Domain-Driven Design

Jesús Sánchez Cuadrado
Última actualización: 26/01/2026

En este capítulo se presenta una introducción a Domain-Driven Design (DDD), un enfoque para el desarrollo de software que enfatiza la colaboración entre expertos en el dominio y desarrolladores de software a través de "un modelo expresado en código". Se exploran los conceptos clave de DDD, incluyendo el lenguaje ubicuo, el modelado del dominio, y los patrones fundamentales como entidades, objetos valor, servicios de dominio, agregados y repositorios. Además, se discuten cuestiones relacionadas con la aplicación de DDD en la práctica.

Índice

1. Introducción 1
   1.1. Objetivo de DDD 1
   1.2. ¿Qué entendemos por Dominio? . 2
2. Lenguaje Ubicuo y modelado del dominio 2
   2.1. Qué incluye un modelo de dominio 2
   2.2. Cuál es el nivel de abstracción correcto 3
   2.3. Modelos ricos vs. modelos anémicos 3
3. Modelo de dominio en software 5
   3.1. Entidades 6
   3.2. Objetos valor (Value Objects) 6
   3.3. Servicios de dominio 7
   3.4. Agregados 8
   3.5. Factorías 11
   3.6. Repositorios 12
4. Eventos de dominio 12
5. Recursos 12
Bibliografía 13

1. Introducción

Domain-Driven Design (DDD) es un enfoque para el desarrollo de software que enfatiza la colaboración entre expertos en el dominio y desarrolladores de software. Fue introducido por Eric Evans en su libro "Domain-Driven Design: Tackling Complexity in the Heart of Software" en 2003 [1]. DDD se centra en comprender profundamente el dominio del problema y utilizar ese conocimiento para diseñar software que refleje con precisión ese dominio.

Hay muchos tipos de sistemas de software (software de sistema, juegos, compiladores, etc.), y DDD no es adeuado para todos ellos. Sin embargo, puede ser particularmente útil para software empresarial y de negocios, donde los desarrolladores deben colaborar con otros equipos no técnicos, donde el dominio del problema no es trivial y requiere una comprensión detallada para crear soluciones efectivas.

DDD es una aproximación que puede aplicarse a diferentes estilos arquitecturales (microservicios, monolitos modulares, arquitecturas basadas en eventos, etc.).

1.1. Objetivo de DDD

Uno de los problemas esenciales en el desarrollo de software es la brecha entre los expertos del dominio (personas que entienden profundamente el área de negocio o problema que el software intenta resolver) y los desarrolladores de software. Independientemente de la metodología de desarrollo utilizada (ágil, en cascada, etc.), el problema fundamental es que una persona técnica (ej., un desarrollador) debe traducir los requisitos y el conocimiento de una persona no técnica (ej., un experto del dominio) en código ejecutable. Esta traducción puede llevar a malentendidos, pérdida de información y soluciones que no reflejan con precisión las necesidades del dominio.

La solución que propone DDD es crear un modelo compartido entre los expertos del dominio y los desarrolladores. Este modelo actúa como un lenguaje común que ambos grupos pueden usar para comunicarse y colaborar (ver Fig. 1).

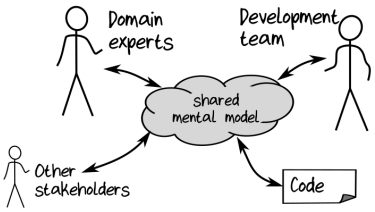

Figura 1: Modelo compartido de DDD

La hipótesis principal es que si el diseño del software es cercano al dominio, y viceversa, el modelo de dominio es cercano al diseño del software, entonces estos no pueden divergir y el software reflejará fielmente las necesidades del negocio. Para ello Eric Evans propone "Model-Driven Design" de manera que el mismo modelo sirva tanto para el análisis como para el diseño del software y que esté reflejado directamente en la implementación[1].

1.2. ¿Qué entendemos por Dominio?

El término **dominio** se refiere al área de conocimiento, actividad o negocio específico para el cual se está desarrollando el software[2]. Es el contexto en el que el software operará y resolverá problemas. Por tanto, el dominio:

- Hace referencia al "qué" del sistema, no el "cómo".
- Se refiere al problema, no la solución técnica.
- Es el conocimiento del negocio, no el código.

DDD parte de la idea de que el software debe modelar fielmente el dominio. La primera cuestión para ello es hablar el mismo lenguaje entre expertos del dominio y desarrolladores. La segunda cuestión es cómo representar ese conocimiento del dominio en el software de manera efectiva.

[1] Esta idea de tener un único modelo que aúne el análisis y el diseño se hace explícita en la discusión de las páginas 28-29 del libro de Evans [1].

[2] En el glosario de [1]: "A sphere of knowledge, influence, or activity."

> ⚠️ **Importante**
>
> Cuando hablamos de dominio no nos referimos al dominio técnico (frameworks, bases de datos, lenguajes), sino al área de conocimiento y actividad del mundo real que el software pretende modelar y resolver.

2. Lenguaje Ubicuo y modelado del dominio

En DDD se denomina **Lenguaje Ubicuo** al conjunto de conceptos y vocabulario que se comparte entre todos los ***stakeholders*** del proyecto: el "lenguaje de todas partes". Este es el lenguaje que define el modelo mental compartido para el dominio del negocio. Es fundamental que este lenguaje se utilice en todas partes del proyecto: requisitos, diseño y código fuente.

> ⚠️ **Importante**
>
> En la práctica lo que signfica usar el lenguaje ubicuo es que los nombres de las clases, métodos, atributos, etc., deben reflejar los términos del dominio.

La motivación principal puede resumirse en la siguiente explicación del libro de Evans (Capítulo 2, página 14 [1]):

Domain experts have limited understanding of the technical jargon of software development, but they use the jargon of their field—probably in various flavors. Developers, on the other hand, may understand and discuss the system in descriptive, functional terms, devoid of the meaning carried by the experts' language. […]

On a project without a common language, developers have to translate for domain experts. Domain experts translate between developers and still other domain experts. Developers even translate for each other. Translation muddles model concepts, which leads to destructive refactoring of code. The indirectness of communication conceals the formation of schisms—different team members use terms differently but don't realize it.

El problema observado por Evans puede resumirse en el clásico juego del teléfono roto. A medida que la información fluye entre diferentes personas, se distorsiona y pierde su significado original.

2.1. Qué incluye un modelo de dominio

El lenguaje ubicuo se hace explícito a través de un **modelo de dominio**, que es una representación conceptual de los aspectos relevantes del dominio.

Este modelo incluye:

- Diagramas

- Representaciones visuales cercanas al dominio
- Ejemplos de escenarios concretos
- Reglas de negocio escritas en lenguaje natural
- Explicaciones textuales

En general es un modelo informal ayuda a comunicar el lenguaje y los conceptos del dominio entre todos los participantes del proyecto.

2.2. Cuál es el nivel de abstracción correcto

La propuesta de DDD es que el modelo de dominio debe estar en un nivel de abstracción que sea comprensible tanto para los expertos del dominio como útil para los desarrolladores. Se trata de que el modelo **sirva tanto para el análisis como para el diseño del software y que sea la guía para la implementación**.

La lógica detrás de esta propuesta es que si se tiene un único modelo que es compartido por todos, entonces se reduce la posibilidad de malentendidos y el software reflejará exactamente las necesidades de los usuarios porque éstas "no se han perdido por el camino". Si el modelo y la implementación son muy cercanos, entonces no hay problema para mantenerlos sincronizados.

> ⚠️ **Importante**
> Por tanto, cuando en DDD hablamos de "modelo", no nos referimos al modelo conceptual del software (como en el modelado UML tradicional), sino a un modelo que une el análisis del dominio y el diseño del software y que está expresado en el código fuente.

2.3. Modelos ricos vs. modelos anémicos

Los modelos anémicos son aquellos que simplemente representan datos sin incluir la lógica de negocio relevante[2]. Este tipo de modelos son poco deseables porque implican todas las incomidades de la orientación a objetos pero sin ninguna de sus ventajas.

Por tanto, lo que se desea son modelos de dominio ricos, en los que los objetos de dominio deben incluir las operaciones que representen las acciones importantes del dominio. Hay que evitar modelos llenos de "getters" y "setters" porque eso serían modelos de datos (o modelos de dominio anémicos) y no modelos de dominio ricos.

Entonces, el modelado del dominio en DDD incluye dos aspectos:

1. Crear un modelo conceptual del dominio que capture los conceptos y relaciones importantes.
2. Diseñar la estructura del software para reflejar fielmente ese modelo conceptual.

Los puntos (1) y (2) van de la mano, ya que el modelo conceptual guía el diseño del software, y el diseño del software debe reflejar el modelo conceptual.

2.3.1. Ejemplo

El siguiente modelo representa una cuenta bancaria pero no tiene ninguna operación, simplemente getters y setters. Al usarlo en la lógica de negocio (clase `ServicioCuenta`) se tienen que añadir validaciones para proteger los invariantes del dominio (saldo positivo, etc.).

```
public class Cuenta {
  private String numero;
  private double saldo;
  // .. getters y setters ...
}

// Servicio con la lógica de negocio
public class ServicioCuenta {

  public void retirar(Cuenta cuenta, double importe) {
    checkArgument(importe > 0,
                        "El importe debe ser positivo");
    checkArgument(cuenta.getSaldo() >= importe,
                        "Saldo insuficiente");

    cuenta.setSaldo(cuenta.getSaldo() - importe);
  }
}
```

El siguiente modelo incluye las operaciones importantes del dominio dentro de la propia clase `Cuenta`, protegiendo así los invariantes del dominio:

```
public class Cuenta {

    private final String numero;
    private double saldo;

    public Cuenta(String numero, double saldoInicial) {
        checkArgument(saldoInicial >= 0,
            "El saldo inicial no puede ser negativo");
        this.numero = numero;
        this.saldo = saldoInicial;
    }

    public void retirar(double importe) {
        checkArgument(importe > 0,
                "El importe debe ser positivo");
        checkState(saldo  >= importe,
                "Saldo insuficiente");
        saldo -= importe;
    }

    public void ingresar(double importe) {
        Preconditions.checkArgument(importe > 0,
                           "El importe debe ser positivo");
        saldo += importe;
    }

    public double getSaldo() {
```

```
        return saldo;
    }

    public String getNumero() {
        return numero;
    }
}
```

```
// Esta clase es de la capa de aplicación.
// Puede llamarse también RetirarDineroUseCase o similar
// (según la convención o estilo utilizado en el proyecto)
public class ServicioRetiradaDinero {

    private final CuentaRepository cuentaRepository;

    public ServicioRetiradaDinero(CuentaRepository
cuentaRepository) {
        this.cuentaRepository = cuentaRepository;
    }

    public void ejecutar(String numeroCuenta, double
importe) {
        // 1. Obtener la entidad
        Cuenta cuenta =
cuentaRepository.buscarPorNumero(numeroCuenta);

        // 2. Delegar la lógica al dominio
        cuenta.retirar(importe);

        // 3. Persistir el estado resultante
        cuentaRepository.guardar(cuenta);
    }
}
```

Esta segunda versión tienen algunas ventajas importantes:

- No se puede dejar la cuenta en un estado inválido
- El comportamiento está repartido en los lugares adecuados: la operaciones del dominio en el modelo de dominio, la lógica de orquestación en el capa de aplicación.
- El modelo expresa el lenguaje ubicuo (retirar, ingresar)
- El servicio (de aplicación) orquesta, no contienen lógica central

Pero más importante que es que **si en otro trozo del código tratamos con una cuenta, es seguro que una cuenta siempre tendrá el comportamiento correcto**.

3. Modelo de dominio en software

Tal y como se ha mencionado anteriormente, DDD propone que el modelo de dominio se refleje directamente en el código fuente. Para ello, DDD define varios patrones que reflejan cómo modelar ciertos conceptos del dominio o cómo organizar el diseño de manera efectiva. Los elementos básicos son:

- Entidades

Si vamos a usar un modelo anémico, entonces es mejor atacar directamente la base de datos con SQL y olvidarse de la orientación a objetos. https://martinfowler.com/eaaCatalog/transactionScript.html

- Objetos valor (Value objects)
- Servicios de dominio
- Agregados
- Factorías
- Repositorios

> ⚠️ **Importante**
>
> Al contrario que al realizar "modelado conceptual" estos elementos incluyen decisiones de diseño cercanas a la implementación.

Utilizaremos entidades, objectos valor y servicios de dominio para modelar el dominio directamente en el software, mientras que los agregados, factorías y repositorios son elementos del diseño que nos permitirán manejar el ciclo de vida de los objetos dentro del software.

3.1. Entidades

Una **entidad** es un objeto que se define principalmente por su identidad, en lugar de sus atributos o propiedades.

Una entidad tiene una identidad única que lo distingue de otros objetos, incluso si sus atributos son idénticos. Una entidad tiene un ***ciclo de vida*** que puede incluir cambios en sus atributos a lo largo del tiempo, pero su identidad permanece constante.

Ejemplos de entidades:

- Un usuario en un sistema (identificado por un ID único)
- Un pedido en un sistema de comercio electrónico (identificado por un número de pedido)
- Una pregunta en un foro en línea (identificada por un ID de pregunta generado automáticamente)

La identificación de una entidad puede venir dada "desde fuera" (ej., el correo electrónico de un usuario), o puede ser un identificador arbitrario creado por y para el sistema (ej., un UUID o un ID generado automáticamente por la base de datos).

> ❓ **Pregunta**
>
> ¿Cómo debe implementarse ***equals*** para entidades? ¿Y para ***value objects***?

Determinar qué tipo de identificador usar para una entidad es una decisión de diseño importante, que no siempre es obvia y depende del dominio. Hay dos tipos básicos de identificadores:

- **Identificadores naturales**: atributos inherentes al objeto (ej., DNI, correo electrónico) o combinaciones de atributos (ej., nombre, fecha y ciudad para identificar un periódico).
- **Identificadores sintéticos**: creados específicamente para identificar el objeto. El sistema debe garantizar que son únicos. Pueden ser:
  - Internos para el sistema (ej., identificador de un producto)
  - Visibles para el usuario (ej., ***tracking number*** de un pedido)

3.2. Objetos valor (Value Objects)

Un **value object** es un objeto que describe algún aspecto de otro objeto del dominio pero que no tiene identidad propia.

Un objeto valor es inmutable y se comparan utilizando todos sus atributos. Dos objetos valor son iguales si todos sus atributos son iguales. Esto simplifica la gestión de este tipo de objetos.

Los objeto valor suelen incluir validaciones en su creación para garantizar la corrección de los datos que representan.

⚙️ **Implementación de Value Objects en Java**

En Java utilizaremos clases inmutables para representar Value Objects. A partir de la versión 16 de Java podemos usar *record* para definir Value Objects de manera muy concisa:

```
public record Color(int rojo, int verde, int azul) {
    // Validación en el constructor
    public Color {
        if (rojo < 0 || rojo > 255 ||
            verde < 0 || verde > 255 ||
            azul < 0 || azul > 255) {
            throw new IllegalArgumentException("Los
valores de color deben estar entre 0 y 255");
        }
    }
}
```

Un *record* implementa automáticamente los métodos *equals* y *hashCode* basándose en todos sus atributos.

Por último, una ventaja importante de los Value Objects es que pueden ser compartidos libremente entre diferentes partes del sistema sin preocuparse por efectos secundarios no deseados, ya que son inmutables.

3.3. Servicios de dominio

Algunos conceptos del dominio no se pueden modelar de manera natural como objetos (entidades o value objects) porque no encapsulan ningún estado sino que son acciones o procesos (e.j., `CalculadoraDePrecios`). A veces también ocurre que hay que tratar con varias entidades no relacionadas (ver Agregados en la sección 3.4). En ese caso definiremos servicios de dominio con el objetivo de no abusar en la definición de entidades simplemente para encajar una operación.

Los **servicios de dominio** son objetos que representan operaciones o acciones importantes en el dominio pero que no encajan naturalmente en una entidad.

Características de los servicios de dominio:

1. Representan operaciones o acciones en el dominio.
2. La interfaz se define en términos de otros elementos del modelo de dominio.
3. No tienen estado (*stateless*). Esto no significa que no puedan tener efectos secundarios (ej., guardar en una base de datos), sino que no mantienen estado interno entre invocaciones.

Por ejemplo, en una biblioteca podríamos tener `ServicioPrestamo` que encapsula las reglas para prestar libros a los usuarios.

⚠️ **Importante**

Los servicios de dominio no deben confundirse con los servicios de aplicación o los servicios de la capa de infrastructura. Por ejemplo, un servicio que envía correos electrónicos es un servicio de infrastructura, no un servicio de dominio, pero un servicio que determina si se debe enviar un correo electrónico de notificación basado en reglas de negocio sí es un servicio de dominio.

```java
public class ServicioPrestamo {
    public void prestar(Usuario usuario, Libro libro) {
        // Lógica para prestar el libro
        // 1. Chequear que el usuario no super su cuota de
libros
        // 2. Chequear que el libro no está prestado
        // 3. Actualizar el usuario y el libro
    }
    public void devolver(Usuario usuario, Libro libro) {
        // Lógica para devolver el libro
    }
}
```

Normalmente los servicios de dominio se implementan como ***singletons***.

3.3.1. Servicios de aplicación

Los servicios de aplicación son responsables de orquestar las operaciones del dominio y coordinar las interacciones entre diferentes objetos del dominio. Por tanto, **no contienen lógica de negocio**.

Los servicios de aplicación son los clientes directos del modelo de dominio.

Si se desea que los servicios de aplicación estén completamente desacoplados del modelo de dominio (en particular de las entidades), se pueden utilizar DTOs (Data Transfer Objects) para transferir datos entre la UI y el servicio de aplicación que es el que actua de intermediario.[3] Para ello:

- El servicio de aplicación recibe DTOs desde la UI.
- El servicio de aplicación utiliza el modelo de dominio para realizar las operaciones necesarias.
  - Obtiene las entidades necesarias desde los repositorios.
  - Llama a los métodos de las entidades y servicios de dominio.
  - Persiste los cambios a través de los repositorios.
  - Gestiona la transaccionalidad.
- El servicio de aplicación devuelve DTOs a la UI.

[3]

⚙️ **Data Transfer Object**

Los DTOs son objetos planos que sirven para transportar datos. Normalmenet son serializables. En Java se puden implementar con ***records***.

3.4. Agregados

Tratar con grafos de objetos complejos puede ser complicado porque una operación sobre un objeto implica cambios en otros objetos relacionados. Por ejemplo, al eliminar un usuario del sistema también hay que eliminar su dirección.

La solución ofrecida por DDD es agrupar estos objetos que forman "una unidad semántica" en lo que se denomina un **agregado**. En lugar de operar sobre los objetos individuales, se opera sobre el agregado como una unidad. Así, el agregado garantiza que se mantienen las invariantes de esos objetos que contiene.

Un **agregado** es conjunto de objetos del dominio que se tratan como una unidad para fines de consistencia de datos, esto es, se manipula y se persisten juntos.

Cada agregado tiene una **raíz del agregado** que es el punto de entrada del agregado de cara a los clientes y se encarga de mantener los invariantes.

3.4.1. Ejemplo de agregado: pedidos

Un ejemplo típico de agregados es la gestión de un pedido y sus líneas de pedido. Una línea del pedido referencia al producto que se está comprando junto con la cantidad de productos. En este ejemplo, la regla de negocio de un pedido es que *el total del pedido no puede superar la 1000 €*.

La figura 1 muestra la solución propuesta. El agregado Pedido contiene LíneaPedido porque es necesario para garantizar el invariante. Además, en esta solución se decide copiar el precio del Producto a la línea de producto por dos razones puesto que una vez que el pedido se quiere fijar el precio[4].

[4]Posiblemente se querría fijar el precio solo cuando el pedido es firme.

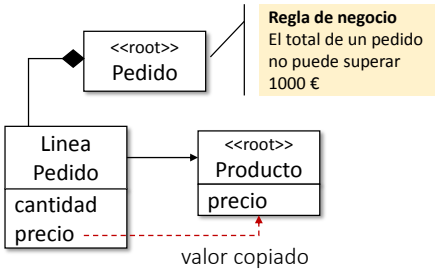

Figura 2: Modelado de preguntas y tipos de preguntas

Además del agregado Pedido tenemos también el agregado Producto. Esto significa que ambos agregados tienen sus propios ciclos de vida. Las referencias entre agregados siempre son entre sus raices (a través de su ID).

> ⚙️ **Modelar referencias entre agregados**
>
> Los agregados pueden tener referencias a otros agregados (a través del ID de la raíz del agregado referenciado). Estas referencias se realizan a través del identificador de la raíz.
>
> Por tanto, es buena idea modelar también el identificador para no tener que usar tipos primitivos (ej., `String` o `int`) para los IDs. Para ello, utilizaremos Value Objects.
>
> Por ejemplo, para el identificador de un `Producto` crearemos una clase `ProductoId`.

3.4.2. Tipos de consistencia

En DDD se distinguen dos tipos de consistencia:

- **Consistencia fuerte o transaccional**: se garantiza que los invariantes del agregado se cumplen al final de una operación que afecta al agregado. Esto se aplica dentro de un único agregado y significa que los cambios de un agregado son consistentes una vez termina la operación: esto es, todos los objetos del agregado

cumplen sus invariantes y la siguiente lectura de esos objetos obtendrá una versión consistente de los mismos.

- **Consistencia eventual** hace referencia al hecho de que los invariantes del modelo no se garantizan atómicamente, sino que se cumplirán dentro de un periodo temporal.

En otras palabras, se garantiza que los invariantes del agregado se cumplirán en algún momento en el futuro, pero no necesariamente al final de la

operación. En DDD, los invariantes entre agregados se mantienen utilizando consistencia eventual. La razón es que garantizar la consistencia fuerte en agregados (o en general grandes grafos de objetos) es complicado y poco escalable.

Como se ha explicado, un agregado debe garantizar los invariantes del grupo de objetos que gestiona. Un agregado es **un límite de consistencia** donde la consistencia es fuerte. En la práctica esto implica cargar tales objetos en memoria[5]. Además, en un entorno distruido pueden existir problemas de concurrencia cuando varios usuarios modifican el mismo agregado y por tanto afectar a la escalabilidad del sistema. La razón es que un agregado se guarda en una transacción y solo un proceso puede completarla al mismo tiempo. Si el agregado es muy grande, hay más contención y más probabilidad que la transacción falle.

[5] Aunque se puede utilizar técnicas como ***lazy loading*** para evitar esto en algunos casos

Una **transacción** de base de datos es un conjunto de operaciones que se ejecutan como una única unidad atómica de trabajo. En una transacción todas las operaciones de la transacción se completan con éxito, o bien ninguna de ellas se aplica a la base de datos.

Por tanto, un agregado debe contener ***solo la información que es necesaria para que las operaciones de negocio del agregado tengan consistencia fuerte***.

> ⚠️ **Importante**
> Un agregado es una ***frontera de consistencia transaccional***. Esto quiere decir que cualquier operación que modifique el estado de un agregado debe garantizar que los invariantes del agregado se cumplen al final de la operación.

> ⚠️ **Importante**
> En DDD la regla es que solo se modifica un agregado por transacción. Si una operación afecta a varios agregados, cada uno se modifica en transacciones separadas, lo que implica consistencia eventual entre ellos.

Cuando una operación incluye varios agregados se utilizará consistencia eventual, normalmente a través de eventos de dominio (ver sección 4)

3.4.3. Reglas para agregados

Identificar y diseñar agregados correctamente es crucial para mantener la integridad del modelo de dominio y garantizar un rendimiento adecuado.

Un agregado define qué se modifica junto conceptualmente y que esas modificaciones tienen que ser consistentes entre sí de manera atómica (*consistencia fuerte*)

En el peor de los casos se podría hacer que todos los objetos del dominio formaran parte de un único agregado, pero esto llevaría a problemas de rendimiento y escalabilidad, ya que cualquier operación sobre el agregado requeriría bloquear y persistir todo el conjunto de objetos relacionados.

A continuación se listan las principales reglas de diseño de agregados en DDD.

1. Cada agregado tiene una raíz (entidad raíz) que es la única que puede ser referenciada desde fuera del agregado.
2. Las operaciones que afectan a objetos dentro del agregado se realizan a través de la raíz del agregado.
3. Los invariantes del agregado se garantizan al final de una operación que afecta al agregado.

4. Los invariantes que deben cumplirse "siempre" deben estar dentro del mismo agregado
5. Un agregado no debe modificar un objeto del que no es dueño.
6. Los cambios a un agregado deben guardarse (***commit***) en una única transacción (el agregado es un límite transaccional)[6].
7. Entre agregados solo hay consistencia eventual.
8. Normalmente hay un repositorio por agregado.

A la hora de diseñarlos hay que tener en cuenta que los agregados deben ser lo más pequeños posible para minimizar los bloqueos y conflictos de concurrencia. Al mismo tiempo deben ser lo suficientemente grandes para garantizar las invariantes del dominio. Es un ***trade-off*** que depende del dominio y de los requisitos de la aplicación.

[6] Razones para esta regla:
1. Si dos agregados tienen que modificarse juntos puede ser un ***smell*** de que en realidad son el mismo agregado
2. Se limita la contención sobre la base de datos.
3. Reducir el acoplamiento. Los agregados podrían ser distruidos fácilmente si no se actualizan juntos.

Ver también https://softwareengineering.stackexchange.com/questions/429898/domain-driven-design-one-change-per--transaction

3.5. Factorías

Este patrón de DDD se utiliza para encapsular la lógica compleja de creación de objetos del dominio, especialmente cuando la creación implica múltiples pasos o reglas de negocio.

> Una **factoría** es un artefacto (método u objeto) que se encarga de crear instancias de otros objetos del dominio. En muchas ocasiones estos objetos son agregados y la factoría se encarga de garantizar que se cumplen todos los invariantes

Al centralizar la lógica de creación en una factoría, se mejora la cohesión del código y se facilita el mantenimiento.

Esencialmente para implementar una factoría se utilizará alguno de los patrones de diseño como ***factory method***, ***abstract factory*** o ***builder***.

Cuando la creación de un objeto es sencilla y no implica lógica compleja, podemos aplicar el patrón GRASP **Creador** (Creator) para asignar la responsabilidad de crear el objeto a una clase que tenga una relación cercana con el objeto a crear. Por ejemplo, una entidad `Pedido` tendría un método `nuevaLinea` ser responsable de crear sus propias `LíneaPedido`, ya que tiene una relación directa con ellas.

```
class Pedido {
    private List<LineaPedido> lineas;

    // Método factoría en la raiz del agregado Pedido
    public LineaPedido nuevaLinea(Producto p, int c) {
        LineaPedido linea = new LineaPedido(p, c);
        lineas.add(linea);
        return linea;
    }
}
```

Sin embargo, cuando la creación de un agregado completo requiere de lógica compleja para añadir todos los objetos relacionados y garantizar las invariantes del agregado, es mejor utilizar el patrón Builder.

3.6. Repositorios

Este patrón trata acerca de cómo obtener referencias a los objetos del dominio sin exponer los detalles de la persistencia.

El funcionamiento y responsabilidades de un repositorios es tal y como se describe en [1] (página 92):

Clients request objects from the REPOSITORY using query methods that select objects based on criteria specified by the client, typically the value of certain attributes. The REPOSITORY retrieves the requested object, encapsulating the machinery of database queries and metadata mapping. REPOSITORIES can implement a variety of queries that select objects based on whatever criteria the client requires. They can also return summary information, such as a count of how many instances meet some criteria. They can even return summary calculations, such as the total across all matching objects of some numerical attribute.

Un repositorio ofrece la ilusión de que los objetos están en memoria, pero en realidad normalmente estarán almacenados en una base de datos y el repositorio se encargará de recuperarlos y persistirlos.

En general, por cada raíz de agregado habrá un repositorio asociado.

❓ **Repositorios y agregados**

¿Qué sucedería si cada entidad tuviera su propio repositorio en lugar de tener un repositorio por agregado? ¿Cómo se resolverían las referencias entre entidades dentro del mismo agregado?

4. Eventos de dominio

Para conseguir un sistema escalable es necesario en ocasiones relajar la consistencia fuerte en favor de la consistencia eventual. Para ello se debe definir algún mecanismo que permita comunicarse a los agregados.

Un **evento de dominio** es un mensaje que describe un evento significativo que ha ocurrido en el dominio. Los eventos se nombran en pasado porque representan acciones que ya han sucedido.

Por ejemplo,

- Un pedido se ha creado (`PedidoCreadoEvent`). Puede ser procesada por el agregado que se encarga de los envíos.
- Un usuario ha pagado una suscripción (`SuscripcionPagadaEvent`). Puede ser procesada por el servicio que da acceso a los productos *premium* de la plataforma.

En DDD los eventos de dominio se utilizan para dos escenarios:

1. Para hacer cumplir la regla de que ***solo se puede modificar un agregado en una transacción***. Así, los agregados se sincronizan utilizando eventos al estilo del patron listener[7].
2. Para comunicar subsistemas diferentes (ej., en ***bounded contexts*** diferentes) de manera eficiente.

[7] Es algo más complicado porque hay que garantizar que el sistema sea robusto

Los eventos deben incluir los datos relativos al evento de manera que los receptores puedan utilizarlos para realizar su trabajo.

5. Recursos

Para profundizar en DDD, se recomienda consultar los siguientes recursos:

- Evans, E. (2004). Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional. [1]
  - Conocido como ***"the blue book"***, es el libro original sobre DDD, que establece los conceptos fundamentales y patrones del enfoque.
- Vernon, V. (2013). Implementing Domain-Driven Design. Addison-Wesley Professional. [3]
  - Conocido como ***"the red book"***, es un libro mucho más práctico que el de Evans y ofrece ejemplos concretos de cómo aplicar DDD en la práctica.
- Khononov, Vlad (2021). Learning Domain-Driven Design. O'Reilly. [4]
  - Explica de manera clara y prácticas los conceptos de DDD. En particular, la parte II (tactical design) es bastante ilustrativa.
- Khalil Stemmler ofrece en su web muchos artículos sobre DDD. Este es un buen punto de partida e incluye referencias a otros artículos: https://khalilstemmler.com/articles/domain-driven-design-intro/

Bibliografía

[1] E. Evans, ***Domain-driven design: tackling complexity in the heart of software***. Addison-Wesley Professional, 2004.

[2] M. Fowler, «Anemic Domain Model». [En línea]. Disponible en: https://martinfowler.com/bliki/AnemicDomainModel.html

[3] V. Vernon, ***Implementing domain-driven design***. Addison-Wesley Professional, 2013.

[4] V. Khononov, ***Learning Domain-Driven Design***. "O'Reilly Media, Inc.", 2021.