Application of methodologies and approaches for the development of domain-specific software in the context of Digital Government

Aplicación de metodologías y enfoques para el desarrollo de software de dominio específico en el contexto de Gobierno Digital

Mauro Cambarieri[1], Luis Vivas [1], Nicolás García Martinez[1], Alejandra Viadana[2]

[1] Universidad Nacional de Río Negro (UNRN), Laboratorio de Informática Aplicada LIA, Sede Atlántica, Viedma, Río Negro
{mcambarieri, lvivas, ngarciam}@unrn.edu.ar

[2] Universidad Nacional de Rosario (UNR)
alejandraviadama@gmail.com

abstract. This paper presents a set of tools, approaches and strategies to help solve the problems that governments face in meeting the challenges of driving a digital transformation. Governments develop Digital Government initiatives with the objective of seeking answers in tools and/or development approaches. From the perspective of an institution, in the context of Digital Government, it is desirable to achieve software reuse in a systematic way in order to achieve benefits associated with using previously built artifacts in each new development that is performed. For software reuse to be systematic, development processes should address the collective construction of families of products related to a domain [21]. On the other hand, to be in line with the business model and processes, a development approach that focuses on a specific business domain should be considered. This is why, in the short and medium term, the need for a revolutionary change in the way software is written is evident. Modernization can be both strategic and technical and institutions must be prepared when the opportunity arises. Modernization is primarily about reengineering to take advantage of the benefits of modern architectures and platforms. This paper discusses approaches and methodologies that enable the development of domain-specific software for Digital Government.

Resumen. Este trabajo presenta un conjunto de herramientas, enfoques y estrategias que ayuden a resolver los problemas que se le presentan a los gobiernos para hacer frente a los desafíos de impulsar una transformación digital. Los gobiernos desarrollan iniciativas de Gobierno Digital con el objetivo buscar respuestas en herramientas y/o enfoques de desarrollo. Desde la perspectiva de una institución, en el contexto de Gobierno Digital, es deseable lograr la reutilización de software de manera sistemática con el fin de lograr beneficios asociados a utilizar los artefactos construidos previamente en cada desarrollo nuevo que se realiza. Para que la reutilización del software sea sistemática, los procesos de desarrollo deberían abordar la construcción colectiva de familias de

Received December 2020; Accepted March 2021; Published June 2021

productos relacionados con un dominio [21]. Por otro lado, para estar en consonancia con el modelo y los procesos de negocio, se debe considerar un enfoque de desarrollo que se centre en un dominio de negocio específico. Es por ello, que es evidente en el corto y mediano plazo, la necesidad de un cambio revolucionario en la forma de escribir el software. La modernización puede ser tanto estratégica como técnica y las instituciones deben estar preparadas cuando se presente la oportunidad. La modernización se trata principalmente de una reingeniería para aprovechar los beneficios de las arquitecturas y plataformas modernas. Este trabajo discute los enfoques y metodologías que permitan el desarrollo de software de dominio específico para Gobierno Digital

Palabras clave: Domain Driven Design, Línea de Productos de Software (LPS), Model-Driven Engineering, Framework. Gobierno Digital, Arquitectura limpia, Arquitectura Hexagonal

1 Introducción

El presente trabajo se enmarca en el proyecto de investigación PI 40-C-551 "Herramientas Informáticas para el Desarrollo de Servicios Digitales Innovadores para Comunidades Urbanas y Rurales en el Marco de Ciudades y Regiones Inteligentes" desarrollado en el Laboratorio de Informática Aplicada, Sede Atlántica, Universidad Nacional del Río Negro (UNRN). La propuesta se encuentra admitido en la "Convocatoria Ideas Proyectos de Desarrollo y Transferencia de Tecnología (IP-DTT)" llevada a adelante por la Secretaría de Investigación, Creación Artística, Desarrollo y Transferencia de Tecnología de la UNRN, para obtener financiamiento. El proyecto tiene una duración de 18 meses y los resultados serán transferidos al Hospital de Viedma.

La contribución del mismo es mostrar la factibilidad de las herramientas, enfoques y estrategias propuestas que requieren los gobiernos para hacer frente a los desafíos de uno de los ejes centrales para impulsar una transformación digital, como es la tecnología. El trabajo está estructurado de la siguiente manera: La Sección 2 explica el contexto, situación - problema u oportunidad. La Sección 3 explica los enfoques y metodologías propuesta. A continuación, la Sección 4 presenta los objetivos del proyecto. La Sección 5 presenta los resultados esperados. Finalmente, la Sección 6 y 7 presenta trabajos relacionados y conclusiones, respectivamente.

2 Contexto - Situación-Problema u Oportunidad

En 1968, en el Congreso sobre Ingeniería del Software que organizó el Comité de Ciencia de la OTAN, se introdujo por primera vez la idea de la reutilización. En ese Congreso el Ing. M.D. McIlroy, de Bell Laboratories, afirmó que "La industria del software se asienta sobre una base débil, y un aspecto importante de esa debilidad es la ausencia de una subindustria de componentes" [1]. En Ingeniería de Software, se entiende por componente, un fragmento reemplazable de un sistema, dentro del cual se

encuentran implementadas un conjunto de funcionalidades [2]. Un componente puede desarrollarse aislado.

Uno de los desafíos centrales en el desarrollo de software, justamente, es la reutilización, lo que permite utilizar nuevamente uno o más artefactos implementados como parte del desarrollo de un nuevo producto o sistema. Existen diferentes técnicas que han facilitado, de alguna manera, el uso de artefactos de desarrollo de granularidad cada vez mayor. Al no disponer de contextos suficientemente amplios como para detectar elementos reutilizables y donde pueden utilizarse se desemboca muchas veces en reutilización oportuna. Esto es, se utiliza algún artefacto ya desarrollado con las correspondientes adaptaciones, en el nuevo sistema.

Desde la perspectiva de una institución, en el contexto de Gobierno Digital, es deseable lograr, en cada desarrollo nuevo, un enfoque de reutilización sistemática. El fin, es obtener los beneficios asociados a utilizar artefactos construidos previamente (por ejemplo, la madurez y calidad). En el proceso de desarrollo se debe plantear la construcción colectiva de familias de productos relacionados por un dominio para lograr que la reutilización sea sistemática.

En un contexto de Gobierno Digital, a medida que las aplicaciones crecen en complejidad y tamaño, es posible que puedan dividirse en contextos o subdominios independientes. DDD es un enfoque de desarrollo de software destinado específicamente a abordar los sistemas complejos. Su base estratégica es el proceso de descubrir dominios. En el que entran en juego dos conceptos muy importantes, el Lenguaje Ubicuo y los Contextos Delimitados (Bounded Context- BC). El Lenguaje Ubicuo es una colección de términos específicos del dominio, el lenguaje de negocios [1]. Los Contextos Delimitados son los límites dentro de los cuales cada modelo de dominio existe y opera. Este es un aspecto crucial y poderoso de la modelización de dominios, ya 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 sencillo 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. Por este motivo, el enfoque permite combatir la complejidad desde el lado de la "modularización". Al dividir grandes procesos en subprocesos más pequeños, es posible diseñar e implementar módulos separados y combinarlos para abordar diferentes aplicaciones [2].

A lo largo del tiempo se ha buscado crear artefactos de software de forma rápida, con menos errores y poniendo más énfasis en los requisitos y necesidades. Con el objetivo de cumplir con esto, MDE brinda más agilidad al desarrollo de aplicaciones. Es una propuesta que se ha venido trabajando desde hace varios años. Es un paradigma para el desarrollo software que considera a los modelos como el principal elemento del proceso de desarrollo [3]. Gracias al uso de modelos, es posible elevar el nivel de abstracción y de automatización. Esto permite atacar los principales problemas en la creación de software: a) la complejidad, b) el desarrollo de marcos de trabajo y lenguajes específicos para lograr la comprensión del dominio; c) mejorar diferentes aspectos de la calidad del software como la productividad y el mantenimiento; y d) obtener provecho de las transformaciones para automatizar trabajo repetitivo y mejorar la calidad del software.

En la actualidad, los objetivos de la industria de software están puesto en la reducción de costos, el aumento de la calidad, el aumento de la productividad y la reducción del tiempo de desarrollo del software. Enfoques como LPS, MDE y DDD buscan alcanzarlos. Adoptarlos dentro de una organización implica desafíos importantes con respecto a los artefactos de software existentes que deben transformarse (MDE) para respetar una arquitectura de software centrada en BC (DDD) y la reutilización (LPS).

3 Conceptos Utilizados

Las siguientes secciones presentan los conceptos utilizados en este trabajo: Ingeniería del Software Basada en Componentes y Reutilización (Sección 3.1), Línea de Producto de Software (Sección 3.2), Ingeniería de Software dirigido por Modelos (Sección 3.3), Domain Driven Design (Sección 3.4), Arquitectura de Software (Sección 3.5) y Arquitectura Limpia (Sección 3.6),

3.1. Ingeniería del Software Basada en Componentes y Reutilización. La Ingeniería del Software basada en componentes (CBSE: component-based software engineering) se ocupa del desarrollo de sistemas a partir de componentes reutilizables. El enfoque busca favorecer la reutilización y, de este modo, evitar la duplicación de componentes, impidiendo que el análisis, el diseño, la implementación y la prueba de la misma solución, se desarrolle una y otra vez en muchos de los productos. Difiere del enfoque tradicional de desarrollo de software que involucra generalmente la construcción de cada una de las piezas desde cero, cada una de ellas implementada a propósito para el producto en curso. Algunas de las piezas o elementos reutilizables provienen de productos anteriores y son reutilizados con ciertos cambios [1]. CBSE, tiende a concentrar el esfuerzo en la composición de componentes de software desarrollados (que por lo general poseen una funcionalidad concreta), trata el desarrollo de sistemas a partir de la composición e integración de componentes reutilizables. Algunos de ellos, pueden requerir una vinculación con otros componentes para proveer la funcionalidad completa. En cuanto a la comunicación, como la integración de los componentes con los elementos del nuevo sistema que se encuentra desarrollando se realiza entendiendo los componentes como cajas negras, esto permite que se utilicen tal y como están disponibles y sin considerar ninguna modificación interna. [1]. A diferencia del enfoque tradicional, CBSE incluye las siguientes actividades: 1) Selección y evaluación de componentes, 2) Adaptación, 3) Composición, y 4) Evolución [1].

Según [12], existen dos factores principales cuando el enfoque que se utiliza está basado en componentes: el primero, la reutilización – la facilidad para reutilizar componentes existentes en la creación de sistemas más complejos – y el segundo, la evolución – crear un sistema altamente formado por componentes pre-existentes facilita su mantenimiento, ya que si el diseño es correcto los cambios estarán muy localizados y los efectos tendrán poca o ninguna propagación en el resto de los componentes del sistema. Estos factores se aplican bajo las siguientes condiciones: en primer lugar, deben existir componentes para su reutilización. En segundo lugar, debe existir un

conjunto de componentes bien diseñados y listos para su utilización. Por otro lado, debe existir un modelo de componentes que soporte el enlazado y la interacción de componentes, esto es, debe existir un marco de referencia estándar en el que los componentes puedan existir y comunicarse. Finalmente, es necesario un proceso y la definición y diseño de arquitecturas que den soporte al CBD.

La reutilización favorece el desarrollo de herramientas de software con un alto grado de flexibilidad, haciendo uso de componentes en diversos proyectos, ampliando las capacidades, mejoras, funcionalidades y mantenimiento del software.

Hay varias definiciones del término Reutilización de Software. Algunas de estas definiciones son las siguientes. “La reutilización de software es el proceso de implementar o actualizar sistemas de software usando activos de software existentes” [12]. “Reutilización de software es el proceso de crear sistemas de software a partir de software existente, en lugar de desarrollarlo desde el comienzo” [13].

Existen varias modalidades de reutilización utilizadas, lo mismo que beneficios y costos, según el enfoque que se aplique. En particular, el oportunista, se beneficia de componentes de software construidos en productos anteriores, pero que no fueron especialmente desarrollados para ser reutilizados. En el enfoque más planificado o sistemático, los componentes se construyen pensando en que serán reutilizados, lo que significa que debe plantearse el componente con mayor generalidad, y por ello en el momento de su desarrollo, se requiere mayor inversión de tiempo y recursos.

3.2. Línea de Producto de Software. La definición más comúnmente aceptada de una LPS procede del autor Clements, quien define “las líneas del producto de software como un conjunto de sistemas software, que comparten un conjunto común de características (features), las cuales satisfacen las necesidades específicas de un dominio o segmento particular de mercado, y que se desarrollan a partir de un sistema común de activos base (core assets) de una manera preestablecida” [5].

Las LPS tienen como objetivo cambiar el enfoque de la creación de aplicaciones de software individuales a la construcción de familias de productos mediante la identificación de elementos comunes y variabilidades en un conjunto de aplicaciones de software [14]. En este contexto, una LPS permite acotar el tiempo de producción de componentes software, con altos niveles de calidad y una alta capacidad de reutilización.

En cuanto a los beneficios y costos, según diferentes casos de estudios documentados, existe una tendencia un aumento en la productividad que duplica o triplica la producción de software con respecto a los enfoques tradicionales. Las LPS pueden incrementar la productividad del desarrollo de software, dada la reducción del esfuerzo y el costo necesario para desarrollar, mantener y poner en marcha el conjunto de productos software similares [15].

Una LPS no es únicamente una mera reutilización de código, sino que los activos base se benefician también de cada producto desarrollado en la línea dado que toman ventaja del análisis, diseño, implementación, planificación, prueba. Potencian la reutilización estratégica [11], dado que en este contexto es más planificado que

oportunista [16]: en desarrollo de software tradicional, se aprecia la posibilidad de reutilizar un componente después de haberlo desarrollado; en LPS, la reutilización es planificada, de manera que se reutilizan la mayor parte de los activos base en todos los productos de la línea [16]. Una LPS comprende esencialmente dos etapas:

a) Ingeniería de dominio- ID (core asset development, por Clements [5]) es la encargada de diseñar los componentes de software que, de acuerdo a su trazabilidad, hacen parte del dominio común y que son sujetos a reutilización. En esta etapa se definen las similitudes y variantes del modelo de dominio en cada una de las fases. Como producto final en esta etapa se considera un componente altamente reusable y flexible que permite a partir de su definición, la consecución de nuevos modelos y artefactos.

b) Ingeniería de Aplicación - IA (product development, por Clements [15]) se encarga de articular los artefactos definidos en la etapa ID a partir de una plataforma específica. El objetivo de esta etapa consiste en lograr la integración de artefactos bajo uno o varios sistemas, logrando con ello un alto grado de reusabilidad de sus componentes a partir de las especificaciones, la definición y desarrollo, documentación e interoperabilidad entre componentes.

3.3. Ingeniería de Software dirigido por Modelos. La Ingeniería Dirigida por Modelos es una metodología de desarrollo de software que se centra en la creación de modelos, o abstracciones, es decir, identificar conceptos del dominio en lugar de concentrarse en el modelo de cómputo subyacente. A lo largo del tiempo se ha buscado crear artefactos de software de forma rápida, con menos errores y poniendo más énfasis en los requisitos y necesidades. Con el objetivo de cumplir con esto, la metodología MDE brinda más agilidad al desarrollo de aplicaciones [19].

A partir del uso sistemático de los modelos en las diferentes etapas del ciclo de vida, han surgido un conjunto de paradigmas de desarrollo de software que conforman lo que se ha denominado “Ingeniería de Software dirigida por Modelos”, MDE por sus siglas en inglés Model-Driven Engineering o “Desarrollo de Software Dirigido por Modelos” MDSD por sus sigleas en inglés Model-Driven Software Development. El uso de modelos promueve elevar el nivel de abstracción y de automatización y con ello combatir la complejidad en la creación de software, además de mejorar la productividad y mantenimiento del software y la interoperabilidad entre sistemas [18].

En particular, la programación dirigida por modelos es un paradigma para el desarrollo software cuyo principal elemento del proceso son los modelos. A partir de estos modelos el código puede ser generado semiautomáticamente. [17], es decir, como “dibujar un sistema y pasar directamente a ejecutarlo” [1].

3.4. Domain Driven Design (DDD). El diseño impulsado por el dominio es un enfoque de desarrollo de software destinado específicamente a abordar los sistemas complejos, el cual establece lo siguiente: 1-. Centrarse en el dominio. 2.- Explorar modelos de dominio en colaboración con los expertos y desarrolladores. 3- . Hablar un lenguaje común (Lenguaje Ubicuo) dentro de un contexto delimitado. Plantea un enfoque diferente para el diseño y construcción de sistemas y tiene como objetivo crear aplicaciones a partir de una comunicación entre contextos. Es por ello que dichos sistemas son flexibles y adaptables dentro de cada contexto [4].

DDD ofrece un enfoque sistemático y completo para el diseño y desarrollo de software. Proporciona un conjunto de principios, herramientas y técnicas que ayudan a combatir la complejidad, manteniendo el modelo de negocio como pieza central del enfoque. Se adapta bien a una implementación de microservicios, dado que permite la división de un sistema en diferentes contextos delimitados. El diseño esta dado de tal manera que sólo el contexto tiene que ser modificado para implementar cambios o introducir nuevas características. Por tal motivo, se obtendrá el máximo beneficio del desarrollo independiente en diferentes equipos (de desarrollo), ya que varias características pueden implementarse en paralelo sin necesidad de una coordinación. La reutilizando de los microservicios como componentes, permite lograr el aumento de la calidad y la reducción del tiempo de implementación de procesos de negocio de una forma ágil [20]. DDD apunta a centrarse fuertemente en el dominio y el desarrollo de servicios independientes que forman aplicaciones de software más grandes cuando se combinan [7].

3.5. 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 [22].

La arquitectura de software conforma la columna vertebral de cualquier sistema y constituye uno de sus principales atributos de calidad [23]. El documento de IEEE Std 1471-2000 [24] 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 [25]: 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 [25]. 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.

3.6 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 [26] y Datos, Contexto e Interacción, (DCI por sus siglas en ingles Data, Context and Interaction) de James Coplien y Trygve Reenskaug [27] 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 [28].

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).

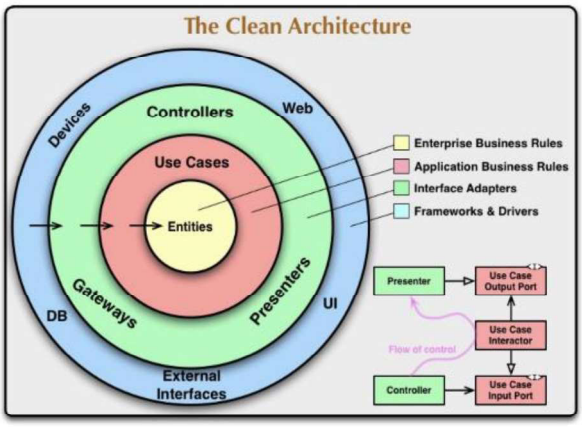

Fig.1 Clean Architecture [28].

3.6.1 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. [28].

3.6.2 Arquitectura Hexagonal.

En el contexto de DDD, Vernon [29] introdujo la arquitectura hexagonal [26]. 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) [30] 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 [31].

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.

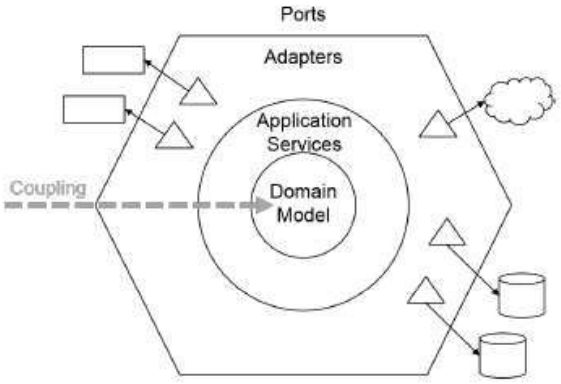

Fig. 2. Arquitectura Hexagonal de A. Cockburn [6]

3.6.3 Implementación de una Arquitectura de Software guiada por el dominio y Entornos de Trabajo (Framework)

En esta sección presentamos las distintas tecnologías y frameworks disponibles para la implementación de la arquitectura hexagonal guiada por el dominio.

Una propuesta para llevar a cabo dicha implementación a partir de los conceptos explicados anteriormente, es definir 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.

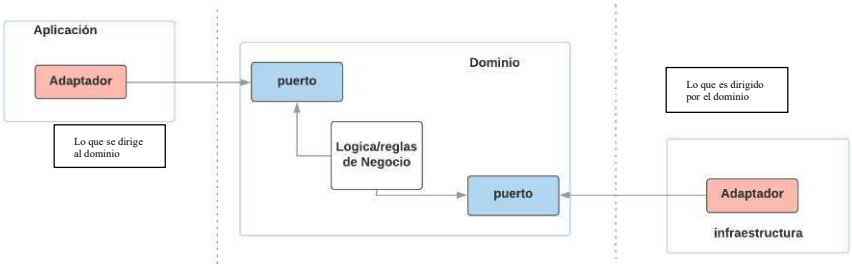

Fig.3 Representación de la arquitectura hexagonal.

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 [26]. 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 [32]. 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 [33] con el soporte de Hibernate [34] y las clases de configuración, a cargo de Spring [36] 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 [36], y el módulo de seguridad de autenticación y autorización como Spring Security [35]. 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 [37]) y representa objetos que son parametrizables por medio de archivos de configuración u anotaciones.

4 Objetivos

El objetivo general de este proyecto es diseñar un marco de trabajo (framework) que permita el desarrollo de software de dominio específico para gobierno digital. El mismo consiste en investigar y desarrollar herramientas informáticas, cuyos dominios sean complejos de resolver.

Los objetivos específicos relacionados con el objetivo general que se abordará son:

1. Integrar herramientas existentes que soportan las metodologías y enfoques de desarrollo de software.
2. Analizar el dominio de aplicación, en Plataformas de servicios con intermediación, en particular se estudiará el contexto del gobierno de la Ciudad de Viedma - Río Negro.
3. Especificar requerimientos para identificar contextos delimitados, que representan unidades organizacionales bajo el enfoque DDD y bajo la metodología LPS los activos base y variables por cada uno (en base a lo analizado en 2).
4. Diseñar una arquitectura de referencia que permite la instanciación de la misma para la aplicación de los enfoques propuestos
5. Construir un marco de trabajo(Framewrok) que en base a la arquitectura (objetivo 4) y los requerimientos (objetivo 3) que considere un modelo metodológico y enfoque de construcción acorde para que permita la reutilización sistemática con el fin de lograr beneficios asociados a utilizar los artefactos(componentes) construidos previamente y la implementación de un modelo en constante evolución.
6. Validar el marco de trabajo producto del Proyecto en el caso de estudio, elaborando el prototipo funcional estable del framework.
7. Identificar resultados concretos sobre la usabilidad y utilidad del framework.

5 Resultados esperados

Como resultado de este proyecto, se espera identificar herramientas, metodologías y modelos conceptuales para el desarrollo de software de dominios complejos que permita, la integración de herramientas de construcción y sea considerado un modelo metodológico acorde para la reutilización sistemática del software construido previamente y la implementación de un modelo y/o reglas de negocio en constante evolución. En cuanto al diseño de la arquitectura de software se espera construir regiones o capas, para desacoplar las mismas y que evolucionen de manera aislada, independientes de la tecnología. También, se espera que el proyecto signifique un cambio revolucionario en la forma de escribir el software y una solución a la modernización, entre otras, para aquellos sistemas que se encuentran obsoletos por la tecnología que utilizan. Se obtendrá mediante un caso de estudio, una Plataforma de comunicación, esta será implementada y transferida al Hospital de Viedma. La elaboración de este prototipo funcional y estable se espera que produzca, un sistema flexible, con capacidad de reutilización de los componentes que lo conforman, mantenible, independiente de la tecnología y tolerante a los cambios (reglas de negocio).

6 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 [38] 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 [39] 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 [40], 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, [41] 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.

7 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 ofrece 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 [29], 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".

En cuanto a Línea de Productos de Software(LPS) nos 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, en la que es posible la reutilización sistemática. Por otro lado, el paradigma Ingeniería del Software Dirigida por Modelos (MDE), es decir, el sistema de modelos tiene suficiente detalle que permite la

generación de código de un sistema completo a partir de los modelos propios, esto es, podamos construir modelos que puedan ser directamente compilados y ejecutados. Los modelos están dados por los Lenguajes Específicos de Dominio (DSL) que de manera similar al desarrollo de LPS, la construcción se basa en un enfoque de reutilización proactivo y la generación de código automática.

Referencias

[1] Usaola, M.: Desarrollo de software basado en reutilización. UOC. 2013.

[2] Montiliva, J et al: Desarrollo de software basado en Componentes. V Congreso de Automatización y Control Mérida. Noviembre 2013.

[3] Markus Voelter ; Iris Groher. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. 2007.

[4] E. Evans, Domain-Driven Design:Tackling Complexity in the Heart of Software. Addison-Wesley, 2003

[5] Clements,P., Northrop,L.: Software Product Lines: Practices and Patterns. Addison-Wesley (2001).

[6] Oquendo, F.: p-Method: A Model-Driven Formal Method for Architecture-Centric Software Engineering. ACM SIGSOFT Software Engineering Notes. Volume 31 Number 3. (2006).

[7] Domain Driven Design Reference. Definitions and Pattern Summaries. Eric Evans.2015.

[8] Cabot, J. http://es.slideshare.net/jcabot/mdd-desarrollo-de-software-dirigido-por-modelos-que-funciona-de-verdad.

[9] C. W. Krueger:Homeaway's transition to software product line practice: Engineering and business results in 60 days. In Proc. of the 12th International Software Product Line Conference (SPLC'08), Ireland, pages 297–306. IEEE, September 2008.

[10] J. Gonzalez-Huerta, S. Abrahão, E. Insfran: Un enfoque Multi-modelo para la Introducción de Atributos de Calidad en el Desarrollo de Líneas de Producto Software. XVI Jornadas de Ingeniería del Software y Bases de Datos. Septiembre de 2011

[11] Peñalvo, F., et al.: Líneas de Productos, Componentes, Frameworks y Mecanos. Informe Técnico, Departamento de Informática y Automática - Universidad de Salamanca (2002).

[12] Sodhi, J., Sodhi,P: Software reuse: Domain analysis and design process. McGraw-Hill. (1999)

[13] Sametinger, J.: Software engineering with reusable components. Springer Verlag, (1997)..

[14] C.Parra, D. Joya, L. Giral, A.Infante: An SOA approach for Automating Software Product Line. Proceeding SAC '14 Proceedings of the 29th Annual ACM Symposium on Applied Computing Pages 1231-1238. Republic of Korea. March2014

[15] Díaz O., Trujillo, S.: Fábricas de Software: experiencias, tecnologías y organización" (2º edición). Editorial Ra-Ma, (2010).

[16] Trujillo, S., Batory, D. y Díaz, O: Feature Oriented Model Driven

Development: A Case Study of Portlets. International Conference on Software Engineering (ICSE) (2007).

[17] H. Escobar, H.Triana, S.Silveira Netto: Conceptualización de arquitectura de gobierno electrónico y plataforma de interoperabilidad para América Latina y el Caribe. Naciones Unidas, Santiago de Chile. julio de 2007.

[18] J. García Molina, García Rubio, V. Pelechano, A. Vallecillo, JM. Vara, C.Vicente-Chicote: Desarrollo de Software Dirigido por Modelos: Conceptos, Métodos y Herramientas. Edición Ra-Ma 2012.

[19] H. Ed-douibi et al. EMF-REST Generation of RESTful APIs from Models. Proceeding SAC '16 Proceedings of the 31st Annual ACM Symposium on Applied Computing Pages 1446-1453.Italy.2016

[20] Microservices: Flexible Software Architecture. Addison-Wesley Professional.2016

[21] LPS mediante un enfoque generativo, disponible en: http://portal.uned.es/portal/page?_pageid=93,56233773&_dad=portal&_schema=PORTAL&idAsignatura=31105043.

[22] Fuentes, L., Troya, J. M. Desarrollo de software basado en componentes, disponible en: http://www.lcc.uma.es/~av/Docencia/Doctorado/tema1.pdf. (2017)

[23] Clements, P., et al, "Software Architecture in Practice", Pearson Education, (2003).

[24] 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

[25] Fowler, M. "Patterns of Enterprise Application Architecture", Addison-Wesley, (2002).

[26] A. Cockburn, "The Pattern: Ports and Adapters," 2005: disponible en: https://alistair.cockburn.us/hexagonal-architecture/ [accedido: 18/08/2020].

[27] 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

[28] Robert C. Martin: Clean Architecture. A Craftsman's Guide to Software Structure and Design. Pearson Education, Inc (2018)

[29] V. Vernon, Implementing Domain-Driven Design. Addison-Wesley (2013)

[30] Johnson, R., et al, "Professional Java Development with the Spring Framework", Wiley Publishing Inc, (2005)

[31] Hexagonal Architecture, disponible en: https://dzone.com/articles/hexagonal-architecture-what-is-it-and-how-does-it [accedido 22/03/2020]

[32] Java Interface, disponible en: https://docs.oracle.com/javase/tutorial/java/IandI/interfaceDef.html [accedido 22/03/2020]

[33] Pollack,M. Gierke, O., Risberg T.,: Spring Data. Oreilly (2012)

[34] Bauer, C., King, G.: Java Persistence with Hibernate, Manning Publications (2007)

[35] Scarioni, C.: Pro Spring Security, Apress (2013)

[36] Craig Walls: Spring in Action 4th Edition, Manning (2014)

[37] Fowler, Martin, “Plain Old Java Object (POJO)”, disponible en: http://www.martin fowler.com/bliki/POJO.html [accedido 07/12/2019]

[38] 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.

[39] Nair V. (2019) Cargo Tracker: Spring Platform. In: Practical Domain-Driven Design in Enterprise Java. Apress, Berkeley, CA

[40] 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

[41] 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.