UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

MÁSTER UNIVERSITARIO EN DESARROLLO ÁGIL DE SOFTWARE PARA LA WEB

Trabajo Fin de Máster

DISEÑO DE SOFTWARE APLICANDO EL PATRÓN DOMAIN-DRIVEN DESIGN

Alberto Murillo Paredes

2021

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

MÁSTER UNIVERSITARIO EN

DESARROLLO ÁGIL DE SOFTWARE PARA LA WEB

Trabajo Fin de Máster

"DISEÑO DE SOFTWARE APLICANDO EL PATRÓN DOMAIN-DRIVEN DESIGN"

Autor: **Alberto Murillo Paredes**

Director: **Antonio García Cabot**

Tribunal:

Presidente: .....

Vocal 1º: ..... Vocal 2º: .....

Calificación: .....

Fecha: ..... de ..... de .....

Este proyecto está dedicado,

a mi familia por su ayuda, paciencia y esfuerzo

a mis compañeros y profesores de la universidad, en especial a Antonio García Cabot, por sus enseñanzas y dedicación

a Eric Evans, Vaughn Vernon, Martin Fowler y Robert C. Martin que sirven como fuente de inspiración y motivación personal

ÍNDICE RESUMIDO

ÍNDICE DETALLADO

ÍNDICE DE FIGURAS

1. INTRODUCCIÓN

En la actualidad no solo se diseñan sistemas informáticos que cumplen con los objetivos para los cuales fueron creados, sino que también se buscan aplicaciones que sean entendibles, mantenibles, cambiables y que permitan ser evolucionadas con gran facilidad.

En el presente documento se pretende exponer un diseño y modelado de aplicaciones en el que el lenguaje que se utiliza a nivel de negocio sea trasladado al código que es implementado, apoyándose en el patrón de diseño denominado "Domain-Driven Design", "Diseño Dirigido al Dominio" o DDD. Este patrón fue expuesto en primera instancia por el autor Eric Evans, el cual hace hincapié tanto en los conceptos y herramienta puramente técnicos como en la coordinación dentro de la estructura organizacional y la comunicación de los equipos de la compañía.

A lo largo del documento se plantean de forma teórica los conceptos y procedimientos que orbitan alrededor de DDD (Figura 1), aderezando con ejemplos los puramente técnicos, es decir, los relacionados con el código implementado por los desarrolladores. Para estos ejemplos prácticos se ha creado un repositorio en GitHub donde se gestiona el subdominio de la prevención de los riesgos laborales asociados a los empleados de una compañía cualquiera: https://github.com/albmurillop/prlmanagement

Es de vital importancia conocer y entender las situaciones en la que estas herramientas y patrones ofrecidos por DDD son efectivos para conseguir aportar valor a las compañías. Por norma general, en grandes organizaciones con una complejidad elevada.

Figura 1: Patrones tácticos y estratégicos de DDD (Evans, E. (2014) Pattern Language Overview [Figura] Recuperado de la página vii del libro Domain-Driven Design Reference: Definitions and Pattern Summaries)

2. OBJETIVOS DEL PROYECTO

Con este proyecto se persigue destilar todos los conceptos que surgen alrededor de una forma desiño de aplicaciones en la que todo gira en torno a los elementos lógicos y físicos del dominio de una organización.

Este patrón de diseño es conocido como Domain-Driven Design o Diseño Dirigido por el Dominio fue divulgado por primera vez en el libro de Eric Evans: "Domain-Driven Design: Tackling Complexity in the Heart of Software". Aunque aporta ciertos ejemplos sobre situaciones reales de su experiencia profesional, este libro está más centrado en la teoría que abarca la utilización de este patrón.

En apoyo a esta teoría de diseño, Vaughn Vernon, escribe un libro denominado "Implementing Domain-Driven Design" que presenta una versión más práctica sobre la implementación en código de los aspectos técnicos asociados al patrón de diseño.

Por lo tanto, y centrando como piedra angular a los dos libros anteriormente mencionados, en este proyecto se presentan todos los conceptos relevantes de manera simplificada y fácil de comprender, junto con ejemplos prácticos en lenguaje Java para los patrones propiamente relacionados con la implementación de la solución una vez se opta por la utilización de este tipo de paradigma.

3. DISEÑO DIRIGIDO POR EL DOMINIO

El patrón "Domain-Driven Design" o "Diseño Dirigido por el Dominio" fue introducido por Eric Evans en su libro "Domain-Driven Desing: Tackling Complexity in the Heart of Software". En este libro Evans define el dominio como "un ámbito de conocimiento, influencia o actividad. El área temática en la que un usuario desarrolla un producto software es el **dominio**".

Por su parte Vernon en su libro "Implementing Domain-Driven Design", ofrece una definición de dominio más sencilla de entender: "Un dominio, en su sentido más amplio, es lo que hace una organización y el modo en el que lo hace. Las empresas identifican un mercado y venden productos y servicios. Cada tipo de organización tiene su propio y único ámbito de conocimiento y sus propias formas de hacer las cosas. Ese ámbito de conocimiento y sus métodos para llevar a cabo sus operaciones es lo que se conoce como **dominio**".

Por otro lado, Evans define **modelo** como "un sistema de abstracciones que describe aspectos seleccionados de un dominio y que puede utilizarse para resolver problemas relacionados con el dominio".

El dominio a su vez se divide en diferentes partes llamadas **subdominios** (Figura 2), los cuales contienen uno o varios modelos. Por lo tanto, son las diferentes partes del problema a modelar y se categorizan de la siguiente manera:

- **Núcleo del dominio o dominio principal (core domain).** Convertir la parte principal del negocio en un activo para la organización. En él destacan los concetos más valiosos y especializados de la organización, haciéndola especial y diferente del resto de compañías de la competencia. Es el subdominio más prioritario, el que requiere más esfuerzo y en el que se concentra el mayor talento
- **Subdominios de apoyo.** Revelan parte del negocio, pero son menos importantes que el dominio principal
- **Subdominio genérico.** No expresan parte del modelado del dominio. Se permite la utilización de estándares para reducir la complejidad y los costes asociados (ERP, CRM, etc.)

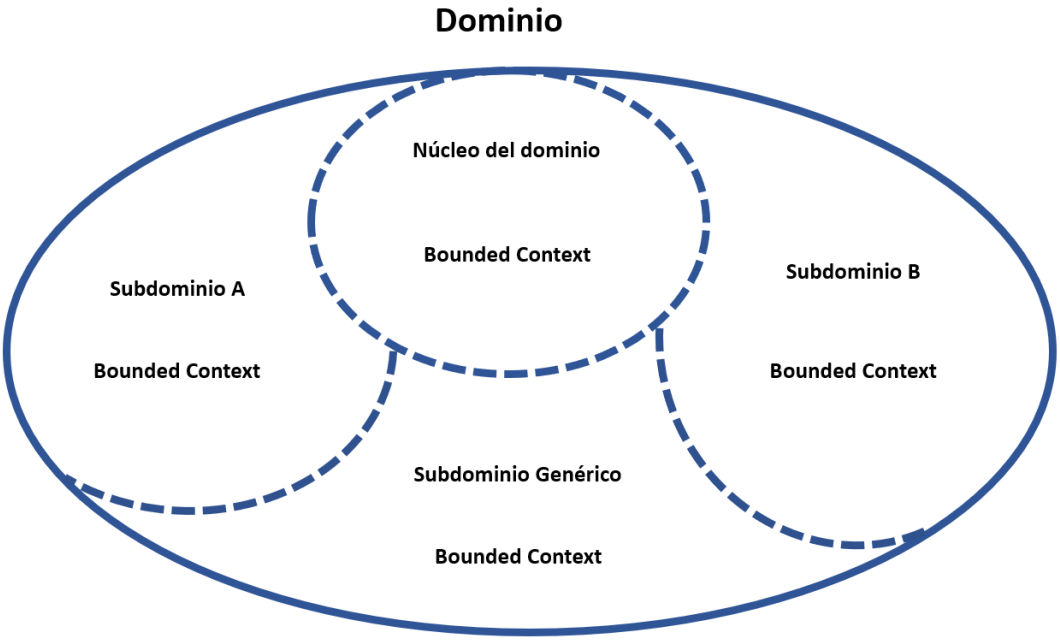

Figura 2: División del dominio de una organización de ejemplo

Cabe destacar que todos los subdominios que conforman al modelado del dominio son importantes para lograr el éxito en conjunto. Los requisitos en la calidad y la cantidad de esfuerzo son las grandes diferencias entre los distintos subdominios. Como se explicará más adelante, el patrón de diseño utilizado para dar solución a estos subdominios recibe el nombre de contextos delimitados (Bounded Context).

Las organizaciones grandes o de gran complejidad a menudo diseñan e implementan su dominio mezclando los diferentes modelos y las relaciones que estos presentan, provocando un caos tanto organizativo como productivo. Con DDD se pretende plasmar la estructura organizacional y su terminología en el código, para que no esté alejada de la jerga utilizada día a día por los integrantes de la compañía. Por lo tanto, entran en juego aspectos tanto **técnico** (tácticos) como **organizativos** (estratégicos).

Gracias a los límites entre subdominios que promueve DDD, se fomenta una cultura de **autonomía** entre los diferentes equipos, permitiendo a cada uno de ellos solucionar sus problemas de la manera más adecuada posible y poder trabajar de manera autónoma. Además, una de las características principales que intenta aporta DDD al diseño de un sistema es la **tolerancia al cambio**, ya que es por todos sabido que el software es volátil.

No hay que volverse loco e intentar instaurar esta "filosofía" en cualquier situación o a cualquier precio. Este proceso es largo y su curva de aprendizaje es elevada, por lo que se debe tener paciencia e ir avanzar paso a paso priorizando el núcleo del dominio y continuando posteriormente según la relevancia. Hay que tener claro que el objetivo principal es aportar valor a la organización [1], [2], [3], [4].

4. LENGUAJE UBICUO

Realizar diseños flexibles y ricos de conocimiento requiere de un lenguaje colaborativo, cambiable y evolutivo. Este tipo de lenguajes normalmente no se suelen utilizar en los proyectos de software convencionales.

Normalmente, los proyectos de software se suelen dividir en dos equipos diferenciados: expertos del dominio y desarrolladores. Los expertos de dominio utilizan su jerga para transmitir el conocimiento y los desarrolladores lo traducen al suyo propio para poder discutir en términos de diseño, entorpeciendo la comunicación y produciendo un conocimiento anémico. Cuando el lenguaje está fracturado de esta manera, el proyecto se enfrenta a serios problemas debido a que la terminología utilizada en el día a día está desconectada de la terminología de la codificación. Incluso las mismas personas llegan a utilizar diferentes lenguajes cuando debaten y cuando codifican, por lo que expresiones importantes del dominio no suelen verse reflejadas en la codificación de la solución.

En DDD se presenta el concepto de un único equipo formado por todos los integrantes del proyecto. El objetivo es utilizar un **lenguaje ubicuo** y robusto que evite el coste asociado de la traducción y el riesgo de perder conocimiento a la hora de modelar el dominio. El equipo utiliza el modelo como pieza clave del lenguaje, comprometiéndose a ejercitarlo sin descanso en la comunicación y en el código. De esta manera, el código transmite los conceptos del negocio y se utilizan tanto en el debate del diseño como en la implementación de las funcionalidades del proyecto. El lenguaje está vivo y es dinámico, por lo que un cambio en el lenguaje supone un cambio en el modelo. Las posibles confusiones en la terminología son resueltas por consenso entre los integrantes del equipo.

Por lo tanto, el lenguaje ubicuo es la utilización de un lenguaje compartido dentro del equipo, en el que no importa el rol de cada miembro. Evidentemente, los expertos del dominio tienen gran influencia en el lenguaje ya que son los que conocen el negocio a gran escala. El trabajo entre los expertos de dominio y los desarrolladores es colaborativo, buscando la construcción de un modelo de dominio representable dentro y fuera del código.

Capturar todos los conceptos de una parte del negocio y representarlos en un lenguaje común no es una tarea sencilla y requiere un esfuerzo y una gran madurez por parte de todo el equipo. Algunos consejos a la hora de poder capturar el mayor conjunto del conocimiento son:

- Realizar diagramas con elementos conceptuales y físicos del dominio. Capturar los elementos e interacciones en el modelo. No tiene por qué ser un lenguaje técnico como UML, puede ser cualquier tipo de diagramas que permita que el equipo se comunique de manera fluida
- Creación de un glosario de términos con una definición lo más simple posible. Incluir una lista de términos alternativos y dejar constancia de por qué son menos adecuados que los consensuados
- Creación de cualquier tipo de documentación que permita entender y revisar el lenguaje utilizado por el equipo
- Todos los integrantes del equipo tienen que ser partícipes de la creación y documentación del lenguaje

El lenguaje ubicuo evoluciona y cambia con el tiempo, por lo que la documentación y el código del proyecto también lo hacen. Incluso es posible que se necesite modificar el comportamiento de cierta parte del código ya que los términos en el modelo hacen que esto se produzca.

El lenguaje común refleja los conceptos, procesos y eventos que se producen en el ámbito de la organización. Este lenguaje es único para cada subdominio y por consiguiente para cada contexto delimitado. A modo de ejemplo, un cliente es diferente desde el punto de vista de la realización de pedidos, de la autenticación en el sistema y el envío de notificaciones. Cada contexto tiene sus propias reglas de integridad y sus propios significados conceptuales. Es un error querer definir un lenguaje a nivel global ya que entonces será extenso, poco manejable y sobre todo poseerá conceptos desalineados con el lenguaje que el equipo utiliza.

Hay que tener presente que el objetivo principal de la creación de un lenguaje ubicuo es servir de reforzamiento de la comunicación dentro del equipo y de la afinidad conceptual del modelo llevado al código implementado [5], [6], [7].

5. PATRONES DE DISEÑO TÁCTICO

Los patrones catalogados como tácticos son los relacionados con el diseño e implementación del código propiamente dicho, es decir, son las herramientas utilizadas para la organización, representación e interacción de los modelos del dominio de un contexto delimitado mediante ficheros u objetos.

5.1. Arquitectura (Architecture)

Para el diseño e implementación de cualquier aplicación se tienen que realizar múltiples tipos de tarea: lógica de negocio, interacción con el usuario, comunicación otros sistemas, etc. Si se mezclan las responsabilidades entre las diferentes tareas se obtienen sistemas acoplados, poco cohesionados, complejos y difícilmente mantenibles y probables.

Las arquitecturas por capas son de gran utilidad para paliar la problemática mencionada anteriormente. El principio esencial que sustenta este tipo de arquitecturas es que un componente de una capa solo depende de otros componentes de esta o de capas inferiores, promoviendo el bajo acoplamiento y la alta cohesión. Evidentemente la elección de cada una de estas capas toma un papel relevante para un buen diseño arquitectónico.

Como definición clásica, este tipo de arquitectura se suele dividir en las siguientes capas:

- **Capa de presentación.** Muestra la información e interpreta las acciones realizadas por los usuarios
- **Capa de aplicación.** Coordina las diferentes interacciones de una tarea y delega el trabajo en los colaboradores de dominio. No contiene lógica de negocio
- **Capa de dominio.** Representa las reglas de negocio. Suele ser denominada como "el corazón del negocio"
- **Capa de infraestructura.** Proporciona mecanismos para la integración con proveedores externos: persistencia de los objetos de dominio, comunicación mediante mensajes, etc.

Es necesario concentrar todo el código relativo al modelado de dominio en una única capa y aislarla del resto de tareas asociadas. Los componentes de la capa de dominio no tienen como responsabilidad mostrar información al usuario, persistir sus propios datos, etc.

Los beneficios que se contienen con el uso de este tipo de arquitectura son:

- Código fácil de leer, entender, cambiar y mantener
- Código simétrico (predecible)
- Alta cohesión y bajo acoplamiento
- Modelos de dominio enriquecido
- Componentes aislados y separados por su responsabilidad
- Facilidad de realización de pruebas
- Útil para sistemas distribuidos
- Flexibilidad en el diseño

Para el diseño de aplicación que siguen el patrón DDD se suele utilizar la **arquitectura hexagonal (puertos y adaptadores)** ya que aísla el modelado del dominio de los componentes externos. Este aislamiento permite que los cambios en los componentes externos no afecten al modelado del dominio. Los cambios en el dominio solo deben permitirse cuando son realizados por criterios del negocio. Los puertos son una definición del contrato público y los adaptadores son la implementación de un puerto para un contexto en concreto.

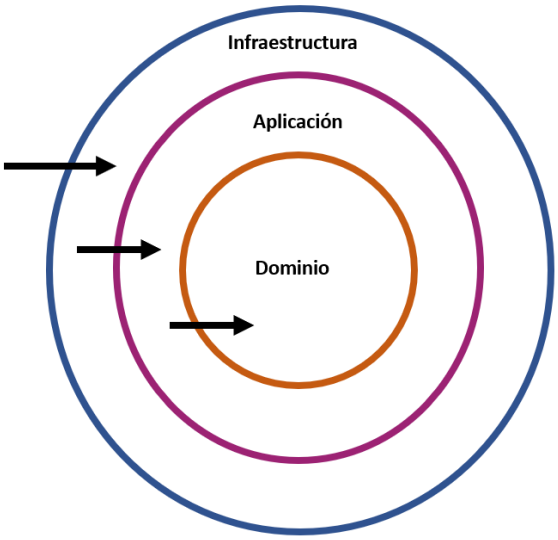

Figura 3: Capas de la arquitectura hexagonal junto con sus reglas de dependencia

En la arquitectura hexagonal (Figura 3), la regla de dependencia entre capas es clara, de fuera hacia adentro. Las capas que componen este tipo de arquitecturas son:

- **Capa de infraestructura.** Componentes externos con los que se tiene que comunicar la aplicación: mensajería, persistencia, etc.
- **Capa de aplicación.** Servicios que definen los puntos de entrada de la aplicación, aislando los elementos de infraestructura del modelado de dominio. Suelen ser los casos de uso
- **Capa de dominio.** Representación del propio contexto y de las reglas de negocio

La representación en código de este tipo de arquitectura se puede consultar en la Figura 4, donde cada módulo del contexto está formado por sus tres capas correspondientes. Como se puede apreciar estos módulos son predecibles y fáciles de mantener, ya que todos poseen la misma estructura de directorios.

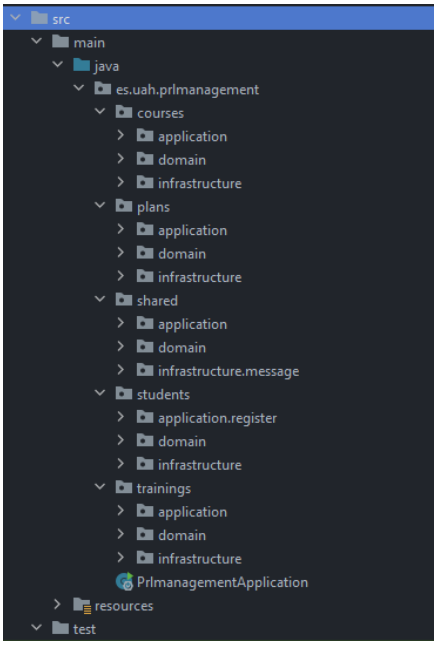

Figura 4: Estructura de carpetas en una arquitectura hexagonal

Adicionalmente a la arquitectura por capas, en el diseño DDD aparecen tres tipos de arquitecturas que pueden llegar a ser muy útiles dependiendo del contexto y que son resumidas a continuación:

- **REST.** Representational State Transfer. Conjunto de buenas prácticas que aportan una serie de contratos para la comunicación HTTP en aplicación web
- **CQRS.** Command Query Responsibility Segregation. Consiste en la separación de los modelos de lectura y escritura. Normalmente en cualquier aplicación se realizan más lecturas que escrituras de los datos, por lo que el trabajo de procesamiento se aconseja trasladarlo a los modelos de escritura, para después obtener los datos de manera rápida y escalable
- **Event-Driven Architecture.** Arquitectura dirigida a eventos. Los eventos son la parte central de la aplicación, donde la comunicación entre los diferentes componentes se realiza mediante estos eventos

El flujo de una petición REST en la arquitectura hexagonal se puede consultar en la Figura 5. El controlador forma parte de la capa de infraestructura y delega en un servicio de la capa aplicación la ejecución de la acción requerida por el usuario. El servicio de la capa de aplicación representa el caso de uso de manera atómica y coordina con ayuda de los elementos de la capa de dominio las tareas asociadas a este. Cabe destacar, que los repositorios son contratos de la capa de dominio que son implementados de manera específica en la capa de infraestructura [8], [9], [10], [11], [12].

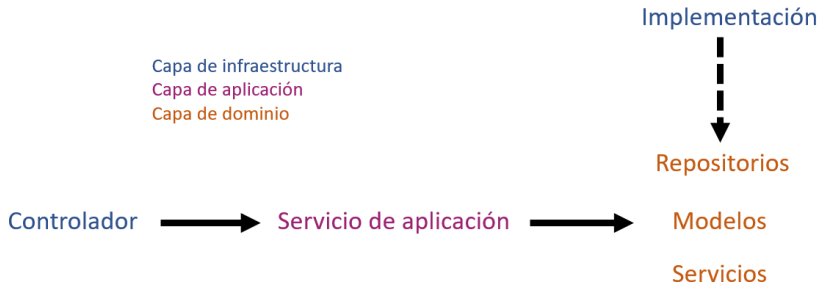

Figura 5: Ejemplo de flujo en una petición REST

5.2. Entidades (Entities)

A la hora de modelar el dominio hay elementos u objetos que tienen importancia por su individualidad, es decir, son distinguibles por su **identidad** y no por los atributos que los representan. Estos objetos tienen la característica de ser mutables, por lo que sus atributos pueden cambiar a lo largo del tiempo, mientras que su identidad evidentemente persiste a lo largo del ciclo de vida. A la hora de realizar estas modificaciones puede ser interesante saber cuándo, cómo y por qué han cambiado o simplemente importa el estado actual en el que se encuentran. Como los objetos se diferencias por su identidad, deben tener la capacidad de distinguirse de los demás incluso cuando sus atributos son idénticos. Por otro lado, deben poder ser iguales, aunque sus atributos no coincidan.

Cuando se van a implementar este tipo de conceptos hay que tener en cuenta varias consideraciones:

- Dotar de mecanismos semánticos a la mutabilidad de sus atributos, es decir, en lugar de utilizar la típica función de asignado de valor (setter), se debería emplear un método con un nombre autodescriptivo en el lenguaje ubicuo en que se encuentre inmerso
- Empujar el comportamiento y la lógica de negocio asociada hacia adentro de la entidad en lugar de tener modelos anémicos, es decir, objetos que solo poseen datos
- Utilización de mecanismos que ayuden a la hora de la generación de identidades: identificadores únicos universales (UUID), secuencias, etc.

En un dominio que gira alrededor de un cine puede ser necesario modelar las **butacas** de una sala donde se sientan los espectadores que quieren visualizar una película específica. Es interesante darse cuenta de la sutiliza a la hora de elegir el tipo de elemento que se debe diseñar, debido a la importancia que toma la numeración o no de estas butacas. En el caso de que estas butacas deban estar numeradas, se poseerá una identidad única que las diferencie del resto de butacas de la sala, pero, por el contrario, si no existe numeración, tampoco existirá identidad única y tomará relevancia el número total de estas. Debido a esta circunstancia, la elección en el diseño de una entidad es una tarea que requiere de un estudio de las necesidades en un dominio específico.

```java
package es.uah.prlmanagement.courses.domain;

import es.uah.prlmanagement.shared.domain.AggregateRoot;
import es.uah.prlmanagement.shared.domain.Identifier;
import es.uah.prlmanagement.trainings.domain.Training;

import java.util.List;
import java.util.Objects;

public class Course extends AggregateRoot {

    private final Identifier id;

    private final CourseName name;

    private final CourseCategory category;

    private List<Training> trainings;

    public Course(final Identifier id, final CourseName name, final CourseCategory category,
                  final List<Training> trainings) {
        this.id = id;
        this.name = name;
        this.category = category;
        this.trainings = trainings;
    }
```

Figura 6: Ejemplo de entidades

A la hora de implementar las entidades en código suelen verse reflejadas como las raíces de los agregados o como campos asociado a los agregados. En la Figura 6 se muestra el ejemplo del agregado **course** en el contexto de la prevención de los riesgos laborales, donde se posee una entidad **course** como raíz del agregado y una lista de entidades llamada **trainings**. Las entidades son parte de la capa de dominio en la arquitectura hexagonal, por lo que todas las importaciones que contienen son únicamente de la capa de dominio [13], [14], [15], [16].

5.3. Objetos de valor (Value Objects)

A diferencia de las entidades algunos conceptos del dominio no poseen identidad, sino que describen **características** de las cosas. Los objetos de valor son instancias que representan un aspecto descriptivo del dominio en el que nos importa el qué en lugar del quién o el cuál, es decir, toman relevancia los atributos y la lógica de un concepto del modelado. Estos objetos, aportan semántica y encapsulación al modelado del dominio, debido a que se obtienen tipos personalizados de datos.

Los objetos de valor no poseen identidad y son más fáciles de crear, utilizar, optimizar y mantener en el tiempo, reduciendo así la batalla constante que se libra con la complejidad inherente al diseño de software. Estos objetos se caracterizan por:

- Medir, cuantificar o describir un elemento conceptual del dominio:
    - La talla cuantifica la estatura de una persona
    - Las coordenadas de un punto describen su ubicación en el espacio
- Son inmutables, por lo que pueden ser completamente reemplazables cuando cambia la medida o la descripción, es decir, solo son modificables por constructor
- Modelados como un todo conceptual en el que se comparten atributos relacionados, es decir, el objeto de valor pueda estar constituido por uno o más atributos del concepto a modelar. Cada atributo aporta importancia al conjunto:
    - La representación de un valor monetario conlleva un valor y una unidad
    - La dirección postal de un cliente está formada por la calle, el número, la ciudad, el código postal, etc.
    - Un rango de fechas está constituido por una fecha de inicio y otra de fin
- Son comparados entre ellos por la igualdad del tipo y valor de sus atributos, es decir, los objetos que son comparados tienen los mismos valores en sus atributos, pero no son la misma instancia del objeto en memoria
- Proporcionan a sus colaboradores un comportamiento sin efectos colaterales (side effects), es decir, funciones que no modifican el estado, sino que producen una salida

Al igual que todos los elementos que forman parte del dominio, los objetos de valor deben contener el lenguaje ubicuo de la aplicación, haciendo que expresen el significado de los atributos que transmiten y de su funcionalidad relacionada.

A la hora de implementarlos en código se deben tener varias consideraciones al respecto:

- No poseen identificador
- Solo se debe poder instanciar los objetos de valor si cumplen con todas las reglas de negocio asociadas. Por lo tanto, las validaciones deben ir en el constructor
- Promover el uso de constructores semánticos, es decir, métodos que posean un nombre con lenguaje ubicuo, aportando semántica a la operación a realizar
- Son comparados por valor en lugar de por referencia
- Sus funciones no producen efectos colaterales, sino que producen salidas
- Se debe promocionar toda la lógica relacionada al objeto de valor, con el objetivo de obtener una alta cohesión. De esta manera, se evitan comprobaciones redundantes por todos los lugares de la aplicación
- Lanzar excepciones en caso de no cumplir con alguna regla de negocio. Estas excepciones deben estar ubicadas en la capa de dominio, ya que es parte del modelado del dominio

Normalmente en cualquier aplicación se poseerá un modelo que represente a los **usuarios** que interactúan con la aplicación. Si un usuario cambia su correo electrónico, contraseña o dirección, sigue siendo el mismo usuario que antes del cambio de estado. Estos conceptos pueden ser modelados como objetos de valor. Además, si por ejemplo la contraseña tiene reglas de negocio como poseer un número mínimo de caracteres o contener un carácter especial, empujaremos la lógica hacía adentro del objeto de valor y será utilizada a la hora de instanciación. Gracias a esto se consigue que cuando se tiene un objeto de valor, se tiene la certeza de que es una instancia válida y cumple con las reglas de negocio pertinentes, evitando de esta manera duplicar lógicas en diferentes lugares de la aplicación.

Cabe destacar, que depende del contexto en el que se realice el diseño del dominio, un elemento conceptual puede ser considerado como una entidad o un objeto de valor, por lo que se debe analizar con detenimiento las características particulares del elemento que se desea diseñar. Como se comentó en el apartado de entidades, una butaca de cine puede ser una entidad o un objeto de valor dependiendo el contexto en el que sea diseñado, es decir, dependerá de si las butacas están numeradas o no para determinar qué tipo de elemento debemos diseñar.

Un ejemplo de implementación en código en el contexto de la prevención de riesgos laborales es el objeto de valor que representa la **duración** en horas de una formación. Como se puede apreciar en la Figura 7, hay una función en la que se asegura que la duración cumple las reglas de negocio marcadas y en caso de violación se lanza una excepción que pertenece a la capa de dominio (ensureValidDuration). De esta manera nunca se puede instanciar un objeto de valor con valores inconsistentes. Estas reglas de negocio pueden ser todo lo complicadas que se necesite, aunque en este caso es simple para no generar ruido alrededor del ejemplo.

```java
package es.uah.prlmanagement.trainings.domain;

import es.uah.prlmanagement.shared.domain.IntegerVO;

public class TrainingDuration extends IntegerVO {

    private static final Integer DURATION_MIN = 1;
    private static final Integer DURATION_MAX = 120;

    public TrainingDuration(Integer value) {
        super(value);
        ensureValidDuration(value);
    }

    private void ensureValidDuration(Integer value) {
        if (value < DURATION_MIN || value > DURATION_MAX) {
            throw new TrainingDurationInvalid(value);
        }
    }
}

```

Figura 7: Ejemplo de objeto de valor que representa la duración de una formación

Todos los objetos de valor extienden de un tipo genérico de estos, para evitar código redundante a lo largo del modelado del dominio. En la Figura 8 se presenta el genérico para los objetos de valor que son números enteros, pero para el resto se sigue una estrategia similar. Todos estos genéricos encapsulan un tipo de dato presente en el lenguaje de programación Java: Integer, String, Boolean, etc. Cabe destacar que gracias al modificador final que proporciona Java se consigue que este tipo de objetos sean inmutables como precondición necesaria para el uso de este patrón [17], [18], [19], [20].

```java
package es.uah.prlmanagement.shared.domain;

import java.util.Objects;

public abstract class IntegerVO {

    private final Integer value;

    public IntegerVO(Integer value) {
        ensureValueNonNull(value);
        this.value = value;
    }

    public Integer value() { return value; }

    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (o == null || getClass() != o.getClass()) {
            return false;
        }
        IntegerVO integerVO = (IntegerVO) o;
        return value.equals(integerVO.value);
    }

    @Override
    public int hashCode() { return Objects.hash(value); }

    private void ensureValueNonNull(Integer value) {
        if (Objects.isNull(value)) {
            throw new ValueObjectIsNull();
        }
    }
}
```

Figura 8: Ejemplo de objeto de valor genérico

5.4. Servicios (Services)

Hay ocasiones en que se tienen que implementar operaciones que no pertenecen conceptualmente a ningún otro objeto en concreto, normalmente en los casos en que se necesita la coordinación entre varios objetos del dominio, pero esto no es un hecho imprescindible.

Como comenta Eric Evans en su libro, un **servicio de dominio** es una operación ofrecida como una interfaz, que se encuentra sola en el modelo y no encapsula estado, al contrario de lo que las entidades y los objetos de valor hacen.

El nombre de servicio enfatiza en la relación que existe entre varios objetos del dominio. Son nombrados con un verbo ya que representan una operación, acción o actividad. Deben ser diseñado con una única responsabilidad definida como parte del dominio con el lenguaje ubicuo presente. Los parámetros y resultados de sus operaciones deben ser objetos de dominio. Dotan de una herramienta muy importante en el diseño de software, la reutilización de código. Estos servicios poseen las siguientes características:

- Contienen la lógica de negocio
- La operación trata sobre un concepto del domino que no es parte natural de una entidad o de un objeto de valor
- El contrato de la operación a realizar tiene que ser definida en términos de otros elementos del modelo de dominio
- Son operaciones sin estado

Los servicios de dominio no son los únicos que nos podemos encontrar en un contexto delimitado, sino que existen otros dependiendo de la capa en la que actúen:

- **Servicios de aplicación.** Son la representación de forma atómica de los casos de uso y la parte visible al exterior. Su función principal es la de coordinar las relaciones entre las entidades, objetos de valor y servicios de dominio para poder realizar una operación de manera satisfactoria. No debería existir la comunicación entre varios servicios de aplicación. Por ejemplo, a la hora de crear un pedido, en el servicio de aplicación se coordina el flujo de la petición: comprobación de existencia del cliente, los productos tienen el suficiente stock, aplicación de los descuentos pertinentes, etc.
- **Servicios de infraestructura.** Resolución de problemas técnicos que no corresponden al dominio, como el proveedor de correo electrónico entre otros

A modo de ejemplo, se muestra el código de un servicio de aplicación en la Figura 9, más en concreto la representación del caso de uso de **registrar un estudiante** en la aplicación de prevención de riesgos laborales. Es importante fijarse que todas las importaciones de módulos son de objetos de dominio y nunca de aplicación o infraestructura. En la medida que sea posible, los objetos de dominio forman parte del mismo módulo o del compartido (shared). La función de registro de estudiantes recibe como parámetro un objeto DTO para traspasar información desde la capa de infraestructura hacia la de aplicación. Gracias al DTO se puede generar una instancia válida del estudiante con el evento de estudiante registrado en el interior. Después, coordina el registro del estudiante en el sistema de persistencia con el repositorio y publica el evento con el publicador de eventos.

```java
package es.uah.prlmanagement.students.application.register;

import es.uah.prlmanagement.shared.domain.EmployeeId;
import es.uah.prlmanagement.shared.domain.EventPublisher;
import es.uah.prlmanagement.shared.domain.Identifier;
import es.uah.prlmanagement.shared.domain.Service;
import es.uah.prlmanagement.students.domain.Student;
import es.uah.prlmanagement.students.domain.StudentActive;
import es.uah.prlmanagement.students.domain.StudentDepartment;
import es.uah.prlmanagement.students.domain.StudentEmail;
import es.uah.prlmanagement.students.domain.StudentFullname;
import es.uah.prlmanagement.students.domain.StudentIncorporatedDate;
import es.uah.prlmanagement.students.domain.StudentLocation;
import es.uah.prlmanagement.students.domain.StudentPosition;
import es.uah.prlmanagement.students.domain.StudentRepository;

@Service
public class StudentRegister {

    private final StudentRepository repository;

    private final EventPublisher eventPublisher;

    public StudentRegister(final StudentRepository repository, final EventPublisher eventPublisher) {...}

    public void execute(final StudentRegisterCommand studentRegisterCommand) {
        final Student student = Student.register(
            new Identifier(studentRegisterCommand.id()),
            new StudentFullname(studentRegisterCommand.fullname()),
            new EmployeeId(studentRegisterCommand.employeeId()),
            new StudentEmail(studentRegisterCommand.email()),
            new StudentActive(studentRegisterCommand.active()),
            new StudentIncorporatedDate(studentRegisterCommand.incorporatedDate()),
            new StudentDepartment(studentRegisterCommand.department()),
            new StudentPosition(studentRegisterCommand.position()),
            new StudentLocation(studentRegisterCommand.location())
        );

        repository.save(student);
        eventPublisher.execute(student.pullEvents());
    }
}
```

Figura 9: Servicio de aplicación que representa el caso de uso de registrar estudiantes

Además, los servicios son una potente herramienta que favorece las pruebas, sobre todo las pruebas unitarias, ya que permiten simular su comportamiento de una manera sencilla y elegante. En la Figura 10, se presenta un ejemplo claro de la simulación del repositorio y del publicador de eventos para el caso de uso de registrar un estudiante [21], [22], [23].

```java
package es.uah.prlmanagement.students.application.register;

import es.uah.prlmanagement.shared.domain.EventPublisher;
import es.uah.prlmanagement.students.domain.StudentRepository;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

import static org.mockito.ArgumentMatchers.any;
import static org.mockito.Mockito.times;
import static org.mockito.Mockito.verify;

@ExtendWith(MockitoExtension.class)
public class StudentRegisterTest {

    private StudentRegister studentRegister;

    @Mock
    private StudentRepository repository;

    @Mock
    private EventPublisher eventPublisher;

    @BeforeEach
    void setUp() {
        this.studentRegister = new StudentRegister(this.repository, this.eventPublisher);
    }

    @Test
    void shouldRegisterAStudent() {
        this.studentRegister.execute(StudentRegisterCommandFactory.example());

        verify(this.repository, times( wantedNumberOfInvocations: 1)).save(any());
        verify(this.eventPublisher, times( wantedNumberOfInvocations: 1)).execute(any());
    }
}
```

Figura 10: Clase de prueba para el caso de uso de registrar estudiantes

5.5. Módulos (Modules)

Los **módulos** son un elemento de gran aceptación dentro de la industria del software, con el objetivo de implementar aplicaciones modulares. Por lo tanto, es una forma que tienen los desarrolladores de organizar el código para conseguir beneficios tales como la alta cohesión, el bajo acoplamiento, la reutilización, etc.

En DDD los módulos deben aparecer como una parte importante del modelo, contando una historia del sistema a alto nivel. Eric Evans en su libro comenta que todos los desarrolladores utilizan módulos, pero pocos lo tratan como una parte completa del dominio. Además, explica que los módulos son un mecanismo de comunicación, incluso muestra una analogía similar a la siguiente: "Si una aplicación o sistema cuenta una historia, los módulos son los capítulos en lo que se divide".

Se deben crear módulos que cuenten historias con el lenguaje ubicuo del dominio, conteniendo un conjunto de conceptos cohesionados y produciendo a menudo un bajo acoplamiento entre estos (divide y vencerás). El código se puede dividir en todo tipo de categorías, desde aspecto arquitectónicos hasta casos de usos implementados por los desarrolladores.

A la hora de diseñar módulos en una aplicación se debe promover la alta cohesión de conceptos con responsabilidades relacionadas y el bajo acoplamiento para minimizar la carga conceptual y la independencia de estos.

En el contexto de DDD, los módulos sirven como contenedores de conceptos del modelo de dominio que están altamente cohesionados entre sí. En el lenguaje de programación Java es lo que se conoce como paquetes. Es recomendable seguir una serie de reglas a la hora de implementarlos en una aplicación planteada con el patrón DDD:

- Nombrado de módulos con el lenguaje ubicuo del dominio
- Diseño de módulos con bajo acoplamiento, con el objetivo de dotar de independencia a cada uno de ellos, en la medida que esto sea posible
- No crear módulos como un elemento estático, sino que puede cambiar con el tiempo. Además, pueden surgir casos en los que necesitemos promocionar de módulo a contexto delimitado
- Diseño de módulos ajustados a los conceptos a modelar

En la Figura 11 se visualizan los diferentes módulos que posee el contexto delimitado para la gestión de la prevención de riesgos laborales. Este contexto delimitado como cualquier otro, está constituido por sus módulos representados en lenguaje ubicuo junto con uno especial denominado compartido (shared). En una situación del mundo real, funcionalidades que aquí pertenecen al módulo compartido podrían estar perfectamente en un contexto delimitado compartido denominado núcleo compartido (shared kernel). Cabe destacar, que el directorio **main** contiene el código de producción y el directorio **test** contiene una representación similar de los módulos y contiene las clases que prueban el código de producción [24], [25], [26].

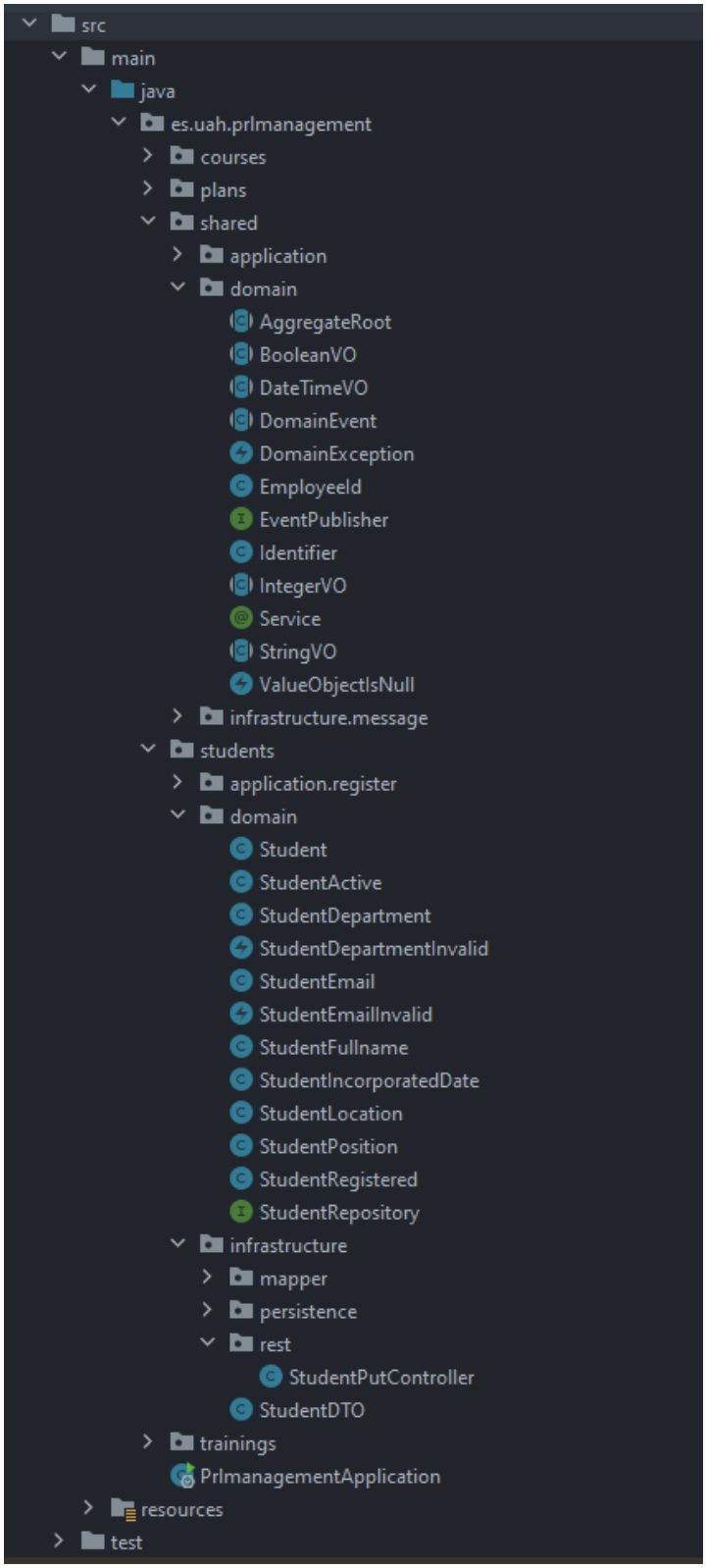

Figura 11: Ejemplo de módulos en el contexto de la gestión de la prevención de riesgos laborales

5.6. Agregados (Aggregates)

En un modelado de dominio con diversas asociaciones, es difícil garantizar la consistencia cuando se producen cambios. Además, es necesario mantener invariantes que se apliquen a un conjunto de objetos fuertemente relacionados, es decir, es imprescindible tratar a un grupo de objetos como una única unidad.

Imaginemos que tenemos una entidad pedido que contiene una lista de entidades denominada línea de pedido. ¿Qué ocurre si borramos el pedido, pero no las líneas o viceversa? A partir de este tipo de circunstancias, en DDD aparece el concepto de **agregado,** que es la forma de agrupar entidades y objetos de valor definiendo sus relaciones y límites, intentando simplificar la complejidad de las relaciones que intervienen. Cabe destacar que este agregado es un elemento conceptual.

Cada agregado está compuesto por un objeto **raíz** (root) y un **límite** alrededor (boundary). La raíz es una entidad específica y el límite define lo que constituye el agregado. Solo se pude acceder al resto de entidades y objetos de valor a través de la raíz, dotando de un mecanismo de control de integridad sobre el conjunto del agregado ante cambios de estado.

En el momento de realizar la implementación de los agregados hay que tener en cuenta:

- La entidad raíz posee una identidad global y es la responsable de controlar la integridad del conjunto del agregado. El resto de los objetos dentro del límite tienen una identidad local
- Solo se puede hacer referencia desde el exterior hacia la entidad raíz
- Solo la entidad raíz puede ser obtenida directamente a través de consultas en base de datos o lo que es lo mismo tener un repositorio. El resto de los objetos deben obtenerse gracias a la unión mediante asociaciones
- Se puede tener una referencia a la raíz de otro agregado, pero no a otra entidad u objeto de valor del exterior del propio agregado
- Una operación de borrado debe actuar en cascada, dicho de otra manera, debe eliminar todo lo que se encuentra dentro del límite del agregado
- Cuando se produce un cambio dentro de un objeto del agregado, todas las restricciones de este deben seguir cumpliéndose
- Mantener agregados pequeños con el objetivo de mejorar la escalabilidad de estos
- Por normal general, un agregado por módulo
- Utilización de dos principios muy utilizados para la generación de código limpio y mantenible:
    - **Law of Demeter.** Evitar el encadenamiento de llamadas de métodos, es decir, se debería llamar a un solo método que realice el conjunto de operaciones necesarias para evitar el acoplamiento y promover la cambiabilidad
    - **Tell, don't ask.** Agrupar los datos con las funciones que operan sobre los datos, es decir, en lugar de extraer de un objeto la información y operar sobre ella, se debe pasar la lógica al objeto en cuestión

Es importante mantener los agregados lo más simples posibles para controlar la expansión de la complejidad. Por ello, una entidad del agregado puede promocionar y convertirse en otro agregado, optando por mantener en el antiguo agregado los conceptos relevantes o una referencia al nuevo agregado. Además, si dos agregados están relacionados tienen que coordinar su lógica a través de los servicios de dominio. Supongamos que un agregado producto tiene una entidad valoración, pero en un

momento dado, la valoración empieza a complicarse y se decide extraer a otro agregado. Se puede optar por dos estrategias para la modificación del antiguo agregado:

- Mantener en el agregado de productos un objeto de valor que representa la puntuación ya que seguramente el resto de los datos no sean necesarios para los productos. De esta manera, las búsquedas serán más rápidas porque nos ahorraremos tener que hacer uniones en base de datos
- Crear un objeto de valor que represente a la identidad del nuevo agregado para poder obtenerlo en caso de necesidad

En un contexto en el que se realizan pagos, se necesitan generar **facturas** asociadas a estos pagos. Estas facturan poseen los productos facturados y otro conjunto de objetos de valor. Todo este grupo de objetos forma un nuevo agregado en el dominio y factura pasa a ser la raíz, ya que la importancia del conjunto gira en torno a esta. Esta raíz se encarga de asegurar la integridad del estado del grupo de objetos. En el momento en que se necesite borrar la factura, todos los demás elementos serán borrados en cascada, debido a que, sin la factura, el resto de los conceptos no tienen sentido de existencia.

Como ejemplo de implementación de este patrón se presenta un ejemplo del agregado **estudiante** en el contexto de la gestión de la prevención de los riesgos laborales. Los agregados son responsables de generar los eventos relacionados, por lo que se ha creado una abstracción para poder reutilizar cierto código en todos los agregados del sistema (Figura 12), es decir, la función de registrar evento dentro del agregado y la de devolverlos al exterior.

```java
package es.uah.prlmanagement.shared.domain;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public abstract class AggregateRoot {

    private List<DomainEvent> events = new ArrayList<>();

    public List<DomainEvent> pullEvents() {
        List<DomainEvent> events = this.events;
        this.events = Collections.emptyList();
        return events;
    }

    protected void registerEvent(final DomainEvent event) {
        this.events.add(event);
    }
}

```

Figura 12: Ejemplo abstracción de los agregados

Además de implementar la abstracción genérica, el agregado estudiante posee una serie campos que lo representan (Figura 13). Como el elemento raíz de este agregado es una entidad se necesita un campo que le de identidad y le diferencie del resto de estudiantes (id). Todos los demás campos son objetos de valor que aportan semántica y validan que se cumplen las restricciones de negocio. Cabe destacar que todos sus campos poseen el modificador final de Java, lo que significa que no pueden ser modificados.

```java
package es.uah.prlmanagement.students.domain;

import es.uah.prlmanagement.shared.domain.AggregateRoot;
import es.uah.prlmanagement.shared.domain.EmployeeId;
import es.uah.prlmanagement.shared.domain.Identifier;

public class Student extends AggregateRoot {

    private final Identifier id;

    private final StudentFullname fullname;

    private final EmployeeId employeeId;

    private final StudentEmail email;

    private final StudentActive active;

    private final StudentIncorporatedDate incorporatedDate;

    private final StudentDepartment department;

    private final StudentPosition position;

    private final StudentLocation location;

    public Student(final Identifier id, final StudentFullname fullname, final EmployeeId employeeId,
                   final StudentEmail email, final StudentActive active, final StudentIncorporatedDate incorporatedDate,
                   final StudentDepartment department, final StudentPosition position, final StudentLocation location) {...}

    public static Student register(final Identifier id, final StudentFullname fullname, final EmployeeId employeeId,
                                   final StudentEmail email, final StudentActive active,
                                   final StudentIncorporatedDate incorporatedDate, final StudentDepartment department,
                                   final StudentPosition position, final StudentLocation location) {
        Student student = new Student(id, fullname, employeeId, email, active, incorporatedDate, department, position,
            location);
        student.registerEvent(new StudentRegistered(id.value(), fullname.value(), employeeId.value(),
            email.value()));
        return student;
    }

    public Identifier id() { return id; }

    public StudentFullname fullname() { return fullname; }
```

Figura 13: Ejemplo del agregado de estudiantes

En caso necesario, es importante modelar dentro de los agregados todas sus funcionalidades asociadas en lugar de desplazar esa lógica a servicios de dominio. Esto se debe hacer en la medida que sea posible para ganar en cohesión, evitar dispersar la lógica asociada del agregado en diferentes componentes y construir modelos de dominio ricos. En la Figura 14 se muestra como la duración de un curso se obtiene desde dentro del agregado, en lugar de delegarlo en un servicio de dominio [27], [28], [29], [30].

```java
package es.uah.prlmanagement.courses.domain;

import es.uah.prlmanagement.shared.domain.AggregateRoot;
import es.uah.prlmanagement.shared.domain.Identifier;
import es.uah.prlmanagement.trainings.domain.Training;

import java.util.List;
import java.util.Objects;

public class Course extends AggregateRoot {

    private final Identifier id;

    private final CourseName name;

    private final CourseCategory category;

    private List<Training> trainings;

    public Course(final Identifier id, final CourseName name, final CourseCategory category,
                  final List<Training> trainings) {
        this.id = id;
        this.name = name;
        this.category = category;
        this.trainings = trainings;
    }

    public static Course create(final Identifier id, final CourseName name, final CourseCategory category,
                                final List<Training> trainings) {...}

    public Identifier id() { return id; }

    public CourseName name() { return name; }

    public CourseCategory category() { return category; }

    public List<Training> trainings() { return trainings; }

    public Integer duration() {
        return trainings.stream().mapToInt(training -> training.duration().value()).sum();
    }

```

Figura 14: Ejemplo del agregado curso

5.7. Factorías (Factories)

Las **factorías** son un patrón de diseño que sirven para crear objetos que llevan asociada una lógica de construcción que se desea ocultar, encapsulando la propia complejidad de instanciación. Cuando la construcción de un objeto se empieza a enmarañar, los constructores se quedan obsoletos, y se debe buscar una estrategia para que el objeto no se instancie a sí mismo, sino que se delegue esta funcionalidad a otro objeto del dominio.

La construcción de un objeto tiene una importancia vital en cualquier tipo de aplicación, pero las operaciones de ensamblaje complejas no son responsabilidad del objeto en sí mismo, ya que dan lugar a diseños difíciles de entender y mantener. Hacer que los clientes coordinen la construcción de elementos del dominio ensucian el diseño del propio cliente, perdiendo el principio de encapsulación del objeto a construir y acoplando excesivamente el cliente con la implementación del objeto. Los clientes no deben conocer la estructura interna del objeto que se desea construir, favoreciendo así la posible cambiabilidad de los elementos que intervienen en el proceso.

Las factorías encapsulan el conocimiento necesario para la creación de objetos complejos o agregados, proporcionando un contrato que refleja los objetivos del cliente y una visión abstracta sobre la creación del objeto en cuestión. Además, promueven el uso del principio de responsabilidad única (SOLID) y centralizan la lógica de construcción en un solo lugar, evitando así la duplicidad de código para la misma tarea. El uso de factorías lleva asociado una serie de características que son de obligado cumplimiento:

- Desplazan la construcción de un objeto complejo o agregado a otro objeto que es parte del dominio
- Cada método de creación es atómico y asegura la integridad y consistencia del objeto o agregado construido
- Utilización del lenguaje ubicuo del dominio presente
- No se añade lógica de negocio en las factorías

Existen diferentes patrones que se aconsejan utilizar a la hora de implementar factorías en DDD:

- **Servicio de dominio.** Función que se encarga de coordinar la creación del objeto complejo o agregado, centralizando de esta manera la lógica de instanciación en un único elemento del dominio
- **Factory method.** Permite que la abstracción delegue la construcción del objeto en la implementación. Suele utilizarse con los agregados, ya que la entidad raíz se encarga de añadir un elemento al propio agregado, ocultando la implementación interna al exterior
- **Abstract factory.** Permite la creación de un grupo de objetos, con el objetivo de garantizar la consistencia del conjunto conceptual que representa
- **Builder.** Dividir la construcción de un objeto complejo en diferentes partes

Hay circunstancias en la que no es necesario añadir la complejidad que la creación de una factoría acarrea y con la utilización de un constructor es suficiente:

- El proceso no supone complejidad
- El objeto es simple y no tiene herencia

- El objeto trabaja de manera aislada, es decir, no requiere la creación de otros elementos conceptualmente ligados
- El cliente posee todos los atributos para la instanciación y no se posee lógica asociada a la construcción
- En situaciones en que la implementación forma parte de la lógica de negocio y la responsabilidad es del objeto o del cliente, entonces se aconseja utilizar el patrón strategy

Imaginemos que en un contexto específico se necesita construir un agregado **producto** para poder manipularlo posteriormente. El flujo necesario para llevar a cabo esta tarea es:

1. El cliente hace uso del método de construcción de la factoría pertinente
2. La factoría se encarga de construir el agregado satisfaciendo las reglas de integridad, es decir, las reglas de negocio
3. La factoría devuelve una instancia consistente del agregado al cliente

Este patrón no solo es útil para la implementación del código de producción, sino que también a la hora de realizar las pruebas de este código. Todo desarrollador se encuentra con el dilema de crear la misma instancia de un objeto en múltiples ocasiones durante la realización de sus pruebas. Con el patrón de las factorías se pueden crear ejemplos de objetos reutilizables a lo largo de los escenarios de prueba que sean implementados. En la Figura 15 se muestra un ejemplo de una factoría de **estudiantes**, en la que se encuentra un ejemplo con los datos válidos. Sería interesante añadir un ejemplo que generase los datos de manera aleatoria como hace con el identificador y el código de empleado, para poder probar diferentes valores cada vez que se lanzan las pruebas tanto en el entorno local del desarrollador como en el proceso de integración continua [31], [32], [33], [34].

```java
package es.uah.prlmanagement.students.domain;

import es.uah.prlmanagement.shared.domain.EmployeeId;
import es.uah.prlmanagement.shared.domain.Identifier;

import java.time.LocalDateTime;
import java.util.UUID;

public class StudentFactory {

    public static Student example() {
        return new Student(
            new Identifier(UUID.randomUUID().toString()),
            new StudentFullname( value: "Alberto Murillo"),
            new EmployeeId(UUID.randomUUID().toString()),
            new StudentEmail( value: "alberto.murillo@uah.es"),
            new StudentActive(Boolean.TRUE),
            new StudentIncorporatedDate(LocalDateTime.now()),
            new StudentDepartment( value: "COMP"),
            new StudentPosition( value: "Engineer"),
            new StudentLocation( value: "Madrid")
        );
    }
}
```

Figura 15: Ejemplo de una factoría de estudiantes para las pruebas

5.8. Repositorios (Repositories)

La parte más importante de cualquier tipo de aplicación son los datos que la componen. En un momento dado, estos datos se encuentran en memoria, pero es necesario persistirlos en cualquier tipo de almacenamiento, para poder tratarlos en el futuro.

Los **repositorios** tienen como objetivo encapsular la lógica necesaria para el acceso y la manipulación de los datos de los objetos de dominio, estableciendo la complejidad técnica de la persistencia en la capa de infraestructura. Cuando se almacenan datos en un sistema de persistencia, más tarde se quieren volver a obtener en el mismo estado en que se encontraban. Por lo general, este tipo de persistencia suele ser un almacenamiento en un sistema gestor de base de datos.

Los repositorios son abstracciones (interfaces) que se encuentran en la capa de dominio, mientras que las diferentes implementaciones se encapsulan en la capa de infraestructura, donde las aplicaciones DDD permiten acoplarse a proveedores externos.

Es importante remarcar que las transacciones deben ser controladas por los clientes que hacen uso de los repositorios, ya que albergan el contexto global de la operación a realizar. Estos clientes ignoran la implementación de la capa de infraestructura ya que utilizan una interfaz (contrato) de la capa de dominio. De esta manera se consigue desacoplar el diseño del dominio de la tecnología utilizada para la persistencia de los datos. Esto se logra gracias al principio de inversión de dependencias (SOLID), que en este caso puntual consiste en utilizar una abstracción de un repositorio que se encuentra en la capa de dominio.

Algunas de las ventajas ofrecidas por el uso del patrón repositorio son:

- Presentar a los clientes un modelo sencillo para la obtención y manipulación de los objetos persistidos
- Desacoplar el diseño del dominio de la tecnología de persistencia utilizada: independiente de la estrategia y fuente de persistencia
- Permiten desarrollar una implementación ficticia, habitualmente en memoria para facilitar las pruebas del sistema. Como se debe poseer un contrato (interfaz) en la capa de dominio, permite simular su funcionamiento en las pruebas unitarias
- Mejora la cambiabilidad debido a que se utiliza una abstracción y no una implementación concreta, que se puede modificar según las necesidades de cada momento puntual
- Promueve el principio de responsabilidad única (SOLID), el bajo acoplamiento y la escalabilidad

Las factorías y los repositorios parecen ser similares, pero tienen distintas responsabilidades. Las factorías se utilizan para la construcción de nuevos objetos. Los repositorios se emplean para la manipulación de objetos persistidos. Aun así, pueden trabajar en conjunto para solucionar un problema del modelado del domino. Por ejemplo, un cliente necesita un agregado cualquiera, para ello se obtienen los datos a través de un repositorio y se apoya en una factoría para generar el agregado que necesita devolver al cliente.

La implementación de los repositorios debe proporcionar métodos para añadir y eliminar, que encapsulen la inserción y el borrado en el sistema de persistencia. Además, de proporcionar métodos para la selección de datos basándose en criterios significativos para los expertos del dominio. Si el filtrado de datos requiere de múltiples criterios, se recomienda el uso del patrón **criteria**. Todo este

conjunto de operaciones debe ser establecido con el lenguaje ubicuo del contexto en el que se está modelando el dominio. Los repositorios son de gran utilidad para los agregados. Solo se implementarán repositorios para las raíces de agregados a los que se necesite acceder.

Imaginemos que nos encontramos en un dominio en el que existe un agregado cuya entidad raíz es **pedido**. Estos pedidos contienen los **artículos** (entidad) elegidos por el cliente y una **dirección** de envío (objeto de valor). Por otro lado, tenemos agregados **clientes** que realizan los pedidos en el sistema. En el almacenamiento elegido se persisten los datos en tres tablas: pedidos, línea de pedidos y clientes. A partir de aquí, cuando se quieran manipular los datos de pedidos o clientes, se hará a través de los repositorios de pedidos o clientes respectivamente. En el código se hará uso de la abstracción (interfaz) de cada repositorio y después se poseerá una o varias implementaciones distintas en la capa de infraestructura acorde a las necesidades particulares: MySQL, PostgreSQL, MongoDB, etc.

A continuación, se presenta un ejemplo de aplicación real del patrón repositorio en el contexto de la gestión de la prevención de los riesgos laborales. El repositorio que se muestra se relaciona con el agregado de **estudiantes**. Como se puede apreciar en la Figura 16, el primer paso es la creación de una interfaz en la capa de dominio, donde se definen los métodos que forman el contrato que se debe cumplir. En este caso, dos funcionalidades para listar estudiantes y una para persistirlos.

```java
package es.uah.prlmanagement.students.domain;

import es.uah.prlmanagement.shared.domain.Identifier;

import java.util.List;
import java.util.Optional;

public interface StudentRepository {

    List<Student> findAll();

    Optional<Student> findOne(final Identifier id);

    void save(final Student student);
}
```

Figura 16: Ejemplo de la interfaz del repositorio de estudiantes

El segundo paso es crear la implementación de este repositorio (Figura 17). La implementación se realiza en la capa de infraestructura acoplándose a uno o varios proveedores externos, en este caso PostgreSQL y Spring Framework. Como hemos mencionado en el apartado de arquitectura, acoplarnos a proveedores externos a nivel de capa de infraestructura está permitido, pero no podemos contaminar la capa de dominio. Cabe destacar que se pueden tener varias implementaciones distintas para un repositorio, tanto para el código de producción como para el de pruebas [34], [35], [36], [37], [38].

```java
package es.uah.prlmanagement.students.infrastructure.persistence;

import es.uah.prlmanagement.shared.domain.Identifier;
import es.uah.prlmanagement.students.domain.Student;
import es.uah.prlmanagement.students.domain.StudentRepository;
import es.uah.prlmanagement.students.infrastructure.mapper.StudentMapper;
import org.springframework.stereotype.Repository;

import java.util.List;
import java.util.Optional;

@Repository
public class PostgresStudentRepository implements StudentRepository {

    private final StudentJpaRepository jpaRepository;

    private final StudentMapper mapper;

    public PostgresStudentRepository(final StudentJpaRepository jpaRepository, final StudentMapper mapper) {
        this.jpaRepository = jpaRepository;
        this.mapper = mapper;
    }

    @Override
    public List<Student> findAll() {
        return mapper.toDomainList(jpaRepository.findAll());
    }

    @Override
    public Optional<Student> findOne(final Identifier id) {
        return jpaRepository.findById(id.value()).map(mapper::toDomain);
    }

    @Override
    public void save(final Student student) {
        jpaRepository.save(mapper.toEntity(student));
    }
}
```

Figura 17: Implementación del repositorio de estudiantes con PostgreSQL

5.9. Eventos de dominio (Domain Events)

Los **eventos de dominio** son una herramienta poderosa para representar acciones que han sucedido en la aplicación. Normalmente un caso de uso codificado en un contexto delimitado conlleva una serie de acciones que se deben realizar, es decir, tiene múltiples responsabilidades. Con los eventos, se consigue que un caso de uso tenga una única responsabilidad y después que publique un evento de la acción realizada, permitiendo que otros componentes escuchen este evento y realicen las acciones asociadas pertinentes. Gracias a esto se consiguen los siguientes beneficios:

- Alta cohesión y bajo acoplamiento
- Promover el principio de responsabilidad única de SOLID
- Fomentar el principio de abierto para extensión, pero cerrado para modificación de SOLID
- Escalabilidad
- Robustez
- Resiliencia. De esta manera si una parte del sistema se encuentra fuera de servicio, el resto puede seguir trabajando hasta que este estado sea solucionado

No todo lo que gira alrededor de la utilización de eventos de dominio son ventajas, por lo que se deben tener en cuenta las siguientes consideraciones:

- Se añade complejidad
- Existe la consistencia eventual. En un momento del tiempo, una funcionalidad que se descompone en varias acciones puede no ser consistente, pero se debe garantizar que en un momento del futuro si lo será
- Gestión de los fallos producidos en la publicación del evento. Se suele utilizar una cola especial para almacenar los eventos que producen fallos (dead queue) y reaccionar según la política establecida
- Los eventos pueden llegar a las colas de mensajería de manera duplicada o desordenada
- Gestión de la transaccionalidad
- Sin un histórico de eventos no se puede conocer los estados intermedios por los que han pasado las instancias de los objetos actuales

Los eventos de dominio tienen como finalidad modelar la información sobre la actividad en el dominio de manera discreta, por lo que son traducidos a objetos de dominio que aportan semántica del negocio expresada con el lenguaje ubicuo pertinente. Están compuestos por:

- Descripción
- Fecha y hora de cuando ocurrió
- Identificador del evento
- Identidad del agregado involucrado en el evento
- Opcionalmente:
    - Fecha y hora de cuando se introdujo en el sistema
    - Identidad del quien lo introdujo

Llevado al terreno de la implementación, la raíz del agregado es la encargada del registro del evento. El publicador de eventos es el responsable de su publicación y los suscriptores son los manejadores para

realizar acciones asociadas. Dependiendo del contexto puede ser necesario tener que almacenar el histórico de eventos en un sistema de persistencia (event sourcing).

A modo de ejemplo se van a presentar los diferentes componentes u objetos que están presentes a la hora de **registrar un estudiante** en el contexto para la gestión de la prevención de riesgos laborales. Cuando se registra un estudiante, se tiene que guardar en el sistema de persistencia, enviar una notificación a su email, etc. Pero como se ha comentado al principio de este apartado, esto supone varias responsabilidades que van más lejos de un simple registro del estudiante. Por ello, el caso de uso que registra estudiantes, lo persiste y publica el evento asociado para que otros componentes realicen las tareas pertinentes. En la Figura 17 se muestra un ejemplo del código que comparten todos los eventos de la aplicación. Aunque aquí está representado dentro del módulo shared, es posible que esto forme parte del núcleo compartido (shared kernel). Al igual que los campos específicos de cada evento particular, son representados como escalares en lugar de como agregados, entidades u objetos de valor, evitando así el acoplamiento a dichos elementos.

```java
package es.uah.prlmanagement.shared.domain;

import java.time.LocalDateTime;
import java.util.UUID;

public abstract class DomainEvent {

    private final String aggregateId;

    private final String eventId;

    private final String occurredOn;

    public DomainEvent(final String aggregateId) {
        this.aggregateId = aggregateId;
        this.eventId     = UUID.randomUUID().toString();
        this.occurredOn  = LocalDateTime.now().toString();
    }

    public abstract String eventName();

    public String aggregateId() { return aggregateId; }

    public String eventId() { return eventId; }

    public String occurredOn() { return occurredOn; }
}
```

Figura 17: Campos comunes de cualquier evento de dominio

Dependiendo de las necesidades particulares, se extiende de la abstracción para crear el evento oportuno (Figura 18). El nombre del objeto para representar el evento está en pasado debido a que representa una acción que ha ocurrido. Si nos fijamos, este objeto sigue el patrón DTO, es decir, sirve como objeto para transportar datos.

```java
package es.uah.prlmanagement.students.domain;

import es.uah.prlmanagement.shared.domain.DomainEvent;

import java.util.Objects;

public class StudentRegistered extends DomainEvent {

    private static final String EVENT_NAME = "student.registered";

    private final String fullname;

    private final String employeeId;

    private final String email;

    public StudentRegistered(final String aggregateId, final String fullname, final String employeeId,
                             final String email) {
        super(aggregateId);
        this.fullname = fullname;
        this.employeeId = employeeId;
        this.email = email;
    }

    @Override
    public String eventName() { return EVENT_NAME; }

    public String fullname() { return fullname; }

    public String employeeId() { return employeeId; }

    public String email() { return email; }

    @Override
    public boolean equals(Object o) {...}

    @Override
    public int hashCode() { return Objects.hash(fullname, employeeId, email); }
}
```

Figura 18: Ejemplo del evento de dominio para el estudiante registrado

Como se indicó en la sección de implementación de agregados, estos son los encargados del registro de los eventos utilizando la función **registerEvent** en el constructor semántico pertinente (Figura 19).

```java
package es.uah.prlmanagement.students.domain;

import es.uah.prlmanagement.shared.domain.AggregateRoot;
import es.uah.prlmanagement.shared.domain.EmployeeId;
import es.uah.prlmanagement.shared.domain.Identifier;

public class Student extends AggregateRoot {

    private final Identifier id;

    private final StudentFullname fullname;

    private final EmployeeId employeeId;

    private final StudentEmail email;

    private final StudentActive active;

    private final StudentIncorporatedDate incorporatedDate;

    private final StudentDepartment department;

    private final StudentPosition position;

    private final StudentLocation location;

    public Student(final Identifier id, final StudentFullname fullname, final EmployeeId employeeId,
                   final StudentEmail email, final StudentActive active, final StudentIncorporatedDate incorporatedDate,
                   final StudentDepartment department, final StudentPosition position, final StudentLocation location) {...}

    public static Student register(final Identifier id, final StudentFullname fullname, final EmployeeId employeeId,
                                   final StudentEmail email, final StudentActive active,
                                   final StudentIncorporatedDate incorporatedDate, final StudentDepartment department,
                                   final StudentPosition position, final StudentLocation location) {
        Student student = new Student(id, fullname, employeeId, email, active, incorporatedDate, department, position,
            location);
        student.registerEvent(new StudentRegistered(id.value(), fullname.value(), employeeId.value(),
            email.value()));
        return student;
    }

    public Identifier id() { return id; }

    public StudentFullname fullname() { return fullname; }
```

Figura 19: Ejemplo del agregado de estudiantes que registra el evento

Por otro lado, el publicador de eventos es una interfaz que forma parte del dominio (Figura 20). En este caso, está alojado dentro del módulo shared pero como se comentó anteriormente, es posible que se ubique en el núcleo compartido (shared kernel).

```java
package es.uah.prlmanagement.shared.domain;

import java.util.List;

public interface EventPublisher {

    void execute(final List<DomainEvent> events);
}
```

Figura 20: Ejemplo de contrato de un publicador de eventos

Cada contexto debe implementar la solución que mejor considere para la publicación de eventos. Esta tarea se realiza en la capa de infraestructura debido al acoplamiento con un proveedor externo, en este caso RabbitMQ (Figura 21).

```java
package es.uah.prlmanagement.shared.infrastructure.message;

import es.uah.prlmanagement.shared.domain.DomainEvent;
import es.uah.prlmanagement.shared.domain.EventPublisher;
import org.springframework.amqp.core.AmqpTemplate;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

import java.util.List;

@Component
public class RabbitEventPublisher implements EventPublisher {

    @Value("${uah.rabbitmq.exchange}")
    private String exchange;

    @Value("${uah.rabbitmq.routingkey}")
    private String routingkey;

    private final AmqpTemplate rabbitTemplate;

    public RabbitEventPublisher(final AmqpTemplate rabbitTemplate) {
        this.rabbitTemplate = rabbitTemplate;
    }

    @Override
    public void execute(final List<DomainEvent> events) {
        events.forEach(event -> rabbitTemplate.convertAndSend(exchange, routingkey, events));
    }
}
```

Figura 21: Ejemplo de implementación de un publicador de eventos con RabbitMQ

En la Figura 22 se visualiza el flujo del caso de uso de registrar estudiantes. Gracias al agregado, primero se crea un objeto de estudiante válido que contiene dentro el evento de estudiante registrado. Después, se persiste el estudiante con el repositorio y se publica el evento con el publicador de eventos [39], [40], [41].

```java
package es.uah.prlmanagement.students.application.register;

import es.uah.prlmanagement.shared.domain.EmployeeId;
import es.uah.prlmanagement.shared.domain.EventPublisher;
import es.uah.prlmanagement.shared.domain.Identifier;
import es.uah.prlmanagement.shared.domain.Service;
import es.uah.prlmanagement.students.domain.Student;
import es.uah.prlmanagement.students.domain.StudentActive;
import es.uah.prlmanagement.students.domain.StudentDepartment;
import es.uah.prlmanagement.students.domain.StudentEmail;
import es.uah.prlmanagement.students.domain.StudentFullname;
import es.uah.prlmanagement.students.domain.StudentIncorporatedDate;
import es.uah.prlmanagement.students.domain.StudentLocation;
import es.uah.prlmanagement.students.domain.StudentPosition;
import es.uah.prlmanagement.students.domain.StudentRepository;

@Service
public class StudentRegister {

    private final StudentRepository repository;

    private final EventPublisher eventPublisher;

    public StudentRegister(final StudentRepository repository, final EventPublisher eventPublisher) {...}

    public void execute(final StudentRegisterCommand studentRegisterCommand) {
        final Student student = Student.register(
            new Identifier(studentRegisterCommand.id()),
            new StudentFullname(studentRegisterCommand.fullname()),
            new EmployeeId(studentRegisterCommand.employeeId()),
            new StudentEmail(studentRegisterCommand.email()),
            new StudentActive(studentRegisterCommand.active()),
            new StudentIncorporatedDate(studentRegisterCommand.incorporatedDate()),
            new StudentDepartment(studentRegisterCommand.department()),
            new StudentPosition(studentRegisterCommand.position()),
            new StudentLocation(studentRegisterCommand.location())
        );

        repository.save(student);
        eventPublisher.execute(student.pullEvents());
    }
}

```

Figura 22: Ejemplo de caso de uso de registrar estudiantes

6. PATRONES DE DISEÑO ESTRATÉGICO

Los patrones que engloban el diseño estratégico son las herramientas que facilitan la representación de la estructura organizacional y la coordinación de los equipos de la compañía. Además, aportan técnicas conceptuales sobre las que luego se apoyarán los patrones tácticos.

El objetivo principal que se persigue con el uso de estos patrones es la autonomía e independencia de los equipos.

6.1. Contextos delimitados (Bounded Context)

En una organización lo suficientemente grande, existen diferentes modelos que representan el dominio del negocio. Estos modelos se suelen unificar para poder ser utilizados en diferentes contextos de la aplicación.

El patrón estratégico más importante que orbita alrededor de DDD son los **contextos delimitados** cuya finalidad es la gestión de diferentes modelos y equipos de manera independiente. En DDD los modelos grandes se dividen en contextos delimitados donde se marca la frontera de responsabilidad de manera expresa, es decir, se define explícitamente el contexto en el que se aplica un modelo, estableciendo los límites en cuanto a la base de código y la organización del equipo.

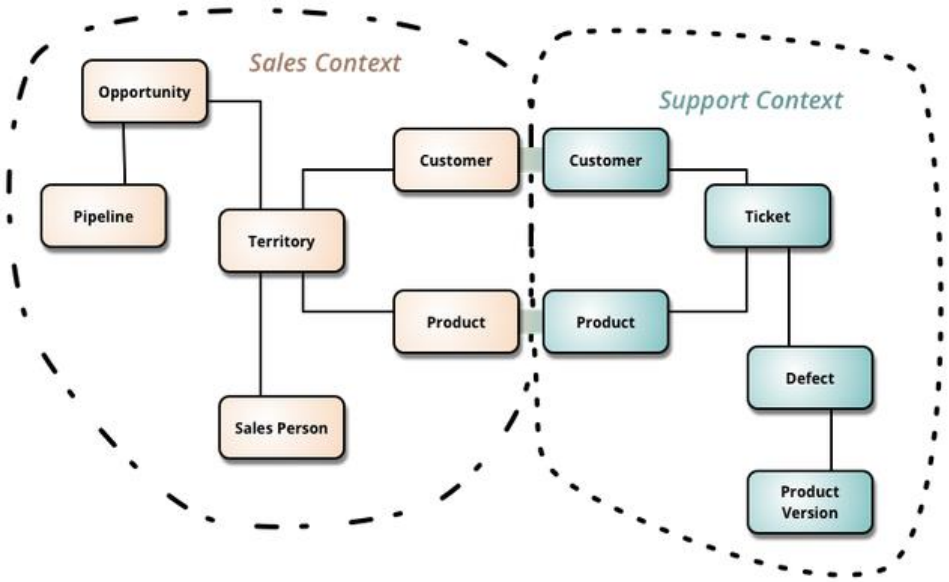

Figura 23: Bounded Context (Fowler, M. (2014). BoundedContext. [Figura]. Recuperado de https://martinfowler.com/bliki/BoundedContext.html)

Como se viene comentando a lo largo del documento, DDD consiste en diseñar e implementar software basándose en los modelos del dominio. Un modelo sirve como base conceptual para el diseño, es decir, su descomposición en objetos del dominio y funcionalidades. Es difícil y poco rentable construir un único modelo unificado para el sistema al completo, debido a que cada equipo posee su propio lenguaje ubicuo con su terminología sutilmente diferente del resto.

Al principio del documento se hablaba sobre subdominios como diferentes partes del problema a modelar. Los contextos delimitados son la traducción a nivel de solución de ese problema que se debía modelar, donde los términos del modelo se agrupan de manera coherente dentro de los límites establecido y sin preocupación por los problemas del exterior. Los objetivos principales que se persiguen con el diseño del software en diferentes contextos delimitados son:

- Promover el principio de responsabilidad única (SOLID) y la alta cohesión dentro del contexto
- División de la complejidad
- Reflejar la estructura organizacional de la empresa
- Dotar de autonomía a los equipos, evitando bloqueos y conflictos

No hay una recomendación específica sobre el tamaño idóneo de los contextos delimitados a la hora de su diseño, sino que se deben analizar las consecuencias:

- Contextos delimitados pequeños:
  - Proceso de integración continua sencillo con equipos y bases de código pequeñas
  - No requiere la unificación de modelos
  - Poseer diferentes modelos para el mismo concepto de dominio permite aplicar necesidades especiales y aplicar la jerga de grupos de usuarios especializados (lenguaje ubicuo)
- Contextos delimitados grandes:
  - La traducción entre dos modelos puede ser complicada e incluso en algunas ocasiones imposible
  - Es más sencillo entender un modelo que dos distintos con un mapeo
  - Con un modelo unificado, las tareas de los usuarios suelen ser más fluidas

Como norma general, a la hora de diseñar los contextos delimitados se debe tener en cuenta las siguientes consideraciones:

- Un subdominio tiene una relación de uno a uno con los contextos delimitados
- Un equipo gestiona uno o más contextos delimitados, pero un contexto nunca puede pertenecer a más de un equipo
- Si un módulo crece lo suficiente debe poder promocionarse de manera sencilla a contexto delimitado, creando un nuevo equipo o asignándole uno existente

Es importante dejar claro que un modelo del dominio en un contexto posee un lenguaje ubicuo propio, y por lo tanto un alcance del modelo como una parte limitada del sistema. La terminología que orbita alrededor de un cliente en el contexto de la realización de pedidos no tiene por qué coincidir con el contexto de notificaciones a estos clientes.

Es de utilidad apoyarse en el patrón estratégico de mapa conceptual, que se explica posteriormente, para representar el conjunto de contextos delimitados que conforman la organización y sus relaciones [3], [42], [43], [44].

6.2. Mapa contextual (Context Maps)

El **mapa contextual** abarca tanto la gestión de proyectos como el diseño de software. Uno de los primeros pasos del modelado de dominio es identificar cada modelo que está presente en la organización y definir sus límites alrededor, es decir, los contextos delimitados. Estos contextos son nombrados con el lenguaje ubicuo presente en la compañía y pueden necesitar la ayuda de otros contextos para conseguir el objetivo general de la organización.

No solo es importante tener una visión global de nuestro modelado del dominio sino también de la forma de integración de este. El mapa contextual consiste en trazar los puntos de contacto entre contextos, traduciendo de manera explícita la comunicación y los resultados obtenidos. La traducción es relevante debido a que dos contextos no tienen por qué compartir el mismo lenguaje ubicuo.

La traducción no tiene por qué ser un tipo de documentación en concreto, sino que se pueden utilizar diferentes mecanismos: diagramas, gráficos, texto, código, etc. El nivel de detalle que se aporta a la documentación varía dependiendo de las necesidades de cada contexto, pero siempre representando la situación actual del contexto oportuno. Además, dependiendo de la situación, pueden ser útiles como una estrategia de diseño efectiva, tanto en una etapa temprana como una tardía. Sobra decir que esta traducción toma un papel vital cuando se llevan a cabo comunicaciones con sistemas complejos.

Vulgarmente el mapa contextual es algo lógico que consiste en mapear términos y relaciones del mundo real en conceptos lógicos y físicos, es decir, una documentación. Los patrones más comunes para la integración entre contextos delimitados son presentados a continuación [45], [46], [47], [48].

6.2.1. Cooperación (Partnership)

Una relación de **cooperación** (Figura 24) se produce cuando los equipos de dos contextos delimitados van a triunfar o fracasar en conjunto. Los contratos (interfaces) se deben ajustar a las expectativas de ambos equipos para poder garantizar el éxito. Estos contratos son evolucionados por los equipos involucrados en la relación y se adaptan a las necesidades de estos.

Esta cooperación entre los equipos de los contextos implicados tiene que garantizar:

- Fácil implementación del productor
- Contrato adecuado para la implementación y desarrollo del consumidor

La cooperación se sustenta en una planificación sincronizada del diseño y una gestión conjunta de la integración. Por lo tanto, las funcionalidades asociadas tienen que realizarse en la misma versión el producto. Evidentemente, si en un lado se producen alteraciones o problemas, estos deben ser solucionados a raíz de la coordinación exhaustiva de ambos equipos y sin comprometer a ninguno de los contextos involucrados [46], [48], [49].

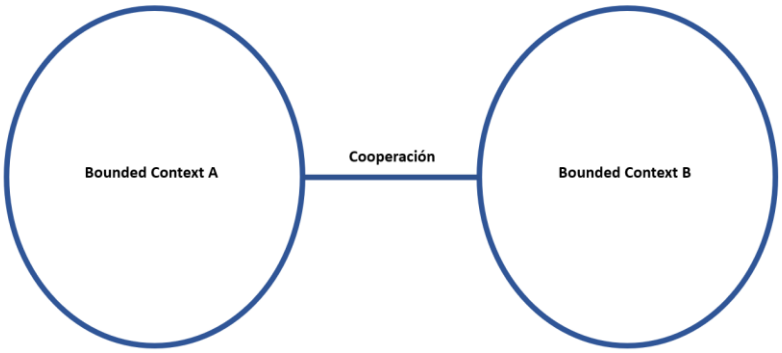

Figura 24: Representación de una relación de cooperación

6.2.2. Núcleo compartido (Shared Kernel)

Hay ocasiones en la que varios contextos delimitados tienen la necesidad de compartir ciertas reglas del negocio que normalmente son parte del dominio principal de la organización.

El **núcleo compartido** (Figura 25) consiste en la creación de un contexto delimitado que sirva para compartir y reutilizar parte del dominio principal entre diferentes contextos delimitados y, por tanto, entre diferentes equipos de la organización. Se delimita un subconjunto del modelo que los diferentes equipos involucrados deciden compartir, formando un contrato que estos deben cumplir. Aunque los diferentes equipos son libres de la modificación del núcleo, estos cambios deben llevarse a cabo con el consenso y autorización de todos los equipos implicados. Apoyarse en un proceso de integración continua facilita mantener el núcleo compartido alineado con el lenguaje ubicuo de los equipos involucrados.

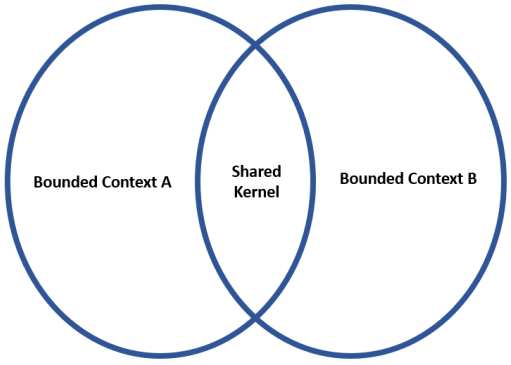

Figura 25: Representación del núcleo compartido

Por norma general un contexto delimitado es gestionado y desarrollado por un equipo, con el objetivo de conseguir la independencia a todos los niveles. En este caso, el núcleo compartido es gestionado y desarrollado por varios equipos de los contextos implicados, desbaratando esa norma, pero aportando un beneficio relevante, la reutilización de un lenguaje común del dominio principal, evitando la duplicidad de la misma funcionalidad en diferentes contextos delimitados.

Se debe tener en cuenta que, si la parte de código que se desea alojar en el núcleo compartido es muy extensa, quizás nos encontremos en una situación en la que los contextos involucrados deban unirse en un contexto mayor.

Este núcleo suele contener ciertas funcionalidades o tipos trasversales compartidos entre los diferentes contextos. Algunos ejemplos son: objetos de valor generales, representación de los identificadores, excepciones comunes, mecanismo de autorización o comunicación, etc. [46], [48], [50], [51]

6.2.3. Productor - Consumidor (Customer - Supplier)

Normalmente un contexto necesita de otro para poder realizar su tarea al completo. El **productor** y el **consumidor** (Figura 26) de este tipo de relaciones están separados en dos contextos delimitados diferentes y actúan en una sola dirección. En este tipo de relaciones, el equipo productor puede lograr el éxito independientemente del consumidor.

El diseño e implementación del equipo productor puede verse obstaculizado si el equipo consumidor tiene poder sobre los cambios que se realizan en el productor o si el proceso de solicitud de cambios es una tarea tediosa. Además, el productor puede dejar de hacer modificaciones en las funcionalidades existentes por miedo a romper los posibles consumidores.

El diseño e implementación del equipo consumidor puede verse mermado por la dependencia en las prioridades que tiene el productor. Es importante establecer una relación transparente entre los equipos del productor y consumidor, por lo que las prioridades del equipo consumidor deben ser tenidas en cuenta en la planificación del equipo productor.

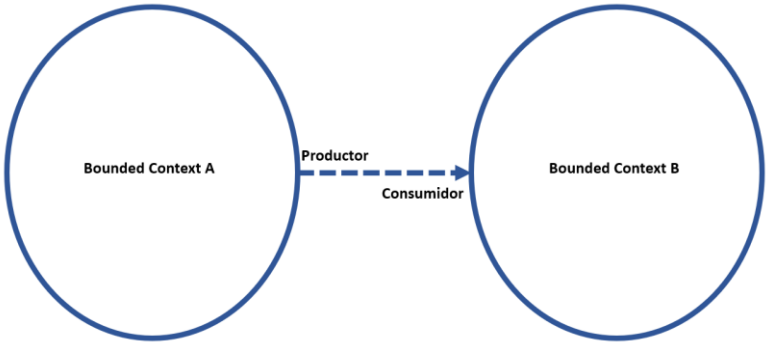

Figura 26: Representación de la relación productor - consumidor

Los equipos ágiles toman una relevancia importante en este tipo de relaciones haciendo que los consumidores tomen el papel de cliente en los productores y poder ser tratados de manera prioritaria. Otro punto relevante es la realización conjunta en las pruebas automáticas de aceptación que validan el contrato que el productor ofrece e incluirlas en el proceso de integración continua del productor, con el objetivo de garantizar que los cambios realizados en el productor no afectan al consumidor.

A modo de resumen, hay dos elementos clave en este tipo de patrón. El consumidor es fundamental. Por lo tanto, se debe producir un proceso de negociación y evaluación en la relación entre productor y consumidor, tanto en el calendario como en los resultados obtenidos. Es necesario realizar un conjunto

de pruebas automáticas de aceptación que permiten cambiar el productor sin miedo de afectar al consumidor [46], [48], [52], [53].

6.2.4. Conformista (Conformist)

Este tipo de integraciones se llevan a cabo cuando el consumidor se acopla, adapta y **conforma** con el modelado de dominio que ofrecen los servicios que expone el productor, es decir, en este tipo de relaciones, el productor no tiene ninguna motivación específica para satisfacer las necesidades del consumidor. Por tanto, el equipo consumidor hereda el lenguaje ubicuo del productor.

En este tipo de relaciones se suele dar la situación en la que el equipo productor realiza promesas que probablemente no se cumplen en el tiempo y compromete los planes del consumidor ya que realizan planes basados en esas promesas. En el equipo consumidor se producen retrasos hasta que el equipo aprende a vivir bajo esta premisa. No se realizan contratos para los clientes específicos, sino que el consumidor se conforma con el modelado del dominio de los servicios ofrecidos por el productor. Este tipo de relaciones suele ocurrir en grandes empresas en que la gestión de los equipos no posee una visión estratégica común.

Este tipo de integraciones no es muy aconsejable, pero sí que son simples de llevar al terreno. Por ello, solo se recomienda cuando la calidad y compatibilidad entre el consumidor y productor es buena. En caso contrario, se debería optar por una de las siguientes acciones:

Productor y consumidor toman caminos separados, analizando las posibles implicaciones asociadas

Creación de una capa de anticorrupción cuando el sistema productor posee un diseño complejo que carece del principio de encapsulación, abstracciones difíciles de utilizar o un modelado que no se ajusta al protocolo del consumidor [46], [48], [54], [55].

6.2.5. Capa de anticorrupción (Anticorruption Layer)

Cuando se diseñan aplicaciones a menudo nos encontramos con sistemas en los que el código implementado es difícil de entender, mantener y evolucionar. En otras ocasiones no se posee control sobre ese sistema o su modelado difiere del nuevo a implementar.

La **capa de anticorrupción** (Figura 27) es un mecanismo que permite comunicarse e integrarse con otro sistema mediante una capa aislada en la que el consumidor posee la traducción de la funcionalidad ofrecida por el productor. Esta capa de anticorrupción se comunica con otros sistemas sin o con pequeñas modificaciones de estos, traduciendo el modelo de dominio en una o en ambas direcciones, evitando posibles contaminaciones entre modelos.

Con este patrón se produce un acoplamiento a nivel de adaptador en la capa de infraestructura, pero nunca a nivel de modelado de dominio, evitando de esta manera contaminar este dominio con el de otros sistemas externos. Como consecuencia, esta capa de anticorrupción debe ser implementada en un servicio de dominio.

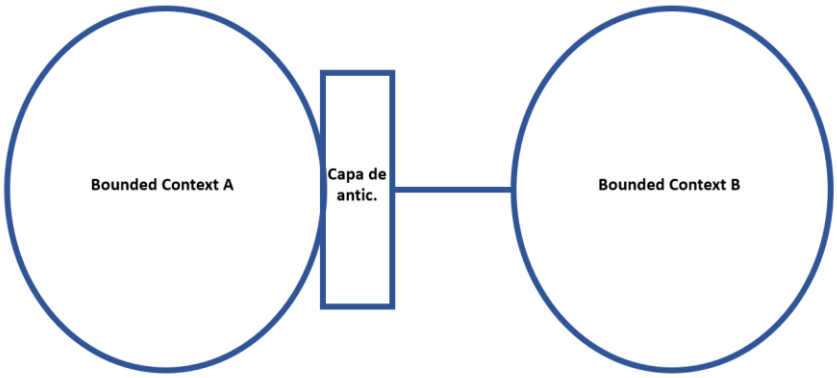

Figura 27: Representación de la capa de anticorrupción

Este patrón es muy útil cuando hay que integrarse con un sistema que es el núcleo del negocio y se caracteriza por su valor y complejidad, incluso cuando hay que realizar una migración de este a otro paradigma o arquitectura diferente. Estos sistemas comúnmente denominados **legacy** no tienen por qué estar mal modelados, sino que es necesario un gran esfuerzo en el mantenimiento y evolución de sus funcionalidades. Gracias a este patrón, la funcionalidad deseada queda encapsulada dentro de una o varias capas de anticorrupción.

Hay algunas consideraciones inherentes a la utilización de este patrón:

- Coste en análisis, diseño e implementación
- Hay que asegurar que se mantiene la cohesión en el proceso de traducción
- Crece el número de servicios de dominio que deben ser mantenidos
- Pueden añadirse latencias

Imaginemos que nos encontramos en un sistema complejo y antiguo con una funcionalidad que forma parte del núcleo del negocio y se desea evolucionar, pero el equipo de desarrollo no se siente seguro modificando este sistema directamente ya que no quieren romper nada de lo que hasta ahora funcionaba correctamente. La solución óptima es la creación de una capa de anticorrupción que utilice la funcionalidad mencionada anteriormente, y realizar las extensiones pertinentes en el sistema que si posee el control y la seguridad pertinente [46], [48], [56], [57], [58].

6.2.6. Caminos separados (Separate Ways)

Es importante ser exhaustivo a la hora de definir los requisitos y posteriormente analizar las implicaciones específicas que conllevan. Si dos funcionalidades no tienen una relación relevante, se deben separar cada una de ellas, fomentado el principio de responsabilidad única y permitiendo seguir **caminos separados**.

La integración entre dos contextos delimitados es costosa y en ocasiones el beneficio obtenido no es relevante. Además, el coste asociado a la coordinación de equipos y al compromiso en la integración continua de las funcionalidades no es un tema baladí.

Por lo tanto, en la medida que sea posible, es interesante definir los contextos delimitados aislados y sin comunicación con los demás, posibilitando encontrar soluciones sencillas y desacopladas. Además,

los equipos son totalmente independientes y pueden realizar evoluciones de los modelos de manera rápida y eficaz.

Cabe destacar que, con esta aproximación, a veces es mejor duplicar cierta lógica que incluir una integración entre diferentes equipos. Como la mayoría de las decisiones organizacionales, deben tomarse con un exhaustivo análisis de las implicaciones asociadas y valorando los beneficios e inconvenientes [46], [48], [59], [60].

6.2.7. Servicio de anfitrión abierto (Open Host Services)

Cualquier contexto delimitado que forma parte del modelado de dominio puede integrarse con uno o varios contextos, por lo que generar un traductor para cada posible interacción sería un trabajo costoso y poco productivo. Cada vez se necesitan más traductores que mantener y por lo tanto más cosas de las que preocuparse cuando se producen cambios.

La solución más natural es generar un protocolo de acceso al contexto que integrarse como un conjunto de **servicios**. Estos servicios están abiertos al "público" para ofrecer funcionalidades a los contextos que necesiten integrarse. Se trata de un protocolo vivo que puede mejorarse y extenderse con el objetivo de atender nuevas necesidades de integración.

Cada consumidor tiene su propia implementación a partir de los servicios expuestos en el productor. Incluso pueden existir ocasiones en las que un equipo tenga necesidades especiales en las que se tenga que crear un traductor único, sin afectar al protocolo compartido (servicios) que puede seguir siendo entendible y mantenible [46], [48], [61], [62].

Este tipo de comunicación se conoce como APIs públicas: API REST, RPC, etc.

6.2.8. Lenguaje publicado (Published Language)

Para que dos contextos delimitados pueden interactuar se precisa una traducción entre modelos como llave a un lenguaje común. El **lenguaje publicado** entra en juego cuando se desea realizar una documentación que exprese la información del dominio necesaria como medio de comunicación entre los contextos del productor y consumidor. Es habitual en organizaciones grandes establecer lenguajes publicados como forma de intercambiar la información.

Este patrón suele utilizarse en conjunto con los servicios de anfitrión abierto del apartado anterior, ya que sirve como mecanismo de documentación de APIs. Además, en la actualidad, se apoya en formatos como JSON para el intercambio de la información [46], [48], [63].

6.2.9. Código espagueti (Big Ball of Mud)

Todos los desarrolladores a lo largo de su carrera profesional se suelen encontrar con grandes partes de una aplicación en la que los modelos se mezclan y los límites a su alrededor son incoherentes. Esto se presenta tanto en sistemas con muchos años de existencia o sistemas diseñados e implementados con rapidez y falta de experiencia.

De esta manera es fácil **empantanarse en el lodo** a la hora de intentar gestionar los límites de los modelos que se encuentran en estos sistemas, por lo que se recomienda no aplicar el patrón DDD para generar modelos sofisticados en este contexto. Por otro lado, se sugiere permanecer atento en el diseño e implementación de DDD para no llegar a esta situación.

Por lo tanto, más que un patrón este apartado habla sobre un anti-patrón a la hora de diseñar DDD, ya que se presenta esta circunstancia con el propósito de evitar generar sistemas que sean un auténtico lío con modelos que se mezclan y límites inconsistentes.

Este tipo de situación es habitual encontrárselas en el código **legacy** de una organización en el que la estructura general nunca estuvo definida explícitamente o se ha deteriorado con el paso del tiempo. La mayoría de su **código** es **espagueti**, estructurado de manera desordenada, acoplado entre sí, poco cohesionado en sus módulos y duplicado.

En el caso de poseer un sistema con estas cualidades en nuestra organización, la recomendación es encapsularlo como si de un contexto delimitado se tratase y utilizar una capa de anticorrupción para consumir los servicios ofrecidos [46], [48], [64].

6.3. Integración Continua

La **integración continua** es una herramienta estrechamente relacionada con el agilismo. El trabajo de los equipos puede producir desalineamiento del modelo en el contexto delimitado para el que trabajan. Como consecuencia, es necesario realizar un proceso de validación de los nuevos cambios introducidos en el contexto.

Cada contexto delimitado debe poseer un proceso de validación y publicación del código implementado, apoyándose en pruebas automatizadas con el objetivo de conseguir:

- Detectar y arreglar el funcionamiento anómalo del código con mayor rapidez
- Mejorar la productividad y la calidad del software
- Entregar nuevas versiones del sistema con mayor agilidad

Este proceso de integración continua suele estar compuesta por las siguientes etapas:

- Garantizar la calidad del nuevo código, con herramientas como el análisis estático de código
- Automatización de la construcción del artefacto con los nuevos cambios
- Ejecución de las pruebas automatizadas

Este proceso de integración continua tiene que ser ágil por lo que no debe prolongarse en el tiempo. Es de gran importancia conseguir un balance entre el conjunto de pruebas automatizadas que se realizan:

- **Unitarias.** Son las más rápidas y prueban un trozo del código
- **Integración.** Comprueban que la interacción entre componentes es la adecuada. Son más lentas que las unitarias porque prueban la conexión y comportamiento real entre los componentes involucrados. A menudo se utilizan para garantizar el buen funcionamiento con el sistema de persistencia, mensajería, etc.
- **Aceptación.** Son lentas y comprueban el flujo completo de una funcionalidad. Sirven como contrato para garantizar que los cambios en el sistema no romperán a sus consumidores
- **Otras.** Depende de las necesidades de la organización. Algunos ejemplos son las pruebas de rendimiento, de estrés, etc.

Dependiendo de las necesidades de cada contexto, este proceso se puede realizar tanto con un simple fichero lanzado manualmente por el desarrollador como con alguna herramienta ofrecida por un proveedor externo (Jenkins, GitLab Runner, etc.) [65].

7. BENEFICIOS E INCONVENIENTES

El código desarrollado en un sistema que implementa el patrón DDD expresa el lenguaje que se habla en la organización, **mejorando la comunicación** entre los diferentes integrantes la compañía, en especial entre los expertos del dominio y los expertos técnicos. El código es más **legible** y **mantenible** desde el primer momento.

El patrón DDD genera un diseño e implementación de arquitecturas que promueven la **alta cohesión**, el **bajo acoplamiento**, la **tolerancia al cambio**, la **flexibilidad**, la **agilidad** y la mejor **organización** y **autonomía** del equipo. Además, aporta una serie de otros patrones que son muy útiles para el diseño rico de los modelos del dominio, incluso permitiendo que la **complejidad de las pruebas** se vea **reducida** al máximo.

Todos los integrantes de la organización comparten la misma **visión estratégica**, es decir, luchan por conseguir el mismo objetivo.

Pero si con la utilización del patrón DDD se obtuvieran solo ventajas, todas las organizaciones lo tomarían como un estándar de diseño e implementación de aplicaciones. Este patrón añade un grado de **complejidad** que no todos los equipos están capacitados para asumir, o en el caso de que lo estuvieran, pueden no resultar beneficiosos para el contexto en particular.

El conocimiento y buena utilización de todos los principios, patrones y procedimientos que orbitan alrededor de DDD requieren de una **alta curva de aprendizaje** para el equipo, tanto la parte de negocio como la parte técnica. Además, es necesario aportar al equipo **expertos en el dominio**.

El diseño de modelos ricos y acotar sus responsabilidades suponen un aumento en el **tiempo**, **esfuerzo** y **coste** de las tareas de análisis, diseño e implementación.

8. VENCER Y CONVENCER

Miguel de Unamuno comentó "Venceréis, pero no convenceréis", remarcando que lo importante no solo es lograr el objetivo sino demostrar que realmente es el adecuado.

A continuación, se exponen una serie de recomendaciones para motivar tanto al equipo como a la organización a la hora de utilizar el patrón de diseño DDD:

1. **Paso a paso.** Tener en cuenta la curva de aprendizaje que supone en el **equipo**, donde nadie se puede quedar atrás
2. **Mejorar en términos de arquitectura.** Identificar la fuente de conflictos en el código actual y exponer los beneficios asociados de diseñar e implementar la complejidad del modelado de dominio con DDD
3. **Aportar valor a la organización, facilitar la comunicación** con cualquier integrante del equipo y **dotar de autonomía** a estos
4. **Alinear la visión estratégica de la organización** con la de todos los integrantes de esta, alejándose de las modas puntuales
5. **Economía del software.** El código legible, mantenible y cambiable aporta beneficios sustanciales. Además, permite aportar nuevas funcionalidades con mayor brevedad

9. RESUMEN Y CONCLUSIÓN

Eric Evans aporta una nueva visión a la hora de diseñar software, centrándose no solo en la parte técnica sino también en la organizativa. Domain-Driven Design, Diseño Dirigido por el Dominio o DDD proporciona una serie de patrones tanto tácticos como estratégicos que permiten un diseño e implementación del modelado del dominio utilizando un lenguaje común entre los expertos del dominio y desarrolladores.

Domain-Driven Design es un tipo de patrón muy útil para grandes organizaciones con una complejidad elevada alrededor de su dominio, ya que aporta una serie de ventajas sustanciales: autonomía, independencia, escalabilidad, cohesión, flexibilidad, agilidad, etc.

9.1. Conclusiones y Futuras Líneas de Trabajo

El patrón de diseño Domain-Driven Design o Diseño Dirigido por el Dominio dota de una serie de herramientas de gran utilidad tanto para la implementación técnica de la solución como para la coordinación y comunicación a nivel organizacional.

Esto no significa que sea el elixir para dar solución a todos los problemas que presentan las aplicaciones que se desean llevar a la práctica en el mercado laboral, sino que actúa de una manera muy eficaz en ciertas situaciones, normalmente en organizaciones grandes que poseen un nivel de complejidad elevado en su negocio.

Aplicando este patrón de diseño en las circunstancias adecuadas se pueden conseguir beneficios relevantes: alta cohesión, bajo acoplamiento, tolerancia al cambio, mantenibilidad, flexibilidad, autonomía de los equipos, mejora en la comunicación, etc.

Todo lo aprendido a la hora de realizar este proyecto constituye la primera piedra en el diseño limpio de aplicaciones para el autor del documento. Este modelado se llevará a la práctica tanto en proyectos personales como profesionales cuando la problemática se ajuste a lo comentado a la largo del documento. Además, este patrón convive con otra serie de herramientas que serán estudiadas en un futuro próximo debido a la relevancia e interés que suscitan en el autor del documento: Command Query Responsibility Segregation, Event-Driven Architecture, etc.

10. BIBLIOGRAFÍA

[1] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Sotware*. United States: Addison-Wesley, 2004, pp. 47-49

[2] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Sotware*. United States: Addison-Wesley, 2004, pp. 397-438

[3] V. Vernon. *Implementing Domain-Driven Design*. United States: Addison-Wesley, 2013, pp. 43-84

[4] E. Evans. *Domain-Driven Design Reference: Definitions and Pattern Summaries*. United States: Dog Ear Publishing, 2014, pp. 40-42

[5] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Sotware*. United States: Addison-Wesley, 2004, pp. 24-29

[6] V. Vernon. *Implementing Domain-Driven Design*. United States: Addison-Wesley, 2013, pp. 20-25

[7] E. Evans. *Domain-Driven Design Reference: Definitions and Pattern Summaries*. United States: Dog Ear Publishing, 2014, pp. 3-4

[8] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Sotware*. United States: Addison-Wesley, 2004, pp. 67-80

[9] V. Vernon. *Implementing Domain-Driven Design*. United States: Addison-Wesley, 2013, pp. 113-169

[10] E. Evans. *Domain-Driven Design Reference: Definitions and Pattern Summaries*. United States: Dog Ear Publishing, 2014, pp. 10

[11] R. C. Martin. (2012, agosto, 13). The Clean Architecture. [En línea]. Disponible: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

[12] E. Salguero. (2018, junio, 22). Arquitectura Hexagonal. [En línea]. Disponible: https://medium.com/@edusalguero/arquitectura-hexagonal-59834bb44b7f

[13] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Sotware*. United States: Addison-Wesley, 2004, pp. 89-96

[14] V. Vernon. *Implementing Domain-Driven Design*. United States: Addison-Wesley, 2013, pp. 171-218

[15] E. Evans. *Domain-Driven Design Reference: Definitions and Pattern Summaries*. United States: Dog Ear Publishing, 2014, pp. 11

[16] J. Sánchez. (2014, febrero, 7). Patrones de diseño en DDD: Entidades. [En línea]. Disponible: http://xurxodeveloper.blogspot.com/2014/02/patrones-de-diseno-en-ddd-entidades.html

[17] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Sotware*. United States: Addison-Wesley, 2004, pp. 97-103

[18] V. Vernon. *Implementing Domain-Driven Design*. United States: Addison-Wesley, 2013, pp. 219-264

[19] E. Evans. *Domain-Driven Design Reference: Definitions and Pattern Summaries*. United States: Dog Ear Publishing, 2014, pp. 12

[20] M. Fowler. (2016, noviembre, 14). ValueObject. [En línea]. Disponible: https://martinfowler.com/bliki/ValueObject.html

[21] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Sotware*. United States: Addison-Wesley, 2004, pp. 104-108

[22] V. Vernon. *Implementing Domain-Driven Design*. United States: Addison-Wesley, 2013, pp. 265-284

[23] E. Evans. *Domain-Driven Design Reference: Definitions and Pattern Summaries*. United States: Dog Ear Publishing, 2014, pp. 14

[24] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Sotware*. United States: Addison-Wesley, 2004, pp. 109-115

[25] V. Vernon. *Implementing Domain-Driven Design*. United States: Addison-Wesley, 2013, pp. 333-346

[26] E. Evans. *Domain-Driven Design Reference: Definitions and Pattern Summaries*. United States: Dog Ear Publishing, 2014, pp. 15

[27] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Sotware*. United States: Addison-Wesley, 2004, pp. 125-135

[28] V. Vernon. *Implementing Domain-Driven Design*. United States: Addison-Wesley, 2013, pp. 347-388

[29] E. Evans. *Domain-Driven Design Reference: Definitions and Pattern Summaries*. United States: Dog Ear Publishing, 2014, pp. 16

[30] Y. Pérez. (2012, febrero, 29). El patrón Agregado (Aggregate). [En línea]. Disponible: http://yagoperez.com/patron_agreggate/

[31] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Sotware*. United States: Addison-Wesley, 2004, pp. 136-146

[32] V. Vernon. *Implementing Domain-Driven Design*. United States: Addison-Wesley, 2013, pp. 389-400

[33] E. Evans. *Domain-Driven Design Reference: Definitions and Pattern Summaries*. United States: Dog Ear Publishing, 2014, pp. 18

[34] J. Cuellar. (2016, diciembre, 9). Domain-Driven Design: Factories & Repositories. [En línea]. Disponible: https://josecuellar.net/domain-driven-design-factories-repositories/

[35] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Sotware*. United States: Addison-Wesley, 2004, pp. 147-162

[36] V. Vernon. *Implementing Domain-Driven Design*. United States: Addison-Wesley, 2013, pp. 401-448

[37] E. Evans. *Domain-Driven Design Reference: Definitions and Pattern Summaries*. United States: Dog Ear Publishing, 2014, pp. 17

[38] G. Sánchez. (2019, agosto, 15). DDD: Repositories. [En línea]. Disponible: https://happydevops.com/2019/08/15/ddd-repositories/

[39] V. Vernon. *Implementing Domain-Driven Design*. United States: Addison-Wesley, 2013, pp. 285-332

[40] E. Evans. *Domain-Driven Design Reference: Definitions and Pattern Summaries*. United States: Dog Ear Publishing, 2014, pp. 13

[41] M. Fowler. (2005, diciembre, 12). Domain Event. [En línea]. Disponible: https://martinfowler.com/eaaDev/DomainEvent.html

[42] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Sotware*. United States: Addison-Wesley, 2004, pp. 335-340

[43] E. Evans. *Domain-Driven Design Reference: Definitions and Pattern Summaries*. United States: Dog Ear Publishing, 2014, pp. 2

[44] M. Fowler. (2014, enero, 14). BoundedContext. [En línea]. Disponible: https://martinfowler.com/bliki/BoundedContext.html

[45] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Sotware*. United States: Addison-Wesley, 2004, pp. 344-353

[46] V. Vernon. *Implementing Domain-Driven Design*. United States: Addison-Wesley, 2013, pp. 87-112

[47] E. Evans. *Domain-Driven Design Reference: Definitions and Pattern Summaries*. United States: Dog Ear Publishing, 2014, pp. 29

[48] M. Plöd. (2021, marzo, 16). Context Mapping. [En línea]. Disponible: https://github.com/ddd-crew/context-mapping

[49] E. Evans. *Domain-Driven Design Reference: Definitions and Pattern Summaries*. United States: Dog Ear Publishing, 2014, pp. 30

[50] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Sotware*. United States: Addison-Wesley, 2004, pp. 354-355

[51] E. Evans. *Domain-Driven Design Reference: Definitions and Pattern Summaries*. United States: Dog Ear Publishing, 2014, pp. 31

[52] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Sotware*. United States: Addison-Wesley, 2004, pp. 356-360

[53] E. Evans. *Domain-Driven Design Reference: Definitions and Pattern Summaries*. United States: Dog Ear Publishing, 2014, pp. 32

[54] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Sotware*. United States: Addison-Wesley, 2004, pp. 361-363

[55] E. Evans. *Domain-Driven Design Reference: Definitions and Pattern Summaries*. United States: Dog Ear Publishing, 2014, pp. 33

[56] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Sotware*. United States: Addison-Wesley, 2004, pp. 364-370

[57] E. Evans. *Domain-Driven Design Reference: Definitions and Pattern Summaries*. United States: Dog Ear Publishing, 2014, pp. 34

[58] M. Narumoto. (2017, julio, 23). Patrón Anti-Corruption Layer. [En línea]. Disponible: https://docs.microsoft.com/es-es/azure/architecture/patterns/anti-corruption-layer

[59] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Sotware*. United States: Addison-Wesley, 2004, pp. 371-373

[60] E. Evans. *Domain-Driven Design Reference: Definitions and Pattern Summaries*. United States: Dog Ear Publishing, 2014, pp. 37

[61] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Sotware*. United States: Addison-Wesley, 2004, pp. 374

[62] E. Evans. *Domain-Driven Design Reference: Definitions and Pattern Summaries*. United States: Dog Ear Publishing, 2014, pp. 35

[63] E. Evans. *Domain-Driven Design Reference: Definitions and Pattern Summaries*. United States: Dog Ear Publishing, 2014, pp. 36

[64] E. Evans. *Domain-Driven Design Reference: Definitions and Pattern Summaries*. United States: Dog Ear Publishing, 2014, pp. 38

[65] D. Radigan. (2021, julio, 1). Integración continua. [En línea]. Disponible: https://www.atlassian.com/es/agile/software-development/continuous-integration

11. APÉNDICE. GLOSARIO

A

API. Application **P**rogramming **I**nterface. Consiste en un conjunto de protocolos y servicios utilizados para la integración entre varios componentes software

C

CQRS. Command **Q**uery **R**esponsibility **S**egregation. Patrón en el que se diferencia separa entre los modelos de lectura y escritura.

Contexto Delimitado. Definición explícita del contexto de aplicación de ciertos modelos del dominio

D

DDD. Domain-**D**riven **D**esign o Diseño Dirigido por el Dominio

DTO. Data **T**ransfer **O**bject. Objeto utilizado para la comunicación entre diferentes capas de la aplicación.

Dominio. Ámbito de conocimiento de una organización

E

Event-Driven Architecture. Arquitectura dirigida a eventos.

H

HTTP. Hyper**t**ext **T**ransfer **P**rotocol. Protocolo de comunicación estándar de comunicación web

J

JSON. Java**S**cript **O**bject **N**otation. Formato para intercambio de datos

L

Lenguaje ubicuo. Lenguaje compartido por todos los integrantes del equipo

M

Modelo. Sistema de abstracción que describe los aspectos seleccionados de un dominio y pueden utilizarse a la hora de resolver problemas relacionados con el dominio

R

REST. Representational **S**tate **T**ransfer. Estilo arquitectónico de software para aplicaciones web

RPC. Remote **P**rocedure **C**all. Permite ejecutar código que está en otra máquina

S

SOLID. **S**ingle responsibility, **O**pen-closed, **L**iskov substitution, **I**nterface segregation and **D**ependency inversion. Es una serie de principios para el buen diseño de software

X

XML. E**x**tensible **M**arkup **L**anguage. Es un lenguaje de marcado utilizado para el traspaso de datos