Software development processPart of a series onSoftware developmentCore activitiesData modelingProcessesRequirementsDesignConstructionEngineeringTestingDebuggingDeploymentMaintenanceParadigms and modelsAgileCleanroomIncrementalPrototypingSpiralV modelWaterfallMethodologies and frameworksASDDADDevOpsDSDMFDDIIDKanbanLean SDLeSSMDDMSFPSPRADRUPSAFeScrumSEMATTDDTSPUPXPSupporting disciplinesConfiguration managementDeployment managementDocumentationProject managementQuality assuranceUser experiencePracticesATDDBDDCCCDCIDDDPPSBEStand-upTDDToolsBuild automationCompilerDebuggerGUI builderIDEInfrastructure as codeProfilerRelease automationUML ModelingStandards and bodies of knowledgeCMMIIEEE standardsIREBISO 9001ISO/IEC standardsITILOMGPMBOKSWEBOKGlossariesArtificial intelligenceComputer scienceElectrical and electronics engineeringOutlinesOutline of software developmentvteDomain-driven design (DDD) is a major software design approach,[1] focusing on modeling software to match a domain according to input from that domain's experts.[2] DDD is against the idea of having a single unified model; instead it divides a large system into bounded contexts, each of which have their own model.[3][4] Under domain-driven design, the structure and language of software code (class names, class methods, class variables) should match the business domain. For example: if software processes loan applications, it might have classes like "loan application", "customers", and methods such as "accept offer" and "withdraw".Domain-driven design is predicated on the following goals:placing the project's primary focus on the core domain and domain logic layer;basing complex designs on a model of the domain;initiating a creative collaboration between technical and domain experts to iteratively refine a conceptual model that addresses particular domain problems.Critics of domain-driven design argue that developers must typically implement a great deal of isolation and encapsulation to maintain the model as a pure and helpful construct. While domain-driven design provides benefits such as maintainability, Microsoft recommends it only for complex domains where the model provides clear benefits in formulating a common understanding of the domain.[5]The term was coined by Eric Evans in his book of the same name published in 2003.[3]Domain-driven design articulates a number of high-level concepts and practices.[3] Of primary importance is a domain of the software, the subject area to which the user applies a program. Software's developers build a domain model: a system of abstractions that describes selected aspects of a domain and can be used to solve problems related to that domain. These aspects of domain-driven design aim to foster a common language shared by domain experts, users, and developersthe ubiquitous language. The ubiquitous language is used in the domain model and for describing system requirements.Ubiquitous language is one of the pillars of DDD together with strategic design and tactical design.In domain-driven design, the domain layer is one of the common layers in an object-oriented multilayered architecture.This article needs additional citations for verification. Please help improve this article by adding citations to reliable sources. Unsourced material may be challenged and removed.Find sources:"Domain-driven design"news newspapers books scholar JSTOR (July 2023) (Learn how and when to remove this message)Domain-driven design recognizes multiple kinds of models. For example, an entity is an object defined not by its attributes, but its identity. As an example, most airlines assign a unique number to seats on every flight: this is the seat's identity. In contrast, a value object is an immutable object that contains attributes but has no conceptual identity. When people exchange business cards, for instance, they only care about the information on the card (its attributes) rather than trying to distinguish between each unique card.Models can also define events (something that happened in the past). A domain event is an event that domain experts care about. Models can be bound together by a root entity to become an aggregate. Objects outside the aggregate are allowed to hold references to the root but not to any other object of the aggregate. The aggregate root checks the consistency of changes in the aggregate. Drivers do not have to individually control each wheel of a car, for instance: they simply drive the car. In this context, a car is an aggregate of several other objects (the engine, the brakes, the headlights, etc.).In domain-driven design, an object's creation is often separated from the object itself.A repository, for instance, is an object with methods for retrieving domain objects from a data store (e.g. a database). Similarly, a factory is an object with methods for directly creating domain objects.When part of a program's functionality does not conceptually belong to any object, it is typically expressed as a service.There are different types of events in DDD, and opinions on their classification may vary. According to Yan Cui, there are two key categories of events: [6]Domain events signify important occurrences within a specific business domain. These events are restricted to a bounded context and are vital for preserving business logic. Typically, domain events have lighter payloads, containing only the necessary information for processing. This is because event listeners are generally within the same service, where their requirements are more clearly understood.[6]On the other hand, integration events serve to communicate changes across different bounded contexts. They are crucial for ensuring data consistency throughout the entire system. Integration events tend to have more complex payloads with additional attributes, as the needs of potential listeners can differ significantly. This often leads to a more thorough approach to communication, resulting in overcommunication to ensure that all relevant information is effectively shared.[6]Context Mapping identifies and defines the boundaries of different domains or subdomains within a larger system. It helps visualize how these contexts interact and relate to each other. Below are some patterns, according to Eric Evans:[7]Partnership: "forge a partnership between the teams in charge of the two contexts. Institute a process for coordinated planning of development and joint management of integration", when "teams in two contexts will succeed or fail together"Shared Kernel: "Designate with an explicit boundary some subset of the domain model that the teams agree to share. Keep this kernel small."Customer/Supplier Development: "Establish a clear customer/supplier relationship between the two teams", when "two teams are in [a] upstream-downstream relationship"Conformist: "Eliminate the complexity of translation [...] choosing conformity enormously simplifies integration", when a custom interface for a downstream subsystem isn't likely to happenAnticorruption Layer: "create an isolating layer to provide your system with functionality of the upstream system in terms of your own domain model"Open-host Service: "a protocol that gives access to your subsystem as a set of services", in case it's necessary to integrate one subsystem with many others, making custom translations between subsystems infeasiblePublished Language: "a well-documented shared language that can express the necessary domain information as a common medium of communication", e.g. data interchange standards in various industriesSeparate Ways": "a bounded context [with] no connection to the others at all, allowing developers to find simple, specialized solutions within this small scope"Big Ball of Mud[8]: "a boundary around the entire mess" when there's no real boundaries to be found when surveying an existing systemAlthough domain-driven design is not inherently tied to object-oriented approaches, in practice, it exploits the advantages of such techniques. These include entities/aggregate roots as receivers of commands/method invocations, the encapsulation of state within foremost aggregate roots, and on a higher architectural level, bounded contexts.As a result, domain-driven design is often associated with Plain Old Java Objects and Plain Old CLR Objects, which are technical implementation details, specific to Java and the .NET Framework respectively. These terms reflect a growing view that domain objects should be defined purely by the business behavior of the domain, rather than by a more specific technology framework.Similarly, the naked objects pattern holds that the user interface can simply be a reflection of a good enough domain model. Requiring the user interface to be a direct reflection of the domain model will force the design of a better domain model.[9]Domain-driven design has influenced other approaches to software development.Domain-specific modeling, for instance, is domain-driven design applied with domain-specific languages. Domain-driven design does not specifically require the use of a domain-specific language, though it could be used to help define a domain-specific language and support domain-specific multimodeling.In turn, aspect-oriented programming makes it easy to factor out technical concerns (such as security, transaction management, logging) from a domain model, letting them focus purely on the business logic.While domain-driven design is compatible with model-driven engineering and model-driven architecture,[10] the intent behind the two concepts is different. Model-driven architecture is more concerned with translating a model into code for different technology platforms than defining better domain models. However, the techniques provided by model-driven engineering (to model domains, to create domain-specific languages to facilitate the communication between domain experts and developers,...) facilitate domain-driven design in practice and help practitioners get more out of their models. Thanks to model-driven engineering's model transformation and code generation techniques, the domain model can be used to generate the actual software system that will manage it.[11]Command Query Responsibility Segregation (CQRS) is an architectural pattern for separating reading data (a 'query') from writing to data (a 'command'). CQRS derives from Command and Query Separation (CQS), coined by Bertrand Meyer.Commands mutate state and are approximately equivalent to method invocation on aggregate roots or entities. Queries read state but do not mutate it. While CQRS does not require domain-driven design, it makes the distinction between commands and queries explicit with the concept of an aggregate root. The idea is that a given aggregate root has a method that corresponds to a command and a command handler invokes the method on the aggregate root. The aggregate root is responsible for performing the logic of the operation and either yielding a failure response or just mutating its own state that can be written to a data store. The command handler pulls in infrastructure concerns related to saving the aggregate root's state and creating needed contexts (e.g., transactions).Event storming is a collaborative, workshop-based modeling technique which can be used as a precursor in the context of Domain-Driven Design (DDD) to identify and understand domain events. This interactive discovery process involves stakeholders, domain experts, and developers working together to visualize the flow of domain events, their causes, and their effects, fostering a shared understanding of the domain. The technique often uses color-coded sticky notes to represent different elements, such as domain events, aggregates, and external systems, facilitating a clear and structured exploration of the domain. Event storming can aid in discovering subdomains, bounded contexts, and aggregate boundaries, which are key constructs in DDD. By focusing on 'what happens' in the domain, the technique can help uncover business processes, dependencies, and interactions, providing a foundation for implementing DDD principles and aligning system design with business goals. [12][13] Event sourcing is an architectural pattern in which entities track their internal state not by means of direct serialization or object-relational mapping, but by reading and committing events to an event store. When event sourcing is combined with CQRS and domain-driven design, aggregate roots are responsible for validating and applying commands (often by having their instance methods invoked from a Command Handler), and then publishing events. This is also the foundation upon which the aggregate roots base their logic for dealing with method invocations. Hence, the input is a command and the output is one or many events which are saved to an event store, and then often published on a message broker for those interested in (such as an application's view).Modeling aggregate roots to output events can isolate internal state even further than when projecting read-data from entities, as in standard n-tier data-passing architectures. One significant benefit is that automatic theorem provers (e.g. Microsoft Contracts and CHESS[14]) are easier to apply, as the aggregate root comprehensively hides its internal state. Events are often persisted based on the version of the aggregate root instance, which yields a domain model that synchronizes in distributed systems through optimistic concurrency.A bounded context, a fundamental concept in Domain-Driven Design (DDD), defines a specific area within which a domain model is consistent and valid, ensuring clarity and separation of concerns. [15] In microservices architecture, a bounded context often maps to a microservice, but this relationship can vary depending on the design approach. A one-to-one relationship, where each bounded context is implemented as a single microservice, is typically ideal as it maintains clear boundaries, reduces coupling, and enables independent deployment and scaling. However, other mappings may also be appropriate: a one-to-many relationship can arise when a bounded context is divided into multiple microservices to address varying scalability or other operational needs, while a many-to-one relationship may consolidate multiple bounded contexts into a single microservice to simplify operations or minimize operational overhead. The choice of relationship should balance the principles of DDD with the system's business goals, technical constraints, and operational requirements. [16]Although domain-driven design does not depend on any particular tool or framework, notable examples include:Actifsource, a plug-in for Eclipse which enables software development combining DDD with model-driven engineering and code generation.Context Mapper, a Domain-specific language and tools for strategic and tactic DDD.[17]CubicWeb, an open source semantic web framework entirely driven by a data model. High-level directives allow to refine the data model iteratively, release after release. Defining the data model is enough to get a functioning web application. Further work is required to define how the data is displayed when the default views are not sufficient.OpenMDX, an open-source, Java-based, MDA Framework supporting Java SE, Java EE, and .NET. OpenMDX differs from typical MDA frameworks in that "use models to directly drive the runtime behavior of operational systems".Restful Objects, a standard for mapping a Restful API onto a domain object model (where the domain objects may represent entities, view models, or services). Two open source frameworks (one for Java, one for .NET) can create a Restful Objects API from a domain model automatically, using reflection.Data mesh, a domain-oriented data architectureEvent stormingKnowledge representationOntology (information science)Semantic analysis (knowledge representation)Semantic networksSemanticsC4 modelStrongly typed identifierIntegrated designSystems science^ Millet, Scott; Tune, Nick (2015). Patterns, Principles, and Practices of Domain-Driven Design. Indianapolis: Wrox. ISBN978-1-118-71470-6.^ Vernon, Vaughn (2013). Implementing Domain-Driven Design. Upper Sadle River, NJ: Addison-Wesley. p. 3. ISBN978-0-321-83457-7.^ a b c Evans, Eric (August 22, 2003). Domain-Driven Design: Tackling Complexity in the Heart of Software. Boston: Addison-Wesley. ISBN978-032-112521-7. Retrieved 2012-08-12.^ martinekuan. "Using tactical DDD to design microservices - Azure Architecture Center". learn.microsoft.com. Retrieved 2024-09-07.^ Microsoft Application Architecture Guide, 2nd Edition. Retrieved from a b c Cui, Yan. Serverless Architectures on AWS. Manning. ISBN978-1617295423.^ Evans, Eric. Domain-Driven Design Reference: Definitions and Pattern Summaries. ISBN978-1457501197.^ Foote, Brian; Yoder, Joseph (1999), Big Ball of Mud, retrieved 2025-05-09^ Haywood, Dan (2009), Domain-Driven Design using Naked Objects, Pragmatic Programmers.^ MDE can be regarded as a superset of MDA^ Cabot, Jordi (2017-09-11). "Comparing Domain-Driven Design with Model-Driven Engineering". Modeling Languages. Retrieved 2021-08-05.^ Learning Domain-Driven Design: Aligning Software Architecture and Business Strategy. ISBN978-1098100131.^ Open Agile ArchitectureTM - A Standard of The Open Group. ISBN9789401807265.^ a MS bug finding tool^ Fundamentals of Software Architecture: An Engineering Approach. O'Reilly Media. 2020. ISBN978-1492043454.^ Building Microservices by Sam Newman. ISBN978-1492034025.^ Stefan Kapferer and Olaf Zimmermann: Domain-driven Service Design - Context Modeling, Model Refactoring and Contract Generation, 14th Symposium and Summer School On Service-Oriented Computing (SommerSoC 2020)[1]Domain Driven Design, Definitions and Pattern Summaries (PDF), Eric Evans, 2015DDD Crew on GitHub: Bounded Context Canvas, Aggregate Canvas, Modeling Process and more repositoriesAn Introduction to Domain Driven Design, Methods & toolsImplementing Aggregate root in C# languageContext Mapper: A Modeling Framework for Strategic Domain-driven Design (Tool, Tutorials, DDD Modeling Examples)Strategic DDC Activity in Design Practice Repository (DPR) and Tactic DDC Activity in Design Practice Repository (DPR)Retrieved from " Design (DDD) is a software development approach that places the primary focus on the business domain and the core business logic, aiming to build a system that truly reflects the complex reality of the business it supports. This approach helps align software architecture with business needs and promotes a modular, maintainable, and adaptable codebase. This guide covers the core principles of DDD, its layered architecture, a practical folder structure, code examples, and the benefits DDD brings to a complex application. Ubiquitous Language: A shared language used by both developers and business stakeholders to describe domain concepts consistently.Bounded Context: Segregating the domain into distinct boundaries (contexts) to prevent overlap and confusion.Entities: Objects with a unique identity that persists over time (e.g., Customer).Value Objects: Immutable objects with no identity, representing descriptive aspects (e.g., Address).Aggregates and Aggregate Roots: Collections of related entities and value objects that form a consistent boundary.Repositories: Interfaces that abstract data access logic, allowing retrieval and storage of aggregates.Domain Services: Implement business operations that don't naturally belong to a single entity or value object.DDD is typically implemented with a layered architecture, where each layer has a specific responsibility. This separation allows the application to be modular and easier to maintain.Domain Layer: Contains core business logic, domain entities, value objects, and domain services.Application Layer: Defines specific use cases and workflows, coordinating between the domain and other layers.Infrastructure Layer: Handles technical details such as database access, external APIs, and file systems.Presentation Layer: Manages user interactions, such as web controllers and user interfaces.Below is a sample folder structure for an e-commerce application, organized according to DDD principles:ecommerce-app/ src/ domain/ entities/ Customer.ts # Customer entity Address.ts # Address value object repositories/ CustomerRepository.ts # Repository interface orders/ Order.ts # Order entity OrderLine.ts # OrderLine entity OrderRepository.ts # Repository interface products/ Product.ts # Product entity ProductRepository.ts # Repository interface application/ RegisterCustomerUseCase.ts # Use case for registering a customer PlaceOrderUseCase.ts # Use case for placing an order infrastructure/ database/ CustomerRepositoryImpl.ts # Implementation of customer repository OrderRepositoryImpl.ts # Implementation of order repository http/ files/ presentation/ controllers/ CustomerController.ts # Handles customer-related HTTP requests OrderController.ts # Handles order-related HTTP requests views/ CustomerView.tsx # UI for displaying customer info OrderView.tsx # UI for displaying order info routes/ index.ts # Routes configuration shared/ tests/ 1. Domain LayerThe domain layer is the core of the application, containing the essential business logic, entities, and repositories.Example: Customer EntityIn an e-commerce app, a Customer entity may contain personal details and domain logic specific to customers.export class Customer { constructor(public id: string, public name: string, public email: string) {} public updateEmail(newEmail: string): void { this.email = newEmail; } }Address Value ObjectA Value Object does not have an identity and is used to describe an entity. Here's an example of an Address value object:export class Address { constructor(public street: string, public city: string, public zipCode: string) {} public toString(): string { return `${this.street}, ${this.city}, ${this.zipCode}`; } }Customer Repository InterfaceRepositories abstract data access, making it possible to swap out database implementations without affecting domain logic.import { Customer } from './Customer'; export interface CustomerRepository { findById(id: string): Promise; save(customer: Customer): Promise; }The Application Layer defines use cases that orchestrate domain logic.Example: RegisterCustomerUseCaseA RegisterCustomerUseCase coordinates the customer registration process.import { Customer } from '../domain/customers/Customer';import { CustomerRepository } from '../domain/customers/CustomerRepository'; export class RegisterCustomerUseCase { constructor(private customerRepository: CustomerRepository) {} public async execute(name: string, email: string): Promise { const customer = new Customer(null, name, email); await this.customerRepository.save(customer); } }The Infrastructure Layer provides technical implementation, such as interacting with databases, external APIs, and infrastructure resources.Example: CustomerRepository ImplementationThis class implements the data access methods defined in the repository interface.import { Customer } from '../../domain/customers/Customer';import { CustomerRepository } from '../../domain/customers/CustomerRepository'; export class CustomerRepositoryImpl implements CustomerRepository { public async findById(id: string): Promise { } public async save(customer: Customer): Promise { } }The Presentation Layer manages user interactions and external interfaces. It includes controllers for handling requests and views for rendering information to users.Example: CustomerControllerA CustomerController handles customer-related HTTP requests, delegating the actual work to use cases.import { RegisterCustomerUseCase } from '../../application/RegisterCustomerUseCase'; export class CustomerController { constructor(private registerCustomerUseCase: RegisterCustomerUseCase) {} public async register(req, res) { const { name, email } = req.body; try { await this.registerCustomerUseCase.execute(name, email); res.status(201).json({ message: 'Customer registered successfully' }); } catch (error) { res.status(400).json({ error: error.message }); } } }Routes ConfigurationRoutes link URL paths to controller actions, defining API endpoints.import express from 'express';import { CustomerController } from '../controllers/CustomerController'; const router = express.Router();const customerController = new CustomerController(); router.post('/customers/register', (req, res) => customerController.register(req, res)); export default router;Customer View (React Component)In applications with a frontend, the Presentation Layer may include user-facing components like this CustomerView.import React from 'react'; export const CustomerView = ({ customer }) => (Customer Information Name: {customer.name} Email: {customer.email});Separation of Concerns: Each layer has a dedicated responsibility, reducing coupling and making the codebase easier to understand and maintain.Modularity: Independent layers make it possible to scale different aspects of the application, add features, and swap out dependencies without major restructuring.Testability: Each layer can be tested in isolation, promoting easier testing of complex business logic and domain behavior.Adaptability: DDD makes it easy to adapt to changing business requirements, as the architecture is aligned with domain concepts and bounded contexts.Alignment with Business Goals: By modeling core business concepts directly in code, DDD ensures that the application is closely aligned with business objectives.Conclusion In summary, Domain-Driven Design provides a robust framework for developing complex applications with a clear structure and focus on domain logic. For complex domains like e-commerce, DDD not only helps in organizing code but also in delivering a maintainable, scalable, and flexible architecture. Domain-Driven Design (DDD) is a method that prioritizes understanding and modeling the specific problem area where a software system functions. It highlights the need for close collaboration with domain experts to gain a thorough understanding of the domain's details and complexities. DDD offers principles, patterns, and practices to help developers accurately capture and represent domain concepts in their software designs.What is Domain-Driven Design (DDD)?DomainThis refers to the specific subject area or problem that the software system aims to address. For instance, in a banking application, the domain involves concepts like accounts, transactions, customers, and relevant banking regulations.Driven"Driven" means that the design of the software system is influenced by the features and needs of the domain. This indicates that design choices are based on a solid understanding of the domain, rather than just technical aspects or implementation details.Design"Design" is the process of making a plan or blueprint of a software system.This includes how different components will interact and how the system will meet its functional and non-functional requirements.Domain-Driven Design is a concept introduced by a programmer Eric Evans in 2004 in his book Domain-Driven Design: Tackling Complexity in Heart of Software.Importance of Domain KnowledgeSuppose we have designed software using all the latest tech stack and infrastructure and our software design architecture is amazing, but when we release this software in the market, it is ultimately the end user who decides whether our system is great or not. Also if the system does not solve business needs, then it is of no use to anyone. No matter how pretty it looks or how well the architecture its infrastructure are. According to Eric Evans, When we are developing software our focus should not be primarily on technology, rather it should be primarily on business. Remember, It is not the customer's job to know what they want" - Steve JobsStrategic Design in Domain-Driven Design(DDD)Strategic Design in Domain-Driven Design (DDD) focuses on defining the overall architecture and structure of a software system in a way that aligns with the problem domain. It addresses high-level concerns such as how to organize domain concepts, how to partition the system into manageable parts, and how to establish clear boundaries between different components. Let us see some key concepts within Strategic Design in Domain-Driven Design(DDD):1. Bounded ContextsA specific area within a problem domain where a particular model or language is consistently used.Sets clear boundaries for terms that may have different meanings in different parts of the system.Allows teams to develop models specific to each context, reducing confusion and inconsistency.Breaks down large, complex domains into smaller, more manageable parts.2. Context MappingThe process of defining relationships and interactions between different Bounded Contexts.Identifies areas where contexts overlap or integrate.Establishes clear communication and agreements between different contexts.Ensures different parts of the system can work together effectively while maintaining boundaries.Includes methods like Partnership, Shared Kernel, and Customer-Supplier for effective mapping3. Strategic PatternsGeneral guidelines for organizing the architecture of a software system in alignment with the problem domain.Helps tackle common challenges in designing complex systems and provides proven approaches for effective structuring.Includes patterns like Aggregates, Domain Events, and Anti-Corruption Layer.Offers solutions to recurring problems in domain-driven design and ensures the architecture accurately reflects underlying domain concepts.A strategic pattern that identifies common areas between different Bounded Contexts and establishes a shared subset of the domain model.This shared subset (or kernel) enables collaboration and integration while allowing each context to maintain its own distinct model.Should be used carefully, as it introduces dependencies between contexts that can lead to coupling if not managed properly.5. Anti-Corruption Layer (ACL)A strategic pattern designed to protect a system from the influence of external or legacy systems that use different models or languages.Acts as a translation layer between the external system and the core domain model.Transforms data and messages to ensure compatibility between systems.Keeps the core domain model pure and focused on the problem domain while allowing necessary integration with external systems.6. Ubiquitous LanguageUbiquitous Language is a shared vocabulary that all stakeholders use consistently during software development, effectively capturing the relevant domain knowledge. Key principles include:The main goal is to create a common understanding among team members, which helps everyone communicate more clearly about domain concepts and requirements.It emphasizes the use of precise terms that have clear meanings, ensuring everyone is on the same page.The language closely mirrors the terminology used in the business context, making sure the software accurately reflects real-world processes.Tactical Design Patterns in Domain-Driven Design (DDD)In Domain-Driven Design (DDD), tactical design patterns are specific strategies or techniques used to structure and organize the domain model within a software system. These patterns help developers effectively capture the complexity of the domain, while also promoting maintainability, flexibility, and scalability. Let us see some of the key tactical design patterns in DDD:1. EntityAn entity is a domain object that has a distinct identity and lifecycle. Entities are characterized by their unique identifiers and mutable state. They encapsulate behavior and data related to a specific concept within the domain. For example, in a banking application, a BankAccount entity might have properties like account number, balance, and owner, along with methods to deposit, withdraw, or transfer funds.2. Value ObjectA value object is a type of domain object that represents a value that is conceptually unchangeable. Unlike entities, value objects lack a unique identity and are usually used to describe attributes or characteristics of entities. They are compared for equality based on their properties rather than their identity.For example, a Money value object might represent a specific amount of currency, encapsulating properties like currency type and amount.3. AggregateAn aggregate is a cluster of domain objects that are treated as a single unit for the purpose of data consistency.Aggregates consist of one or more entities and value objects, with one entity designated as the aggregate root. Aggregates enforce consistency boundaries within the domain model, ensuring that changes to related objects are made atomically. For example, in an e-commerce system, an Order aggregate might consist of entities like OrderItem and Customer, with the Order entity serving as the aggregate root.4. RepositoryRepositories separate data access logic from the domain model.They provide a consistent interface for querying and storing domain objects.Repositories hide the specifics of how data is retrieved or stored.They encapsulate the translation between domain objects and underlying data storage methods, such as databases or external services.For example, a CustomerRepository might provide methods for querying and storing Customer entities.5. FactoryA factory is a creational pattern used to encapsulate the logic for creating instances of complex domain objects. Factories abstract the process of object instantiation, allowing clients to create objects without needing to know the details of their construction. For example, a ProductFactory might be responsible for creating instances of Product entities with default configurations.6. ServiceA service is a domain object that represents a behavior or operation that does not naturally belong to any specific entity or value object. Services encapsulate domain logic that operates on multiple objects or orchestrates interactions between objects. Services are typically stateless and focus on performing specific tasks or enforcing domain rules. For example, an OrderService might provide methods for processing orders, applying discounts, and calculating shipping costs.Benefits of Domain-Driven Design(DDD)Below are the main benefits of Domain-Driven Design:Promotes effective communication among domain experts, developers, and stakeholders using a common language.Helps teams prioritize the most valuable areas of the application to meet business objectives.Encourages designs that adapt to evolving business needs and market conditions.Maintains a distinct separation between domain logic, infrastructure, and user interface.Supports well-defined domain objects for easier and more focused testing.Challenges of Domain-Driven Design (DDD)Below are the challenges of domain-driven design:DDD can introduce complexity, especially in large domains. Accurately modeling intricate business areas requires a deep understanding and careful management of ambiguity.In complex domains, aligning different models and bounded contexts can be difficult. Clear communication and coordination are essential to avoid inconsistencies.Implementing DDD may require new technologies and frameworks, complicating integration with existing systems. Addressing performance and scalability issues is crucial for successful adoption.Team members may resist DDD due to familiarity with traditional methods. Overcoming this requires effective communication and education about DDD's benefits.Use-Cases of Domain-Driven Design (DDD)Below are the use cases of domain-driven design:Finance and Banking: Models complex financial instruments and ensures system integrity for better risk management.E-commerce and Retail: Manages product catalogs and inventory for features like personalized recommendations and dynamic pricing.Healthcare and Life Sciences: Models patient records and workflows to support electronic health record systems and telemedicine.Insurance: Manages products, policies, and claims to enhance policy management and risk assessment.Real Estate and Property Management: Handles properties, leases, and tenants to enable features like property listings and lease management.Real-world Example of Domain-Driven Design (DDD)Let's understand the real-world example of Domain-Driven Design through a problem statement below:Lets say, we are developing a ride-hailing application called "RideX." The system allows users to request rides, drivers to accept ride requests, and facilitates the coordination of rides between users and drivers.1. Ubiquitous LanguageUser: Individuals who request rides through the RideX platform.Driver: Individuals who provide rides to users on the RideX platform.Ride Request: A users request for a ride, detailing the pickup location, destination, and ride preferences.Ride: A specific instance of a ride that includes pickup and drop-off locations, fare, and duration.Ride Status: Indicates the current state of a ride, such as "Requested," "Accepted," "In Progress," or "Completed."2. Bounded ContextsManages the lifecycle of rides, including handling ride requests, assigning drivers, and updating ride statuses.Oversees user authentication, registration, and features like ride history and payment methods.Manages driver authentication, registration, availability, and features like earnings and ratings.3. Entities and Value ObjectsUser Entity: Represents a registered user on the RideX platform, with properties like user ID, email, password, and payment information.Driver Entity: Represents a registered driver, including properties such as driver ID, vehicle details, and driver status.Ride Request Entity: Represents a users ride request, including properties like request ID, pickup location, destination, and ride preferences.Ride Entity: Represents an instance of a ride, detailing ride ID, pickup and drop-off locations, fare, and ride status.Location Value Object: Represents a geographical location with properties for latitude and longitude.4. AggregatesRide Aggregate: The central component is the Ride Entity, along with related entities like Ride Request, User, and Driver. This aggregate manages the lifecycle of a ride, including processing ride requests, assigning drivers, and updating ride statuses.5. RepositoryRide Repository: Provides methods for querying and storing ride-related entities, including retrieving ride details, updating ride statuses, and saving ride data in the database.6. ServicesRide Assignment Service: Responsible for assigning available drivers to ride requests, considering factors like driver availability, proximity to the pickup location, and user preferences.Payment Service: Manages payment processing for completed rides, calculating fares, handling payments, and updating payment information for users and drivers.7. Domain EventsRideRequestedEvent: Triggered when a user requests a ride, containing details about the ride request and the user ID.RideAcceptedEvent: Triggered when a driver accepts a ride request, including information like the ride ID, driver ID, and pickup location.8. Example ScenarioUser Requesting a Ride: A user inputs their pickup location, destination, and preferences. RideX creates a new ride request entity and triggers a RideRequestedEvent.Driver Accepting a Ride: A driver accepts the ride request on the RideX platform. The ride status changes to "Accepted," the driver is assigned, and a RideAcceptedEvent is triggered.Ride In Progress: Once the driver arrives at the pickup location, the ride status updates from "Accepted" to "In Progress."Ride Completion: After reaching the destination, the ride status is updated to "Completed." RideX calculates the fare, processes payment, and updates the payment information for both the user and the driver. Domain-Driven Design (DDD) is a method that prioritizes understanding and modeling the specific problem area where a software system functions. It highlights the need for close collaboration with domain experts to gain a thorough understanding of the domain's details and complexities. DDD offers principles, patterns, and practices to help developers accurately capture and represent domain concepts in their software designs.What is Domain-Driven Design (DDD)?DomainThis refers to the specific subject area or problem that the software system aims to address. For instance, in a banking application, the domain involves concepts like accounts, transactions, customers, and relevant banking regulations.Driven"Driven" means that the design of the software system is influenced by the features and needs of the domain. This indicates that design choices are based on a solid understanding of the domain, rather than just technical aspects or implementation details.Design"Design" is the process of making a plan or blueprint of a software system.This includes how different components will interact and how the system will meet its functional and non-functional requirements.Domain-Driven Design is a concept introduced by a programmer Eric Evans in 2004 in his book Domain-Driven Design: Tackling Complexity in Heart of Software.Importance of Domain KnowledgeSuppose we have designed software using all the latest tech stack and infrastructure and our software design architecture is amazing, but when we release this software in the market, it is ultimately the end user who decides whether our system is great or not. Also if the system does not solve business needs, then it is of no use to anyone. No matter how pretty it looks or how well the architecture its infrastructure are. According to Eric Evans, When we are developing software our focus should not be primarily on technology, rather it should be primarily on business. Remember, It is not the customer's job to know what they want" - Steve JobsStrategic Design in Domain-Driven Design(DDD)Strategic Design in Domain-Driven Design (DDD) focuses on defining the overall architecture and structure of a software system in a way that aligns with the problem domain. It addresses high-level concerns such as how to organize domain concepts, how to partition the system into manageable parts, and how to establish clear boundaries between different components. Let us see some key concepts within Strategic Design in Domain-Driven Design(DDD):1. Bounded ContextsA specific area within a problem domain where a particular model or language is consistently used.Sets clear boundaries for terms that may have different meanings in different parts of the system.Allows teams to develop models specific to each context, reducing confusion and inconsistency.Breaks down large, complex domains into smaller, more manageable parts.2. Context MappingThe process of defining relationships and interactions between different Bounded Contexts.Identifies areas where contexts overlap or integrate.Establishes clear communication and agreements between different contexts.Ensures different parts of the system can work together effectively while maintaining boundaries.Includes methods like Partnership, Shared Kernel, and Customer-Supplier for effective mapping3. Strategic PatternsGeneral guidelines for organizing the architecture of a software system in alignment with the problem domain.Helps tackle common challenges in designing complex systems and provides proven approaches for effective structuring.Includes patterns like Aggregates, Domain Events, and Anti-Corruption Layer.Offers solutions to recurring problems in domain-driven design and ensures the architecture accurately reflects underlying domain concepts.A strategic pattern that identifies common areas between different Bounded Contexts and establishes a shared subset of the domain model.This shared subset (or kernel) enables collaboration and integration while allowing each context to maintain its own distinct model.Should be used carefully, as it introduces dependencies between contexts that can lead to coupling if not managed properly.5. Anti-Corruption Layer (ACL)A strategic pattern designed to protect a system from the influence of external or legacy systems that use different models or languages.Acts as a translation layer between the external system and the core domain model.Transforms data and messages to ensure compatibility between systems.Keeps the core domain model pure and focused on the problem domain while allowing necessary integration with external systems.6. Ubiquitous LanguageUbiquitous Language is a shared vocabulary that all stakeholders use consistently during software development, effectively capturing the relevant domain knowledge. Key principles include:The main goal is to create a common understanding among team members, which helps everyone communicate more clearly about domain concepts and requirements.It emphasizes the use of precise terms that have clear meanings, ensuring everyone is on the same page.The language closely mirrors the terminology used in the business context, making sure the software accurately reflects real-world processes.Tactical Design Patterns in Domain-Driven Design (DDD)In Domain-Driven Design (DDD), tactical design patterns are specific strategies or techniques used to structure and organize the domain model within a software system. These patterns help developers effectively capture the complexity of the domain, while also promoting maintainability, flexibility, and scalability. Let us see some of the key tactical design patterns in DDD:1. EntityAn entity is a domain object that has a distinct identity and lifecycle. Entities are characterized by their unique identifiers and mutable state. They encapsulate behavior and data related to a specific concept within the domain. For example, in a banking application, a BankAccount entity might have properties like account number, balance, and owner, along with methods to deposit, withdraw, or transfer funds.2. Value ObjectA value object is a type of domain object that represents a value that is conceptually unchangeable. Unlike entities, value objects lack a unique identity and are usually used to describe attributes or characteristics of entities. They are compared for equality based on their properties rather than their identity.For example, a Money value object might represent a specific amount of currency, encapsulating properties like currency type and amount.3. AggregateAn aggregate is a cluster of domain objects that are treated as a single unit for the purpose of data consistency.Aggregates consist of one or more entities and value objects, with one entity designated as the aggregate root. Aggregates enforce consistency boundaries within the domain model, ensuring that changes to related objects are made atomically. For example, in an e-commerce system, an Order aggregate might consist of entities like OrderItem and Customer, with the Order entity serving as the aggregate root.4. RepositoryRepositories separate data access logic from the domain model.They provide a consistent interface for querying and storing domain objects.Repositories hide the specifics of how data is retrieved or stored.They encapsulate the translation between domain objects and underlying data storage methods, such as databases or external services.For example, a CustomerRepository might provide methods for querying and storing Customer entities.5. FactoryA factory is a creational pattern used to encapsulate the logic for creating instances of complex domain objects. Factories abstract the process of object instantiation, allowing clients to create objects without needing to know the details of their construction. For example, a ProductFactory might be responsible for creating instances of Product entities with default configurations.6. ServiceA service is a domain object that represents a behavior or operation that does not naturally belong to any specific entity or value object. Services encapsulate domain logic that operates on multiple objects or orchestrates interactions between objects. Services are typically stateless and focus on performing specific tasks or enforcing domain rules. For example, an OrderService might provide methods for processing orders, applying discounts, and calculating shipping costs.Benefits of Domain-Driven Design(DDD)Below are the main benefits of Domain-Driven Design:Promotes effective communication among domain experts, developers, and stakeholders using a common language.Helps teams prioritize the most valuable areas of the application to meet business objectives.Encourages designs that adapt to evolving business needs and market conditions.Maintains a distinct separation between domain logic, infrastructure, and user interface.Supports well-defined domain objects for easier and more focused testing.Challenges of Domain-Driven Design (DDD)Below are the challenges of domain-driven design:DDD can introduce complexity, especially in large domains. Accurately modeling intricate business areas requires a deep understanding and careful management of ambiguity.In complex domains, aligning different models and bounded contexts can be difficult. Clear communication and coordination are essential to avoid inconsistencies.Implementing DDD may require new technologies and frameworks, complicating integration with existing systems. Addressing performance and scalability issues is crucial for successful adoption.Team members may resist DDD due to familiarity with traditional methods. Overcoming this requires effective communication and education about DDD's benefits.Use-Cases of Domain-Driven Design (DDD)Below are the use cases of domain-driven design:Finance and Banking: Models complex financial instruments and ensures system integrity for better risk management.E-commerce and Retail: Manages product catalogs and inventory for features like personalized recommendations and dynamic pricing.Healthcare and Life Sciences: Models patient records and workflows to support electronic health record systems and telemedicine.Insurance: Manages products, policies, and claims to enhance policy management and risk assessment.Real Estate and Property Management: Handles properties, leases, and tenants to enable features like property listings and lease management.Real-world Example of Domain-Driven Design (DDD)Let's understand the real-world example of Domain-Driven Design through a problem statement below:Lets say, we are developing a ride-hailing application called "RideX." The system allows users to request rides, drivers to accept ride requests, and facilitates the coordination of rides between users and drivers.1. Ubiquitous LanguageUser: Individuals who request rides through the RideX platform.Driver: Individuals who provide rides to users on the RideX platform.Ride Request: A users request for a ride, detailing the pickup location, destination, and ride preferences.Ride: A specific instance of a ride that includes pickup and drop-off locations, fare, and duration.Ride Status: Indicates the current state of a ride, such as "Requested," "Accepted," "In Progress," or "Completed."2. Bounded ContextsManages the lifecycle of rides, including handling ride requests, assigning drivers, and updating ride statuses.Oversees user authentication, registration, and features like ride history and payment methods.Manages driver authentication, registration, availability, and features like earnings and ratings.3. Entities and Value ObjectsUser Entity: Represents a registered user on the RideX platform, with properties like user ID, email, password, and payment information.Driver Entity: Represents a registered driver, including properties such as driver ID, vehicle details, and driver status.Ride Request Entity: Represents a users ride request, including properties like request ID, pickup location, destination, and ride preferences.Ride Entity: Represents an instance of a ride, detailing ride ID, pickup and drop-off locations, fare, and ride status.Location Value Object: Represents a geographical location with properties for latitude and longitude.4. AggregatesRide Aggregate: The central component is the Ride Entity, along with related entities like Ride Request, User, and Driver. This aggregate manages the lifecycle of a ride, including processing ride requests, assigning drivers, and updating ride statuses.5. RepositoryRide Repository: Provides methods for querying and storing ride-related entities, including retrieving ride details, updating ride statuses, and saving ride data in the database.6. ServicesRide Assignment Service: Responsible for assigning available drivers to ride requests, considering factors like driver availability, proximity to the pickup location, and user preferences.Payment Service: Manages payment processing for completed rides, calculating fares, handling payments, and updating payment information for users and drivers.7. Domain EventsRideRequestedEvent: Triggered when a user requests a ride, containing details about the ride request and the user ID.RideAcceptedEvent: Triggered when a driver accepts a ride request, including information like the ride ID, driver ID, and pickup location.8. Example ScenarioUser Requesting a Ride: A user inputs their pickup location, destination, and preferences. RideX creates a new ride request entity and triggers a RideRequestedEvent.Driver Accepting a Ride: A driver accepts the ride request on the RideX platform. The ride status changes to "Accepted," the driver is assigned, and a RideAcceptedEvent is triggered.Ride In Progress: Once the driver arrives at the pickup location, the ride status updates from "Accepted" to "In Progress."Ride Completion: After reaching the destination, the ride status is updated to "Completed." RideX calculates the fare, processes payment, and updates the payment information for both the user and the driver. In our daily development work, we often hear about DDD. But what exactly is DDD? There have been many articles online before, but most of them are lengthy and hard to understand. This article aims to give you a clearer picture of what DDD is all about. What is DDD? DDD (Domain-Driven Design) is a software development methodology for building complex systems by focusing on the business domain. Its core idea is to tightly integrate the code structure with real business needs. In one sentence: DDD is about using code to reflect the essence of the business, rather than just implementing functionality. In traditional development, we follow PRD documents and write CRUD operations step by step. In DDD, we work together with business stakeholders to build domain models. The code mirrors the business (when business changes, code adapts accordingly). Traditional Development Model: A Simple Registration Example Honestly, it's easy to forget abstract concepts after a while, right? Let's look at a code example. Suppose we have a user registration feature with the following business rules: The username must be uniqueThe password must meet complexity requirementsA log must be recorded after registration In traditional development, we might quickly write the following code:@Controllerpublic class UserController { public void register(String username, String password) { // Validate password // Check username // Save to database // Record log // All logic mixed together } } Some may say, "There's no way all the code is in the controller we should separate concerns using layers like controller, service, and DAO." So the code might look like this:// Service layer: only controls the flow, business rules are scatteredpublic class UserService { public void register(User user) { // Validation Rule 1: implemented in a utility class ValidationUtil.checkPassword(user.getPassword()); // Validation Rule 2: implemented via annotation if (userRepository.exists(user)) { ... } // Data is passed directly to DAO userDao.save(user); } } To be fair, this version has a much clearer flow. But wait, is this DDD, right? The answer is: Not DDD! In the code above the layering is clear and structurally divided, it is not DDD. In that traditional layered code, the Service layer acts as a coordinator carrier (anemic model), and the business logic is scattered across Service, Util, Controller... It looks elegant and clean that this must be DDD. Let's say: "Hey, we've already layered the code! It looks elegant and clean that this must be DDD" In DDD, some logic should be encapsulated within the domain object like password validation. For this registration example, the DDD approach (rich model) would look like this:// Domain Entity: encapsulates business logicpublic class User { public User(String username, String password) { // Password rules encapsulated in the constructor if (!isValidPassword(password)) { throw new InvalidPasswordException(); } this.username = username; this.password = encrypt(password); } // Password complexity validation is the responsibility of the entity private boolean isValidPassword(String password) { ... } } Here, the password validation is pushed down into the User domain entity. In professional terms, business rules are encapsulated inside the domain object which is no longer just a "data bag." Key Design Concepts in DDD So, is DDD just about pushing some logic into domain objects? Not exactly. Besides layering, the essence of DDD lies in deepening business expression through the following patterns: Aggregate RootDomain Service vs Application ServiceDomain Events Aggregate Root Scenario: A user (User) is associated with shipping addresses (Address) Traditional approach: manage User and Address separately in the Service layerDDD approach: treat User as the aggregate root and control the addition/removal of Address through itpublic class User { private List addresses; // The logic to add an address is controlled by the aggregate root public void addAddress(Address address) { if (addresses.size() >= 5) { throw new AddressLimitExceededException(); } addresses.add(address); }} Domain Service vs Application ServiceDomain Service: Handles business logic that spans multiple entities (e.g., transferring money between two accounts)Application Service: Coordinates the overall process (e.g., calling domain services + sending messages)// Domain Service: handles core business logicpublic class TransferService { public void transfer(Account from, Account to, Money amount) { from.debit(amount); // Debit logic is encapsulated in Account entity to.credit(amount); } } // Application Service: orchestrates the process, contains no business logicpublic class BankingAppService { public void executeTransfer(Long fromId, Long toId, BigDecimal amount) { Account from = accountRepository.findById(fromId); Account to = accountRepository.findById(toId); transferService.transfer(from, to, new Money(amount)); messageQueue.send(new TransferEvent(...)); // Infrastructure operation } } Domain Events Use events to explicitly express business state changes. Example: After a user successfully registers, trigger a UserRegisteredEventpublic class User { public void register() { // ...registration logic this.events.add(new UserRegisteredEvent(this.id)); // Record domain event } } Differences Between Traditional Development and DDD Lets briefly summarize the differences between traditional development and DDD. Traditional Development: Ownership of Business Logic: Scattered across Service, Utils, ControllersRole of the Model: Data carrier (anemic model)Impact on Technical Implementation: Schema is driven by database table design DDD: Ownership of Business Logic: Encapsulated in domain entities or domain servicesRole of the Model: Business model that carries behavior (rich model)Impact on Technical Implementation: Schema is driven by business needs A DDD Example: Placing an E-Commerce Order To help you better understand, heres a concrete DDD case to quench your thirst. Suppose theres a requirement: When placing an order, the system must: validate stock, apply coupons, calculate the actual payment, and generate an order. Traditional Implementation (Anemic Model)// Service layer: bloated order placement logicpublic class OrderService { @Autowired private InventoryDAO inventoryDAO; @Autowired private CouponDAO couponDAO; public Order createOrder(Long userId, List items, Long couponId) { // 1. Stock validation (scattered in Service) for (ItemDTO item : items) { Integer stock = inventoryDAO.getStock(item.getSkuId()); if (item.getQuantity() > stock) { throw new RuntimeException("Insufficient stock"); } } // 2. Calculate total amount BigDecimal total = items.stream() .map(i -> i.getPrice().multiply(i.getQuantity())) .reduce(BigDecimal.ZERO, BigDecimal::add); // 3. Apply coupon (logic hidden in utility class) if (couponId != null) { Coupon coupon = couponDAO.getById(couponId); total = CouponUtil.applyCoupon(coupon, total); // Discount logic is in util } // 4. Save order (pure data operation) Order order = new Order(); order.setUserId(userId); order.setTotalAmount(total); orderDAO.save(order); return order; } } Problems with the traditional approach: Stock validation and coupon logic are scattered across Service, Util, DAOThe Order object is just a data carrier (anemic), no one owns the business rulesWhen requirements change, developers have to "dig" through the Service layer DDD Implementation (Rich Model): Business Logic Encapsulated in Domain// Aggregate Root: Order (carries core logic)public class Order { private List items; private Coupon coupon; private Money totalAmount; // Business logic encapsulated in the constructor public Order(User user, List items, Coupon coupon) { // 1. Stock validation (domain rule encapsulated) items.forEach(item -> item.checkStock()); // 2. Calculate total amount (logic resides in value objects) this.totalAmount = items.stream() .map(OrderItem::subtotal) .reduce(Money.ZERO, Money::add); // 3. Apply coupon (rules encapsulated in entity) if (coupon != null) { validateCoupon(coupon, user); // Coupon rule encapsulated this.totalAmount = coupon.applyDiscount(this.totalAmount); } } // Coupon validation logic (clearly owned by the domain) private void validateCoupon(Coupon coupon, User user) { if (!coupon.isValid() || !coupon.isApplicable(user)) { throw new InvalidCouponException(); } } } // Domain Service: orchestrates the order processpublic class OrderService { public Order createOrder(User user, List items, Coupon coupon) { Order order = new Order(user, convertItems(items), coupon); orderRepository.save(order); domainEventPublisher.publish(new OrderCreatedEvent(order)); return order; } } Benefits of the DDD Approach: Stock validation: Encapsulated in the OrderItem value objectCoupon logic: Encapsulated within methods of the Order entityCalculation logic: Ensured precision by the Money value objectBusiness changes: Only require changes to the domain object Now suppose there's a new product requirement: Coupons must offer $20 off for orders over $100, and apply only to new users. With traditional development, you have to modify: CouponUtil.applyCoupon() logicThe Service layer to add new-user validation With DDD, you only need to modify: Order.validateCoupon() method in the domain layer When Should You Use DDD? So, should DDD be used in every situation? Not really that would be overengineering. When the business is complex (e.g., e-commerce, finance, ERP) When requirements change frequently (90% of internet businesses) When it's simple CRUD (admin panels, data reports) I think this quote makes a lot of sense: When you find that modifying business rules only requires changes in the domain layer, without touching the Controller or DAO thats when DDD is truly implemented. We are Leapcell, your top choice for hosting backend projects. Leapcell is the Next-Gen Serverless Platform for Web Hosting, Async Tasks, and Redis: Multi-Language Support Develop with Node.js, Python, Go, or Rust. Deploy unlimited projects for free pay only for usage no requests, no charges. Unbeatable Cost Efficiency Pay-as-you-go with no idle charges.Example: $25 supports 6.94M requests at a 60ms average response time. Streamlined Developer Experience Intuitive UI for effortless setup.Fully automated CI/CD pipelines and GitOps integration.Real-time metrics and logging for actionable insights. Effortless Scalability and High Performance Auto-scaling to handle high concurrency with ease.Zero operational overhead just focus on building. Explore more in the Documentation! Follow us on X: @LeapcellHQ Read on our blog