Implementación de una Arquitectura de Software guiada por el Dominio

Mauro Germán Cambarieri[1], Federico Difabio[1], Nicolás García Martínez[1]

[1] Laboratorio de Informática Aplicada, Universidad Nacional de Río Negro
Viedma, Río Negro
{mcambarieri,fdifabio,ngarciam}@unrn.edu.ar

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 diseño dirigido por el dominio ofrece la posibilidad de contar con principios, patrones y actividades para construir un modelo de dominio, que es el artefacto principal [25]. Además, ayuda a garantizar que la arquitectura de software permanezca centrada en las funcionalidades del negocio. En este trabajo proponemos 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, como valor estratégico, el cual proporciona mapear la idea del dominio del negocio para el desarrollo de los artefactos del software. El trabajo propuesto presenta 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, se valida mediante un caso de estudio.

1 Introducción

La Ingeniería de Software ofrece métodos, técnicas y herramientas dirigidas a mejorar el proceso de desarrollo de software, realizarlo con éxito y de calidad depende de varios factores como, las 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 [1]. 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 [2], 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. Es por ello, que es evidente en el corto o mediano plazo, la necesidad de un cambio en la forma de escribir el software. La modernización puede ser tanto estratégica como técnica y se trata principalmente de una reingeniería para aprovechar los beneficios de las arquitecturas y plataformas modernas. En muchos casos, esto significa extraer el conocimiento del negocio de los sistemas y re-

implementarlos, cuando los sistemas heredados están tan lejos de la tecnología, una reescritura se convierte en la única opción viable. Sin embargo, hay herramientas, técnicas y metodologías que pueden ayudar en este proceso y se fundamentan en los conceptos de Ingeniería de Software. En los últimos años, la industria del software se ha enfrentado a nuevos desafíos en cuanto a complejidad, costos, tiempo de comercialización, estándares de calidad y evolución. Para enfrentar estos desafíos tanto en la academia como en la industria, se encuentran 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 completo 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 [3]. 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 [4], 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. Existen varios factores que trazan los límites entre los contextos. La cultura humana, es una de los factores dominantes, dado que se necesita un modelo diferente cuando cambia el lenguaje [5].

Por su parte las arquitecturas limpias permiten separar las responsabilidades mediante regiones o capas, de esta forma se consigue desacoplar las mismas para que evolucionen de manera aislada, cuyo núcleo central es el dominio.

Una arquitectura comúnmente usada es la definida en capas. En el caso de una aplicación empresarial puede dividirse en tres capas lógicas bien definidas [13]: 1) capa de presentación, 2) capa de negocio y 3) capa de persistencia. Este trabajo explora como adaptar un proyecto desarrollado bajo una arquitectura de software típica, implementando el enfoque DDD y la arquitectura hexagonal (arquitectura limpia) [6].

La contribución de este trabajo es mostrar la viabilidad sobre la transformación y adaptación de una arquitectura de software de tres capas típicas a una arquitectura hexagonal para el desarrollo de los artefactos del 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 valida mediante un caso de estudio y está estructurado de la siguiente manera: La Sección 2 presenta los conceptos relacionados, incluyendo Diseño dirigido por el dominio, arquitectura de software, arquitectura limpia y arquitectura hexagonal. La Sección 3 explica el enfoque propuesto y las tecnologías seleccionadas. A continuación, la Sección 4 valida la propuesta a través de un caso de estudio y muestra cómo se adopta DDD y la arquitectura hexagonal. La Sección 5 presenta otros aportes científicos en esta línea de investigación y discute la contribución de este trabajo. Por último, la Sección 6 brinda conclusiones y explica los trabajos futuros.

2 Conceptos Utilizados

Las siguientes secciones presentan los conceptos utilizados en este trabajo: Diseño dirigido por el dominio (Sección 2.1), Arquitectura de software (Sección 2.2), Arquitectura Limpia (Sección 2.3) y Arquitectura hexagonal (Sección 2.4).

2.1 Diseño dirigido por el dominio

El diseño dirigido por el dominio permite desarrollar proyectos de software cuya complejidad está dada por dominios complejos [7]. Su autor, Eric Evans, expone un extenso conjunto de prácticas, técnicas y principios de diseño.

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 un diseño exitoso dado por la complejidad del dominio debe tratar sistemáticamente este aspecto central en el desarrollo de software. Para ello debe tenerse en cuenta: 1) Para la mayoría de los proyectos de software, el enfoque principal debería ser el dominio y la lógica del mismo; 2) Los diseños de dominios complejos deberían basarse en un modelo.

Su objetivo es crear un entorno de colaboración entre los expertos en dominios (personas que conocen el negocio) y los técnicos(desarrolladores). Esto permite descubrir dominios, perfeccionando iterativamente [8] un modelo diseñado para abordar un problema empresarial que se está tratando de resolver.

Este proceso de hallazgo de dominios es la base estratégica de DDD en el cual entran en juego dos conceptos muy importantes, el lenguaje ubicuo y los contextos delimitados (Bounded Context). El lenguaje ubicuo es una colección de términos específicos del dominio, es utilizado por los expertos del dominio del negocio. Este lenguaje emplea todas las formas de comunicación entre el equipo de desarrollo y esencialmente los mismos términos se utilizan 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 son 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 de 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 se hace mucho 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 [9]. 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 [7]. Este diseño táctico consiste en definir los modelos de dominio con más precisión. Los patrones tácticos se aplican dentro de un contexto delimitado. Interesan destacar especialmente los patrones de agregados, entidades y servicios de dominio. Aplicar estos patrones ayuda a identificar los límites de los servicios en nuestra aplicación.

2.2 Arquitectura de Software

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 de aplicar esos patrones [10].

La arquitectura de software conforma la columna vertebral de cualquier sistema y constituye uno de sus principales atributos de calidad [11]. 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".

En particular, una arquitectura típica comúnmente usada es la definida en capas. En el caso de una aplicación empresarial puede dividirse en tres capas lógicas bien definidas [13]: 1) Presentación, 2) Negocio y 3) Persistencia. El principio para la separación en capas es que cada una esconde su lógica al resto y solo brinda puntos de acceso a dicha lógica. En la capa de presentación los objetos trabajan directamente con las interfaces de negocios, implementando el patrón arquitectónico Model-View-Controller [13]. En este, el modelo (Model) es modificable por las funciones de negocio, siendo estas solicitadas por el usuario, mediante el uso de un conjunto de vistas (View) que solicitan dichas funciones de negocio a través de un controlador (Controller), que es quien recibe las peticiones de las vistas y las procesa.

La capa de negocio está formada por servicios implementados por objetos de negocio. Estos delegan gran parte de su lógica en los modelos del dominio que se intercambian entre todas las capas. Finalmente, la capa de persistencia facilita el acceso a los datos y su almacenamiento en una base de datos.

2.3 Arquitectura Limpia (Clean Architecture)

En las últimas décadas surgieron diferentes ideas sobre las arquitecturas de los sistemas, como la Arquitectura Hexagonal (AH, también conocida como Puertos y Adaptadores), desarrollada por Alistair Cockburn [6] y Datos, Contexto e Interacción, (DCI por sus siglas en ingles Data, Context and Interaction) de James Coplien y Trygve Reenskaug [14] entre otras.

Aunque estas arquitecturas varían un poco en sus detalles, son muy similares. Todas tienen el mismo objetivo, que es la separación de las responsabilidades, dividiendo el software en capas. Cada una tiene al menos una capa para las reglas de negocio, y otra capa para las interfaces graficas de usuario [15].

Este tipo de arquitecturas produce sistemas que tienen las siguientes características:

- *Independiente de los frameworks*. La arquitectura no depende de la existencia de una librería o biblioteca de software.
- *Testeable*. Las reglas de negocio pueden ser probadas sin la interfaz de usuario, base de datos, servidor web, o cualquier otro elemento externo.
- *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.

- *Independiente de la base de datos*. Dado que las reglas de negocio no están ligadas a la base de datos, es posible cambiar el motor de base de datos.
- *Independiente de cualquier agente externo*. Las reglas de negocio no conocen en absoluto sobre las interfaces con el mundo exterior.

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

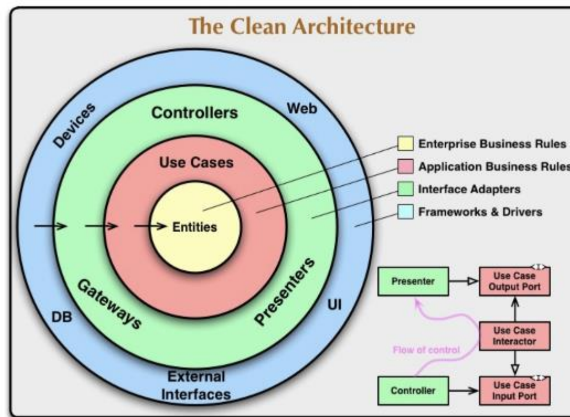

Fig.1 Clean Architecture [15].

2.3.1 Regla De Dependencia.

Es la regla primordial que hace que esta arquitectura funcione:

"Las dependencias del código fuente deben dirigirse sólo hacia adentro, hacia las políticas de nivel superior".

El círculo interno no conoce en absoluto algo del círculo externo. En particular, una declaración en el círculo exterior no debe ser mencionado por el código en el círculo interno, esto incluye, clases, variables, o cualquier otra entidad de software nombrada. Por la misma razón, los formatos de datos declarados en un círculo exterior, especialmente si estos son generados por un framework, no deben ser utilizados por el círculo interior. Nada en el círculo exterior debe impactar en los círculos interiores.

Los círculos tienen la intención de ser simplificados, es probable que se necesiten más que cuatro y no hay ninguna regla que lo impida. Sin embargo, la Regla de Dependencia siempre se aplica. El código fuente de las dependencias siempre apuntan hacia adentro, el software se vuelve más abstracto y encapsula detalles de alto nivel. El círculo más interno es el más general y de mayor nivel. El círculo exterior consiste en detalles concretos de bajo nivel. [15].

2.4 Arquitectura Hexagonal

En el contexto de DDD, Vernon [16] introdujo la arquitectura hexagonal [6]. Es un patrón estructural para diseñar software. La idea detrás de esta es establecer 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. La intención original de la arquitectura se define a continuación: “Permita 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”.

La arquitectura hexagonal fue un cambio con respecto a la arquitectura típica en capas, donde es posible usar la inyección de dependencia (DI por sus siglas en inglés, Dependency Injection) [17] y otras técnicas para posibilitar pruebas sobre esta. Pero hay una diferencia clave con el modelo hexagonal: la interfaz de usuario también se puede intercambiar, y esta fue una de las motivaciones principales de su creación [18].

La arquitectura resultante aplicando este patrón, se puede ver en la figura 2. Según Cockburn, la arquitectura hexagonal consiste en el modelo de dominio, los servicios de aplicación y los puertos con los adaptadores. Cada lado del hexágono representa un puerto concreto, aunque en la práctica podría haber más puertos distintos junto con su correspondiente adaptador. Basándose en el principio de inversión de dependencia, el modelo de dominio como núcleo y por lo tanto la lógica de negocio es independiente de los servicios de aplicación y adaptadores que lo rodean, esto simplifica el traspaso o cambio de decisiones tecnológicas. Es posible escribir un nuevo adaptador, en el caso que se quiera cambiar un framework o herramienta utilizada. Alrededor de la lógica de negocio (el hexágono) debe estar libre de cuestiones de tecnología, solo el exterior del hexágono habla con el interior mediante interfaces, llamadas puertos. Lo mismo al revés. Al cambiar la implementación(adaptador) de un puerto, se cambia la tecnología. Las capas de dependencias se aplican desde el exterior hacia el núcleo.

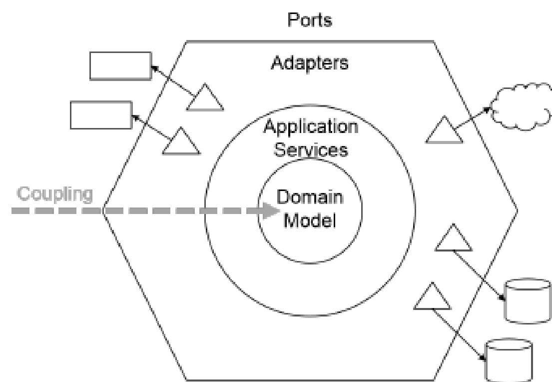

Fig. 2. Arquitectura Hexagonal de A. Cockburn [6]

3 Arquitectura de Software y Entornos de Trabajo (Framework)

La solución se describe en términos de los conceptos explicados anteriormente, en esta sección presentamos las distintas tecnologías y frameworks disponibles para la implementación de la arquitectura hexagonal, como primer paso, se definen los bloques fundamentales (Aplicación, Dominio e Infraestructura) del sistema como muestra la figura 3. La arquitectura hexagonal hace una separación explícita de qué código es interno y externo al núcleo, y qué se usa para conectar el código en ambos extremos.

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

1. Lo que hace posible ejecutar una interfaz de usuario (sea cual sea);
2. La lógica de negocio o núcleo del sistema, que es utilizada por la interfaz de usuario para hacer que las cosas sucedan realmente;
3. El código de infraestructura, que conecta el núcleo de aplicación con herramientas como base de datos, motor de búsqueda o APIs de terceros.

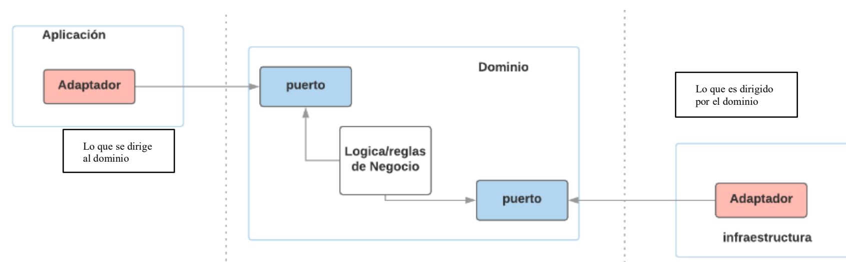

Fig.3 Representación de la arquitectura hexagonal.

La diferencia clave es que, mientras que la capa de aplicación se utiliza para decirle a la capa de dominio que haga algo, la capa de infraestructura es informada por el sistema para hacer algo. Esta es una distinción muy relevante, ya que tiene 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, capa de dominio, está expresado por las unidades de código denominados adaptadores [6]. 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 [19]. Es importante destacar que los puertos permanecen dentro de la capa de dominio, mientras que los adaptadores se encuentran afuera.

La característica principal de la arquitectura hexagonal, a diferencia del estilo de arquitectura en capas típicas, es que las dependencias entre los componentes apuntan "hacia adentro", hacia los objetos de dominio.

El principio para la separación en capas es que cada una esconde su lógica 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 con Spring MVC brindando servicios REST (Representational State Transfer). La capa de dominio está formada por servicios implementados por 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 [20] con el soporte de Hibernate [21] y las clases de configuración, a cargo de Spring [23] 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 contenedor, por lo que las posibilidades sobre esta arquitectura quedan abiertas para la integración de nuevos módulos como Spring AOP [11], y el módulo de seguridad de autenticación y autorización como Spring Security [22]. Además, Spring se encarga de administrar el ciclo de vida de los objetos, implementados mediante POJOs (Plain Old Java Object - sigla creada por Martin Fowler, Rebecca Parsons y Josh MacKenzie [24]) 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 (caso de uso), que se centra en la implementación de una plataforma de empleo.

Ilustración del escenario:

La oficina de empleo(OE) del gobierno de Viedma es un organismo que tiene como objetivo principal ser el nexo entre la oferta y la demanda de empleo en el mercado laboral de la ciudad, concretando políticas activas y aprovechando recursos tecnológicos. Llevan adelante una política abierta profundizando el contacto de las empresas y organismos públicos con las personas que aspiran a mejorar su empleabilidad. Actualmente se está impulsando un servicio a los ciudadanos mediante una plataforma digital de empleo que permite que se registren para ofrecer sus oficios/profesiones, exponer sus habilidades y contar con referencias, por otro lado, que las empresas y entidades público-privada puedan registrar la demanda de trabajos que necesitan. De esta manera se logra a través de una herramienta colaborativa, contar con un mayor acceso a la información de calidad, contribuyendo a la mejora en la calidad del servicio y a generar confianza entre los actores que intervienen en el proceso.

Para este escenario, se plantea resolver y mostrar el caso de uso “Contratación de servicios” el cual se define de la siguiente manera: Cuando un usuario demandante realiza una contratación de un servicio ofrecido, el sistema registra dicha contratación y notifica al usuario oferente sobre un nuevo contacto. Una vez que la contratación finaliza, el usuario demandante puede calificar al usuario oferente. Esto permite que el perfil del usuario oferente pueda tener una calificación general de sus trabajos.

4.1 Análisis del dominio: Identificación del modelo de la Plataforma Digital de Empleo (PDE)

Antes de comenzar a escribir código, es necesario tener una visión general de la PDE. Aplicando conceptos de DDD, se crea un modelo abstracto en el ámbito del dominio. Esto permite extraer y organizar el conocimiento del mismo, y proporciona el lenguaje ubicuo para los desarrolladores y los expertos de negocio. A continuación, se muestra en la figura 4 el planteo general.

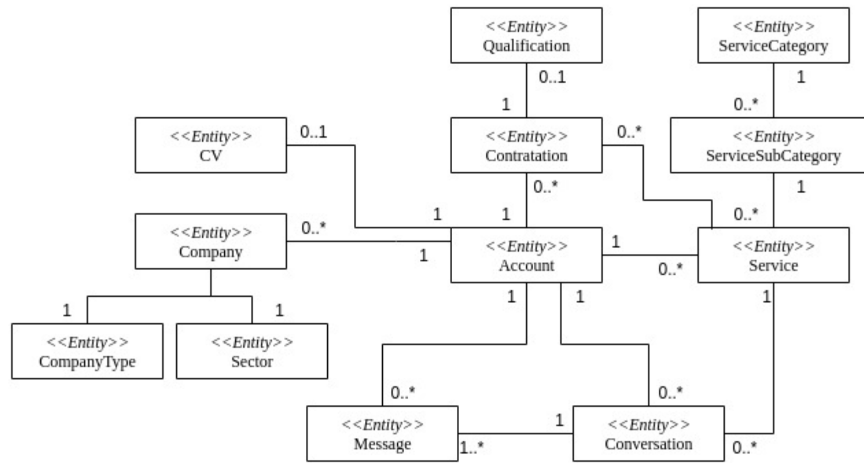

Fig. 4: Representación del modelo de dominio.

4.2 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.

Por ejemplo, para un subsistema de gestión de servicios se tendrá una representación de la entidad Servicio con muchas más características, como descripción, nombre, tipo, precio, reputación, etc. En cambio, para un subsistema de gestión de contratación, solo se necesita saber si un Servicio está disponible o no.

Un contexto delimitado define el límite de un dominio. De la figura 4, es posible agrupar la funcionalidad teniendo en cuenta si estas compartirán un único modelo de dominio. En la figura 5 se muestra el mapa de contexto de PDE, este permite representar e identificar los contextos delimitados, los mismos no se encuentran aislados entre sí, dado que el contexto Bounded Context-Contratation depende de Bounded Context-Profile y Bounded Context-Service.

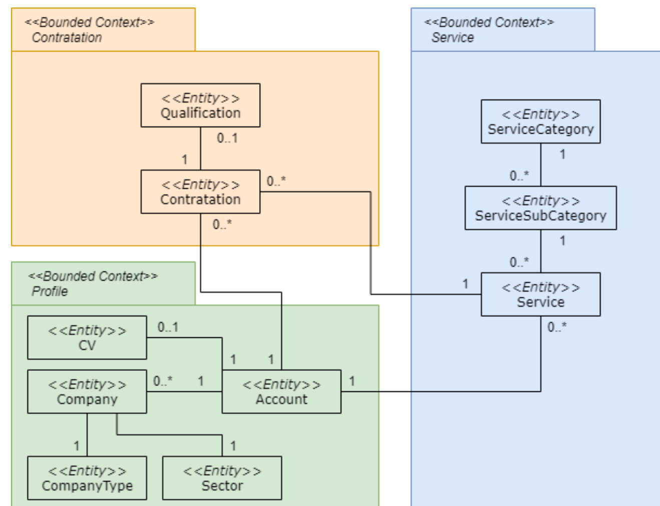

Fig. 5: Mapa de contextos

Durante esta fase estratégica de DDD, se asignó el dominio de negocio y se definieron los contextos delimitados. El diseño basado en dominios táctico consiste en definir los modelos de dominio con más precisión. Presentamos a continuación, el caso de uso que se encuentra en el contexto delimitado Bounded Context-Contratation.

4.3 Definición del Caso de Uso

Caso de uso: Contratación de un servicio
Actor principal: Usuario demandante
Actor secundario: Usuario oferente
Propósito: permite a un usuario(demandante) de la plataforma efectuar la contratación de un servicio ofrecido por otro usuario(oferente).
Resumen: 1. Un usuario demandante de la aplicación comienza el proceso al solicitar la contratación de un servicio. 2. El sistema valida que el servicio esté disponible. 3. El sistema genera la contratación en estado pendiente, enviando la solicitud al usuario oferente para que pueda “Aceptar” o “Rechazar”. 4. El usuario oferente podrá “Aceptar” o “Rechazar”. En ambos casos el sistema notificará al usuario demandante. 5. En caso de “Aceptar”, el sistema pasará la contratación a estado “Aceptado”. 6. En caso de “Rechazo”, se le solicitará un motivo y la contratación finalizará con estado “Cancelada” 7. El usuario demandante, podrá ponerse en contacto con el oferente para coordinar y dar curso a la contratación. 8. El usuario demandante podrá “Finalizar” y calificar la contratación.

Del caso de uso presentado se muestra la construcción y diseño de la arquitectura hexagonal de acuerdo al enfoque DDD a partir de una arquitectura en capas típica como se muestra en la figura 6. Las secciones a continuación explican, en detalle, el resultado de la transformación y adaptación de las diferentes capas típicas de la arquitectura.

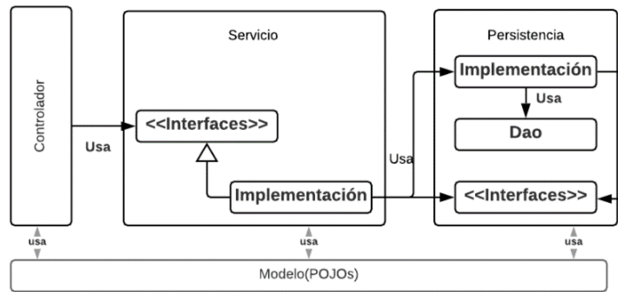

Fig. 6 Arquitectura en capas típica.

Al realizar una revisión de la arquitectura presentada en la figura 6, es posible identificar la utilización del patrón POJO como implementación del modelo [24], totalmente anémico [26], es decir, las entidades mismas son muy delgadas, proporcionan campos para contener el estado y métodos getter/setter. Por otro lado, se encuentra el módulo de servicio, con lógica de negocio que es compartida a través de todas las capas con código técnico (creación de patrón DAOs [27]). Esto se debe a que no hay una separación limpia entre el código técnico y la lógica de negocio.

4.4 Separación de las responsabilidades: Transformación a la arquitectura hexagonal

Es posible buscar un modo de separación, de manera que, si hay que reemplazar o cambiar el "stack" tecnológico, se pueda reutilizar totalmente la lógica de dominio (negocio). Es aquí donde entra en juego la arquitectura hexagonal. Uno de los conceptos clave de esta arquitectura es tener como pieza central de la misma, el modelo de dominio en un solo lugar. El dominio (es decir, el hexágono) debe depender sólo de sí mismo para garantizar el desacoplamiento entre la lógica de negocio y las capas técnicas. Al usar el patrón de Inversión de Control [28] se logra este desacoplamiento, quedando la arquitectura, de acuerdo al esquema anterior de la siguiente manera:

Fig. 7: Transformación de la arquitectura típica

El patrón de inversión del control garantiza el aislamiento del hexágono, con lo cual es posible invertir las dependencias de las capas externas. La transformación y adaptación es posible como se puede ver en la figura 8:

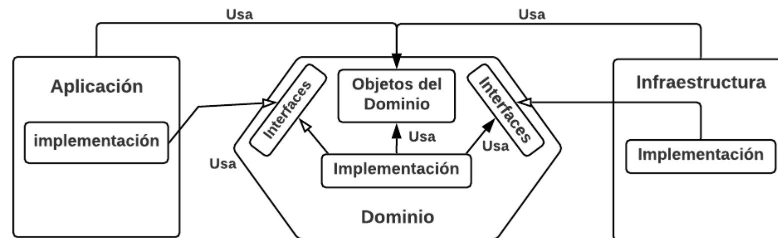

Fig. 8: Representación de la arquitectura hexagonal

La lógica de dominio se especifica en un núcleo, que llamaremos parte interna, el resto son partes externas (aplicación e infraestructura). A continuación, se explica y se detalla cada una de las capas de la arquitectura hexagonal:

A la izquierda, el lado de la capa de aplicación. A través de esta capa el usuario interactúa con la aplicación. Esta área puede contener elementos como interfaces de usuario, controladores RESTful, etc. Este es el lado donde encontramos a los actores que manejan el dominio. Para el caso de uso, definimos la capa de aplicación a través de una API RESTful como muestra en la implementación 1 de la clase ContratationController:

```
public class ContratationController {
   private ContratationService contratationService;
   @PostMapping
  public  ResponseEntity createContratation(ContratationDTO  dto){
     contratationService.createContataton(dto.getService(),dto.getProfile());
  }
}
```

Implementación 1. Clase ContratationController

Centro, el dominio, es el código que implementa la lógica de negocio. Esta capa debe aislarse de la capa de aplicación e infraestructura. Los elementos del dominio deben estar diseñados para mantener el estado y el comportamiento de negocio correctamente. Comenzamos con la creación los objetos para el caso de uso:

```
public class Contratation {
   // otros atributos
   private List<ContratationRecord> contratationRecords;
   private Service service;
   private Profile service;
   public void createContratation(Service service, Profile profile) {
      validateService(service);
      validateProfile(profile);
      this.status = ContratationStatus.CREATED;
      this.contratationRecords.add(new ContratationRecord(this));
   }
```

```
    public void removeContratationRecord () {
        //……..
    }
    public void addContratationService (Service service) {
        this.service = service;
    }
}
```

Implementación 2: Clase de negocio Contratation

La clase Contratation es nuestra raíz agregada. El agregado es un patrón táctico importante en DDD, que ayuda a mantener la consistencia de los objetos de negocio. Cualquier cosa relacionada con la lógica de negocios pasará por esta clase. Por lo tanto, los agregados, se guardan y se actualizan como un todo dentro de una transacción [7]. Esta clase será la responsable de mantener el estado correcto y crear otros objetos, como por ejemplo ContratationRecord.

```
public class ContratationRecord {
private UUID id;
private ContratationStatus status;
private Date dateCreated;
private Contratation contratation;

public ContratationRecord(Contratation contratation) {
            this.contratation = contratation;
            this.status = contratation.getStatus();
            this.dateCreated = new Date();
  }
}
```

Implementación 3: Clase de negocio ContratationRecord.

A continuación, se establecen los límites de la capa de dominio que se encuentran aislados por las interfaces (puertos). El código de la aplicación se dirige hacia el dominio por la interface ContratationService. El código que es dirigido por el dominio hacia la infraestructura es a través de la interface ContratationRepository, ambos objetos se encuentran dentro del límite de la capa de dominio. Estas interfaces actúan como aislantes explícitos entre el interior y el exterior de la arquitectura. A continuación, se describe cada una:

```
public interface ContratationService {
    void createContratation(Contratation contratation);
}

public interface ContratationRepository {
    void save (Contratation contratation);
}
```

Implementación 4: Interfaces de ContratationService y ContratationRepository

Por último, la definición de un servicio de dominio es utilizado como punto de entrada al modelo de dominio. Este es representado por el caso de uso, el cual traduce en llamadas de método orquestadas a las entidades de dominio que hacen el trabajo real. Muchas de las reglas de negocio están localizadas en las entidades en lugar de los servicios de dominio (implementación del caso de uso).

```
public class DomainContratationService  implements ContratationService {
    private final ContratationRepository contratationRepository;
```

```
  @Override
  public  void createContratation(Service service, Profile profile){
          Contratation contratation = new Contratation();
          contratation.createContatation(service, profile);
          contratationRepository.save(contratation);
  }
}
```

Implementación 5: Servicio de dominio DomainContratationService

A la derecha, la capa de infraestructura. contiene todo lo que la aplicación necesita para funcionar, por ejemplo, la configuración de Spring, la base de datos, etc. Además, implementa interfaces dependientes de la capa de dominio. Se muestra, la clase que registrará el servicio DomainContratationService como un bean Spring:

```
@Configuration
public class BeanConfiguration {
   @Bean
  ContratationService contratationService(ContratationRepository repository)
 {return new DomainContratationService(repository);}
}
```

Implementación 6: clase de configuración.

A continuación, se muestra la implementación de la interface ContratationRepository desde la capa de dominio:

```
public class ContratationRepositoryJPA  implements ContratationRepository {
   private final JPARepository jpaRepository;

  @Override
  public  void save(Contratation contratation){
          jpaRepository.save(contratation);
  }
}
```

Implementación 7: Clase ContratationRepositoryJPA

La arquitectura hexagonal utiliza la figura de los puertos y adaptadores para representar las interacciones entre el interior y el exterior. Para finalizar con el caso de estudio planteado para la PDE se muestra a continuación, la comunicación entre los componentes de las capas de aplicación, dominio e infraestructura que se definieron para el caso de uso "Contratación de servicios". En la figura 9 se muestra que la capa de dominio define los puertos, a los que se pueden conectar indistintamente cualquier tipo de adaptadores, implementados por la capa de infraestructura si siguen la especificación definida por el mismo.

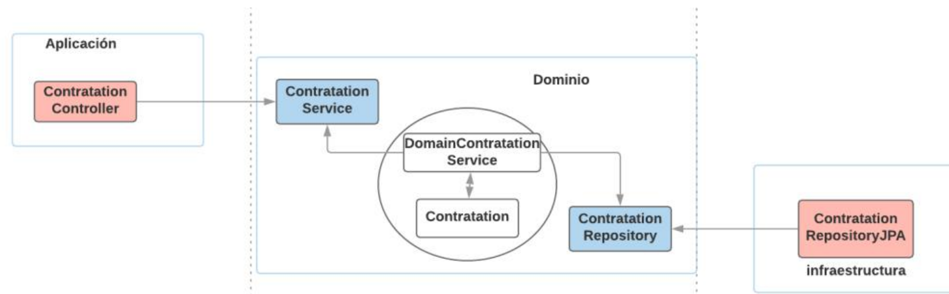

Fig. 9 representación del caso de uso “Contratación de servicio”

5 Trabajos Relacionados

Este trabajo está basado en tareas de investigación que analizaron los aportes científicos que se encuentran en esta misma línea. En particular, se investigaron trabajos relacionados con la aplicación del enfoque DDD que proporciona un medio para representar el dominio del negocio, como pieza fundamental y la creación de una arquitectura, que pudiera brindar un enfoque riguroso y un marco de calidad que resultará en mejoras de los procesos de diseño y de desarrollo de aplicaciones. Algunos de los trabajos más importantes se discuten a continuación.

En [29] plantea un prototipo de arquitectura creada con el propósito de utilizar el enfoque DDD- para acortar el desarrollo de proyectos de software. Discuten sobre el papel importante que desempeña DDD en los proyectos de Tecnología. Presentan una arquitectura e indica que, con la ayuda de técnicas estándar y controladas, es posible alcanzar ganancias significativas en la programación y el costo del software. Vijay Nair [30] muestra cómo DDD se combina con Jakarta EE MicroProfile o Spring Boot para ofrecer un paquete completo para crear aplicaciones de nivel empresarial. En este artículo plantea cómo conceptos de DDD (contextos delimitados, UL y Agregados) y las herramientas disponibles correspondientes (CDI, JAX-RS y JPA) dentro de la plataforma Jakarta EE se unen en una de las formas más eficientes para desarrollar software complejo. Como caso de estudio construye una aplicación monolítica.

En [31], utiliza a DDD como guía para el diseño de microservicios basado en modelos. Plantea la relevancia que DDD adquiere al descomponer los dominios en contextos, donde cada uno de estos corresponden a microservicios funcionales que proveen las capacidades de negocio específicas. Presenta, además, la arquitectura de microservicios (MSA, por sus siglas en inglés, MicroService Architecture) como un estilo arquitectónico para sistemas de software distribuido con altos requisitos de escalabilidad y adaptabilidad. Por otro lado, aplica DDD a MSA y discute varios problemas en lo que respecta a la derivación de servicios de los modelos de dominio, la modelización de los componentes de la infraestructura y la modelización de los dominios en equipos autónomos. Demuestra mediante herramientas del desarrollo dirigido por el modelo (MDD, por sus siglas en inglés Model Driven Development) los desafíos que se presentan al construir los artefactos del sistema.

Finalmente, [32] plantea como DDD se ha convertido en una técnica popular para

descomponer un dominio en los llamados contextos delimitados, esto es debido a que las arquitecturas orientadas a microservicios han recibido mucha atención en los últimos años; sobre todo en las empresas que adoptan estos conceptos y tecnologías para aumentar la agilidad, la escalabilidad y la capacidad de mantenimiento de sus sistemas. Aplicar los patrones estratégicos de DDD como Bounded Context y Context Map, puede servir de apoyo a los analistas de negocios, los arquitectos y los que adoptan a los microservicios cuando se presentan los desafíos de descomponer e integrar múltiples servicios independientes de un sistema. Presenta una herramienta de código abierto denominada Context Mapper, esta solución ofrece un lenguaje específico de dominio (DSL por sus siglas en ingés, Domain Specific Language) que expresa los patrones estratégicos de DDD dado que los lenguajes de descripción de arquitectura existentes no apoyan suficientemente las pautas estratégicas del DDD. El DSL y las herramientas propuestas ayudan a los arquitectos de software en el proceso de encontrar descomposiciones de servicios.

Al comparar el estado del arte con los resultados de este trabajo, es posible fomentar el enfoque DDD, donde se plantea un cambio de estrategia en el desarrollo de software desde la perspectiva del dominio relacionándolo con una determinada arquitectura (Hexagonal, como la propuesta en este trabajo), e identificando un conjunto de tecnologías para su implementación, aportando, a nuestro criterio, evidencia práctica para su aplicación en sistemas monolíticos como el presentado en el caso de estudio. De este modo, aportamos contribución empírica para demostrar las ventajas de combinar principios, técnicas y patrones de DDD con arquitecturas de software.

6 Conclusiones y Trabajos futuro

En el corto o mediano plazo, es indispensable pensar en la necesidad de un cambio revolucionario en la forma de escribir el software. Esta modernización puede ser tanto estratégica como técnica y se trata principalmente de aprovechar los beneficios de las arquitecturas y plataformas modernas. Es por ello que abordamos un enfoque para el desarrollo de software basado en el dominio. La ventaja de aplicarlo es afrontar problemas complejos que requieren ser resueltos separando la lógica y/o reglas de negocio de la tecnología. El enfoque DDD nos ofreció la posibilidad de contar con principios, patrones y actividades de cómo construir un modelo de dominio como artefacto central del sistema. La adopción del mismo implica un cambio de mentalidad en el desarrollo de software y en la construcción de la arquitectura porque permite centrarnos en construir y gestionar los aspectos de negocio por sobre la tecnología. La elección de la arquitectura hexagonal fue posible en este contexto dado que fue introducido por Vernon [16], el cual tiene un efecto beneficioso, separar la complejidad del negocio como pieza central y probar automáticamente el comportamiento independientemente de todo lo demás, esto es la razón por la que esta arquitectura fue adoptada y se alinea perfectamente con DDD. Es importante destacar que alrededor de la lógica de negocio, el hexágono(núcleo) se encuentra libre de cuestiones de tecnología (API REST, Base de datos, etc) y la posibilidad de incorporar o cambiar de tecnología basta con solo implementar, para un puerto especifico un nuevo adaptador, es decir construir un nuevo “plugin”.

DDD es un concepto poderoso que cambia la forma en que los arquitectos,

desarrolladores y tester perciben el software, desde nuestra perspectiva podemos demostrar las ventajas de combinar los principios, técnicas y patrones que ofrece para cambiar nuestra forma de construir el software aplicando la filosofía de "primero el dominio y segundo la infraestructura".

Trabajos a futuro incluyen extender el trabajo aplicando otras técnicas y metodologías de desarrollo como Línea de productos de software(LPS) en la que es posible la reutilización sistemática. LPS brinda un enfoque diferente para la creación de aplicaciones individuales a partir de la construcción de Familias de Productos, este proceso se basa en la selección de elementos comunes y las variabilidades entre los miembros de la Familia. Por otro lado, el paradigma Ingeniería del Software Dirigida por Modelos (MDE por sus siglas en ingles Model-Driven Engineering), es decir, el sistema de modelos tiene suficiente detalle para permitir la generación de código de un sistema completo a partir de los modelos propios, esto es, construir modelos que puedan ser directamente compilados y ejecutados. Por último, explorar los Lenguajes Específicos de Dominio (DSL por sus siglas en inglés, Domain Specific Language) y la Programación Generativa que de manera similar al desarrollo de LPS ofrece la construcción, con un enfoque de reutilización proactivo, de familias de productos relacionados por algún dominio en particular, en lugar de productos independientes.

Referencias

1. Vivas, L; Cambarieri, M: Un Marco de Trabajo para la Integración de Arquitecturas de Software con Metodologías Ágiles de Desarrollo. CACIC. (2013)
2. InfoQ, “Domain Driven Design and Development In Practice”, disponible en https://www.infoq.com/articles/ddd-in-practice/ [accedido: 01/03/2020].
3. Nair V: “Domain Driven Design. In: Practical Domain-Driven Design in Enterprise Java”. Apress, Berkeley, CA. (2019).
4. Eric Evans. Domain-Driver Design Reference. Definitions and Pattern Summaries. Domain Language, Inc. 2015
5. Martin Fowler, “Bounded Context” https://martinfowler.com/bliki/BoundedContext.html [accedido: 27/04/2020].
6. A. Cockburn, “The Pattern: Ports and Adapters,” 2005: disponible en: https://alistair.cockburn.us/hexagonal-architecture/ [accedido: 18/08/2020].
7. E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional (2003).
8. Duc Minh Le et al., Domain-driven design patterns: A metadata-based approach. Conference: 2016 IEEE RIVF International Conference on Computing & Communication Technologies, Research, Innovation, and Vision for the Future (2016).
9. Wayne Zammit, “How Domain Driven Design is playing a role in modernizing software in Government”, https://mita.gov.mt/en/ict-features/Pages/2018/How-Domain-Driven-Design-is-playing-a-role-in-modernising-software-in-Government.aspx [accedido: 05/03/2020].
10. Fuentes, L., Troya, J. M. Desarrollo de software basado en componentes, disponible en: http://www.lcc.uma.es/~av/Docencia/Doctorado/tema1.pdf. (2017)
11. Clements, P., et al, “Software Architecture in Practice”, Pearson Education, (2003).
12. IEEE Standards Association, “1471-2000 - IEEE Recommended Practice for Architectural Description for Software-Intensive Systems”, disponible en: http://standards.ieee.org/findstds/standard/1471-2000.html
13. Fowler, M. “Patterns of Enterprise Application Architecture”, Addison-Wesley, (2002).

14. James O Coplien, Trygve Mikkjel Heyerdahl Reenskaug: “The data, context and interaction paradigm. Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity.” October 2012 Pages 227–228 https://doi.org/10.1145/2384716.2384782
15. Robert C. Martin: Clean Architecture. A Craftsman's Guide to Software Structure and Design. Pearson Education, Inc (2018)
16. V. Vernon, Implementing Domain-Driven Design. Addison-Wesley (2013)
17. Johnson, R., et al, “Professional Java Development with the Spring Framework”, Wiley Publishing Inc, (2005)
18. Hexagonal Architecture, disponible en: https://dzone.com/articles/hexagonal-architecture-what-is-it-and-how-does-it [accedido 22/03/2020]
19. Java Interface, disponible en: https://docs.oracle.com/javase/tutorial/java/IandI/interfaceDef.html [accedido 22/03/2020]
20. Pollack,M. Gierke, O., Risberg T.,: Spring Data. Oreilly (2012)
21. Bauer, C., King, G.: Java Persistence with Hibernate, Manning Publications (2007)
22. Scarioni, C.: Pro Spring Security, Apress (2013)
23. Craig Walls: Spring in Action 4th Edition, Manning (2014)
24. Fowler, Martin, “Plain Old Java Object (POJO)”, disponible en: http://www.martin fowler.com/bliki/POJO.html [accedido 07/12/2019]
25. Hippchen, Benjamin & Giessler, Pascal & Steinegger, Roland & Schneider, Michael & Abeck, Sebastian. (2017). Designing Microservice-Based Applications by Using a Domain-Driven Design Approach. International Journal on Advances in Software (1942-2628). 10. 432 - 445
26. Fowler, Martin, Anemic Domain Model, disponible en: https://www.martinfowler.com/bliki/AnemicDomainModel.html [accedido 22/02/2020]
27. Oracle, “Data Access Object (DAO)”, disponible en: https://www.oracle.com/technetwork/java/dataaccessobject-138824.html [accedido 01/03/2020]
28. Fowler, Martin, Injection, disponible en: https://martinfowler.com/articles/injection.html, [accedido 22/03/2020]
29. F. P. Marzullo, J. M. de Souza and G. B. Xexeo, "A domain-driven approach for enterprise development, using BPM, MDA, SOA and Web Services," 2008 International Conference on Innovations in Information Technology, Al Ain, 2008, pp. 150-154.
30. Nair V. (2019) Cargo Tracker: Spring Platform. In: Practical Domain-Driven Design in Enterprise Java. Apress, Berkeley, CA
31. F. Rademacher, J. Sorgalla and S. Sachweh, "Challenges of Domain-Driven Microservice Design: A Model-Driven Perspective," in IEEE Software, vol. 35, no. 3, pp. 36-43, 2018.
32. Kapferer, S. and Zimmermann, O. (2020). Domain-specific Language and Tools for Strategic Domain-driven Design, Context Mapping and Bounded Context Modeling.In Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD, ISBN 978-989-758-400-8, pages 299-306.