributes might mutate and their life cycles can drastically change, but their identity persists. Identity is maintained via a unique key or a combination of attributes guaranteed to be unique.

In the e-commerce domain for example, an order has a unique identifier and it goes through several different stages: open, confirmed, shipped, and others, therefore it's considered a Domain Entity.

```
export class Customer {

    private id: number;
    private name: string;

    protected constructor(name: string) {
        // A uuid guarantees a unique identity for the Customer Entity
        this.id = uuidv4();
        this.name = this.setName(name);
    }

    private setName(name: string): string {
        // Business invariant: Customer name should not be empty
        if (name === undefined || name === '') {
            throw new Error('Name cannot be empty');
        }
        return name;
    }

    public static create(name: string): Customer {
        return new Customer(name);
    }
}
```

Customer.ts hosted with ♡ by GitHub view raw

Value Objects

Objects that describe characteristics, and that do not possess any unique identity are called Value Objects, they care only about what they are, not who they are.

Value Objects are attributes of, and can be shared by multiple entities, for example: two customers can have the same shipping address. Although there's a risk — if one of their attributes needs to change, all the entities that share them would get impacted. To prevent this, Value Objects must be immutable, forcing the system to replace them by fresh new instances, when an update is required.

Also, Value Object creation should always depend on the validity of the data used to create them and how it respects business invariants. Therefore, if data is invalid, the object instance will not be created. For example, in North America, a postal code with non-alphanumeric characters would violate a business invariant and would trigger an exception on the Address creation.

```
export class Address {

    private readonly streetAddress: string;
    private readonly postalCode: string

    protected constructor(streetAddress: string, postalCode: string) {
        this.streetAddress = this.getValidStreetAddress(streetAddress);
        this.postalCode = this.getValidPostalCode(postalCode);
    }

    private getValidStreetAddress(streetAddress: string): string {
        // Business invariant: street address should not be longer than 128 characters
        if (streetAddress.length > 128) {
            throw new Error('Address should not be longer than 128 characters');
        }
        return streetAddress;
    }

    private getValidPostalCode(postalCode: string): string {
        // Business invariant: Should be a valid canadian postal code
        const pattern = /[a-z]\d[a-z][ \-]?\d[a-z]\d/g;
        if (!postalCode.match(pattern)) {
            throw new Error('Postal code should only contain alphanumeric caracters and spaces')
        }
        return postalCode;
    }

    public getStreetAddress(): string {
        return this.streetAddress;
```

```
30      }
31
32      public getPostalCode(): string {
33          return this.postalCode;
34      }
35
36      public static create(streetAddress: string, postalCode: string): Address {
37          return new Address(streetAddress, postalCode);
38      }
39
40      public equals(otherAddress: Address): boolean {
41          // Value Objects equality is based on their propertie's values
42          return objectHelper.isEqual(this, otherAddress);
43      }
44  }
```

Address.ts hosted with ♡ by GitHub view raw

Services

In many cases the Domain Model requires certain actions or operations that are not directly related to an Entity or a Value Object, forcing these into their implementations causes a distortion of their definition. Services are classes that offer stateless operations. They are commonly named as verbs, as opposed to nouns for Entities and Value Objects, and are named based on the Ubiquitous Language.

Services should be carefully crafted, always ensuring that they do not strip Entities and Value objects of their direct responsibilities and behavior. They should also be stateless, such that clients can use any given instance of a Service disregarding that instance's history during the application's lifetime. Having Entities and Value Objects with no domain logic is considered an anti-pattern called the Anemic Domain Model.

Domain Objects and their Life Cycle

Domain objects commonly have complex life cycles, they are instantiated, go through several changes, interact with other objects, execute operations, get persisted, get reconstituted, get deleted, and so on. Maintaining their integrity while ensuring that the system doesn't mismanage their complex lifecycle is one of the main challenges that implementing a proper Domain Model represents.

Minimizing the relationship and interaction between domain objects to maintain manageable levels of complexity within the Domain Model is difficult, especially in complex business domains, or as Eric Evans describes it:

> *"It is difficult to guarantee the consistency of changes to objects in a model with complex associations. Invariants need to be maintained that apply to closely related groups of objects, not just discrete objects. Yet cautious locking schemes cause multiple users to interfere pointlessly with each other and make a system unusable"*

Aggregates

To mitigate the aforementioned challenge, an aggregation of Entities and Value Objects is needed, restricting the violation of business invariants.

Aggregates are collections of related Entities and Value Objects, clustered together representing a transactional boundary. Each Aggregate has an Entity that faces outwards and controls all access to the objects inside the boundary, this entity is called the Aggregate Root, and it's the only object that other objects can interact with. No objects within the Aggregate can be called directly from the outside world, thus maintaining consistency within.

Business Invariants are business rules that guarantee the integrity of an Aggregate and its contents, in other words, it is a mechanism that ensures that their state is always consistent with the business rules. For example, an order can never be placed when the stock quantity of a certain product is zero.

```
1   export class Order {
2
3       private id: number;
4       private isConfirmed: boolean;
5       private total: number;
6       private shippingAddress: Address;
7       private customer: Customer;
8       private items: Product[];
9       private payments: Payment[];
10
11      constructor(
12          customer: Customer,
13          shippingAddress: Address,
14          items: Item[],
```

```
        payments: Payment[]
    ) {
        // Generate a unique identifier (UUID) for the Order Entity
        this.id = uuidv4();
        this.isConfirmed = false;
        this.total = 0;
        this.customer = customer;
        this.shippingAddress = shippingAddress;
        this.items = items.length ? items : [];
        this.payments = payments.length ? payments : [];
    }

    private getPaymentsTotal(): number {
        return this.payments.reduce((accumulator, payment) => accumulator + payment.total);
    }

    public addPayments(payment: Payment): void {
        this.payments.push(payment);
        this.total += payment.total;
    }

    public addItems(product: Product): void {
        // Business invariant: an order should not have items which are not in stock
        if (!product.getStockQuanity()) {
            throw new Error(`No stock for product id: ${product.id}`);
        }
        this.items.push(product);
    }

    public confirm(): void {
        // Business invariant: only fully paid orders can be confirmed
        if (this.total === this.getPaymentsTotal()) {
            throw new Error('Total amount paid does not equal order total');
        }
        this.isConfirmed = true;
    }
}
```

Order.ts hosted with ♡ by GitHub view raw

Factories

Creating complex object and aggregate instances can be a daunting task, and can also disclose too much of the object's internal details. Using Factories, we can solve this

problem and provide the necessary encapsulation.

A Factory should be able to construct domain objects or aggregates in one atomic operation, requiring all data needed to be provided by the client when called, and enforcing all invariants on the created object. This activity is not part of the Domain Model, but still belongs in the Domain Layer as it is part of the business rules that apply to the system.

```
export class OrderFactory implements Factory {

    private customerEntity: Customer;
    private addressValue: Address;
    private productsRepository: Repository;
    private paymentsRepository: Repository;

    constructor(customerEntity: Customer, addressValue: Address, productsRepository: Repository,
        this.customerEntity = customerEntity;
        this.addressValue = addressValue;
        this.productsRepository = productsRepository;
        this.paymentsRepository = paymentsRepository;
    }

    public async createOrder(customerName: string, addressDto: AddressDto, itemDtos: ItemDto[],
        try {
            const customer = this.customerEntity.create(customerName);
            const shippingAddress = this.addressValue.create(addressDto.streetAddress, addressDt
            const items = await this.productsRepository.getProductCollection(itemDtos);
            const payments = await this.paymentsRepository.getPaymentCollection(paymentDtos);

            return new Order(customer, shippingAddress, items, payments);
        } catch(err) {
            // Error handling logic should go here
            throw new Error(`Order creation failed: ${err.message}`);
        }
    }
}
```

OrderFactory.ts hosted with ♡ by GitHub view raw

Repositories

To be able to retrieve objects from our persistence, be it in-memory, filesystem, or database, we need to provide an interface that hides the implementation details from the client, so that it does not depend on the infrastructure specifics, but merely on an abstraction.

Repositories provide an interface that the Domain Layer can use to retrieve stored objects, avoiding tight-coupling with the storage logic and giving the client an illusion that the objects are being retrieved directly from memory.

It's important to mention that all repository interface definitions should reside in the Domain Layer, but their concrete implementations belong in the Infrastructure Layer.

```
export class OrderRepository implements Repository {
    private model: OrderModel;
    private mapper: OrderMapper;
    private productsRepository: Repository;
    private paymentsRepository: Repository;

    constructor(orderModel: OrderModel, orderMapper: Mapper, productsRepository: Repository, pay
        this.model = orderModel;
        this.mapper = orderMapper;
        this.productsRepository = productsRepository;
        this.paymentsRepository = paymentsRepository;
    }

    public async getById(orderId: number): Promise<Order> {
        const order = await this.model.findOne(orderId);

        if (!order) {
            throw new Error(`No order found with order id: ${orderId}`);
        }
        return this.mapper.toDomain(order);
    }

    public async save(order: Order): Promise<Boolean> {
        const orderRecord: OrderRecord = this.mapper.toPersistence(order);

        try {
            await this.productsRepository.insert(order.items);
            await this.paymentsRepository.insert(order.payments);

            if (!!await this.getById(order.id)) {
```

```
                await this.model.update(orderRecord);
            } else {
                await this.model.insert(orderRecord);
            }
        } catch (err) {
            // call to rollback mechanism should go here
            return false;
        }
        return true;
    }
}
```

OrderRepository.ts hosted with ♡ by GitHub view raw

Isolating the Domain from Other Concerns

The part of the code that is written to specifically solve domain problems is a fraction of the whole codebase. If this portion is intertwined with code that solves other concerns, it will be difficult to understand and improve. Clearly separating domain logic from all other functionality will reduce the leakage and will avoid confusion in a large and complex system.

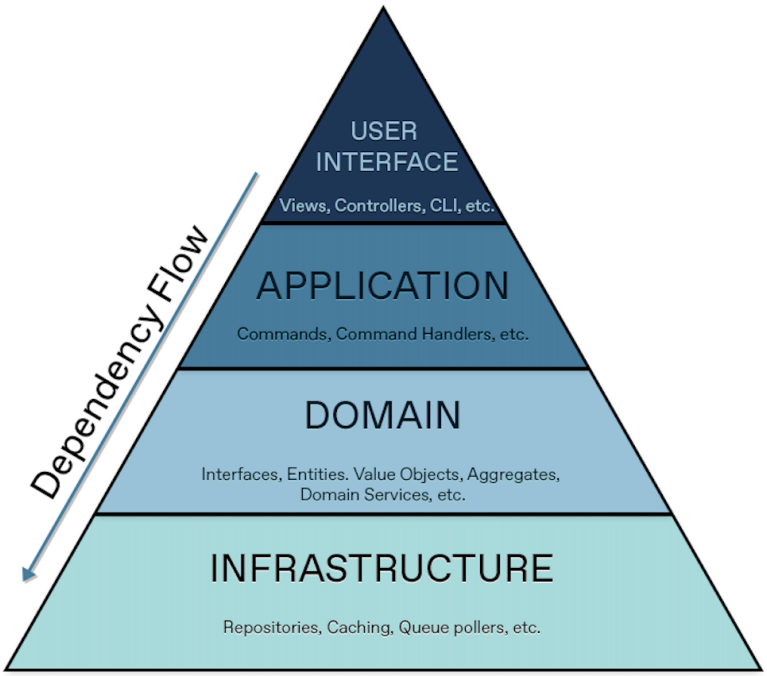
DDD proposes a Layered Architecture as a way to separate concerns and avoid responsibility confusion by splitting the codebase in 4 main layers: User Interface, Application, Domain, and Infrastructure.

The main rule here is that components in each layer should depend only on components in the same layer or any layer beneath it. Upper layers can use components of lower ones just by calling their public interfaces and lower layers can *only* communicate upwards by Inversion of Control (IoC).

- **User Interface Layer:** is responsible for displaying data and capturing user's commands.
- **Application Layer:** serves as an orchestrator of domain work, it does not know domain rules, but organizes and delegates domain objects to do their job. It is also the only layer accessible to other bounded contexts.
- **Domain Layer:** holds the business logic and rules, as well as the business state. It is where the Domain Model lives.

- **Infrastructure Layer:** implements all the technical functionalities the application needs to support the higher layers, persistence, messaging, communication between layers, etc…

Even though not all layers are required in every system, the existence of the Domain Layer is a prerequisite in DDD.



Conclusion

All in all, DDD is a holistic approach for solving business problems through rich collaboration with domain experts and strict design patterns, it is not a universal solution for all software projects, but it can provide significant benefits when properly applied.

There are several books on this subject, and various concepts were purposely left out of this article, but If I have sparked your interest in Domain-Driven Design, I invite you to start by reading the blue and red books, in that order, which are the starting point of this massive topic.

Editorial reviews by Anas Trabulsi, Mario Bittencourt, Mikhail Levkovsky, Deanna Chow, Liela Touré, & Prateek Sanyal.

Want to work with us? Click here to see all open positions at SSENSE!

Software Architecture Design Patterns Software Software Engineering Domain Driven Design