“DDD (diseño dirigido por el dominio) y aplicaciones Enterprise: ¿fidelidad al modelo o a las herramientas?

Adriana Echeverría, Gustavo López, María Delia Grossi, Arturo Servetto, Ismael Jeder, Adrián Paredes, Pablo Linares.
Laboratorio de Aplicaciones Informáticas - Departamento de Computación - Facultad de Ingeniería
Universidad de Buenos Aires

1. Resumen

Teniendo como marco de trabajo la cátedra de “Trabajo Profesional” de la Facultad de Ingeniería de la Universidad de Buenos Aires, se presenta una experiencia de desarrollo, en la que se trató de respetar el conocimiento teórico, el modelo así como también contemplar maneras de resolver la distancia entre lo que puede ofrecer una herramienta de desarrollo que no se ajusta completamente a la teoría o el modelo elegido para construir la aplicación Enterprise correcta.
El prototipo desarrollado pertenece al tipo de sistemas de información soporte del negocio de transporte de pasajeros, viajes, itinerarios y pasajes.
Las diferencias conceptuales entre el modelo de diseño dirigido por el dominio o DDD y la plataforma EJB elegida llevaron a considerar como solución disponer en el modelo tanto Entity Beans como Session Beans. Lo mismo se propone para cada objeto de valor de DDD, disponiendo en la capa de dominio, los servicios implementados como Session Beans.

2. Abstract

Having as a working framework the course “Professional Work” at the School of Engineering, University of Buenos Aires, we describe a development experience in which we tried to respect the theoretical knowledge, the model and also contemplate the ways of resolving the distance between what a development tool that doesn’t conform completely to the theory or the model chosen to build an Enterprise application correctly can offer.
The prototype developed is of the type of information systems support of business of passenger transport, travel, itineraries and tickets.
The conceptual differences between domain driven design or DDD and the platform model chosen EJB led us to consider as a solution to have in the model both: Entity Beans and Session Beans. The same is proposed for each target value of DDD, providing at the domain layer, the services implemented as Session Beans.

3. Palabras clave

DDD, EJB 3.0, dominio portable, Enterprise

4. Aplicaciones Enterprise

Se entiende por aplicación Enterprise a un sistema de información con una arquitectura de n capas, es decir, bastante más sofisticada que una arquitectura cliente – servidor típica, físicamente distribuida a lo largo de una red de computadoras. Alguna de las capas da soporte especialmente al modelo del negocio que soporta el sistema y se caracteriza principalmente, porque posibilita la integración de los servicios, funcionalidades y características no funcionales que requiere la empresa. En efecto, es la integración lo que distingue a las aplicaciones Enterprise:

pueden ejecutarse de manera independiente entre sí pero coordinarse entre sí con bajo acoplamiento.
Esto posibilita a cada aplicación, enfocarse en un conjunto de funcionalidades y delegar a otras aplicaciones, funcionalidades relacionadas. Las aplicaciones integradas se comunican asincrónicamente no debiendo esperar por respuestas a los mensajes enviados; pueden actuar sin una respuesta o realizar otras tareas de manera concurrente hasta que la respuesta esté disponible [1].

5. DDD

DDD o diseño dirigido por el dominio es tanto una manera de pensar como un conjunto de prioridades, con el objeto de acelerar el desarrollo de proyectos de software que deben lidiar con dominios complicados. [2]
Se entiende por dominio del software a todo aquello que está relacionado con alguna actividad relevante para el usuario y su negocio. De esta manera, es posible encontrar una creciente gran variedad de dominios pertenecientes al mundo físico tangible como un asiento en un ómnibus de larga distancia o intangible como el valor de una imagen acompañada de sonidos. También son fundamentales ciertos dominios específicos a la ingeniería del software tales como sistemas de verificación automática de aplicaciones informáticas.
En aplicaciones Enterprise, las necesidades y expectativas de los usuarios provocan una gran complejidad en el manejo de información, el cumplimiento de funcionalidades requeridas, así como varias características no funcionales.
Es decir que las actividades de los usuarios para el funcionamiento del negocio deben ser captadas y realizadas lo más ajustadamente posible por el equipo de desarrollo de manera que el producto resuelva de manera óptima las necesidades de manejo del negocio. Esto resulta, a veces, extremadamente complejo para ser resuelto mediante sistemas basados en computadora.
Por lo tanto, se hace imprescindible elaborar un modelo del negocio, más precisamente, del dominio del problema que dirija el desarrollo hasta lograr el producto final.
Entre todos los aspectos que caracterizan DDD se han considerado dos en particular:

- arquitectura de cuatro capas,
- lenguaje omnipresente (“ubiquitous”).

5.1.1. Arquitectura de cuatro capas

Aun cuando Eric Evans no establece una arquitectura definida, sin embargo enuncia lineamientos de diseño propiciando la separación de capas.
El objetivo principal de DDD es aislar el dominio del resto de la aplicación.
La separación de la capa de dominio de las capas infraestructura y de la interfaz de usuario da lugar a un diseño mucho más limpio de cada capa. Capas aisladas son mucho menos costosas de mantener puesto que tienden a evolucionar a diferentes tasas de rapidez y respondiendo a diferentes necesidades. La separación también ayuda al despliegue en sistemas distribuidos, permitiendo que diferentes capas se ubiquen en diferentes sitios físicos, servidores o clientes, de manera de minimizar las comunicaciones y optimizar el rendimiento. [3]
Siguiendo a Evans, se ha considerado el siguiente conjunto de capas:

- de presentación o interfaz de usuario,
- de aplicación,
- de dominio o capa del modelo y
- de infraestructura

5.1.2. Lenguaje omnipresente.

El modelo de dominio también dirige el uso de un lenguaje para un proyecto de software, el cual será compartido por todos los involucrados en el mismo. A este lenguaje

común, DDD lo denomina “ubiquitous language” o lenguaje omnipresente.
El alcance del lenguaje omnipresente de un proyecto deberá considerar:

- nombres de clases y sus operaciones destacadas,
- términos para discutir reglas de negocio que se han incorporado al modelo
- nombres de patrones de análisis y de diseño que se aplican al modelo de dominio

Disponer de un lenguaje común para todo el proyecto implica un proceso iterativo, de continua revisión y re elaboración, en el que se deben mantener sincronizados modelo, lenguaje y código, ya que el código fuente también debe ser escrito respetando el lenguaje único.
Un cambio en el lenguaje implica un cambio en el modelo y en el código; los tres artefactos deben evolucionar juntos, como un único bloque de conocimiento.
Cabe destacar que este principio ya fue establecido por Bertrand Meyer en su clásico libro [4]

6. Caso de estudio.

El sistema de información desarrollado es un prototipo básico parametrizable, un “core” de negocio capaz de adaptarse a las distintas empresas de transporte de larga distancia. En lo que respecta a una arquitectura Enterprise típica, lo que este prototipo ofrece es una capa de acceso al negocio, una capa de dominio y las capas de infraestructura necesarias. Sobre este “core” se pueden disponer variadas presentaciones. Las aplicaciones que se construyan sobre el sistema no estarán limitadas a una empresa de transporte en particular (como aerolíneas, barcos, micros o trenes) y por supuesto tampoco a una vista (pudiéndose construir aplicaciones web, de escritorio, para dispositivos móviles, etc.). El prototipo es genérico, por lo cual no sólo sirve para construir aplicaciones de transporte de personas, sino que también brinda la posibilidad de construir aplicaciones para transporte de carga o paquetes.

7. DDD vs herramientas elegidas para la implementación

La arquitectura diseñada para la aplicación consta de las siguientes capas:

- presentación,
- acceso al negocio,
- dominio
- persistencia.

Sólo las capas de acceso al negocio y de persistencia interactúan directamente con la capa de dominio.
La capa de Dominio no tiene ninguna dependencia con otra capa, como puede apreciarse en la figura 1.
De hecho, son las capas de Persistencia y de Servicio las que interactúan directamente con la capa de Dominio y no al revés. Esto es así dado que el dominio es portable, genérico y reutilizable, independientemente de la tecnología, plataforma y/o arquitectura que se utilice.

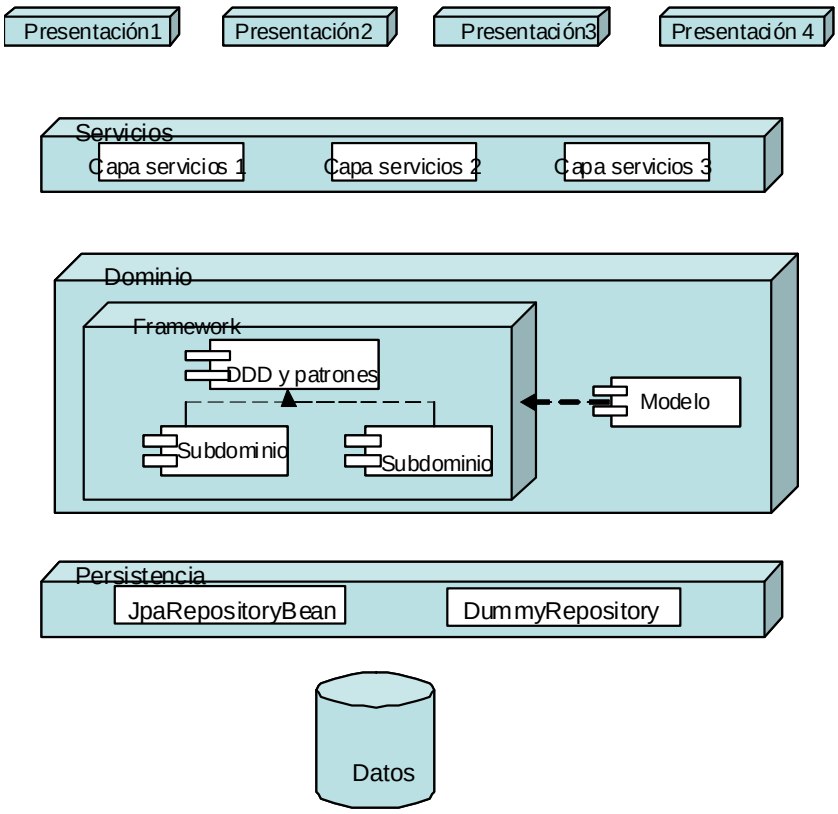

Figura 1

El hecho de que el sistema de información se pensó como aplicación Enterprise, debiendo soportar concurrencia, transacciones de negocio y por supuesto, funcionando en ambientes distribuidos, se eligió EJB 3.0 para su implementación porque esta tecnología ha mostrado ser tanto robusta como actual para aplicaciones Enterprise escalables.

EJB 3.0 [5] es un Framework ofreciendo un conjunto de bibliotecas las cuales orientan la definición de la arquitectura puesto que para aprovecharlo de manera óptima, debe correr en un servidor de aplicaciones que implemente la especificación correspondiente de SUN.

Una arquitectura típica construida con EJB 3.0 muestra una capa de presentación en la que se ubican servletes, JSP y JSF; una capa de lógica de negocio donde se ubica SessionBeans y una capa de persistencia donde se encuentra EntityBeans.

Esta arquitectura no resulta compatible con una arquitectura DDD típica, ni con la arquitectura que propone el producto. La incompatibilidad fundamental se basa en que en EJB 3.0 no existe una capa de dominio aislada donde resida el Modelo de manera independiente a la infraestructura y/o aplicación.

Lo que EJB 3 llama Modelo es una capa de Persistencia que se encuentra por sobre la capa de Base de Datos. En esta capa se ubican los *Entity Beans*, esto es los objetos de dominio con atributos y métodos de manipulación de los mismos aunque sin lógica de negocio. Toda la lógica de negocio se ubica en la capa de Lógica de Negocio, es decir, en donde se encuentran los *Session Beans*.

Por estas razones, el conocimiento del dominio se encuentra interesando dos capas

separadas y, en el caso de la capa de Persistencia, mezclada con aspectos de infraestructura y, por ende, de tecnología.
Evans, en su propuesta DDD no acepta este tipo de arquitecturas. Es más, declara que atentan contra la programación orientada a objetos, y a favor de la programación estructurada, debido a la separación de los datos y los algoritmos que los manipulan.
Otra cuestión a tener en cuenta, en la propuesta DDD es que cuando se elige utilizar un framework, se debe respetar la filosofía esencial del mismo. No hacer esto podría llevar fácilmente a un producto de baja calidad en cuanto a mantenibilidad e inteligilibilidad.
La solución que propone este trabajo es establecer la convención de que en el modelo se ubicarán los Entity Beans así como los Session Beans. Cada entidad que deba ser persistida estará compuesta por un Entity Bean conteniendo los atributos y un Session Bean que implemente la lógica de negocio para manipular tales atributos.
Además, cada objeto de valor de DDD que deba ser persistido estará compuesto por un Entity Bean conteniendo los atributos y un Session Bean implementando la lógica de negocio para manipular dichos atributos. Por último, en la capa de dominio se ubicarán los servicios. Los servicios se implementarán como Session Beans.
Resumiendo:

- Entity = EntityBean + SessionBean + Interface

- ValueObject = EntityBean + SessionBean + Interface

- Service = SessionBean + Interface

A partir de EJB 3.0, tanto el mapeo ORM [6] como la configuración de los Session Beans pueden hacerse mediante anotaciones imbuidas en el código.
Sin embargo, las anotaciones resultan naturales en una arquitectura EJB 3.0, pero no en una arquitectura DDD pues agregar anotaciones en una clase resulta en una dependencia de tecnología.
Dado que la capa de Dominio no puede tener absolutamente ninguna dependencia con el framework de persistencia o la plataforma EJB3, se ha tomado la decisión de utilizar descriptores XML (orm.xml y ejb-jar.xml) para que los POJOs [7] que pasarán a ser Entity Beans o Session Beans no tuvieran ninguna dependencia de tecnología.
De esta manera se podría desarrollar un paquete y usarlo en cualquier aplicación (no necesariamente una aplicación Java EE o EJB 3.0). Éste es el concepto de Dominio Portable.

8. Conclusiones

Una propuesta de diseño como DDD puede no ser totalmente compatible con las herramientas y tecnologías elegidas para el desarrollo de un proyecto. Si se quieren mantener en el mismo, resulta necesario resolver las inconsistencias o contradicciones que puedan surgir.
El caso práctico es una muestra de posibles caminos de solución al problema planteado.
En el diseño, los objetos del modelo estarán compuestos por dos clases y una interfaz. Una de las clases será un EJB del tipo *Entity*, donde no se escribirá lógica de negocio; el *Entity* EJB solamente guardará los atributos que se quieran persistir y sus correspondientes primitivas. La otra clase será un EJB *Session Bean*, que contendrá la lógica de negocio y hará uso de los datos del EJB *Entity*. La interfaz expondrá los servicios que podrán ser invocados remotamente desde el *Session Bean* y será, por supuesto, la interfaz que implemente la lógica de negocio.
Bajo ningún concepto se permite separar en distintos paquetes los datos de un objeto de dominio de su lógica de negocio. Si existen dos clases para representar un objeto del modelo en el código, las dos deben estar contenidas en el mismo paquete y siguiente la convención de los puntos anteriores.
No se deben usar anotaciones que pertenezcan a la aplicación, en ninguna clase de la capa de

dominio. Si deben incluirse parámetros de configuración o mapeo ORM para el motor de persistencia, se especificarán mediante el uso de archivos de configuración externos como XML o properties. Fuera de la capa de dominio, podrán utilizarse las anotaciones, que resultan más cómodas para el desarrollo.
La capa de dominio debe construirse para ser completamente independiente de la aplicación que la utiliza. No debe existir ninguna dependencia de la capa de dominio hacia cualquier otra capa. La capa de dominio podrá ejecutarse fuera de la aplicación, ya que será la representación del modelo y la fuente de mayor valor para el negocio.

9. Estudios futuros

En el marco de esta actividad de alta interacción docente – alumno, se pretende seguir estudiando los avances en DDD y su relación con SOA y aplicaciones distribuidas en general, de manera de aplicar los resultados en casos prácticos.
También, se pretende analizar la compatibilidad entre herramientas de desarrollo presentes en el mercado con los lineamientos de las nuevas tendencias teóricas que se propongan en relación con desarrollo de aplicaciones distribuidas, en particular diseño e implementación.

10. Referencias

1. Hohpe, G. Woolf, B. “Enterprise integration patterns, designing, building and deploying messaging solutions”, The Addison Wesley signature series.
2. Evans, E., “Domain-Driven Design: Tackling Complexity in the Heart of Software”, Eric Evans, Addison Wesley, 2003.
3. Fowler, M. “Analysis Patterns: Reusable Object Models”, Addison-Wesley, 1997.
4. Meyer, B, “Object-Oriented Software Construction”, Prentice Hall, 1997.
5. http://java.sun.com/products/ejb/
6. http://madgeek.com/Articles/ORMapping/EN/mapping.htm
7. http://www.martinfowler.com/bliki/POJO.html