A Crash Course on Domain-Driven Design

BYTEBYTEGO
AUG 01, 2024

Developing software for complex domains is a challenging task.

As the complexity of the problem domain grows, it becomes increasingly difficult to create software that accurately represents the business concepts, rules, and processes. Poorly designed software can quickly turn into an incomprehensible tangle of code that is difficult to understand, maintain, and extend.

Domain-Driven Design (DDD) offers a solution to this problem.

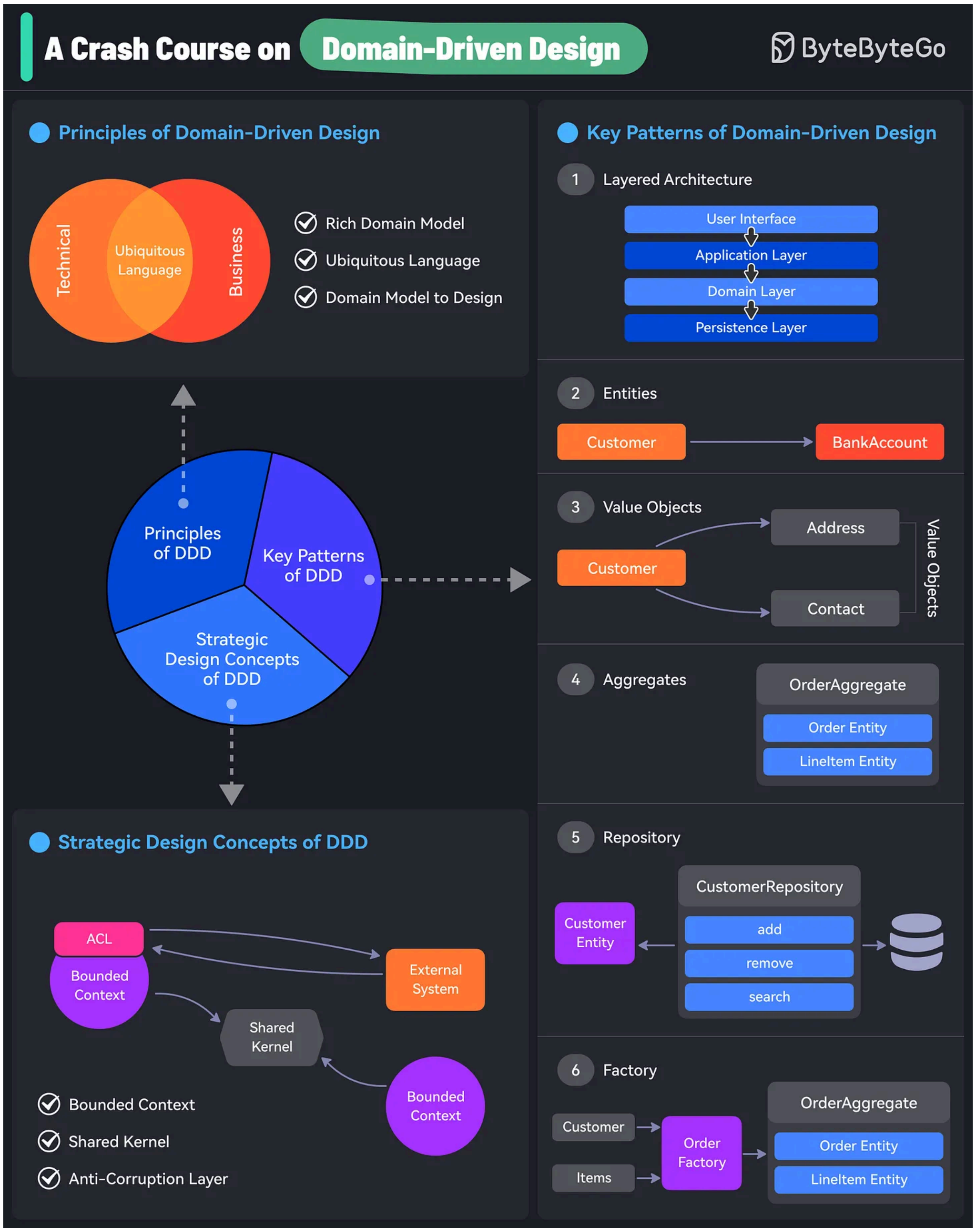
DDD is an approach to software development that tackles domain complexity by emphasizing the importance of modeling the core domain and business logic and using those models as the foundation for software design.

At its heart, Domain-Driven Design is about:

- Placing the primary focus on the core domain.
- Basing complex designs on a model of the domain
- Building collaboration between technical and domain experts.

The need for Domain-Driven Design has become more pressing in recent years. Architectures based on microservices and cloud computing have resulted in systems composed of numerous small components that interact in intricate ways. Without a clear and well-defined model of the domain guiding their design, such systems can quickly become a "big ball of mud".

In this article, we'll understand the basics of Domain-Driven Design and its key concepts that can help us build more maintainable and extensible systems that are aligned with the core domain and business logic.



Core Principles of Domain-Driven Design

Domain-Driven Design (DDD) focuses on creating software systems that closely align with the underlying business domain.

It aims to bridge the gap between the technical implementation and the business requirements by placing the domain model at the center of the development process.

There are three core principles of DDD:

- Creating a rich domain model based on input from domain experts
- Using a ubiquitous language based on the domain model
- Driving the design of the software from the domain model

Let's explore each of these principles in more detail.

Creating a Rich Domain Model

The foundation of DDD lies in the creation of a rich domain model that accurately captures the key concepts, relationships, and business rules of the problem domain. This model is not created in isolation by the development team but emerges through close collaboration with domain experts who possess deep knowledge of the business.

The process of distilling domain knowledge into a usable model is known as knowledge crunching.

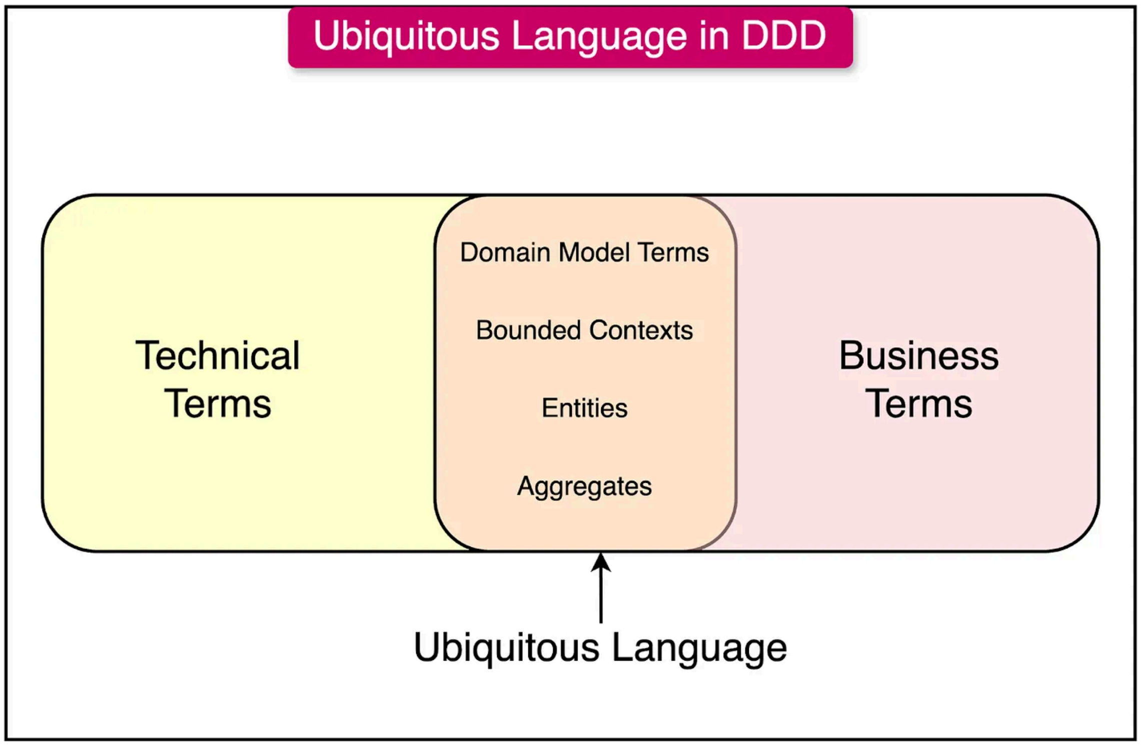
During this process, the development team works closely with the domain experts to identify and refine the most relevant concepts and rules for the problem. This collaborative effort often includes brainstorming sessions, experimentation with different model designs, and iterative refinement based on feedback and insights gained during development.

For example, in a banking system, a knowledge session with domain experts from the finance department would focus on understanding core concepts such as accounts, customers, transactions, and interest calculations.

The goal is to create a model that captures these fundamental concepts and their relationships, which are understandable to the domain experts and serve as a practical foundation for the software system.

Using a Ubiquitous Language

As the domain model takes shape, it gives rise to a common language shared by the development team and domain experts. DDD refers to this as the ubiquitous language, as it permeates all aspects of the project.



The ubiquitous language is directly based on the domain model.

The names of classes, methods, and variables in the code are derived from the model concepts. Similarly, in discussions and documents, the team uses the terms and phrases of the ubiquitous language to ensure clear communication and avoid ambiguity.

With a consistent use of the ubiquitous language, the code directly reflects the model, making it easier to understand and modify as the model involves.

For instance, if the banking domain model includes concepts like "Account," "Deposit," and "Withdrawal," these same terms will be used consistently in the code, discussions between developers and domain experts, and project documentation. Any ambiguities or inconsistencies in understanding are quickly brought to light when everyone uses the same language.

Here's a simple code example demonstrating the use of the ubiquitous language while writing the class and its methods.

```
public class Account {
    private String accountNumber;
    private double balance;
    public void deposit(double amount) {
        // Perform deposit logic
        balance += amount;
    }

    public void withdraw(double amount) {
        // Perform withdrawal logic
        if (balance >= amount) {
            balance -= amount;
        } else {
            throw new InsufficientFundsException("Insufficient funds for
withdrawal");
        }
    }
}
```

Domain Model to Software Design

In DDD, the domain model is not just a conceptual tool. It is the very foundation of the software design. The structure and behavior of the software mirror the structure and behavior of the model.

This approach is known as model-driven design.

In practice, it means that the classes, relationships, and behaviors in the code directly correspond to the concepts, relationships, and rules in the domain model. The design is not driven by technical concerns or infrastructure details but by the need to effectively express the domain model.

For example, if the banking domain model defines an "Account" concept with behaviors like "deposit" and "withdraw," the software design will include an "Account" class with methods for "deposit" and "withdraw." The implementation details of how these methods work will be guided by the rules and requirements captured in the model. We've already seen this in action in the previous section.

Key Patterns to Craft a Domain Model

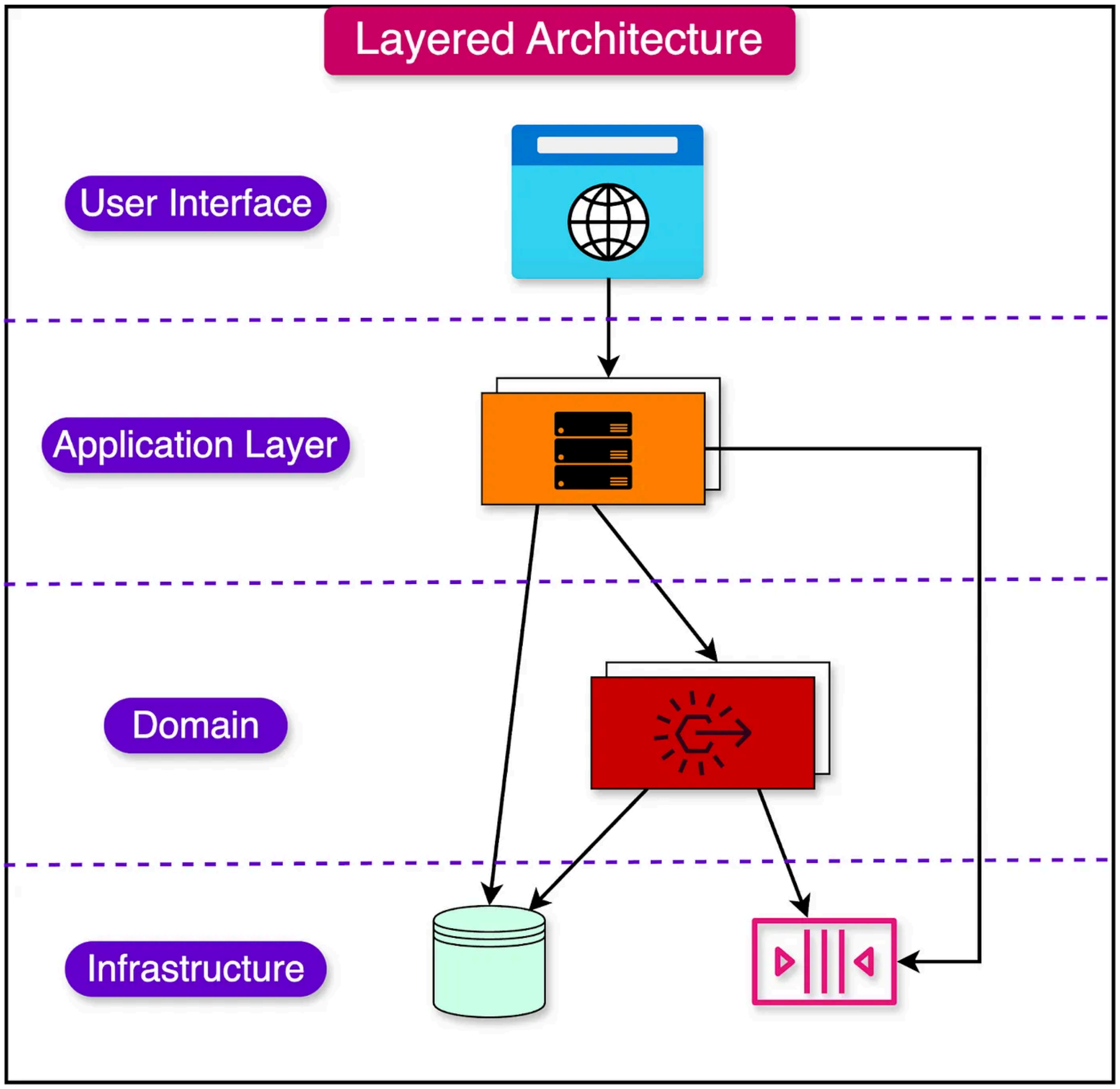
Crafting a domain model is the most important aspect of Domain Driven Design. However, some key patterns and practices can help us achieve optimal results.

Let's look at the major ones in detail.

Layered Architecture

In the context of Domain-Driven Design (DDD), a layered architecture helps isolate the domain model and promote a clear separation of concerns.

The diagram below shows how an application can be structured into layers.



Typically, a layered architecture consists of four main layers:

- **User Interface (Presentation) Layer:** This layer is responsible for presenting information to the user and interpreting user commands. It communicates with the Application Layer to send user requests and receive responses.
- **Application Layer:** The Application Layer coordinates the overall application activity and orchestrates the data flow between the UI and the Domain Layer. It does not contain business logic but delegates it to the Domain Layer. The Application Layer handles application-specific tasks, such as transaction management and security.
- **Domain Layer:** The Domain Layer is the core of the software system and the primary focus of Domain-Driven Design. The Domain Layer represents the real-world concepts, entities, and their relationships within the problem domain. In other words, it encapsulates the information about the business domain and contains the business rules and logic.
- **Infrastructure Layer:** The Infrastructure Layer provides generic technical capabilities and supports the higher layers of the architecture. It includes cross-cutting concerns such as

persistence, messaging, logging, and other technical services.

For example, in an e-commerce system, the Domain Layer would contain core business concepts like Product, Order, and Customer. The UI Layer would have screens for product display, shopping cart, and checkout. The Application Layer would handle the flow of the order placement process. The Infrastructure Layer would provide functionalities like persistence, messaging, etc.

Entities

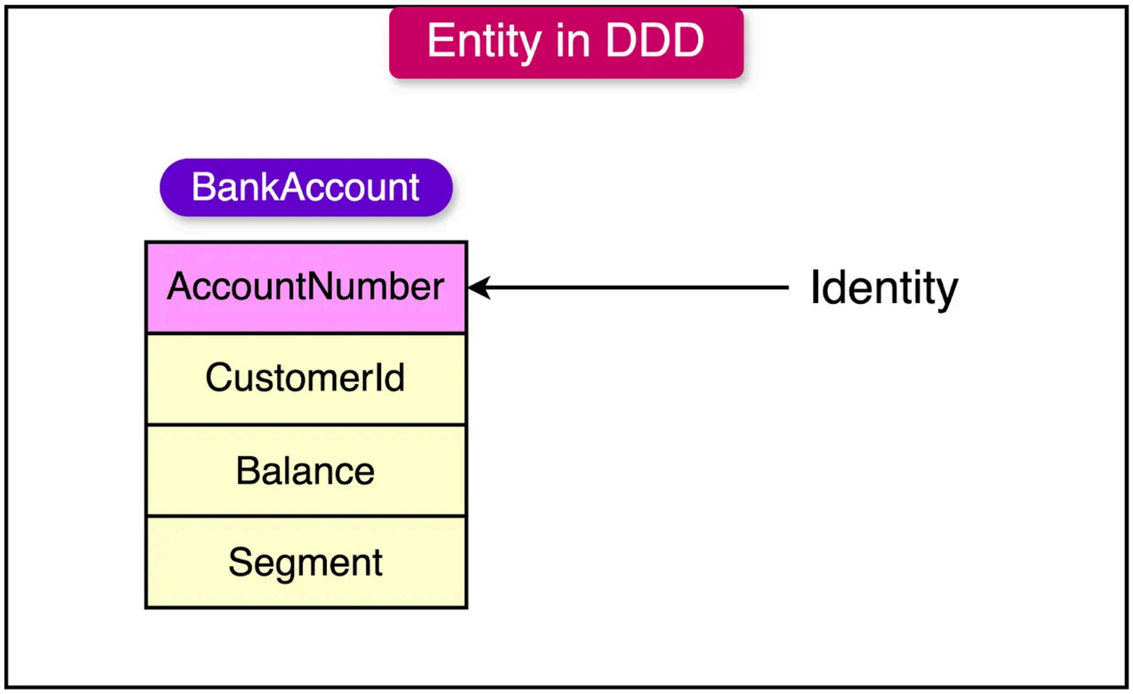
In Domain-Driven Design (DDD), an entity is a fundamental concept that represents an object with a distinct identity and a thread of continuity.

Entities possess several key characteristics that distinguish them from other objects in the domain:

- Entities are defined by their identity rather than just a collection of attributes. The identity makes an entity unique.
- Entities can change their state and attributes with time while still maintaining their identity.
- The identity of an entity is used to track and reference it within the system. Other entities can establish relationships with an entity based on its identity.

To understand the concept of an entity, consider a banking system. In this context, BankAccount would be a prime example of an entity.

Each bank account has a unique account number that serves as its identity. This account number distinguishes one account from another, regardless of any other attributes different accounts may have in common.



A bank account's information can change over time, due to deposits, withdrawals, or updates to the account holder's information. Despite these changes, the account retains its identity. It remains the same account throughout its lifecycle.

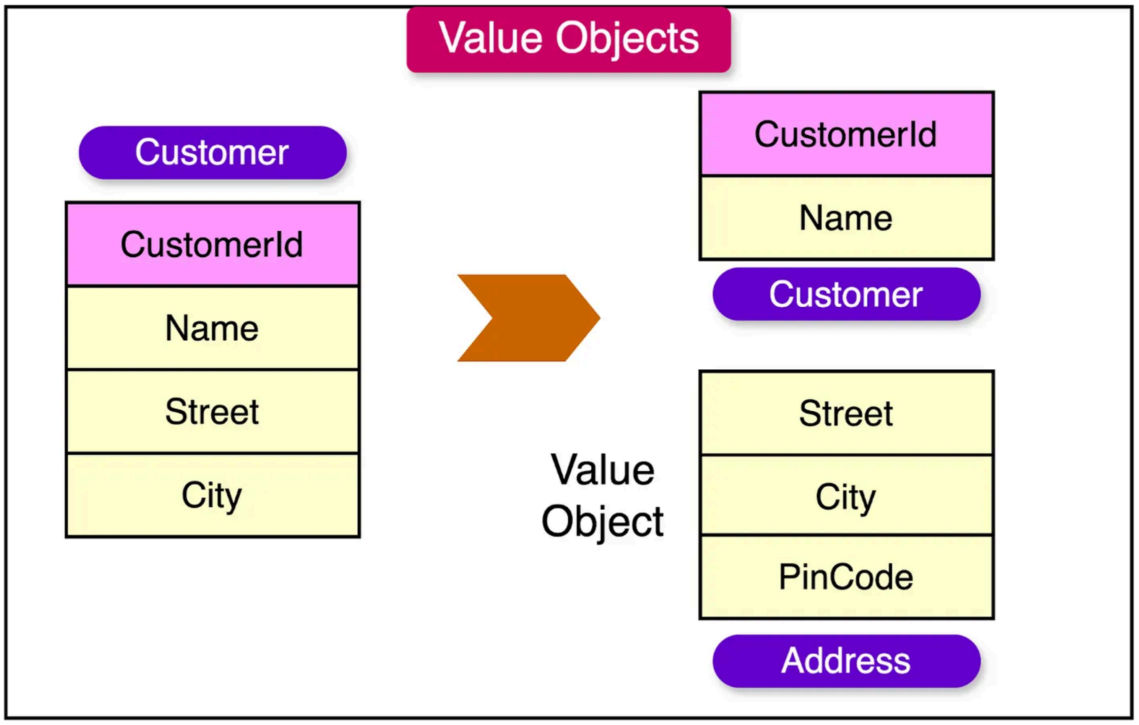
Value Object

A value object represents a descriptive aspect of the domain without having its own identity. They are defined solely by their attributes, rather than having a separate identity that distinguishes them from other objects.

Some key characteristics of value objects are as follows:

- Unlike entities, value objects do not have a unique identity. They are defined entirely by their attributes.
- Value objects are immutable, meaning their state cannot be modified after creation. Changing a value object means creating a new instance with the desired attribute values.
- Any two value objects with the same attribute values are considered equal and interchangeable.

For example, within a Customer entity, the Address representing a specific location is a prime example of a value object. It is defined by attributes such as street, city, and pin code. Also, any two addresses with the same details are considered the same address.



Some other value object examples are “MonetaryAmount”, “GeographicLocation”, “Color”, etc.

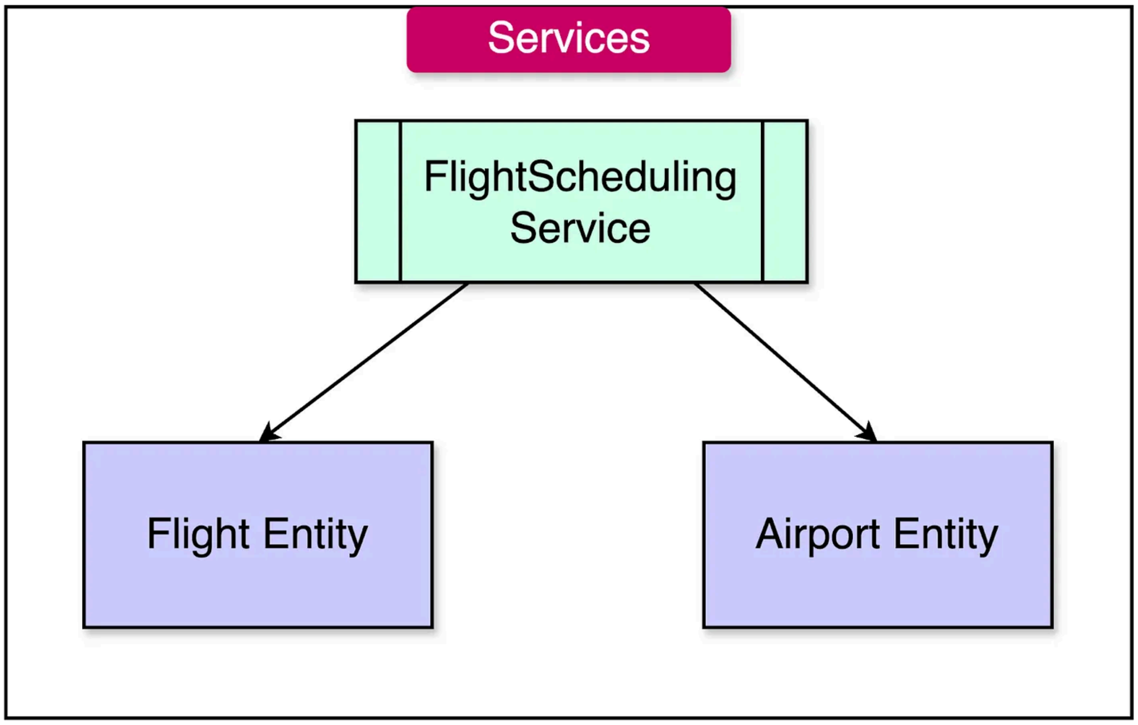
Services

A service is a concept that represents an operation or a behavior that doesn't naturally fit within the responsibility of an entity or a value object. They encapsulate logic that doesn’t belong to any particular object but still plays a significant role in the domain model.

The key characteristics of services are as follows:

- Services operate on entities and value objects, but they don’t have an internal state that affects their behavior.
- The interface of a service is defined using other elements. For example, services take entities or value objects as input, perform operations on them, and may return other entities or value objects as output.
- Services are stateless. In other words, any service instance can be used by any client without affecting other clients.

For example, an airline booking system can have a “FlightSchedulingService” responsible for finding available flights based on the provided criteria. A method in this service can take the departure and the destination airport as input and return a list of available flights. Such an operation doesn’t naturally belong to any single entity, such as the “Airport” or “Flight”, because it involves coordinating information from multiple sources and applying business rules.



Aggregates

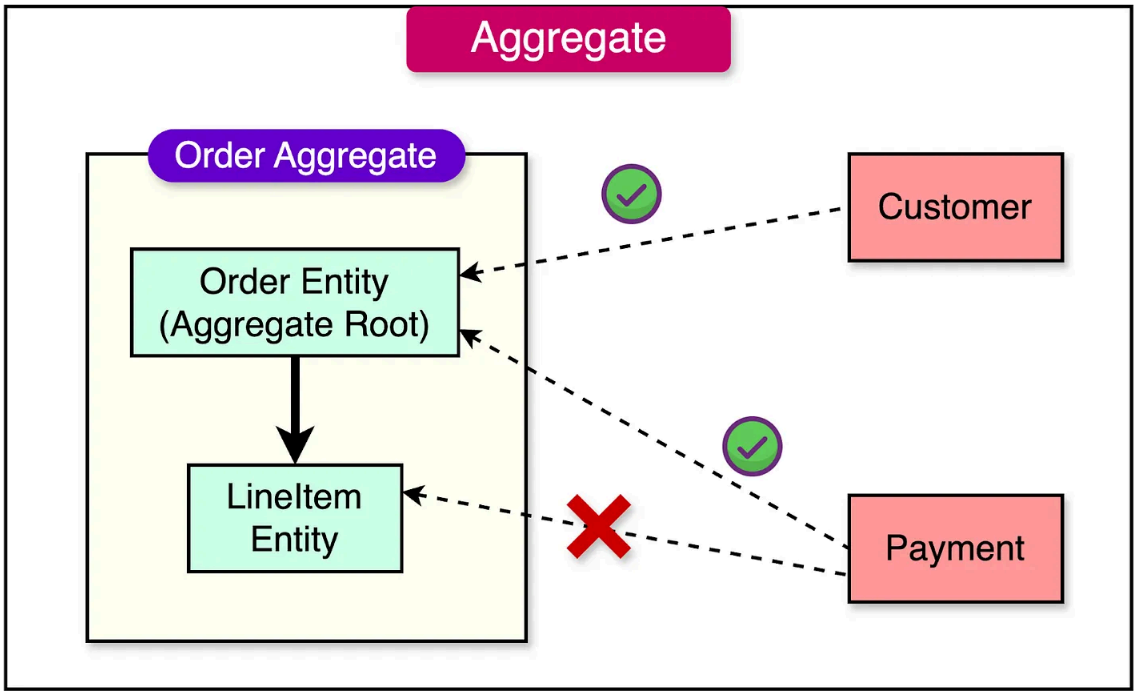
In Domain-Driven Design (DDD), an aggregate is a pattern that helps manage the complexity of the domain model by grouping related objects into a cohesive unit.

They have several key characteristics such as:

- Aggregates define a consistent boundary around a group of related objects. This means that the objects within an aggregate are always in a consistent state with each other.
- Every aggregate has a designated root entity known as the aggregate root. The root entity is the only member of the aggregate that outside objects can hold references to. All interactions with the objects inside the aggregate must go through the root entity.
- Aggregates should be designed to be small enough to ensure consistency and maintainability, but large enough to encompass all the necessary related objects.

To illustrate the concept of an aggregate, let's consider an example from an order processing system.

In this example, an “Order” and its associated “LineItems” form a natural aggregate. The “Order” encapsulates the business rules and invariants related to order processing, such as calculating the total amount, applying discounts, and ensuring the validity of the line items.



From a code point of view, the Order aggregate can look like this when it comes to dealing with LineItems:

```
public class Order {
    private String orderId;
    private List<LineItem> lineItems;
    private BigDecimal totalAmount;
    private OrderStatus status;
    public void addLineItem(LineItem lineItem) {
        lineItems.add(lineItem);
        totalAmount = totalAmount.add(lineItem.getSubtotal());
    }

    public void removeLineItem(LineItem lineItem) {
        lineItems.remove(lineItem);
        totalAmount = totalAmount.subtract(lineItem.getSubtotal());
    }
}

public class LineItem {
    private String productId;
    private int quantity;
    private BigDecimal unitPrice;
}
```

External objects, such as a “Customer” or a “Payment”, would hold references only to the “Order” entity and not directly to the “LineItems”. Any operations or modifications to the

“LineItems”, such as adding, removing, or updating items, would be performed through the “Order” entity.

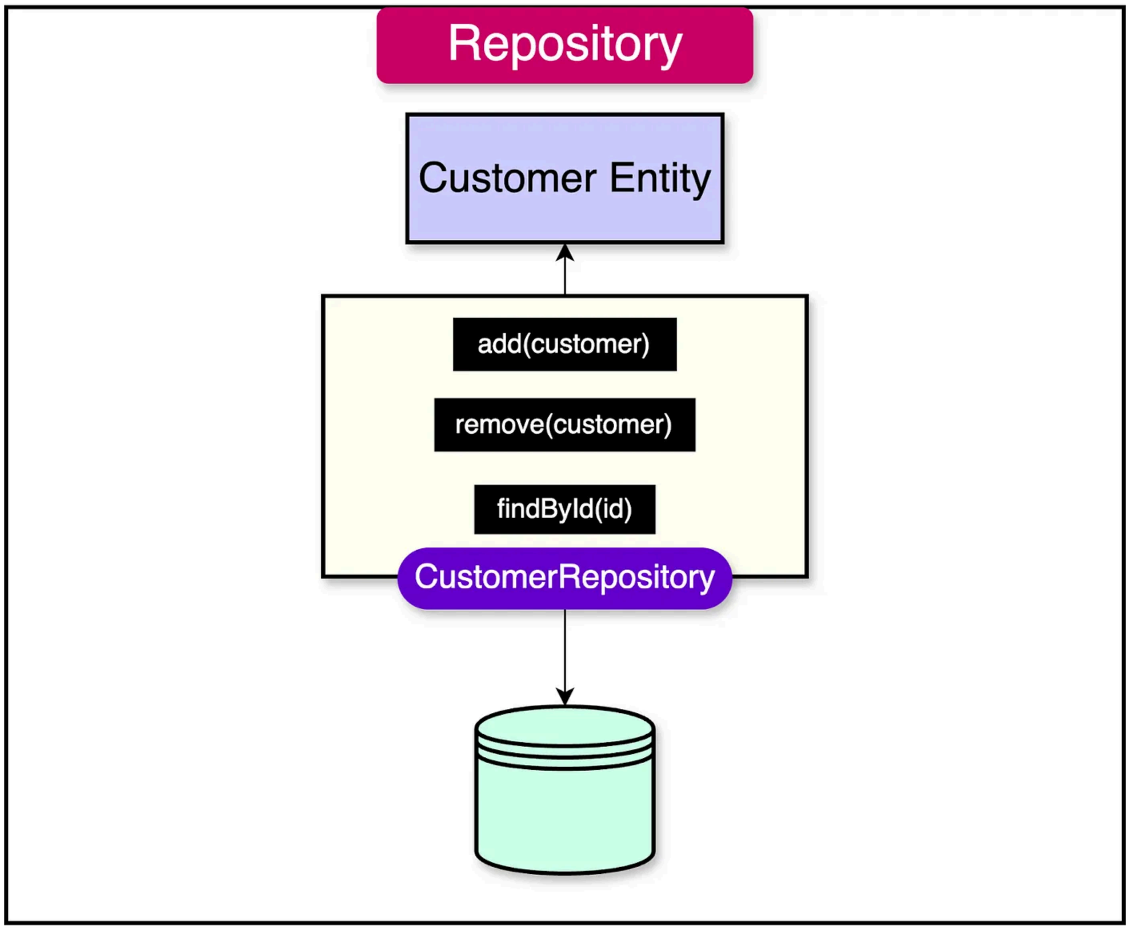
Repository

A repository is a pattern that provides an abstraction over the persistence layer for a specific type of entity or aggregate root. It acts as a collection-like interface, encapsulating the underlying data access and storage mechanisms.

A few key points about repositories are as follows:

- Repositories provide methods that simulate the behavior of a collection, such as adding, removing, and querying objects. The client code can interact with the repository as if it were a simple collection, without exposing the complexities of the underlying persistence mechanism.
- Repositories act as a gateway to the persistence layer for a specific aggregate or entity.
- Repositories decouple the application and the domain design from the specifics of the persistence layer.

For example, A “CustomerRepository” would provide methods like add(customer), remove(customer), and findByID(id) to manage the persistence of Customer entities, without exposing details of the storage approach.



These methods allow the client code to interact with the “CustomerRepository” in a way that mimics a collection of “Customer” objects.

The client can add, remove, and query customers without worrying about how the data is stored in the database. Behind the scenes, the repository may use an ORM tool, SQL query, or any other persistence mechanism to interact with the underlying storage system.

Factory

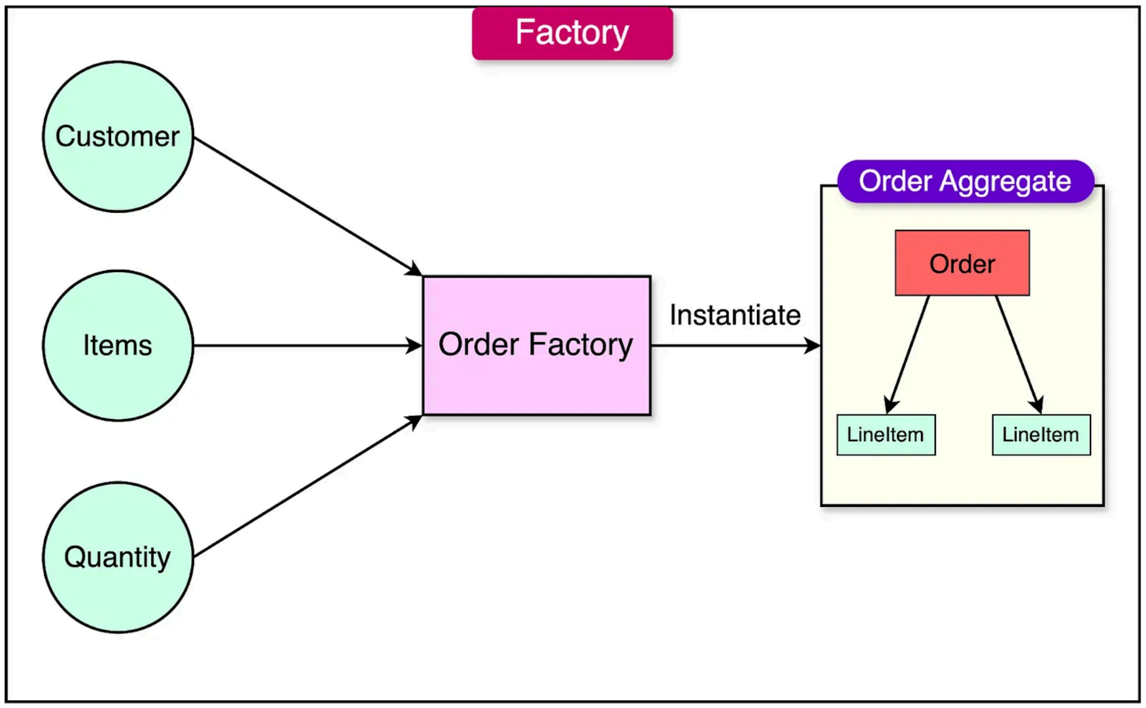
In domain-driven design, a factory is a pattern that encapsulates the creation of complex entities or aggregates. It provides an interface for clients to create objects without exposing the details of the creation process or the dependencies involved.

Some key characteristics of factories are as follows:

- Factories hide the intricacies of object instantiation, validations, and dealing with dependencies.
- Factories allow the creation process to be changed independently of the clients that use them. This improves the flexibility and maintainability of the system.

- Factories ensure that the objects they create are always in a consistent and fully initialized state.

For example, an “OrderFactory” in an e-commerce system can take all the necessary details (customer, items, quantities, etc.), create the “Order” and “LineItem” instances, link them together appropriately, and return the resulting Order aggregate to the client.



Strategic Design Concepts of DDD

When dealing with large models in big projects, some strategic design concepts should be followed to gain the benefits of domain-driven design.

Bounded Contexts

In large-scale software projects, the domain model often encompasses multiple subdomains, each with its own set of concepts, entities, and business rules.

Attempting to maintain a single, unified model across all these subdomains can lead to a complex and tangled model that becomes difficult to understand, reason about, and maintain over time.

The solution to this challenge is to introduce the concept of Bounded Contexts.

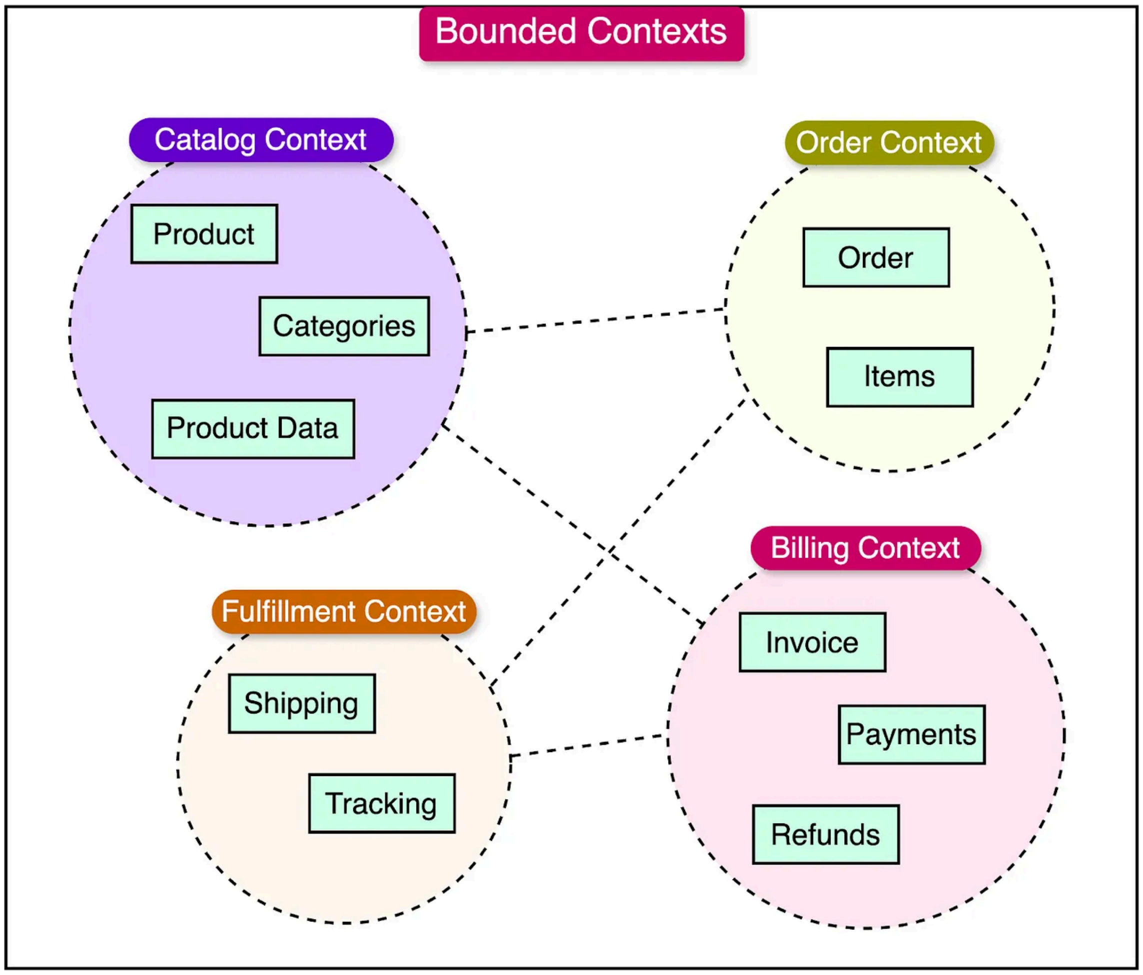
A Bounded Context is a specific area within the domain model that represents a particular subdomain or a cohesive set of related concepts. Each Bounded Context encapsulates its unified model, which is tailored to the specific needs and requirements of that subdomain.

A well-defined Bounded Context exhibits the following characteristics:

- **Unified Model:** Within a Bounded Context, there should be a consistent and coherent model that represents the concepts, entities, and relationships specific to that subdomain.
- **Ubiquitous Language:** Each Bounded Context should have its ubiquitous language.
- **Clear Scope and Boundary:** A Bounded Context should have a well-defined scope and boundary.

To understand the concept of Bounded Contexts, consider the example of an e-commerce system. In this system, we can identify several subdomains, each of which can be a separate Bounded Context:

- **Catalog Context:** This context focuses on managing the product catalog, including product information, categories, and attributes. It has its model that represents products, their variations, and the relationships between them.
- **Order Context:** The Order Context deals with the process of placing and managing orders. It has a model that represents orders, order items, and the associated business rules for order processing and fulfillment.
- **Billing Context:** The Billing Context handles the financial aspects of the e-commerce system, including invoicing, payments, and refunds. It has a model representing invoices, payment transactions, and the associated business rules.
- **Fulfillment Context:** The Fulfillment Context is responsible for the logistics and delivery of orders. It has a model representing shipping methods, tracking information, and the processes involved in fulfilling orders.



Relationships Between Bounded Contexts

There can be relationships and interactions between different Bounded Contexts. These relationships are typically represented through shared concepts or translations between the models of the different contexts.

Some key patterns to establish relationships between Bounded Contexts are as follows:

1 - Context Maps

In a complex domain-driven design (DDD) project, multiple Bounded Contexts represent different subdomains or parts of the system. It might become difficult to understand the overall system landscape. It becomes beneficial to create a Context Map.

A Context Map is a high-level visual representation of the Bounded Contexts in a project and the relationships between them. It provides a bird's-eye view of the system, highlighting the key boundaries and integration points.

The Context Map should provide a clear overview of:

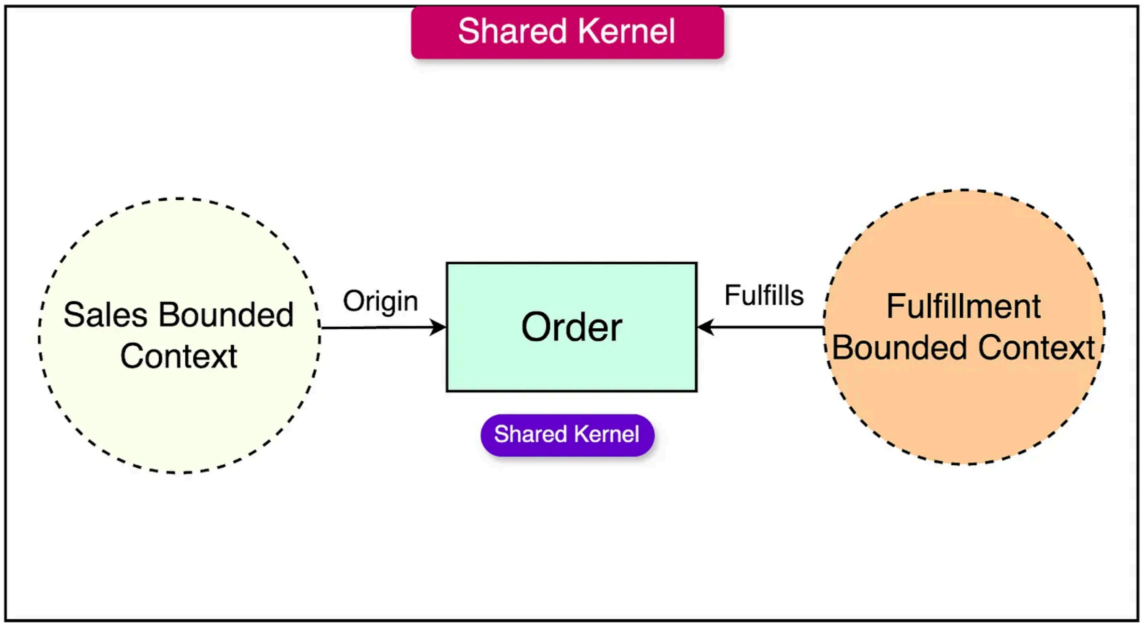
- **Bounded Contexts:** The distinct subdomains or logical boundaries within the system, each representing a specific area of the domain model.
- **Relationships:** The connections and interactions between the Bounded Contexts, indicating how they relate to and communicate with each other.
- **Integration Points:** The key points of integration where Bounded Contexts exchange data or coordinate their behavior.

2 - Shared Kernel

A shared kernel represents a portion of the domain model commonly used and relied upon by multiple teams or Bounded Contexts. It encapsulates the core concepts, entities, and their relationships that are fundamental to the domain and relevant across different parts of the system.

For example, consider that there are two Bounded Contexts - Sales and Fulfillment. The Sales Bounded Context includes the product catalog, pricing, promotions, the shopping cart, and order placement. The Fulfillment Bounded Context handles inventory, suppliers, shipment, and delivery.

While these Bounded Contexts have distinct concerns, the “Order” is one concept shared by both of them. The “Order” originates in the Sales Bounded Context, but once it is placed, it needs to be fulfilled by the Fulfillment Bounded Context. Therefore, the Order forms a shared kernel between Sales and Fulfillment. Both teams need to agree on what an “Order” means and its lifecycle.

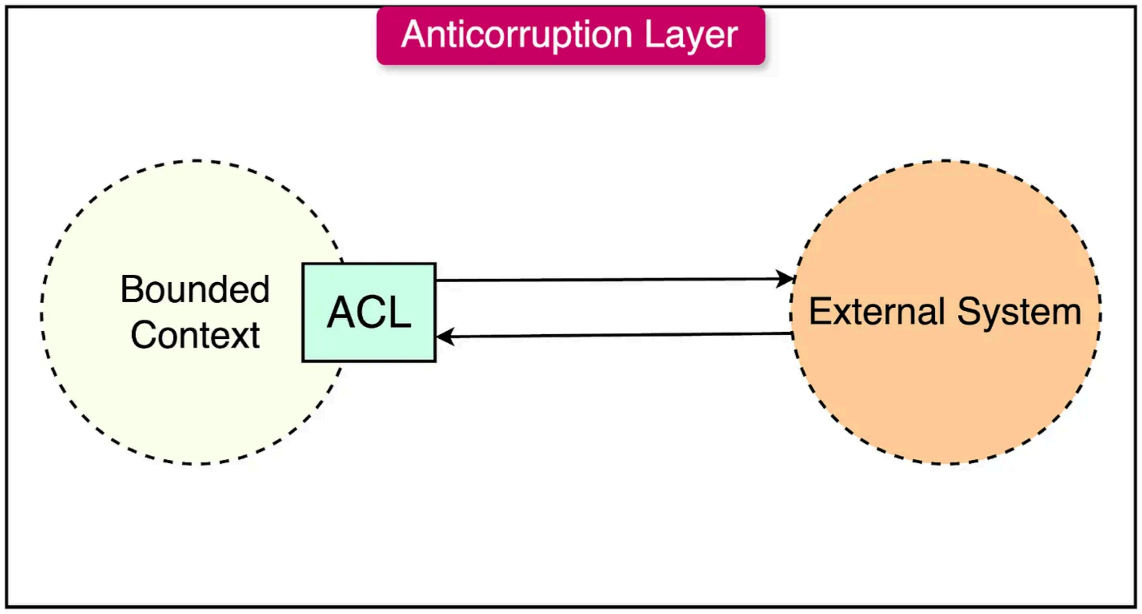


3 - Anticorruption Layer

When integrating a new system with a legacy system or an external partner's system that has an incompatible or undesirable model, it's essential to protect the integrity and purity of the new system's model.

This is where the Anticorruption Layer pattern comes into play.

The Anticorruption Layer acts as an insulating layer between the new system and the existing system or external partner's system. It provides a translation mechanism that allows the new system to interact with the other system without compromising its own model and design principles.



Here’s how it works:

- The new system sends requests to the Anticorruption Layer instead of directly interacting with the legacy or external system.
- The Anticorruption Layer receives the requests from the new system and translates them into a format compatible with the legacy or external system’s model. It communicates with the external system using these translated requests.
- When the external system responds, the Anticorruption Layer translates the response back into the format that aligns with the new system’s domain model before letting it through.

Summary

In this article, we’ve learned the fundamentals of Domain-Driven Design and how it can help you design systems based on the core domain.

Let’s summarize the learnings in brief:

- DDD focuses on creating a rich domain model that reflects a deep understanding of the business domain. The domain model is based on input from domain experts and serves as a conceptual framework for the software.
- A ubiquitous language is cultivated based on the domain model. This language is used consistently by developers and domain experts in all communication - in conversations, documentation, and the code itself. Using the ubiquitous language helps keep the model and implementation aligned.
- The software design is driven by the domain model, rather than technical concerns. This is called model-driven design.
- DDD manages complexity by isolating the domain layer from infrastructure and user interface concerns and dividing large models into Bounded Contexts.
- To build a domain model, DDD utilizes several patterns such as layered architecture, entities, value objects, services, aggregates, repositories, and factories.
- For more complex domains, strategic design patterns like bounded contexts are leveraged.
- To manage relationships between multiple Bounded Contexts, techniques like context maps, shared kernels, and anticorruption layers are used.

319 Likes · 21 Restacks

Discussion about this post

Comments Restacks

Write a comment...

Ruslan Aug 26

Thank you for the great overview of the Domain-Driven Design concepts!

It would be even better if you included Domain Events as a part of tactical DDD components or patterns. Though it's a newer concept it still plays a crucial role in the domain model when raising business logic events in event-driven systems.

LIKE (5) REPLY SHARE