



Arquitectura Moderna de Servicios en Java

Círculo Siete Capacitación

Clase 5 de 30
10 Octubre 2024

¿Qué es Domain-Driven Design?

- Domain-Driven Design (DDD) es una metodología para desarrollar software centrada en modelar y resolver problemas complejos del dominio del negocio.
- Pone énfasis en la colaboración entre expertos del dominio y desarrolladores.
- Enfoque clave: Crear un modelo de dominio basado en las necesidades del negocio.

Principios Clave del DDD

- Lenguaje Ubicuo (Ubiquitous Language):
 - Un lenguaje compartido entre expertos del negocio y desarrolladores.
 - Mejora la comunicación y elimina malentendidos entre el equipo técnico y no técnico.
- Modelar según el dominio:
 - El modelo del software refleja fielmente las realidades del negocio.
 - Enfocado en capturar las reglas y comportamientos esenciales del dominio.

Colaboración con los Expertos del Dominio

- **Experto del dominio:** persona con amplio conocimiento sobre el problema que se está resolviendo.
- **Desarrollador:** transforma ese conocimiento en código.
- Colaboración continua entre ambos roles es esencial para el éxito del modelo.

Contexto Delimitado (Bounded Context)

- Define los límites donde un modelo es aplicable.
- Cada Bounded Context tiene su propio modelo de dominio.
- Evita conflictos entre modelos al establecer fronteras claras entre ellos.

Beneficios de los Contextos Delimitados

- **Escalabilidad:** Los sistemas pueden dividirse en partes manejables y autónomas.
- **Separación de preocupaciones:** Cada equipo puede trabajar en su contexto sin interferir con otros.
- **Claridad y mantenimiento:** Los límites permiten gestionar mejor el código y los cambios.

Colaboración entre Contextos

- Integración entre contextos delimitados:
- Los modelos de diferentes contextos interactúan de manera controlada.
- Técnicas como ***Context Mapping*** y ***Anti-Corruption Layer*** ayudan a gestionar esta interacción.

El Enfoque Estratégico

- DDD no solo se centra en la estructura técnica del software, sino también en las decisiones estratégicas.
- Impulsa un diseño que responde a las necesidades del negocio, optimizando el impacto del software en los resultados empresariales.

Introducción

- DDD ayuda a gestionar la complejidad de dominios empresariales avanzados.
- Al establecer un lenguaje común y dividir el sistema en contextos delimitados, permite crear software más alineado con los objetivos del negocio.

¿Qué es un Contexto Delimitado?

- Un ***Bounded Context*** establece los límites donde un modelo de dominio es válido.
- Cada contexto tiene su propio lenguaje ubicuo y lógica de negocio.
- Ayuda a gestionar la complejidad del sistema, dividiéndolo en partes manejables.

¿Por qué son importantes los límites?

- Los límites evitan que los modelos se mezclen y generen confusión.
- Clara separación de responsabilidades entre contextos.
- Facilita la colaboración entre equipos, ya que cada uno trabaja en su propio contexto.

Relación entre Contextos

- Los Bounded Contexts pueden interactuar, pero deben hacerlo de manera controlada.
- Técnicas clave para gestionar la interacción entre contextos:
 - **Context Maps:** Mapa visual de las relaciones entre los diferentes contextos.
 - **Anti-Corruption Layer:** Barrera para proteger un contexto de la complejidad de otro.

Mapa de Contextos (Context Map)

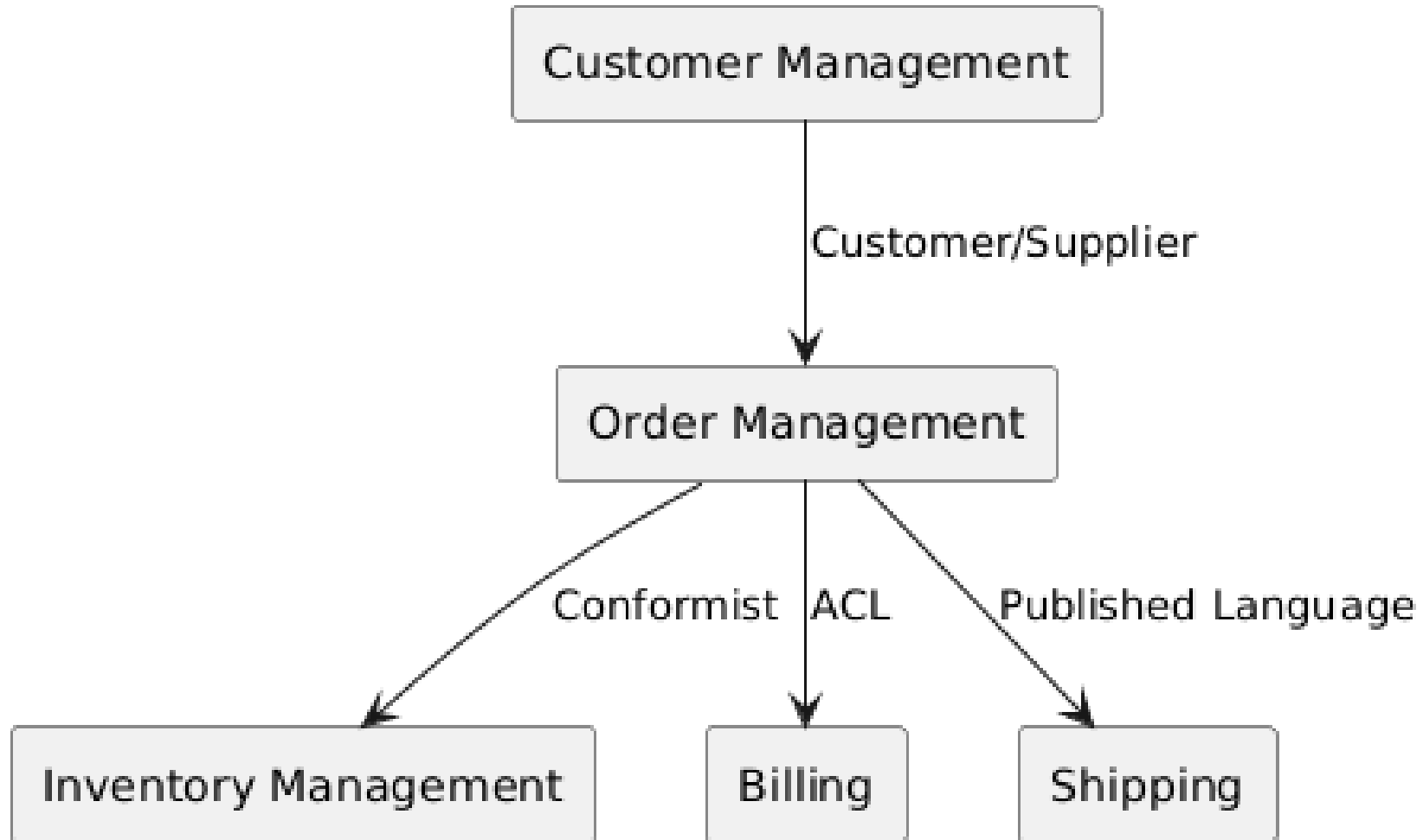
- **Context Map:** una herramienta que muestra las relaciones y dependencias entre distintos contextos.
- Proporciona una visión clara de cómo interactúan los contextos dentro del sistema.
- Identifica las zonas de integración y las posibles fuentes de conflicto entre modelos.

Context Map para un sistema de e-commerce

- **Customer Management Context:** Gestiona la información de los clientes, perfiles, historial de compras, etc.
- **Order Management Context:** Responsable de gestionar los pedidos, su creación, procesamiento y estado.
- **Inventory Management Context:** Controla el stock de productos, la disponibilidad en el almacén y las actualizaciones del inventario.
- **Billing Context:** Encargado de la facturación, pagos y cobros.
- **Shipping Context:** Gestiona el envío de los productos, seguimiento de paquetes, y comunicación con proveedores logísticos.

Relaciones y patrones del Context Map

- Customer Management → Order Management: Relación de ***Customer/Supplier***.
 - Customer Management proporciona datos del cliente cuando se crea un pedido en Order Management.
 - Esta relación indica que Order Management depende de Customer Management para obtener información actualizada del cliente.
- Order Management → Inventory Management: Relación de ***Conformist***.
 - Order Management depende directamente del modelo de inventario de Inventory Management para verificar si los productos están en stock. Aquí, Order Management acepta los términos y modelo de datos que le dicta Inventory Management sin intentar cambiar o interferir en ese modelo.
- Order Management → Billing: Relación de ***Anti-Corruption Layer (ACL)***.
 - Order Management necesita interactuar con Billing, pero los modelos internos de cada uno son diferentes. Para evitar que Order Management se "contamine" con el modelo de Billing, se crea una capa de anti-corrupción que traduce y adapta los datos entre los dos contextos.
- Order Management → Shipping: Relación de ***Published Language***.
 - Order Management publica eventos de dominio cuando un pedido está listo para ser enviado. Shipping escucha estos eventos para proceder con el proceso de envío. Ambos contextos acuerdan utilizar un lenguaje de eventos compartido (por ejemplo, un "pedido listo para enviar") para facilitar esta comunicación.



Tipos de relaciones

- Las relaciones entre los diferentes Bounded Contexts son fundamentales para definir cómo interactúan y colaboran las distintas partes del sistema.
- Eric Evans y Vaughn Vernon describen varios tipos de relaciones entre contextos, y cada una define una forma particular de acoplamiento o interacción
 - Customer/Supplier
 - Conformist
 - Anti-Corruption Layer (ACL)
 - Published Language
 - Shared Kernel
 - Separate Ways
 - Partnership
 - Open Host Service

Customer/Supplier

- Esta relación indica que un contexto (el Supplier) proporciona datos o servicios que otro contexto (el Customer) necesita para funcionar. En este caso, el Customer depende de los datos o servicios del Supplier, y el Supplier define las reglas o el contrato que debe seguir el Customer.
- Ejemplo: En un sistema de e-commerce, el Order Management necesita información del Customer Management para crear un pedido. Customer Management actúa como el Supplier, y Order Management es el Customer.
- Implicaciones: El Customer depende del Supplier para recibir datos y debe adaptarse al modelo y las políticas del Supplier. Sin embargo, los cambios en el Supplier pueden afectar al Customer, por lo que debe haber una colaboración cercana entre equipos.

Conformist

- En una relación Conformist, un contexto depende de otro y acepta el modelo y las reglas del otro contexto sin intentar imponer cambios o influir en su comportamiento. El contexto que se conforma debe adaptarse al modelo del contexto dominante.
- Ejemplo: El Order Management podría depender del Inventory Management para verificar la disponibilidad de productos. El Order Management acepta y usa el modelo de Inventory Management tal cual, sin intentar modificarlo.
- Implicaciones: El Conformist no tiene control sobre los cambios del otro contexto. Esto puede ser una solución rápida, pero también significa un fuerte acoplamiento y menos flexibilidad.

Anti-Corruption Layer (ACL)

- Cuando dos contextos necesitan interactuar, pero tienen modelos muy diferentes, se puede utilizar una Anti-Corruption Layer. Esta capa actúa como un intermediario que traduce y adapta las estructuras y comportamientos de un contexto al otro, evitando que uno "contamine" al otro.
- Ejemplo: El Order Management podría necesitar interactuar con un sistema de facturación (Billing), pero sus modelos de datos son diferentes. Para evitar que el modelo de Billing afecte el diseño del Order Management, se implementa una ACL que traduce las interacciones entre ambos.
- Implicaciones: Esta capa agrega complejidad, pero permite mantener ambos contextos independientes y proteger sus modelos internos, evitando acoplamientos indeseados.

Published Language

- En una relación de Published Language, los contextos acuerdan utilizar un lenguaje compartido o estándar para intercambiar datos o eventos. Este lenguaje puede tomar la forma de mensajes de eventos o APIs bien definidas, lo que permite la comunicación sin tener que conocer los detalles internos del otro contexto.
- Ejemplo: En un sistema de e-commerce, el Order Management podría publicar un evento llamado "Order Shipped" que Shipping escucha para comenzar el proceso de envío. Ambos contextos se ponen de acuerdo en el formato de ese evento.
- Implicaciones: Este tipo de relación reduce el acoplamiento entre los contextos, ya que dependen solo del lenguaje acordado (eventos o mensajes) y no de los detalles internos de cada uno.

Shared Kernel

- En esta relación, dos contextos comparten una pequeña parte del modelo o código, generalmente en forma de librerías o módulos comunes. El Shared Kernel contiene componentes que ambos contextos necesitan y deben mantener en conjunto, lo que requiere una fuerte colaboración entre equipos para garantizar que estos componentes compartidos sean consistentes.
- Ejemplo: Un sistema de pagos y un sistema de pedidos podrían compartir un módulo que contiene la lógica de cálculo de impuestos, que es necesaria para ambos. Esta lógica compartida sería parte del Shared Kernel.
- Implicaciones: Los equipos deben colaborar estrechamente para mantener el Shared Kernel, lo que puede generar dependencias y complejidad adicional si no se gestiona bien.

Separate Ways

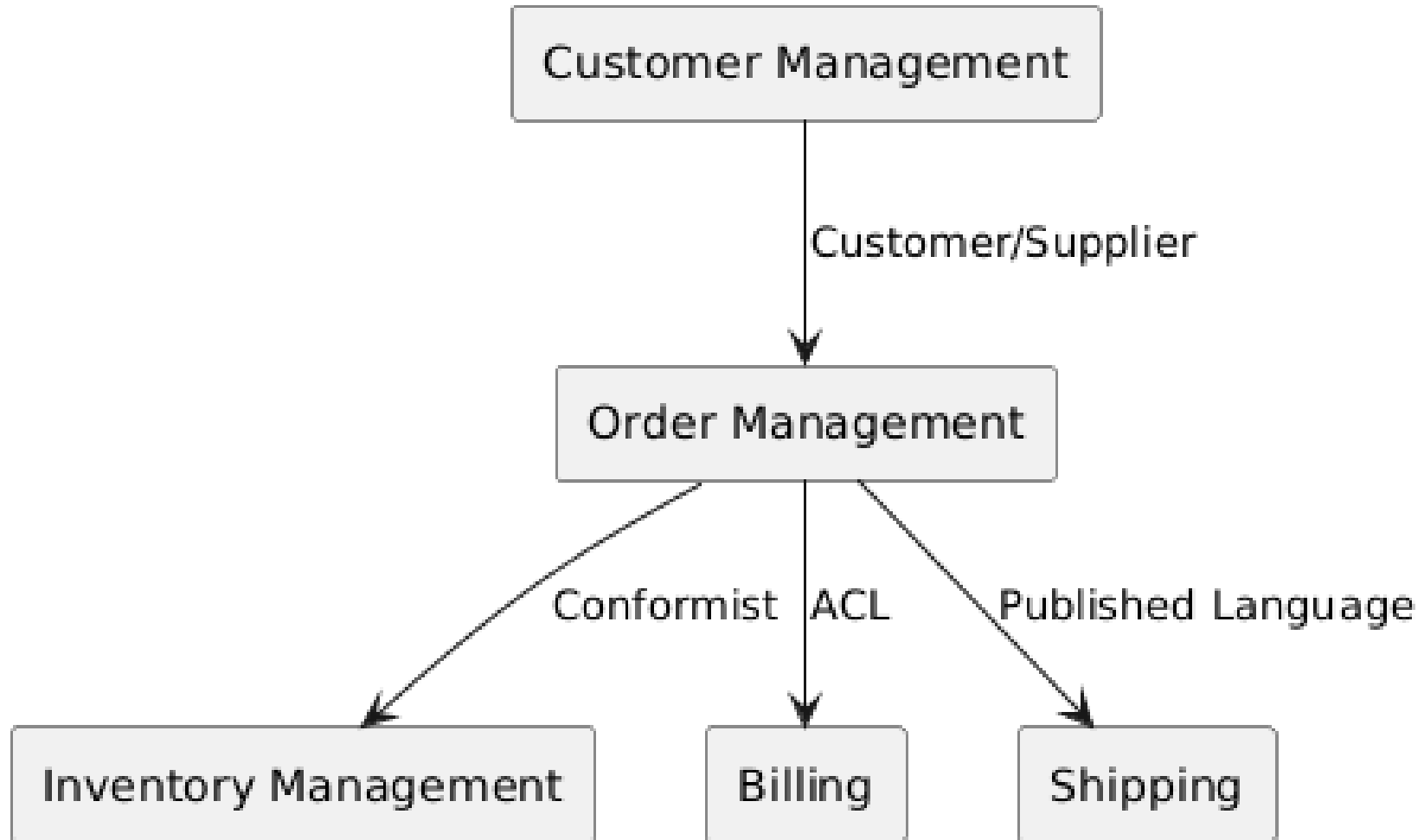
- En una relación de Separate Ways, los contextos no necesitan interactuar entre sí, y cada uno es independiente. A veces, es más eficiente que ciertos módulos sigan su propio camino y no interfieran o dependan de otros.
- Ejemplo: Un sistema de gestión de inventario y un sistema de marketing pueden no necesitar comunicarse directamente, ya que gestionan funciones completamente diferentes.
- Implicaciones: Los contextos son completamente autónomos, lo que permite mayor flexibilidad y menos acoplamiento. Sin embargo, esto solo es útil si realmente no hay necesidad de interacción entre ellos.

Partnership

- En una relación de Partnership, dos contextos trabajan muy estrechamente y de manera igualitaria para lograr un objetivo común. Ambos contextos dependen del otro, y los cambios en uno pueden afectar al otro. A diferencia de Customer/Supplier, aquí no hay un contexto dominante.
- Ejemplo: El Order Management y el Shipping pueden trabajar como socios igualitarios, ya que ambos necesitan coordinarse muy de cerca para garantizar que los pedidos se procesen y envíen correctamente.
- Implicaciones: Los equipos necesitan una colaboración constante, lo que puede resultar en una mayor coordinación y mayor riesgo de acoplamiento, pero también una integración más fluida.

Open Host Service (OHS)

- Su propósito es exponer un servicio claro y accesible para que otros contextos puedan interactuar con un determinado contexto de manera controlada y predecible.
- Este patrón permite que un contexto ofrezca su funcionalidad a otros contextos de forma pública, mediante una API o servicio, sin necesidad de que los detalles internos de su modelo de dominio sean expuestos.
- Ejemplo: Inventory Management expone un servicio que permite consultar el stock de productos, quizás con una API REST o mensajes en un sistema de eventos. Order Management y Shipping consumen este servicio para verificar si un producto está disponible antes de procesar el pedido o proceder con el envío.



Modularidad y Contextos

- Al dividir el sistema en contextos delimitados, se fomenta la modularidad.
- Cada módulo puede evolucionar de forma independiente, facilitando:
 - Mantenibilidad.
 - Escalabilidad.
 - Independencia entre equipos de desarrollo.
- Los Bounded Contexts son esenciales para gestionar la complejidad de un sistema de dominio.
- Definen límites claros que mejoran la colaboración, el diseño modular y la independencia entre los equipos.
- Técnicas como Context Maps y Anti-Corruption Layer ayudan a mantener la coherencia y la claridad en la integración entre contextos.

¿Qué es el Lenguaje Ubicuo?

- Es un lenguaje compartido entre expertos del dominio y desarrolladores.
- Evoluciona a medida que se profundiza en la comprensión del modelo de dominio.
- Sirve como base para todo el desarrollo del software y la comunicación dentro del equipo.

Importancia del Lenguaje Ubicuo

- Evita malentendidos entre el equipo técnico y los expertos del negocio.
- Refuerza la colaboración continua entre desarrolladores y expertos del dominio.
- Mejora la calidad del código, ya que los términos utilizados en el código reflejan con precisión el modelo de negocio.

Lenguaje Ubicuo en el Código

- El lenguaje ubicuo debe estar presente en todos los artefactos del software, incluyendo:
 - Clases.
 - Métodos.
 - Nombres de variables.
 - Documentación.
- Ejemplo:
 - Si en el negocio se habla de "pedido", la entidad en el código debe llamarse "Pedido" en lugar de usar términos genéricos como "Transaction" o "Order".

Colaboración con los Expertos del Dominio

- El lenguaje se desarrolla y refina a través de conversaciones continuas con los expertos del dominio.
- Las discusiones sobre el dominio deben ser traducidas directamente en modelos y código.
- Crea un ciclo de retroalimentación entre el desarrollo y el entendimiento del dominio.

Desarrollo Incremental del Lenguaje

- El lenguaje ubicuo no es estático, evoluciona conforme cambia la comprensión del dominio.
- Las iteraciones en el desarrollo y la retroalimentación del negocio permiten refinar tanto el lenguaje como el modelo.
- Un lenguaje bien definido puede ayudar a detectar inconsistencias en el modelo.

Técnicas para Desarrollar el Lenguaje Ubicuo

- Workshops de descubrimiento:
 - Sesiones colaborativas con expertos del dominio para identificar y definir términos clave.
- Documentación activa:
 - Registrar continuamente términos, reglas y relaciones del dominio conforme surgen.
- Pruebas colaborativas:
 - Utilizar pruebas unitarias y tests de aceptación para reflejar el lenguaje y comportamiento esperado.

Ventajas del Lenguaje Ubicuo

- Cohesión del equipo: Todos los miembros del equipo, incluidos los no técnicos, hablan el mismo idioma.
- Código legible: El código es más fácil de entender y mantener, ya que utiliza términos precisos del negocio.
- Menos errores: Minimiza los malentendidos y discrepancias en la implementación del modelo de negocio.

¿Qué es un Modelo de Dominio?

- Un modelo de dominio es una representación conceptual del área de negocio que está siendo implementada en el software.
- Refleja las reglas de negocio, los comportamientos y las relaciones entre los objetos del dominio.
- Se construye a través de una colaboración constante entre desarrolladores y expertos del dominio.

Entidades y Valores

- Entidades: Objetos que tienen una identidad única dentro del sistema y persisten a lo largo del tiempo.
- Ejemplo: Cliente, Pedido, Factura.
- Objetos de Valor: No tienen identidad propia, se definen por sus atributos y son inmutables.
- Ejemplo: Dirección, Dinero, Fecha.

¿Qué es un Agregado?

- Un agregado es un conjunto de entidades y objetos de valor relacionados que se comportan como una unidad de consistencia.
- Garantiza la integridad de los datos al definir los límites dentro de los cuales las operaciones deben ser consistentes.
- Cada agregado tiene una raíz de agregado que actúa como punto de entrada para todas las interacciones externas.

Diseño de Agregados

- Un agregado debe:
 - Mantenerse lo más pequeño posible.
 - Evitar la dependencia entre agregados.
 - Estar diseñado para que las operaciones puedan ser transaccionales dentro de sus límites.
- Ejemplo:
 - Un agregado Pedido puede incluir entidades como Cliente y Línea de Pedido, asegurando que cualquier cambio en el pedido sea consistente en su conjunto.

Raíz de Agregado

- La raíz de agregado es la entidad principal que controla el acceso a los demás objetos dentro del agregado.
- Solo se puede acceder a las entidades internas a través de la raíz.
- La raíz asegura la coherencia de los datos y las reglas de negocio.

Regla de Consistencia

- Los agregados garantizan que las reglas de negocio dentro de sus límites se cumplan.
- La consistencia fuerte debe mantenerse dentro de los agregados, mientras que la consistencia eventual se puede permitir entre agregados, especialmente en sistemas distribuidos.

Consejos para Diseñar Agregados

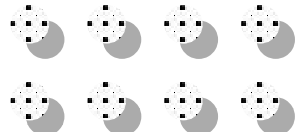
- Mantén los agregados pequeños:
 - Limita el número de entidades dentro de un agregado para reducir la complejidad y mejorar la escalabilidad.
- Revisa las transacciones:
 - Un agregado debe permitir que las operaciones sean atómicas y consistentes dentro de sus límites.
- Distingue responsabilidades:
 - Usa agregados para separar diferentes responsabilidades en el dominio, garantizando un modelo más limpio y mantenible.

Ejemplo de Diseño de Agregado

- Sistema de Comercio Electrónico:
 - Agregado: Pedido
 - Entidades internas: Cliente, Línea de Pedido
 - Objetos de Valor: Dirección de Envío, Moneda
- El agregado Pedido asegura que todas las operaciones (agregar productos, modificar cliente, calcular precio) se realicen de manera coherente.

Revisar código

- ***agregados.zip***



Agregados Conclusiones

- Los agregados son esenciales para estructurar modelos de dominio complejos y asegurar la consistencia de los datos.
- El diseño correcto de los agregados permite crear sistemas escalables, modulares y alineados con las reglas del negocio.
- La raíz del agregado juega un papel clave en la protección de la integridad del dominio.

¿Qué es un Repositorio?

- Un repositorio es una abstracción que proporciona un mecanismo para almacenar y recuperar agregados.
- Actúa como una colección en memoria de agregados, aunque en realidad interactúa con la base de datos u otro sistema de persistencia.
- Proporciona acceso a los agregados sin exponer los detalles de cómo están almacenados.

Rol del Repositorio

- Los repositorios permiten:
 - Agregar nuevos agregados.
 - Recuperar agregados existentes por su identidad.
 - Eliminar agregados.
 - Abstraer la complejidad del sistema de persistencia subyacente (bases de datos, APIs, etc.).

Interacción con los Agregados

- Los repositorios solo deben operar sobre agregados completos.
- Deben encapsular la lógica de acceso a la base de datos y garantizar que los agregados recuperados sean consistentes.
- Mantienen la integridad de los datos entre las transacciones.

Diseño de un Repositorio

- Un repositorio suele implementar métodos como:
 - ***save(Aggregate aggregate)***: guarda un agregado.
 - ***findById(AggregateId id)***: recupera un agregado por su identidad.
 - ***delete(Aggregate aggregate)***: elimina un agregado.

Relación con la Persistencia

- El repositorio abstrae la interacción con la capa de persistencia, evitando que el código de dominio conozca detalles de cómo y dónde se almacenan los agregados.
- Puede estar respaldado por:
 - Bases de datos relacionales (ORMs como Hibernate).
 - Bases de datos NoSQL.
 - Sistemas de almacenamiento distribuido.

Patrón Repositorio vs. DAO

- Repositorio:
 - Operaciones enfocadas en agregados completos.
 - Más cercano al dominio del negocio.
- DAO (Data Access Object):
 - Operaciones más granulares, enfocadas en entidades individuales o datos sin estructura.
 - Más técnico, centrado en la persistencia de datos.

Implementación del Repositorio

- Al diseñar un repositorio, se debe:
 - Asegurar que sea específico del dominio (trabajar con agregados, no con entidades aisladas).
 - Encapsular las reglas de negocio que deben aplicarse al guardar y recuperar datos.
 - Mantenerlo independiente de la infraestructura específica.
- Ejemplo:
 - Si usas JPA o Hibernate, el repositorio interactúa con el EntityManager, pero ese detalle no debe estar expuesto al dominio.

Repositorios y Consistencia

- Los repositorios deben manejar la consistencia de los agregados, lo que incluye:
 - Garantizar que los agregados sean recuperados en un estado consistente.
 - Aplicar las reglas de negocio pertinentes al persistir o eliminar agregados.
 - En sistemas distribuidos, puede ser necesario manejar la consistencia eventual.

¿Qué son los Eventos de Dominio?

- Los eventos de dominio representan hechos que han ocurrido en el dominio del negocio.
- Son inmutables y describen cambios significativos en el estado del sistema.
- Ejemplo: "Pedido Creado", "Pago Aceptado", "Cliente Registrado".

Beneficios de los Eventos de Dominio

- **Desacoplamiento:** Permiten que distintas partes del sistema reaccionen sin necesidad de estar fuertemente acopladas.
- **Rastreo del Estado:** Los eventos de dominio proporcionan un historial detallado de los cambios del sistema.
- **Reactivo:** Facilita un diseño orientado a eventos donde los cambios se propagan automáticamente.

Eventos en el Modelo de Dominio

- Los eventos son parte del lenguaje ubicuo y deben reflejar hechos del negocio, no detalles técnicos.
- Se pueden utilizar para:
 - Informar a otros agregados de cambios.
 - Sincronizar diferentes sistemas.
 - Disparar procesos asíncronos, como notificaciones o integraciones.

Estructura de un Evento de Dominio

- Un evento de dominio típicamente incluye:
 - Nombre descriptivo: Ejemplo, PedidoCreado.
 - Datos relevantes: Información sobre el hecho ocurrido, como el ID del pedido, fecha de creación, monto total, etc.
 - Momento del evento: Tiempo en que el evento ocurrió.

Implementación de Eventos de Dominio

- Crear el Evento: Cuando ocurre una acción significativa, se genera el evento en el código.
 - Ejemplo: Al crear un nuevo pedido, se dispara un evento PedidoCreado.
- Publicar el Evento: El evento se publica y es procesado por los interesados en él.
 - Puede ser publicado sincrónicamente o asincrónicamente.
- Procesar el Evento: Otros componentes del sistema escuchan y reaccionan al evento (enviar un email, actualizar inventario, etc.).

Publicación y Manejo de Eventos

- Los eventos de dominio se pueden publicar de diferentes maneras:
 - **Sincrónico:** Los eventos se manejan en el mismo proceso y transacción.
 - **Asincrónico:** Los eventos se envían a través de un sistema de mensajería y se procesan en un momento posterior.
- **Infraestructura:** Sistemas como RabbitMQ, Kafka o mensajería interna pueden ser utilizados para manejar eventos asíncronos.

Consistencia y Eventualidad

- Los eventos de dominio ayudan a manejar la consistencia eventual en sistemas distribuidos.
- Los sistemas que reaccionan a eventos pueden actualizarse de manera eventual, sin necesidad de una transacción global.
- Ejemplo: Un sistema de inventario puede recibir un evento de PedidoCreado y ajustar sus cifras en un proceso separado.

Ventajas del Diseño Basado en Eventos

- Escalabilidad: Los sistemas desacoplados y basados en eventos pueden escalar más fácilmente.
- Auditoría y Rastreo: Los eventos proporcionan un registro claro de lo que ha ocurrido, útil para auditorías.
- Resiliencia: Al manejar los eventos de manera asincrónica, los sistemas son menos propensos a fallos críticos debido a desacoplamiento.

¿Qué es una Fábrica?

- Una fábrica es un patrón de diseño utilizado para la creación de objetos complejos.
- Encapsula la lógica necesaria para construir un agregado o una entidad, asegurando que se respeten las reglas de negocio.
- Facilita la creación de objetos sin que el código cliente necesite conocer los detalles de su construcción.

¿Por qué usar una Fábrica?

- Simplificación: Evita que los clientes del modelo de dominio tengan que preocuparse por la lógica de creación de objetos.
- Consistencia: Asegura que las reglas y restricciones del dominio se cumplan al crear objetos.
- Desacoplamiento: Separa la responsabilidad de crear objetos de otras partes del código, promoviendo la cohesión.

¿Cuándo utilizar una Fábrica?

- Utiliza una fábrica cuando:
- La creación del objeto es compleja y requiere de múltiples pasos o validaciones.
- El proceso de construcción puede cambiar en el futuro y deseas encapsular ese cambio.
- Se debe garantizar que los objetos sean creados en un estado válido desde el inicio.


Fábricas en el Contexto del DDD

- En Domain-Driven Design, las fábricas son responsables de crear:
 - Entidades: Cuando la creación involucra lógica compleja.
 - Agregados: Para garantizar que un agregado completo se construya correctamente, respetando todas las reglas del dominio.
- Las fábricas ayudan a proteger la consistencia y las reglas inmutables del modelo de dominio.

Beneficios de Usar Fábricas

- Encapsulación: Las reglas y la complejidad de la creación del objeto están encapsuladas en la fábrica, no en el código cliente.
- Reducción de errores: Evita la creación incorrecta de objetos fuera de un estado válido.
- Flexibilidad: Si el proceso de creación cambia, solo la fábrica necesita modificarse, no el código cliente.
- Reutilización: Puedes reutilizar la lógica de creación en diferentes partes del sistema.

Ejemplo de una Fábrica para un Agregado



```
public class PedidoFactory {  
    public Pedido crear(Cliente cliente, List<LineaPedido> lineas) {  
        // Validaciones y lógica de negocio  
        if(lineas.isEmpty()) {  
            throw new IllegalArgumentException("El pedido debe tener al menos una línea.");  
        }  
        return new Pedido(cliente, lineas);  
    }  
}
```

Fábricas vs Constructores

- Constructor:
 - Adecuado para la creación de objetos simples que no requieren validaciones complejas.
 - Se usa cuando la lógica de creación es trivial y no viola las reglas del dominio.
- Fábrica:
 - Útil para la creación de objetos complejos o agregados.
 - Ideal cuando hay lógica de negocio o validaciones involucradas.

¿Y LA CHEYENNE 'APÁ?



¿Y LA CHEYENNE 'APÁ?

Service Layer



¿Qué es un Servicio de Dominio?

- Un Servicio de Dominio es un patrón que encapsula lógica de negocio que no pertenece a una entidad o un objeto de valor.
- Responde a preguntas del tipo: "¿Dónde coloco esta lógica que no encaja naturalmente en ninguna entidad o valor?"
- Representa operaciones que son parte del dominio pero que no tienen estado propio.

Características de un Servicio de Dominio

- Sin estado: No guarda información persistente, solo ejecuta operaciones sobre otros objetos del dominio.
- Foco en el negocio: Representa acciones del dominio que involucren reglas de negocio.
- Lógica compleja: Encapsula comportamientos que no pertenecen a una única entidad o agregado.

¿Cuándo usar un Servicio de Dominio?

- Usar un servicio de dominio cuando:
 - La lógica no puede ser naturalmente asignada a una entidad o un objeto de valor.
 - La operación afecta a múltiples entidades o agregados.
 - Quieres mantener el modelo de dominio limpio y cohesivo.

Ejemplos de Servicios de Dominio

- Cálculo de Impuestos:
 - La lógica de cálculo de impuestos puede involucrar múltiples entidades (pedidos, clientes, etc.), pero no pertenece a una entidad específica.
 - Encapsular esta lógica en un servicio mejora la cohesión.
- Transferencia de Dinero:
 - La operación de transferencia de dinero puede involucrar múltiples cuentas (agregados) y necesita reglas de validación y consistencia.

Tipos de Servicios en DDD

- En DDD, generalmente encontramos tres tipos de servicios:
 - Application Services (Servicios de Aplicación)
 - Domain Services (Servicios de Dominio)
 - Infrastructure Services (Servicios de Infraestructura)

Application Services (Servicios de Aplicación)

- Los Application Services son responsables de orquestar la lógica de negocio, actuando como una capa entre la interfaz del usuario (o las APIs) y el dominio.
- No contienen lógica de negocio; su principal tarea es coordinar las operaciones del dominio, a menudo interactuando con repositorios, entidades, y otros servicios de infraestructura.
- Responsabilidades:
 - Orquestar los casos de uso de la aplicación.
 - Interactuar con Repositorios y Domain Services.
 - Delegar las reglas de negocio al dominio.
 - Coordinar transacciones y flujos de trabajo.

Ejemplo de un Application Service

```
public class OrderApplicationService {
    private final OrderRepository orderRepository;
    private final PaymentService paymentService;

    public OrderApplicationService(
        OrderRepository orderRepository, PaymentService paymentService) {
        this.orderRepository = orderRepository;
        this.paymentService = paymentService;
    }

    public void confirmOrder(UUID orderId) {
        // Cargar el agregado (Order) desde el repositorio
        Order order = orderRepository.findById(orderId);

        // Validar y confirmar el pedido
        order.confirmOrder();

        // Procesar el pago usando un servicio externo
        paymentService.processPayment(order);

        // Guardar el agregado
        orderRepository.save(order);
    }
}
```

Domain Services (Servicios de Dominio)

- Los Domain Services encapsulan lógica de negocio que no pertenece naturalmente a una entidad o a un objeto de valor.
- Cuando una operación involucra múltiples entidades o conceptos que no pueden estar claramente asociados a una única entidad, se utiliza un servicio de dominio.
- Un Domain Service contiene reglas de negocio que no tienen un "hogar" natural en una entidad específica.
- Responsabilidades:
 - Implementar lógica de negocio que involucra varias entidades.
 - Asegurar que las operaciones que no pertenecen claramente a una entidad estén encapsuladas correctamente.

Ejemplo de un Domain Service

```
public class ShippingService {  
    public double calculateShippingCost(Order order, Address deliveryAddress) {  
        double baseCost = 5.0; // Costo base  
        double distanceFactor = deliveryAddress.distanceFromWarehouse();  
        // Cálculo simplificado  
        return baseCost + distanceFactor * order.getTotalWeight();  
    }  
}
```

Infrastructure Services (Servicios de Infraestructura)

- Los Infrastructure Services se encargan de las operaciones relacionadas con la infraestructura, como la persistencia, la integración con sistemas externos, y otras tareas que no pertenecen al dominio del negocio en sí.
- Estos servicios son usados tanto por los Application Services como por las entidades del dominio, pero su lógica no forma parte del core del dominio.
- Responsabilidades:
 - Manejar la interacción con sistemas externos.
 - Proporcionar servicios que no están directamente relacionados con la lógica de negocio.
 - Acceso a bases de datos, colas de mensajes, APIs externas, etc.

Ejemplo de un Infrastructure Service

```
public class PaymentService {  
    public void processPayment(Order order) {  
        // Lógica para procesar el pago con un sistema externo  
        System.out.println("Procesando pago para el pedido: " + order.getOrderId());  
        Map<String, Object> chargeParams = new HashMap<>();  
        chargeParams.put("amount", chargeRequest.getAmount());  
        chargeParams.put("currency", chargeRequest.getCurrency());  
        chargeParams.put("description", chargeRequest.getDescription());  
        chargeParams.put("source", chargeRequest.getStripeToken());  
        return Charge.create(chargeParams);  
    }  
}
```


Buenas Prácticas para Servicios de Dominio

- Mantenerlos sin estado: No deben almacenar información entre llamadas, asegurando que sean reutilizables y predecibles.
- Centrarse en una única responsabilidad: Cada servicio debe tener una única responsabilidad en el dominio, siguiendo el principio de responsabilidad única (SRP).
- Usar el lenguaje ubicuo: Nombres y métodos que reflejen claramente la lógica del negocio en el lenguaje común a los expertos del dominio.

¿Qué es un Módulo en DDD?

- Un módulo es una agrupación lógica de conceptos del dominio que están relacionados entre sí.
- Se utiliza para organizar el modelo de dominio y mejorar la comprensión, mantenibilidad y escalabilidad del sistema.
- Un módulo debe ser cohesivo, con responsabilidades claras y bien definidas.

¿Por qué usar Módulos?

- Claridad y Organización: Ayudan a organizar grandes modelos de dominio en partes manejables.
- Encapsulación: Los módulos agrupan elementos relacionados y ocultan detalles innecesarios a otras partes del sistema.
- Mantenibilidad: Facilitan la evolución del sistema, permitiendo que los módulos cambien de manera independiente.

Diseño de Módulos

- Cohesión: Todos los elementos dentro de un módulo deben estar estrechamente relacionados con el dominio que el módulo representa.
- Bajo Acoplamiento: Los módulos deben interactuar entre sí a través de interfaces bien definidas, minimizando las dependencias.
- Lenguaje Ubicuo: Cada módulo debe seguir el lenguaje ubicuo del contexto delimitado al que pertenece.

Modularidad en el Modelo de Dominio

- Los módulos en DDD se utilizan para organizar las entidades, objetos de valor, agregados, repositorios y servicios de dominio en agrupaciones lógicas.
- Permiten que los desarrolladores se enfoquen en partes específicas del dominio sin verse abrumados por la complejidad global del sistema.

Módulos en un Sistema de ecommerce

- Módulo Pedidos:
 - Contiene entidades como Pedido, Línea de Pedido, y servicios relacionados con la gestión de pedidos.
- Módulo Inventario:
 - Agrupa la lógica relacionada con la gestión del inventario, como las entidades Producto, Almacén, y servicios de inventario.
- Módulo Pagos:
 - Maneja todo lo relacionado con el procesamiento de pagos, como Pago, Método de Pago, y servicios de validación de transacciones.

Buenas Prácticas de Modularidad

- Agrupar por Dominio de Negocio: Organiza los módulos según los conceptos de negocio, no por aspectos técnicos.
- Evitar Dependencias Cíclicas: Los módulos no deben depender unos de otros en un ciclo. Mantén las dependencias claras y dirigidas.
- Interfaces Claras: Define contratos claros entre los módulos para que interactúen sin necesidad de conocer detalles internos.

Relación entre Módulos y Contextos Delimitados

- Los Bounded Contexts pueden contener varios módulos, agrupando conceptos más específicos dentro del contexto general del dominio.
- Cada módulo dentro de un contexto debe:
 - Respetar los límites del contexto delimitado.
 - Seguir el lenguaje ubicuo definido por el contexto.
- El uso correcto de módulos dentro de contextos facilita la escala del sistema y reduce la complejidad global.

Modularidad y Escalabilidad

- La modularidad permite:
 - Evolución Independiente: Los módulos pueden evolucionar sin afectar a otras partes del sistema.
 - Pruebas Unitarias: Facilita la pruebas y validación de cada parte de manera independiente.
 - Distribución: En arquitecturas distribuidas, los módulos bien definidos pueden convertirse fácilmente en microservicios.