Diseño e implementación de una Arquitectura de Software guiada por el dominio

Federico Ezequiel Difabio

Director: Mauro Germán Cambarieri

UNRN Sede Atlántica

2021

Trabajo final de grado para optar al título de Licenciado en Sistemas

Agradecimientos

En primer lugar a mi familia, papá y mamá que siempre me alientan a seguir, a cumplir mis objetivos, acompañándome en todo momento, creyendo y aportando en cada paso que doy. Demostrando que cada logro tiene su esfuerzo.

A Lucas y Guille, mis ejemplos a seguir, los que me guiaron en el camino de esta profesión que comenzó desde la primaria y continúa hasta el día de hoy. A ustedes infinitas gracias.

A Caro, por tanta paciencia y compañía en esta etapa y especialmente por siempre motivarme a seguir formándome y crecer profesionalmente.

A todos mis amigos/as y compañeros/as, por toda la ayuda brindada durante estos años, tantas horas de estudio, cursadas y proyectos compartidos.

A la Universidad Pública por darme la oportunidad de estudiar y en particular a la Universidad Nacional de Río Negro por concretar esta etapa educativa y profesional.

Agradezco a todos mis profesores por haber colaborado en mi formación y desarrollo profesional, en especial a todos mis compañeros del Laboratorio de Informática Aplicada. A mi director de trabajo final Mauro, por toda la ayuda, motivación y horas de trabajo para llegar a este momento.

Y por último quisiera hacer extensiva mi gratitud al equipo del proyecto IPDTT, que sin ellos esto no sería posible.

A todos, gracias, sin ustedes esto no sería posible.

Índice

Resumen

El diseño de software bajo un enfoque sólido, sistemático, y completo cuenta con un conjunto de herramientas y técnicas, que permiten separar la complejidad del negocio, dejando como pieza central del mismo, el dominio.

El enfoque de diseño dirigido por el dominio permite contar con principios, patrones y actividades para construir un modelo de dominio, que es el artefacto principal. Además, garantiza que la arquitectura de software permanezca centrada en las funcionalidades del negocio.

Por su parte las arquitecturas limpias, entre ellas la arquitectura hexagonal, permiten separar las responsabilidades mediantes regiones o capas, permitiendo desacoplar las mismas y que evolucionen de manera aisladas, cuyo núcleo central es el dominio.

En este trabajo se propone un desarrollo de software y la aplicación de una arquitectura orientada al dominio. La contribución del mismo es mostrar la viabilidad sobre la adopción del enfoque, así como el valor estratégico, el cual proporciona mapear la idea del dominio del negocio para el desarrollo de los artefactos de software.

El trabajo presenta el diseño de una arquitectura que permite el desarrollo de software guiado por el dominio y la selección de tecnologías para su implementación, el cual se valida mediante un caso de estudio.

De esta forma, se desarrolla una plataforma digital orientada a una red de perfiles del personal de salud del Hospital de Viedma, que permite la comunicación e interacción entre la comunidad hospitalaria.

Palabras claves
Diseño Dirigido por el Dominio, Arquitectura Limpia, Arquitectura Hexagonal.

1. Objetivos

El objetivo general de este proyecto es diseñar e implementar una arquitectura de software que permita el desarrollo guiado por el dominio. El mismo consiste en investigar y desarrollar herramientas informáticas, cuyos dominios sean complejos de resolver.

Los objetivos específicos son:

1. Integrar herramientas existentes que soportan las metodologías y enfoques de desarrollo de software.
2. Diseñar una arquitectura de referencia que permita el desarrollo de software guiado por el dominio, manteniendo el modelo de negocio en sincronía con la implementación de la solución.
3. Implementar la solución propuesta mediante un caso de estudio.

2. Marco teórico

2.1. Introducción

La ingeniería de software ofrece métodos, técnicas y herramientas dirigidas a mejorar el proceso de desarrollo de software. Existen varios factores para garantizar el éxito o la calidad del mismo, como herramientas a utilizar, la arquitectura de software y la metodología que guiará el proceso. Es un factor de éxito la elección de las mismas.

La arquitectura de software brinda una visión abstracta de alto nivel de sus componentes, y la relación entre ellos, permitiendo plantear la reutilización y la evolución del código (Vivas; 2013). Por otro lado, existen enfoques de desarrollo de software cuyo valor estratégico es mapear la idea del dominio del negocio en los artefactos del software (Penchikala; 2008), identificando el problema relevante y permitiendo construir arquitecturas de software para conectar la implementación a un modelo en evolución de la idea principal del negocio. En la academia y en la industria, existen enfoques y herramientas como el diseño dirigido por el dominio (DDD por sus siglas en inglés Domain Driven Design) y las arquitecturas limpias (CA por sus siglas en inglés Clean Architecture).

El diseño dirigido por el dominio ofrece un enfoque sólido, sistemático y complejo para el diseño y desarrollo de software. Proporciona un conjunto de herramientas y técnicas que ayudan a separar la complejidad del negocio mientras mantiene como pieza central del enfoque, el modelo de dominio (Nair; 2019). DDD proporciona un medio de representar el mundo real en la arquitectura de software, a través de un patrón central y estratégico, como son los contextos delimitados (Evans; 2015), y permiten tener un modelo unificado. El modelo en el contexto delimitado actúa como lenguaje ubicuo para ayudar a la comunicación entre técnicos y expertos en el dominio.

Por su parte las arquitecturas limpias permiten separar las responsabilidades mediantes regiones o capas, permitiendo desacoplar las mismas y que evolucionen de manera aisladas, cuyo núcleo central es el dominio.

La contribución del mismo es mostrar la viabilidad sobre la adopción del enfoque, así como el valor estratégico, el cual proporciona mapear la idea del dominio del negocio para el desarrollo de los artefactos de software. Se presenta la construcción de la arquitectura centrada en el dominio específico del negocio y la selección de tecnologías que permiten su implementación.

El trabajo propuesto se validó mediante un caso de estudio, en el cual se diseñó una arquitectura que permite el desarrollo de software guiado por el dominio y la selección de tecnologías que permitieron su implementación. A su vez, para mostrar su viabilidad se desarrolló una plataforma digital, el cual utiliza la arquitectura propuesta.

A continuación el trabajo, se estructuró de la siguiente manera:

2.2. Conceptos utilizados, donde se investigaron Diseño dirigido por el dominio, diseño estratégico, diseño táctico, separación de responsabilidades comandos y consultas, lenguaje específico del dominio, arquitectura de software, arquitectura limpia, regla de dependencia y arquitectura hexagonal.

3. Arquitectura de software y entorno de trabajo, se describe el enfoque y la arquitectura propuesto y las tecnologías seleccionadas.

4. Caso de estudio donde se describen los casos de uso más relevantes y se eligió un caso de uso para clarificar y ejemplificar la solución. Además se especifica la construcción, diseño e implementación de la arquitectura propuesta.

5. Herramientas y entornos donde se mencionan las herramientas utilizadas para la construcción, gestión y documentación del mismo.

2.2. Conceptos utilizados

2.2.1. Diseño dirigido por el dominio

El diseño dirigido por el dominio (DDD por sus siglas en inglés Domain Driven Design) es un enfoque de desarrollo de software que permite tener un modelo de negocio en sincronía con la implementación de la solución. Este enfoque, permite desarrollar proyectos de software cuya complejidad está dada por dominios complejos (Vivas; 2013). Eric Evans, su autor, expone un conjunto de prácticas, técnicas y principios de diseño, que al aplicarlos asegura un aumento en la capacidad de modelar e implementar problemas complejos.

La complejidad más significativa de muchas aplicaciones, no es técnica, se encuentra en el dominio como el proceso o reglas de negocio. Es por ello que para el éxito de un diseño, se debe tratar sistemáticamente este aspecto central en el desarrollo de software. Hay dos aspectos a tener en cuenta: 1) El enfoque principal

debe ser el dominio y la lógica del negocio; 2) Los diseños de dominios complejos deben basarse en un modelo.

Evans, propone el diseño estratégico y táctico para la implementación del enfoque, cuyo orden es definido según el estado de avance del proyecto.

2.2.2. Diseño estratégico

A medida que los sistemas crecen se vuelven complejos, difíciles de manipular y comprender. Por ello, este enfoque define principios de diseño estratégicos que proporcionan una guía para las decisiones de diseño del modelo que reducen interdependencia de las partes y mejora la claridad y facilidad de comprensión y análisis (Vivas et. al.; 2013).

Su objetivo es crear un proceso iterativo con continuas reuniones entre expertos en dominios (personas que conocen el negocio) y los técnicos (equipo de desarrollo), permitiendo descubrir dominios y perfeccionando un modelo diseñado para resolver un problema empresarial.

Este proceso de hallazgo de dominio es la base estratégica de DDD en el cual entran en juego dos conceptos muy importantes, el lenguaje ubicuo y los contextos delimitados (BC, por sus siglas en inglés Bounded Context).

El lenguaje ubicuo es una colección de términos específicos del dominio, definido por los expertos del dominio del negocio. Este lenguaje emplea todas las formas de comunicación entre el equipo de desarrollo del proyecto y expertos del dominio y esencialmente es utilizado en el código fuente. Esto asegura que los conceptos del negocio y el modelo de dominio que se está diseñando permanezcan en sincronía y que el modelo, en efecto, hable sobre el negocio. Es decir, el modelo actúa como un lenguaje ubicuo.

Los contextos delimitados determinan los límites dentro de los cuales existe y opera cada modelo de dominio. Los conceptos de dominio dependen del contexto en el que existen. Este es un aspecto crucial y poderoso para la modelización de dominios, dado que permite que los modelos en diferentes contextos evolucionen independientemente unos de otros. Esto tiene un efecto significativo en el mantenimiento porque hace más simple aplicar y probar los cambios en un modelo que se centra únicamente en su propio dominio. El modelo sólo cambiará si cambian las reglas en su propio contexto (Nair; 2019).

2.2.3. Diseño táctico

Posteriormente de la etapa estratégica se inicia el proceso táctico donde se adoptan los patrones de arquitectura para diseñar la solución, al igual que las tecnologías a utilizar. Para llegar a una implementación sin perder la fuerza de DDD se requiere plantear la conexión con el modelo para hacerse a nivel de detalle (Vivas et. al.; 2013). Este diseño táctico consiste en definir los modelos de dominio con más precisión. Los patrones tácticos se aplican dentro de un único contexto delimitado.

Entre ellos se destacan los patrones de agregados, entidades, objetos de valor, servicios de dominios. Las entidades representan continuidad e identidad, por medio de diferentes estados o incluso en diferentes implementaciones, y los objetos de valor son un atributo el cual describe el estado. Los agregados pueden verse como un objeto de valor, la diferencia radica en la dependencia, el agregado si o si debe ser referenciado al objeto raíz (Vivas et. al.; 2013). Es decir, que el agregado no puede ser una colección, ya que hace referencia a más de un objeto de dominio.

Por otra parte, los servicios de dominio representan las acciones u operaciones. Aplicar estos patrones ayuda a identificar los límites naturales de los servicios en nuestra aplicación.

Existen dos escenarios posibles para implementar este enfoque: 1) Proyecto Iniciado o 2) Proyecto nuevo. Para los proyectos con alguna funcionalidad implementada, se inicia por la parte táctica y de a poco la parte estratégica. En este caso, se implementó en un proyecto nuevo.

2.2.4. Separación de responsabilidades comandos y consultas

En una arquitectura tradicional, el sistema realiza operaciones de negocio y permite realizar consultas sobre la información generada. Por lo tanto el sistema se encarga de hacer dos cosas distintas y con requisitos diferentes desde distintos puntos de vista: funcional, tiempo de respuesta, escalabilidad, criticidad, etc. Es por ello, que entra en juego un estilo arquitectónico modular, el cual plantea tener dos subsistemas diferentes, uno responsable de los comandos y otro responsable de las consultas. Cuando un usuario u otro sistema realiza una consulta, para realizar una operación de negocio que genera un cambio de un estado a otro, es considerado un comando. Este patrón, denominado separación de responsabilidades comandos y consultas (CQRS por sus siglas en inglés Command Query Responsibility Segregation) permite dividir el modelo conceptual en modelos separados para actualización y visualización, a los que identifica como Comando y Consulta

respectivamente siguiendo el vocabulario de separación de comandos y consultas (CQS por sus siglas en inglés Command Query Separation).

Este término, separación de comandos y consultas fue creado por Bertrand Meyer en su libro “Object Oriented Software Construction” en el cual propone dividir los métodos de un objetos en dos categorías claramente separadas: 1 Consultas: devuelven un resultado y no cambian el estado interno del objeto. 2 Comando: cambia el estado de un objeto sin devolver ningún resultado (Meyer; 1997). Esta categorización permite en dominios complejos facilitar su uso y por otro lado al tener la carga de lectura y escrituras separadas pueden ser escaladas de forma independiente para mejorar su rendimiento.

En particular, CQRS sólo debe utilizarse en partes específicas de un sistema y no en el sistema como un todo, por lo tanto, es aplicable en los bounded context de DDD, en el que cada contexto acotado necesita sus propias decisiones sobre cómo debe modelarse los datos. Por lo tanto, CQRS sigue el principio de CQS y el principio de responsabilidad única de Martin (2006). Esto implica tener una clase para insertar, actualizar o eliminar un registro y otra o varias para consultar un registro, dando lugar a un modelo como el de la figura. Por un lado, los comandos que escriben en la base de datos y por otro las consultas.

Esto implica que cada subsistema tiene un diseño, modelo y quizás mecanismo de persistencia diferente, optimizado para su fin. El subsistema de comando recibe la petición y sólo la ejecuta si es consistente con el estado actual del sistema. Como resultado, el estado del sistema cambia y es comunicado al subsistema de consultas mediante mecanismos de sincronización.

El uso de CQRS permite definir una arquitectura modular, donde cada módulo puede ser diseñado, mantenido, escalado y desplegado por separado. Añadiendo nuevas funcionalidades sin tener que parar el resto del sistema.

2.2.5. Lenguaje específico del dominio

En la ingeniería basada en modelos, un lenguaje específico de dominio (DSL por sus siglas en inglés Domain-Specific Language) es un lenguaje específico y orientado a problemas concretos, el cual describe con precisión un dominio de conocimiento (Langlois et. al.; 2007).

Un usuario de DSL se enfoca en describir el dominio, mientras que el diseño e implementación están ocultos dentro de una función que interpreta el dominio y lo

transforma en código fuente. Esta característica permite elevar el nivel de abstracción del software facilitando el desarrollo, productividad y calidad del mismo.

Un DSL está acoplado a un intérprete y generador de código, como es el caso de JHipster. Una plataforma de desarrollo para generar, desarrollar e implementar rápidamente aplicaciones web, la cual tiene su propio DSL denominado JDL (JHipster Domain Language). JDL es el lenguaje específico de dominio de JHipster que permite generar el código fuente con una sintaxis de alto nivel, dando como resultado aplicaciones, implementaciones, entidades y sus relaciones.

JHipster, al ser un generador de código y tener su propio DSL, permite generar aplicaciones con solo describir el dominio, facilitando entonces el desarrollo, productividad y calidad del software.

2.2.6. Arquitectura de software

Un aspecto crítico a la hora de construir sistemas complejos es el diseño de la estructura del sistema, y por ello el estudio de la Arquitectura Software se ha convertido en una disciplina de especial relevancia en la ingeniería del software (Shaw y Garlan; 1996).

La arquitectura de software es la representación de alto nivel de la estructura de un sistema, describe las partes que la integran, las interacciones entre ellas, los patrones que supervisan su composición y las restricciones que aplican esos patrones (Troya et. al.; 2017).

En general, la arquitectura de software se realiza en términos de componentes y de sus interacciones, la cual da lugar a arquitecturas basadas en componentes y conectores. Este tipo de arquitecturas son denominadas modulares, ya que favorecen a la reutilización de sus componentes, incluyendo sus conectores.

Una arquitectura de software forma una estructura, la cual es lograda por la unión de enlaces específicos entre sus componentes y conectores que la componen. Esta estructura permite identificar patrones genéricos que definen a una familia de sistema, dando lugar a diferentes estilos arquitectónicos, los cuales son de gran importancia para la reutilización.

La arquitectura de software conforma la columna vertebral de cualquier sistema y constituye uno de sus principales atributos de calidad (Bass et. al.; 2003). El documento de IEEE Std 1471-2000[12] define: “La arquitectura de software es la

organización fundamental de un sistema encarnada en sus componentes, las relaciones entre ellos y el ambiente y los principios que orientan su diseño y evolución".

2.2.7. Arquitectura limpia

En las últimas décadas surgieron diferentes ideas sobre las arquitecturas de los sistemas las cuales se engloban dentro de las arquitecturas limpias, como la Arquitectura Hexagonal (AH, también conocida como Puertos y Adaptadores), desarrollada por Cockburn (2008) y Datos, Contexto e Interacción, (DCI por sus siglas en inglés Data, Context and Interaction) de Coplien y Reenskaug (2012) entre otras.

Aunque estas arquitecturas varían un poco en sus detalles, son muy similares. Ambas tienen el mismo objetivo, la separación de las responsabilidades, dividiendo el software en capas. Roben Martin, expone en su libro "Clean Architecture" (2018), que cada una tiene al menos una capa para las reglas de negocio, y otra capa para las interfaces gráficas de usuarios y que este tipo de arquitecturas produce sistemas con las siguientes características:

- Independiente de los frameworks: La arquitectura no depende de la existencia de una librería o biblioteca de software. El modelo de negocio no está acoplado al código técnico, lo que permite flexibilidad a la hora de cambiar algunas implementaciones.
- Testeable: Las reglas de negocio pueden ser probadas sin la interfaz de usuario, bases de datos, servidor web, o cualquier otro elemento externo. Esta característica incrementa exponencialmente la capacidad de prueba del dominio.
- Independiente de la Interfaz de Usuario: Se puede cambiar fácilmente, sin cambiar el resto del sistema. Una interfaz de usuario web podría ser reemplazada por una consola, por ejemplo, sin cambiar las reglas del negocio.
- Independencia de cualquier agente externo: Las reglas de negocio no conocen en absoluto sobre las interfaces con el mundo exterior. Por lo tanto, basta implementar cualquier interfaz de usuario para comunicarse con el modelo, ya sea un servicio REST, un servidor web, una consola de comandos, etc.

La figura 1 representa la integración de ambas arquitecturas en una sola idea. Los círculos representan diferentes áreas del software, los externos son dispositivos, los internos son políticas (reglas de negocio, dominio).

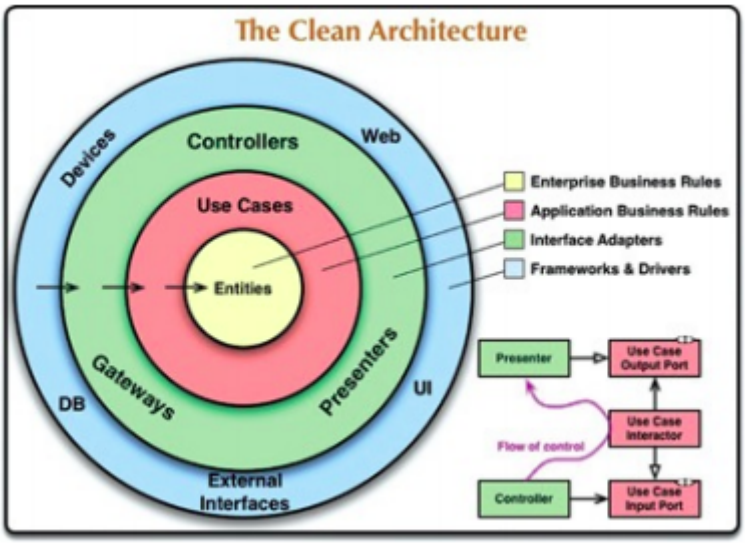

Fig. 1. Fuente: Libro “Clean Architecture" de Robert Martin, Cap. 22. (2018)

Su autor, la describe como un conjunto de reglas y patrones en 4 capas: entidades: casos de usos, adaptador de interfaz y frameworks y drivers, con el objetivo de que las dependencias de código sean hacia las capas internas. Para lograr la separación entre las capas, su autor propone una regla, llamada regla de dependencia.

II.2.8. Regla de dependencia

Es la regla primordial que hace que esta arquitectura funcione:
“Las dependencias del código fuente deben dirigirse solo hacia adentro, hacia las políticas de nivel superior” (Martin; 2018).

La capa interna no conoce sobre su capa externa. En particular, una declaración en la capa externa, no debe ser mencionada por el código de la capa interna, esto incluye, clases, variables, o cualquier otra entidad de software nombrada. Por la misma razón, los formatos de datos declarados en un círculo externo, especialmente si estos son generados por un framework, no deben ser utilizados por el interno. Esta regla, permite que cualquier cambio en el círculo externo no impacte en los internos (Martin; 2018).

Su autor, no define ninguna regla sobre la cantidad de círculos, ya que tienen la intención de ser simplificados y probablemente se necesiten más que cuatro. Sin embargo, la Regla de dependencia siempre se aplica (Martin; 2018).

En la capa de entidades, la más interna, se encapsula el modelo y la lógica de negocio para ser utilizadas por diferentes aplicaciones de la empresa.

Hacia el exterior se encuentra la capa coordinadora, denominada casos de usos, encargada de encapsular e implementar cada caso de usos del sistema. Es acá donde se coordina el caso de uso a ejecutar, utilizando las reglas de negocio para lograr el objetivo.

Un nivel más arriba, se encuentra la capa adaptador de interfaz, la cual convierte según su capa externa los datos al formato más conveniente para los casos de usos y entidades, ya sea una base de datos o la web.

En la última capa, se encuentran las interfaces externas, como por ejemplo la base de datos, Frameworks Web, interfaz de usuario, dispositivos, etc.

Es así, como esta arquitectura, define una serie de reglas para separar el software en capas y cumplir con la regla de dependencia, lo que implica aislar el modelo y la lógica de negocio del código externo. Por lo tanto, cuando cualquier capa externa del sistema quede obsoleta puede ser reemplazada sin que afecte a las capas internas.

II.2.9. Arquitectura hexagonal

Permite que una aplicación sea manejada igualmente por usuarios, programas, pruebas automatizadas o scripts por lotes, y que se desarrolle y pruebe aisladamente de sus eventuales dispositivos y bases de datos en tiempo de ejecución (Cockburn; 2008).

En el contexto de DDD, Vernon, introdujo la arquitectura hexagonal. Es un patrón estructural para diseñar software, que establece entradas y salidas en los bordes del diseño, lo que significa que es posible intercambiar los “manejadores” sin cambiar el código del núcleo. Al hacerlo, el núcleo de la aplicación está aislado de las partes externas.

A diferencia de la arquitectura típica en capas, es posible usar la inyección de dependencia y otras técnicas para posibilitar pruebas, aunque la diferencia está en el modelo hexagonal: La interfaz de usuario también se puede intercambiar, y esta fue una de las motivaciones principales de su creación.

Como puede verse en la figura 2, la arquitectura hexagonal consiste en el modelo de dominio, los servicios de aplicación y los puertos con sus adaptadores. Cada lado del hexágono representa un puerto concreto, aunque en la práctica podría haber más puertos con sus adaptadores correspondientes.
Basándose en el principio de inversión de dependencia, el modelo de dominio y la lógica de negocio como núcleo independiente de los servicios de aplicación y adaptadores que lo rodean, simplificando el traspaso o cambio de decisiones tecnológicas.

Es posible escribir un nuevo adaptador, en el caso que se requiera cambiar un framework o herramienta utilizada, aunque la lógica de negocio (el hexágono), debe estar libre de ellas. Solo el exterior del hexágono habla con el interior mediante interfaces, llamadas puertos. Lo mismo al revés.
Las capas de dependencia se aplican desde el exterior hacia el núcleo (Cockburn; 2008).

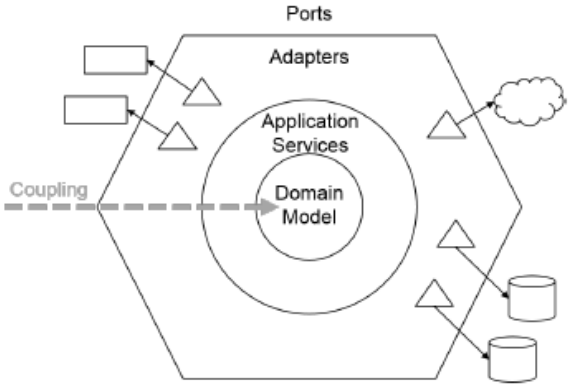

Fig. 2. Arquitectura Haxagonal de Cockburn

3. Arquitectura de Software y Entorno de Trabajo (Framework)

La solución se describe en términos de los conceptos explicados anteriormente. Para implementar la arquitectura hexagonal, existen distintas tecnologías y frameworks. Como primer paso, se definen los tres bloques fundamentales del sistema, como puede verse en la figura 3 (Aplicación, Dominio e Infraestructura).
La arquitectura hexagonal hace una separación explícita del código interno y externo al núcleo, y cuál se usa para conectar el código en ambos extremos.

Se identifican explícitamente tres capas fundamentales de código en el sistema:

- Aplicación: hace posible ejecutar una interfaz de usuario, sea cual sea.
- Dominio: la lógica de negocio o núcleo del sistema, que es utilizada por la interfaz de usuario para hacer que las cosas sucedan realmente.
- Infraestructura: el código que conecta el núcleo de aplicación con herramientas como bases de datos, motor de búsqueda o APIs de terceros.

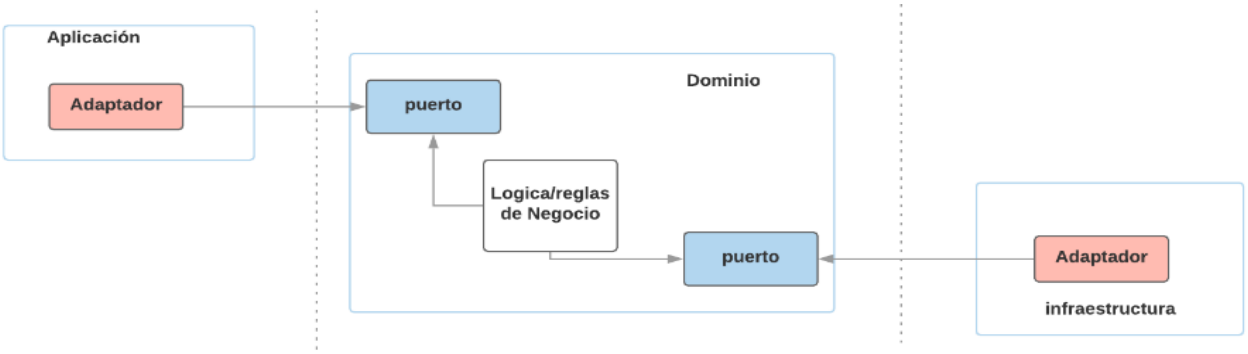

Fig. 3 Representación de la arquitectura hexagonal

La diferencia clave es que, mientras la capa de aplicación se utiliza para informarle a la capa de dominio que haga algo, la capa de infraestructura es informada por el sistema para hacer algo. Esta distinción es muy relevante, debido a las fuertes implicaciones en la forma en que se construye el código que se conecta con el núcleo.

La conexión de las herramientas con el núcleo o la capa de dominio está expresada por las unidades de códigos denominados adaptadores (Cockburn; 2005). Los adaptadores son los que implementan efectivamente el código que permitirá a la lógica de negocio comunicarse con una herramienta específica y viceversa.

Los adaptadores se crean para adecuarse a un punto de entrada muy específico del núcleo de la aplicación a través de un puerto. Un puerto es una especificación de

cómo la herramienta puede usar o ser usado por el núcleo de la aplicación. El puerto, es una Interface Java (Java Interface; 2021). Es importante destacar que los puertos permanecen dentro de la capa de dominio, mientras que los adaptadores se encuentran fuera del dominio.

La característica principal de la arquitectura hexagonal, es que las dependencias entre componentes apuntan “hacia adentro”, hacia los objetos de dominio.

El principio para la separación en capas es que cada una oculta la lógica de negocio al resto y solo brinda puntos de acceso a dicha lógica. En cuanto a los frameworks y herramientas utilizadas en cada una se explican a continuación.

En la capa de aplicación los objetos trabajan directamente con los puertos de la capa de dominio, se implementa el patrón arquitectónico Model-View-Controller son Spring MVC brindando servicios REST (Representational State Transfer - Transferencia de estado representacional).

La capa de dominio está formada por servicios implementados por los objetos de negocio. Estos delegan gran parte de su lógica en los modelos del dominio, implementando en dichos servicios las operaciones (casos de usos).

Finalmente, la capa de infraestructura facilita el acceso a los datos y su almacenamiento en una base de datos mediante la tecnología Spring Data JPA (Pollack; 2012) con soporte de Hibernate (Bauer; 2007) y las clases de configuración, a cargo de Spring (Walls; 2015) a través de su contenedor de inversión de control (IoC) de Beans y su paradigma de Inyección de dependencias.

Spring es un framework que aporta aún más funcionalidades a esta arquitectura además de su comportamiento como contendor, por lo que las posibilidades sobre esta arquitectura quedan abiertas para la integración de nuevos módulos como Spring AOP (Bass, *et.al.; 2003*), y el módulo de seguridad de autenticación y autorización como Spring Security (Scarioni; 2013). Además, Spring se encarga de administrar el ciclo de vida de los objetos, implementados mediante POJOs (Plain Old Java Object - Fowler; 2003) y representa objetos que son parametrizables por medio de archivos de configuración u anotaciones.

4. Caso de estudio

A continuación, se presenta el caso de estudio mediante un escenario, que se centra en la implementación de la plataforma digital del Hospital de Viedma.

El Hospital de Viedma es un organismo público que tiene a cargo diferentes centros de salud dentro de su estructura jerárquica. Actualmente se está impulsando un servicio mediante una plataforma digital que permite generar una red de perfiles del personal de salud del Hospital de Viedma y generar un canal de comunicación e interacción entre la comunidad hospitalaria. Su objetivo principal es gestionar los perfiles y crear un canal de comunicación directo entre los mismos.

Además, permite identificar la estructura jerárquica hospitalaria y el personal de salud asignado, dando lugar al canal de comunicación e interacción para informar sobre eventos, avisos y comunicación en general.

De esta manera se logra a través de una herramienta, generar una comunidad digital que permite relacionar a todo el personal de salud con sus respectivas dependencias o centros de salud.

4.1. Definición de requerimientos funcionales

A nivel funcional, la plataforma digital posee tres tipos de cuentas que permiten acceder a diferentes funcionalidades. Cada una de ellas representa un rol dentro del organismo. Este apartado es meramente informativo, permitiendo contextualizar el caso de estudio e identificar el dominio y su alcance para tenerlo en cuenta a la hora de definir la arquitectura de software.

La plataforma digital es una aplicación web que está destinada al personal del Hospital Viedma. Como se mencionó anteriormente, se puede acceder con diferentes tipos de cuentas.

1. La cuenta “Administrador” permite crear cuentas de gobierno, como por ejemplo la cuenta del Hospital Viedma. Supongamos que la plataforma administra todos los hospitales de Rio Negro, por lo tanto la cuenta “Adminsitrador” podra crear a cada hospital bajo el tipo de cuenta de gobierno.

2. La cuenta “Gobierno” permite crear la estructura jerárquica, administrar su personal y asignarlos a cada dependencia. Entonces, por ejemplo la cuenta

“Gobierno” del Hospital Viedma podrá gestionar sus dependencias o centros de salud, geolocalizarlos, gestionar su personal y asignar/desasignar el lugar de trabajo de cada uno de ellos.

3. La cuenta “Personal” permite gestionar el perfil, acceder a la red de contactos y al canal de comunicación entres los diferentes perfiles. Permite a todo el personal de salud comunicarse con él/los referentes de cada centro de salud o dependencia que requiera.

4.2. Análisis del dominio

Identificación del modelo de la plataforma digital del Hospital de Viedma (PDHV):

Antes de comenzar a escribir código se requiere tener una visión general de la PDHV. Aplicando conceptos de DDD, se creó un modelo abstracto en el ámbito del dominio. Esto permitió extraer y organizar el conocimiento del mismo, y proporcionar el lenguaje ubicuo para los desarrolladores y los expertos de negocio. Lenguaje utilizado a lo largo del proyecto, incluso utilizado en el código fuente.

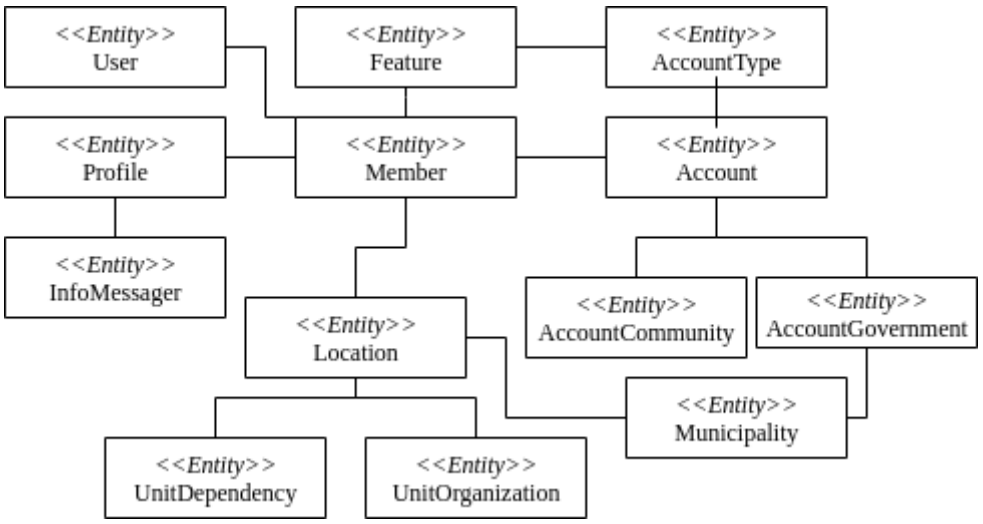

Fig 4. Representación del modelo de dominio

Como puede verse en el gráfico, la representación del modelo de dominio refleja el contexto de la plataforma digital y propone un panorama de las entidades más importantes y la relación entre las mismas.

De este gráfico se puede distinguir que la plataforma digital posee 5 posibles módulos principales: 1. Seguridad y Accesos; 2. Gestión de cuentas; 3. Gestión de perfiles para personal de salud; 4. Gestión de comunicación entre perfiles; 5.

Gestión de la estructura Jerárquica. los cuales serán evaluados en la etapa de definición de contextos delimitados.

4.3. Definición de contextos delimitados

El modelo de dominio, incluye representaciones del mundo real, lo cual no significa que todos los contextos delimitados deban usar las mismas representaciones.

De la representación del modelo del dominio, se distingue los posibles contextos delimitados, perfiles y mensajería, como puede verse en la imagen 5. Ambos contextos tendrán una representación de Perfil, la cual no necesariamente debe ser la misma. Como se mencionó anteriormente, los conceptos de dominio dependen del contexto en el que existe.

Por ejemplo, para el contexto de gestión de perfiles se tiene una representación de la entidad Perfil con muchas más características, como nombre, apellido, fecha de nacimiento, cuil, tipo y número de documento, teléfono, correo, etc. En cambio, para el contexto de gestión de la comunicación entre perfiles, solo se necesita saber si el perfil está activo o no, nombre, apellido y foto. Este aspecto permite que los modelos y en particular la entidad Perfil, evolucionen independientemente, mejorando el mantenimiento y pruebas centradas únicamente en su propio dominio.

Por lo tanto, el modelo y/o la entidad Perfil solo cambiará si cambian las reglas en su propio contexto.

Un contexto delimitado define el límite de un dominio dentro de los cuales existe y opera. De la figura 4, es posible agrupar funcionalidad teniendo en cuenta si estas compartirán un único modelo de dominio. En la figura 5 se muestra el diagrama de mapa de contexto de PDHV. Este permite representar e identificar los contextos delimitados, los mismos no se encuentran aislados entre sí, dado que el contexto delimitado “Estructura Jerárquica”, depende de los contextos “Perfil” y “Cuenta”.

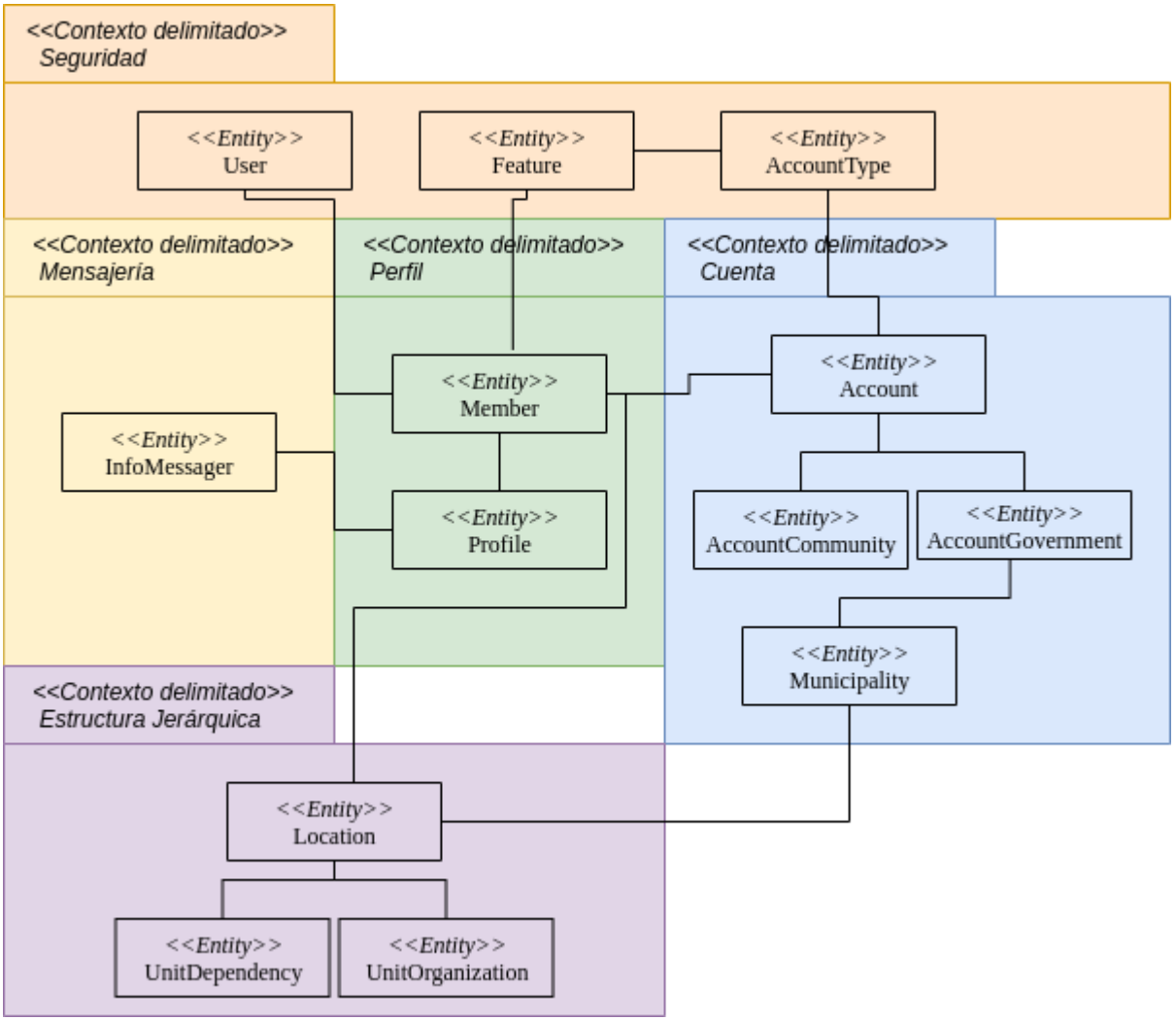

Fig. 5. Mapa de contextos

Durante esta fase estratégica de DDD, se definió los contextos delimitados. El diseño basado en dominios tácticos permitió definir los modelos de dominios con más precisión aportando una mejor mantenibilidad, tolerancia al cambio, testeabilidad y a los posibles cambios de conceptos.

4.4. Casos de usos

A continuación se listan los casos de usos más relevantes de la plataforma digital.

- Creación de cuenta de gobierno: El usuario administrador debe poder dar de alta cuentas de gobierno.
- Administración de localizaciones/Estructura Jerárquica: El usuario con cuenta de gobierno, puede gestionar localizaciones. Dar de alta, editar y eliminar. Las localidades pueden depender de otra localidad, dando lugar a generar la estructura jerárquica.

- Invitación a miembros de la cuenta (Personal de salud): El usuario con cuenta de gobierno, puede invitar/añadir miembros a su cuenta, cargando sus datos personales y asignando las localidades donde desempeña su trabajo.
- Gestión de perfiles.
- Comunicación por mensajería: Permite enviar dos tipos de mensaje, por notificación o email. Ambas opciones permiten seleccionar los destinatarios, enviar a todos, por localizaciones seleccionando las mismas o personalizado donde se pueden seleccionar los miembros de la cuenta. Luego se debe ingresar un título y descripción del mismo.
- Consultar perfiles registrados: Búsqueda de perfiles según diferentes criterios
- Gestión de actividades/Eventos.
- Gestión de Notificaciones.
- Administrar miembros de una localidad: El usuario con cuenta de gobierno puede asignar o desasignar puestos de trabajo a través de las localizaciones de cada miembro.

4.5. Definición del caso de uso “Administrar miembros de una localidad”

Presentamos a continuación, el caso de uso que se encuentra en el contexto delimitado Perfil.

La determinación a cuál contexto pertenece está dada por los modelos de dominio involucrados y sobre todo si conceptualmente es apropiado. Para este caso de uso corresponde que esté dentro del contexto Perfil donde son administrados los miembros.

Caso de uso: Administrar miembros de una localidad.
Actor principal: Usuario con cuenta “Gobierno”.
Propósito: Permite a un usuario con cuenta “Gobierno” de la plataforma administrar los miembros de cada localidad de la cuenta. De esta manera, la cuenta “Gobierno” podrá asignar o desasignar puestos de trabajo a través de las localizaciones de cada miembro.
Resumen: 1. Un usuario con cuenta “Gobierno”, comienza el proceso en ingresar al menú “Gestionar localidades”. 2. El sistema lista las localidades y el usuario selecciona una localidad, e ingresa a la opción “Administrar miembros”. 3. El sistema abre un sidenav (componente de contenido lateral). 4. De este caso de uso, se desprenden dos casos de usos **a. Asignar**

i. El sistema muestra un listado de miembros disponibles. Miembros que pertenezcan a la cuenta gobierno y que no estén asignados a la localidad en cuestión.
ii. El usuario por cada miembro, podrá agregarlo a la localidad.
iii. El sistema solicita una confirmación.
iv. En caso de confirmar, envía la solicitud.
v. El sistema valida que el miembro pertenezca a la cuenta y que no esté asignado a la localidad.
vi. En caso de éxito, el sistema informa con un mensaje, agrega el miembro al listado de asignados y lo quita del listado de disponibles.

b. Desasignar

i. El sistema muestra un listado de miembros asignados. Miembros que pertenezcan a la cuenta gobierno y que estén asignados a la localidad en cuestión.
ii. El usuario por cada miembro, podrá quitarlo de la localidad.
iii. El sistema solicita una confirmación.
iv. En caso de confirmar, envía la solicitud.
v. El sistema valida que el miembro pertenezca a la cuenta y que esté asignado a la localidad.
vi. En caso de éxito, el sistema informa con un mensaje, quita el miembro al listado de asignados y lo agrega del listado de disponibles.

A continuación, se muestran capturas de pantallas del funcionamiento de los casos de uso “Asignar” y “Desasignar”.
Al ingresar al menú “Gestionar localidades”, el sistema lista las mismas como puede verse en la imagen 6. Tal como se describe el caso de uso general, el segundo paso es seleccionar la opción “Administrar miembros” de una localidad.

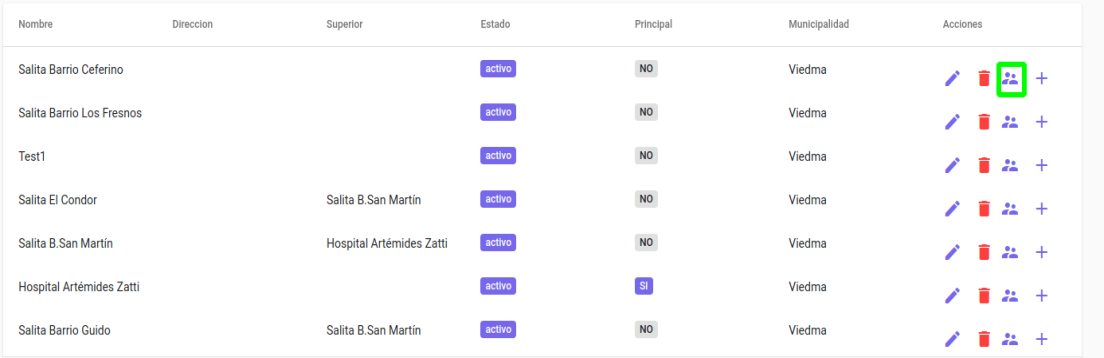

Imagen 6. Localidades de la cuenta.

El sistema abre un sidenav, como puede verse en la imagen 7 y 9 con dos listados. El listado de asignados que permite quitar esa asignación (desasignar) y el listado de disponible que permite asignarlos.

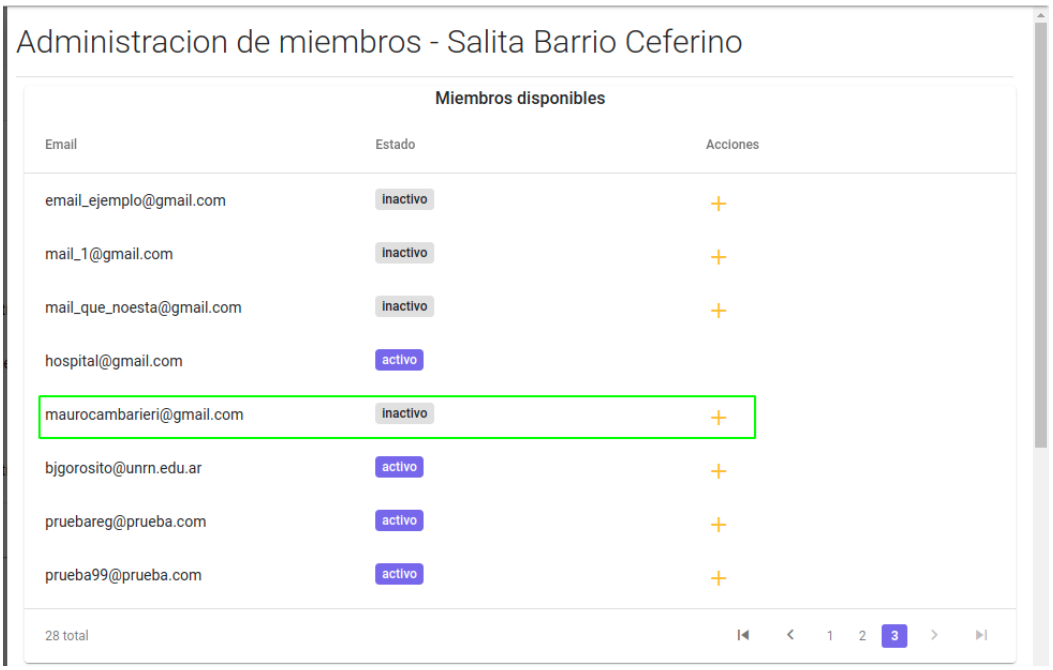

Imagen 7. Miembros disponibles.

Al hacer clic en el icono + sobre cualquier miembro del listado de disponibles, el sistema pide confirmación al usuario para realizar esa acción.

Imagen 8. Confirmación al asignar.

Confirmada la acción, el sistema envía al backend la solicitud.

En cambio, el listado de asignados permite desasignar a cada uno de los miembros.

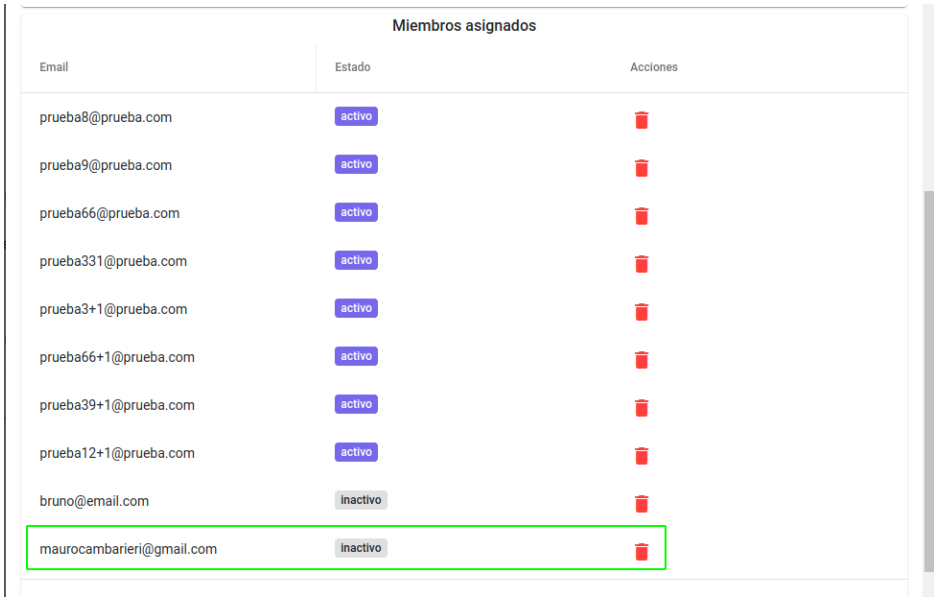

Imagen 9. Miembros asignados.

Al hacer clic en el icono de eliminar sobre cualquier miembro del listado de asignados, el sistema pide confirmación al usuario para realizar esa acción.

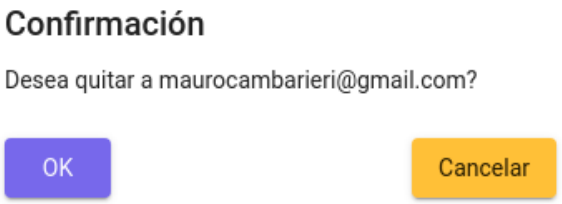

Imagen 10. Confirmación al desasignar.

Confirmada la acción, el sistema envía al backend la solicitud.

4.6. Construcción y diseño de la Arquitectura Hexagonal

En una arquitectura hexagonal se tienen entidades, casos de usos, puertos de entrada y salida y sus respectivos adaptadores como principales elementos. En esta etapa se definió una estructura de paquetes que refleja la arquitectura y que permite organizar la construcción y diseño de la misma.

Es por ello que una vez definidos los contextos delimitados, el siguiente paso es la creación de un módulo por cada contexto definido. Por ende, se crea un módulo con el patrón bc-nombrecontexto, el cual representará un contexto delimitado específico. En el caso de este trabajo se creó el módulo llamado “bc-profile” del contexto delimitado Perfil.

Cada módulo posee la siguiente estructura de paquetes:

- Dominio: se crea con el patrón bc-nombrecontexto-domain.
- Infraestructura: se crea con el patrón bc-nombrecontexto-persistence.
- Aplicación: se crea con el patrón bc-nombrecontexto-web.

Los paquetes hacen referencia a cada capa de la arquitectura hexagonal. En la imagen 11 podemos ver un ejemplo de la estructura del contexto Perfil (Profile en inglés).

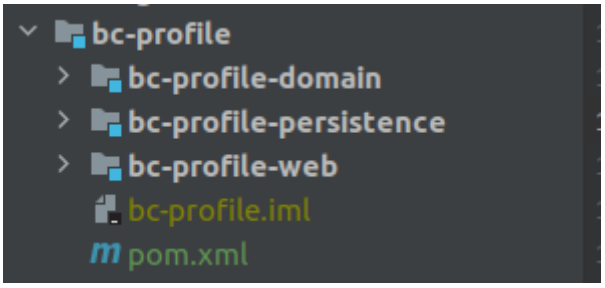

Imagen 11. Estructura del contexto Perfil (bc-profile).

Esta construcción y organización se llevó a cabo con la herramienta Maven, la cual permite la construcción de módulos y submódulos y el manejo de dependencias en proyectos Java.

El módulo Dominio en su interior también está organizado con una estructura de paquetes, la cual permite separar los principales elementos de la arquitectura.

- Command: se encuentran los modelos de entradas, uno por cada caso de uso específico. Podemos encontrar dos tipos de comandos: 1. Lectura o consulta; 2. Escritura. Al mirar las clases creadas dentro de este paquete se puede saber los casos de usos implementados en el contexto.
- Domain: se encuentran las clases, las cuales representan el modelo de dominio.
- Exception: se encuentran las implementaciones de cada una de las excepciones que agregamos en nuestro dominio, las cuales validan nuestros casos de usos.
- Handle: se encuentran las implementaciones de cada uno de los casos de usos, por lo tanto se corresponden con los comandos.
- Port: se encuentran las interfaces de salida.

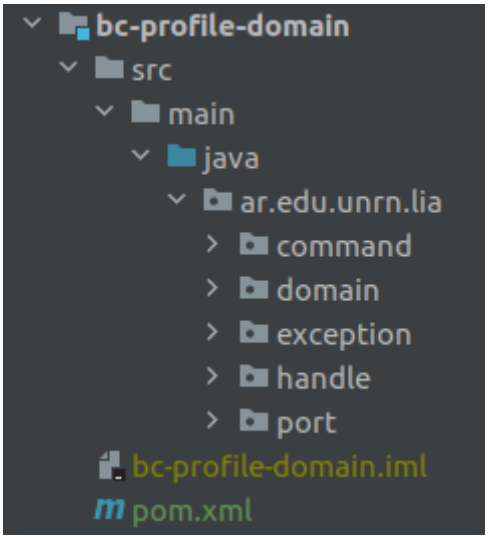

Imagen 12. Estructura del módulo dominio del contexto Perfil (bc-profile-domain).

En este módulo, el cual es el núcleo de esta arquitectura, se generan los comandos que son la puerta de entrada al núcleo. Los handles implementan los comandos y presentan toda la lógica necesaria para ejecutar el caso de uso.

Los programadores desarrollan cada caso de uso dentro de este módulo sin preocuparse por las interfaces de usuario, ni la persistencia de datos.

El módulo Aplicación es el medio de comunicación con el mundo exterior. Esta comunicación es posible mediante adaptadores de entrada. Estos aceptan peticiones del exterior y las traducen en llamadas a nuestro núcleo (bc-profile-domain), diciéndole que comando ejecutar.

El flujo de control va desde los controladores en el adaptador web, a los handle en el módulo Dominio, el cual expone los puertos, llamados comandos, necesarios para que el adaptador web pueda comunicarse. Los handles implementan estos comandos, y es acá donde se aplica el principio de inversión de dependencia, ya que los adaptadores web se comunican a través de los comandos con el módulo de dominio.

Un adaptador web comúnmente se encarga de:

- Mapear la solicitud HTTP a los objetos Java.
- Realiza verificaciones de autorización.
- Validar y mapear la entrada al modelo de caso de uso, es decir, la construcción del comando a ejecutar. Los comandos son validados por sí solo en su constructor.
- Llamar al caso de uso.
- Mapear el resultado del caso de uso a JSON.
- Respuesta HTTP.

En caso de que la verificación de autenticación y autorización, no sea exitosa el adaptador web devuelve un error.
Otra responsabilidad de una adaptador web es llamar al caso de uso con el modelo de entrada, es decir con el comando correspondiente. Luego mapea el resultado del caso de uso a JSON y construye una respuesta HTTP que se envía a la llamada. En caso de error, se lanza una excepción, y el adaptador web debe enviar ese mensaje a la llamada.

El módulo Infraestructura es el medio de comunicación con herramientas externas. Esta comunicación es posible al igual que en el módulo Aplicación, mediante adaptadores pero en este caso son de salida. Estos proporcionan funcionalidad de persistencia a los handle del módulo Dominio.

Los handle llaman a las interfaces (puertos) para acceder a la funcionalidad de persistencia. Estos puertos son implementados por adaptadores de persistencia que hacen el trabajo de persistencia real y son responsables de comunicarse con la base de datos o cualquier otra fuente.

Los puertos son una capa de indirección entre los handle y la persistencia. Esta capa de indirección permite evolucionar el código del dominio, sin tener que pensar en problemas de persistencia. Esta característica permite cambiar la capa de persistencia, sin necesidad de cambiar el código del módulo Dominio.

Un adaptador de persistencia comúnmente se encarga de:

- Tomar la entrada de datos.
- Mapear la entrada al formato de la base de datos, puede ser opcional.
- Enviar la entrada a la base de datos.
- Mapear la salida de la base de datos al modelo de domino.
- Retornar la salida.

El adaptador de persistencia toma la entrada a través del puerto (Interface). El modelo de entrada puede ser un dominio o un objeto dedicado a una operación específica de la base de datos, según lo especificado por la interface.

4.7. Implementación del caso de uso

A continuación, se mostrará el circuito del caso de uso en cuestión, para comprender el flujo de control y cómo interactúan las capas entre sí.

El circuito comienza en el controlador en el que se puede observar dos métodos, “assign” y “unassign” los cuales corresponden a “asignar” y “desasignar”, uno por caso de uso. Para que el ejemplo no sea tan extenso, se toma el caso de uso “asignar”.

El método asignar, recibe como parámetro un objeto “AssignMemberLocationCommand”, el cual representa la estructura de datos necesaria para ejecutar el caso de uso.

```java
@WebAdapter
@RestController
@RequiredArgsConstructor
class MemberLocationController {

    private final CommandBus commandBus;
    private final QueryBus queryBus;
    private final QueryFindAllBus queryFindAllBus;

    private final static String relativePath = "/api/member-location";

    @PostMapping(path = relativePath + "/assign")
    void assign(@RequestBody AssignMemberLocationCommand command) throws CommandHandlerExecutionError {
        commandBus.dispatch(command);
    }
    @PostMapping(path = relativePath + "/unassign")
    void unassign(@RequestBody UnassignMemberLocationCommand command) throws CommandHandlerExecutionError {
        commandBus.dispatch(command);
    }
```

Imagen 13. Controlador del módulo aplicación del contexto Perfil (bc-profile-web).

Este comando, es el responsable de validar los datos de entrada y ser el nexo entre la capa de aplicación y dominio.

```java
@Getter
public class AssignMemberLocationCommand implements Command {

    @NotNull
    private final Long memberId;
    @NotNull
    private final Long locationId;

    public AssignMemberLocationCommand(@NotNull Long memberId, @NotNull Long locationId) {
        this.memberId = memberId;
        this.locationId = locationId;
    }
}
```

Imagen 14. Comando del módulo dominio del contexto Perfil (bc-profile-domain).

Luego que los datos de entrada se validan con la construcción del comando, el controlador despacha el comando a través de una implementación de bus de comando, como se muestra en la imagen 13.

Esta implementación de comando, determina a que handle llamar en tiempo de ejecución, y es determinado por el comando despachado. Cada handle, implementa un comando y esa es la forma que determina su correspondencia.

El circuito continúa en el handle, quien se conecta a través de puertos a la base de datos y así realizar las validaciones necesarias para ejecutar el caso de uso.

En este caso de uso en particular, no devuelve resultado pero en caso de necesitarlo el handle retorna el objeto solicitado.

```java
@ServiceDomain
@RequiredArgsConstructor
@Feature(context = "bc-account")
class AssignMemberLocationHandle implements CommandHandler<AssignMemberLocationCommand> {

    private final MemberPort memberPort;
    private final LocationPort locationPort;
    private final MemberLocationPort memberLocationPort;


    @Override
    @Transactional
    public void handle(AssignMemberLocationCommand command) {
        Member member = memberPort.getFull(command.getMemberId());
        LocationAccount locationAccount = locationPort.findById(command.getLocationId());
        MemberLocation memberLocation =
                MemberLocation.withoutId(member, locationAccount);
        memberLocationPort.create(memberLocation);
    }
}
```

Imagen 15. Handle del módulo dominio del contexto Perfil (bc-profile-domain).

Cada uno de los puertos utilizados en el handle, son interfaces las cuales definen su comportamiento y luego deben ser implementadas en la capa de infraestructura por adaptadores. Se observa en la imagen 16 que el puerto “MemberLocationPort”, define un método llamado create, el cual recibe como parámetro un objeto MemberLocation.

```java
public interface MemberLocationPort {

    MemberLocation create(MemberLocation domain);
```

Imagen 16. Puerto del módulo dominio del contexto Perfil (bc-profile-domain).

Como se mencionó anteriormente, el puerto es implementado por un adaptador, el cual se observa en la imagen 17 que utiliza una clase de mapeo para convertir el objeto del dominio a un objeto de persistencia.

```java
@PersistenceAdapter
class MemberLocationPersistenceAdapter extends GenericCrudAndQueryPersistenceAdapter
        <MemberLocationJpaEntity, MemberLocation, Long, MemberLocationRepository>
        implements MemberLocationPort {

    public MemberLocationPersistenceAdapter(MemberLocationRepository repository,
                                            MemberLocationMapper mapper) {
        super(repository, mapper);
    }
    @Override
    public MemberLocation create(MemberLocation domain) {
        MemberLocationJpaEntity entity = getMapper().mapToJpaEntity(domain);
        getRepository().save(entity);
        return getMapper().mapToDomainEntity(entity);
    }
```

Imagen 17. Adaptador del módulo infraestructura del contexto Perfil (bc-profile-persistence).

La clase de mapeo, convierte un objeto del modelo de dominio a un objeto JPA y viceversa.

```java
@Component
@AllArgsConstructor
class MemberLocationMapper implements GenericMapper<MemberLocationJpaEntity,
        MemberLocation> {

    private final MemberMapper memberMapper;
    private final LocationMapper locationMapper;


    public MemberLocation mapToDomainEntity(
            MemberLocationJpaEntity entity) {...}

    public MemberLocation mapToFullDomainEntity(
            MemberLocationJpaEntity entity) {...}


    public MemberLocationJpaEntity mapToJpaEntity(MemberLocation domain) {...}

}
```

Imagen 18. Clase de mapeo del módulo infraestructura del contexto Perfil (bc-profile-persistence).

Una vez construido el objeto de persistencia, el adaptador delega la responsabilidad a JPA Repository, quien termina persistiendo los datos en la base de datos.

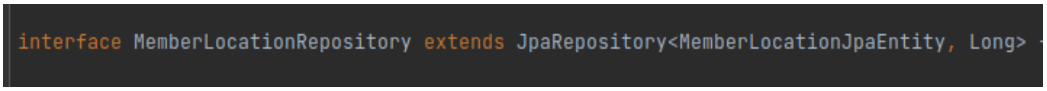

Imagen 19. Repositorio JPA del módulo infraestructura del contexto Perfil (bc-profile-persistence).

En la imagen 19 se observa que para utilizar el API de persistencia de Java es necesario extender de JpaRepository, clase que provee Spring Data JPA. Además esta clase nos obliga a enviar dos parámetros, una entidad JPA y un identificador.

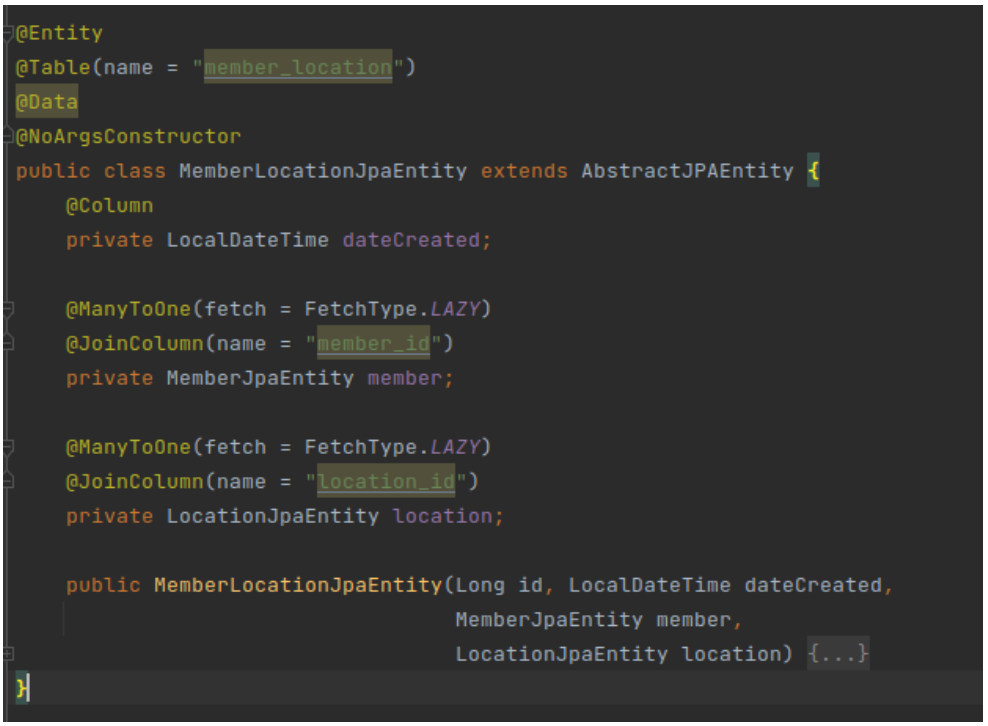

Imagen 20. Entidad JPA del módulo infraestructura del contexto Perfil (bc-profile-persistence).

Es por ello, que mapeamos el objeto MemberLocationJpaEntity con la anotación @Entity, la cual le indica a Spring que es un objeto persistente.

Ahora que se comprende el flujo de control y la interacción entre las capas, se muestran los pasos que se recomiendan para crear cada caso de uso.

El primer paso es comenzar por el módulo de dominio. Se define el comando con los datos necesarios para ejecutar el caso de uso. Luego se implementa el comando a través del handle donde se desarrolla toda la lógica de negocio. En caso de necesitar datos de un externo o persistencia, se crearán puertos.

Esta forma de construir software permite a los programadores desarrollar cada caso de uso dentro de este módulo sin preocuparse por las interfaces de usuario, ni la persistencia de datos.

El programador podrá enfocarse en realizar test unitarios de los casos de usos dentro del módulo de dominio, sin necesidad de instalar y configurar agentes externos como bases de datos.

Una vez testeados los casos de usos, se continúa con la implementación de los adaptadores, tanto de la capa de aplicación como de infraestructura. Este es el momento de realizar un test de integración entre las capas de la arquitectura.

5. Herramientas y Entornos

En la industria del software hay disponible una gran variedad de herramientas para implementar un proyecto web. Dada esta situación, se debió seleccionar la metodología de trabajo, el lenguaje de programación a utilizar, el tipo de base de datos para el servidor, editores de código, bibliotecas externas y el ambiente de desarrollo.

5.1. Sistema de control de versiones de código fuente

GitLab es un servicio web de control de versiones y desarrollo de software colaborativo basado en Git. Además de gestor de repositorios, el servicio ofrece también alojamiento de wikis y un sistema de seguimiento de errores, todo ello publicado bajo una Licencia de código abierto. Contiene todo lo necesario para llevar a cabo el proyecto bajo la metodología ágil, desde la creación de la idea a la producción de la misma.

5.2. Gestión de proyecto

OpenProject es una herramienta web de gestión de proyectos pensada para metodologías ágiles. Permite controlar y organizar fácilmente todas las actividades del proyecto, tareas, errores, así como asignar responsabilidades, realizar un seguimiento de las fechas de vencimiento, ver historial de cambios, etc. Todo ello publicado bajo una Licencia de código abierto.

5.3. Documentación técnica del proyecto

A la hora de crear un proyecto de software, es clave para el éxito una buena documentación que permita a los demás desarrolladores comprender acerca del mismo. Gitlab, permite documentar un proyecto a través de archivos de textos llamados README, los cuales son una manera ágil y sencilla de que otros usuarios puedan comprender de qué se trata el proyecto. El mismo deberá contener solamente la información necesaria para que los desarrolladores comiencen a utilizar y contribuir sobre el proyecto, ejemplo, modo de instalación, modo de uso y forma de colaboración.

5.4. Entornos

Para el desarrollo de este proyecto, fue necesario separarlo en dos proyectos de software. Proyecto Backend y proyecto Frontend.

Backend es el código que se escribe del lado del servidor, el cual contiene la arquitectura propuesta y que se conecta desde la capa de infraestructura como en nuestro caso con la base de datos MySQL. Además, la capa de aplicación expone servicio REST para que el proyecto frontend (proyecto web) pueda interactuar e intercambiar datos. Para la gestión de todos los componentes del proyecto, módulos y manejo de dependencias se utilizó Maven. El lenguaje utilizado es JAVA.

Frontend es la parte de un sitio web que interactúa con los usuarios, por eso decimos que está del lado del cliente. Para este caso, se desarrolló con el framework Angular, comúnmente llamado angular 2. Es un framework para aplicaciones web desarrollado en el lenguaje TypeScript, de código abierto, mantenido por Google, que se utiliza para crear y mantener aplicaciones web de una sola página.

Para ambos proyectos se utilizó el entorno de desarrollo integrado (IDE) IntelliJ IDEA, desarrollado por JetBrains.

6. Verificacion y Validacion

Se entiende por verificación de software el proceso que se realiza con el objetivo de asegurar que el software satisface por completo todos los requisitos esperados.

A lo largo de este trabajo, se validó un caso de uso y la arquitectura propuesta.

En primer lugar, para determinar el correcto funcionamiento de la plataforma digital, fue necesario verificar cada uno de los casos de usos. El propósito general de la evaluación es encontrar errores y defectos con el fin de corregirlos. Es importante comprobar que el sistema cumpla con los requerimientos establecidos por el usuario y tenga un rendimiento adecuado en el ambiente donde se encuentre.

Para el caso de la arquitectura propuesta se verificó el correcto funcionamiento del flujo de control, es decir, la comunicación entre componentes. Además, que cada componente cumpla con su responsabilidad.

Ambas pruebas, se llevaron a cabo en conjunto en el apartado 4.5. Los resultados fueron los esperados según el conocimiento empírico.

Por otro lado, se validó este trabajo, entiéndase como un proceso de evaluación del software, mediante un proyecto de investigación presentado y expuesto en las Jornadas Argentinas de informática (JAIIO) 2020. El mismo fue titulado “Implementación de una Arquitectura de Software guiada por el Dominio”. Este propone la transformación y adaptación de una arquitectura de software de tres capas típicas a una arquitectura centrada en el dominio específico del negocio y la selección de tecnologías que permiten su implementación. La contribución del mismo, permitió la construcción de este Trabajo Final de Grado.

También se validó mediante el caso de estudio de la plataforma digital del Hospital de Viedma. La misma se encuentra en entornos de testing y en etapa de pruebas por parte de directivos y personal del Hospital de Viedma.

7. Conclusiones y Recomendaciones

Con el desarrollo de la plataforma digital en el Hospital de Viedma, se concluye la viabilidad de la arquitectura propuesta. Ya que esta permite el desarrollo de software basado en el dominio específico, y también permite resolver y llevar a cabo una solución de un dominio complejo.

En cuanto a la construcción y diseño de la arquitectura, fue un desafío cambiar el enfoque y el paradigma arquitectónico para resolver un problema. El cambio no fue solo a nivel de código y de componentes arquitectónicos, sino que obligó a cambiar la forma de pensar e invertir los flujos de comunicación entre los componentes de la arquitectura.

Más allá de las dificultades, esta solución mejora notablemente la forma en que se construye software en varios aspectos, como por ejemplo la calidad, testeabilidad e independencia de cualquier agente externo, ya sea framework, interfaz de usuario o base de datos.

Es cierto que con la arquitectura planteada se escriben más código, clases e interfaces para poder cumplir con la regla de la dependencia. Aún así, esta arquitectura redujo drásticamente errores de código, al tener las responsabilidades definidas y separadas.

Por otro lado, a la hora de realizar cambios se identifica con mayor claridad cuáles son las clases o componentes que deben ser modificados. Estos cambios permiten validar que la modificación de un caso de uso impacte solamente en el mismo. De esta forma genera independencia respecto a los demás casos de uso del sistema, permitiendo a su vez que los contextos evolucionen.

Por último se concluye con este Trabajo Final de Grado, que el desarrollo de software guiado por el dominio es una herramienta que permite tener el modelo de negocio en sincronía con la implementación de la solución.

Finalmente, se recomienda en un futuro seguir esta línea de investigación con el fin de agregar mejoras a la solución propuesta. Con esto, se podrá ampliar el número de beneficios de esta implementación.

8. Bibliografía

Vivas, H. L., Cambarieri, M., García Martínez, N., Muñoz Abbate, H., & Petroff, M. (2013). Un Marco de Trabajo para la Integración de Arquitecturas de Software con Metodologías Ágiles de Desarrollo.

Penchikala, S. (12 de junio de 2008). Domain Driven Design and Development In Practice. InfoQ, disponible en https://www.infoq.com/articles/ddd-in-practice/ [accedido: 30/06/2021].

Evans, E. (2015). Domain Driven Design, Definitions and Pattern Summaries.

Nair, V. (2019). Domain Driven Design. In Practical Domain-Driven Design in Enterprise Java (pp. 1-15). Apress, Berkeley, CA.

Fowler, M. (14 de enero de 2014). Bounded Context. https://martinfowler.com/bliki/BoundedContext.html [accedido: 30/06/2021].

Fowler, M. (8 de diciembre de 2003). Plain Old Java Object (POJO). http://www.martinfowler.com/bliki/POJO.html [accedido 25/05/2021]

Java Interface (2021). Disponible en: https://docs.oracle.com/javase/tutorial/java/IandI/interfaceDef.html [accedido 22/06/2021]

Pollack, M., Gierke, O., Risberg, T., Brisbin, J., & Hunger, M. (2012). Spring Data: modern data access for enterprise Java. " O'Reilly Media, Inc.".

Bauer, C. (2007). G, King, Java Persistence with Hibernate.

Scarioni, C. (2013). Pro Spring Security. Apress.

Walls, C. (2015). Spring Boot in action. Simon and Schuster.

Cockburn, A. (2005). The pattern: Ports and adapters (‘’object structural”).

Evans, E. (2004). Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional.

Le, D. M., Dang, D. H., & Nguyen, V. H. (2016, November). Domain-driven design patterns: A metadata-based approach. In 2016 IEEE RIVF International Conference on Computing & Communication Technologies, Research, Innovation, and Vision for the Future (RIVF) (pp. 247-252). IEEE.

Troya, J. M., Vallecillo, A., & Fuentes, L. (2017). Desarrollo de software basado en componentes.

Bass, L., Clements, P., & Kazman, R. (2003). Software architecture in practice. Addison-Wesley Professional.

Fowler, M. “Patterns of Enterprise Application Architecture”, Addison-Wesley, (2002).

Coplien, J. O., & Reenskaug, T. M. H. (2012, October). The data, context and interaction paradigm. In Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity (pp. 227-228).

Martin, R. C. (2018). Clean architecture: a craftsman's guide to software structure and design. Prentice Hall.

Cockburn, A. Hexagonal Architecture: Ports and Adapters (“Object Structural”). June 19, 2008.

Johnson, R., Hoeller, J., Arendsen, A., & Thomas, R. (2009). Professional Java development with the Spring framework. John Wiley & Sons.

Meyer, B. (1997). Object-oriented software construction (Vol. 2, pp. 331-410). Englewood Cliffs: Prentice hall.

Langlois, B., Jitia, C. E., & Jouenne, E. (2007, October). DSL classification. In OOPSLA 7th Workshop on Domain specific modeling.