Domain Driven Design
(Diseño dirigido por el dominio)

Kelly Stefanía Tobón Montoya.
Juan Guillermo Hernández Alarcón.
Oscar Darío Botero Vargas.

Diciembre 2020

Universidad de Antioquia
Ingeniería de sistemas
Arquitectura de software

¿Qué es modelo de dominio?

Dividamos los conceptos.

Modelo

Trata de cómo se va a plantear la solución a un problema, normalmente, es mediante la abstracción de la realidad. “Un modelo es una simplificación, es una interpretación de la realidad que abstrae los aspectos relevantes para la solución del problema en cuestión e ignora detalles superfluos, por tanto, esa área temática a la que el usuario aplica el programa es el dominio del software. Para crear software que está valiosamente involucrado en actividades de los usuarios, un equipo de desarrollo debe poseer altos conocimientos relacionados con esas actividades, los modelos son herramientas que sirven para recopilar toda esa información, así pues, un modelo es una forma simplificada, selectiva y conscientemente estructurada de conocimientos, también se puede decir que un modelo adecuado le da sentido a la información y la enfoca en un problema”.

Dominio

Trata de lo que se va a resolver, es decir el negocio y sus reglas, sus procesos y de entender cómo opera la compañía. Esto se puede descomponer en subdominios, ejemplo: área contable, área de recursos humanos, proveedores, entre otros subdominios.
¿Se puede construir un software bancario robusto sin los conocimientos necesarios? La respuesta es no, debes conocer “el dominio”, es decir el negocio, sus reglas y procesos.

Entonces, **el modelo de dominio** son todas las partes con sus atributos, métodos y relaciones que surgen entre los mismos, con el fin de representar los conceptos claves del dominio del problema. Un modelo de dominio no es un diagrama en particular; es la idea que el diagrama pretende transmitir

¿Cómo se hace sin DDD?

Es bueno pensar en las técnicas que actualmente tienen mayor impacto positivo en el mundo del desarrollo de software y la arquitectura de software, pero, es también oportuno mencionar los disparadores de acogidas a nuevas técnicas (no tan nueva) como el Domain Driven Design o diseño guiado por el dominio.

Anteriormente, y aún en la actualidad, hay empresas, desarrolladores freelance y otros que no se preocupan por este tema de la buena y limpia arquitectura del software, en algunos casos porque no saben el daño que hacen o porque creen que eso es algo insignificante.

Sin DDD, entonces, se suele trabajar pensando en dar la solución al problema bruto, es decir sin tener en cuenta más factores de los que se ven en la superficialidad. Podríamos pensar que está bien construir algo como el siguiente ejemplo: una plataforma que permite comprar libros y/o realizar reseñas de estos. Observe que los usuarios pueden comprar libros o realizar reseñas de estos. Para comprar el libro harán falta ciertos datos especiales. Entonces, el usuario y su definición tendrán sentido con ciertos campos, puede ser: nombre y email, para realizar la reseña; sin embargo, la definición del usuario que incluye “tarjeta de crédito” para las compras no tiene mucho sentido en Reseña.

El anterior es un ejemplo que no está fuera de la realidad de muchas aplicaciones, sin embargo, eso no implica que duren poco, implica que cuesta que ese software evolucione a través de sucesivas iteraciones del diseño.

¿Qué es DDD?

"El diseño guiado por el dominio es un enfoque de diseño de software que enlaza el modelado de dominio y el diseño del software, con el objetivo de crear un modelo del dominio que evolucione a través de sucesivas iteraciones del diseño".

En otras palabras, el diseño dirigido por el dominio consta de darle un enfoque claro y preciso al dominio, es decir a las reglas que posee el negocio, sus restricciones, sus procesos y tener una visión amplia de ellas para poder así tener una solución concreta al problema.

El diseño guiado por el dominio tiene las siguientes premisas:

1. Para la mayoría de los proyectos de software, el enfoque principal debe estar en la lógica del dominio y en el dominio, ya que poseer un modelo, el cual exprese fielmente los requerimientos del cliente, hace que todo el equipo de desarrollo entienda fácilmente el problema y, por ende, detecte rápidamente la solución; mejorando así la productividad.

2. Diseños de dominio complejo deben basarse en un modelo, es decir que para modelar software que posea un gran rango de acción o dominio grande, es fundamental usar un modelo como guía de todo el proceso de desarrollo, pues de esta forma se segmenta adecuadamente el dominio.

Según Eric Evans, escritor del libro Domain-Driven Design: Tackling Complexity in the Heart of Software, para llevar a cabo los postulados del DDD se debe usar alguna metodología de desarrollo ágil, y los desarrolladores y expertos del dominio deben trabajar juntos.

Para que la comunicación entre las partes desarrolladores y expertos del dominio se debe definir un lenguaje ubicuo, es un lenguaje común entre ellos para referirse al modelo.

Principales conceptos de DDD

Entities: son objetos con identidad, es decir tiene un ID único. Tienen la capacidad de ser buscadas, almacenadas y recuperadas. Y tiene una alta cohesión en su definición, es decir, podemos decir que tiene una única responsabilidad.

Value Objects: Un objeto que representa un aspecto descriptivo del dominio sin identidad conceptual se denomina Value Objects. No tienen identidad. Características:

- Inmutabilidad: si queremos nuevos valores, no podemos modificar sus atributos, sino crear uno nuevo.
- No tiene identidad, entonces son objetos que pueden ser creados y descartados en cualquier momento.
- A nivel implementación, no se debe poder alterar el estado interno y todos los métodos públicos deben comportarse como creacionales, retornando un nuevo objeto con los datos modificados.

- La igualdad entre objectos va a estar dada por el valor de sus atributos.

Services: son usados para almacenar lógica que no pertenece ni a una entidad ni a un value object. Sirven de orquestadores de varias entidades que colaboran entre si, y de lógica que no pertenece a las propias entidades.

El nombre del servicio debe representar la acción a realizar, y solo debe tener una intensión de cambio (Single Responsibility Principle nuevamente).

Se pueden asociar o agrupar varios servicios, implementando el patrón Façade.

Ocurre que, al utilizar estos servicios, estemos programando con interfaces de estos y sus implementaciones se encuentre en la capa de infraestructura. Esta diferenciación suele darse así:

Los servicios que solo representan lógica de negocio e interactúan con objetos del dominio, se deben encontrar en el core domain.

Los servicios que interactúan con entidades externas deben estar en la capa de infraestructura, como, por ejemplo: EmailSender, SmsSender, ReporterPrinter.

Factories: Puede ocurrir que la construcción de Entidades y Agregaciones sea demasiado compleja. Y en realidad, lo que nos interesa es la utilización y no tanto la fabricación de este: es como cuando utilizamos cualquier electrodoméstico, solamente lo usamos y delegamos la fabricación en algún señor que se dedique a eso.

Ocurre lo mismo con las Entidades y Agregaciones, internamente puede ocurrir que sean estructuras complejas, y conocer estructuras internas para su fabricación sería

violar el principio de encapsulamiento. Por ende, vamos a delegar la fabricación en un objeto Factory.

Este objeto, tendrá la responsabilidad de fabricar objetos complejos, centralizando el conocimiento de la fabricación (y que no quede desperdigada por todo nuestro código), mejorando la calidad de este, generando código testeable.

Repositories: Las entidades desconocen completamente su forma de persistirse, no tienen ese conocimiento ni tampoco le interesa: solo debe interesarse por cumplir las reglas de negocio.

Entidades van al repositorio. No habrá acoplamiento con un motor de bases de datos.

Lenguaje Ubicuo: La comunicación efectiva entre los desarrolladores y los expertos del dominio es esencial para el proyecto. Un lenguaje común, que sea representado en el dominio, tanto como en los bounded contexts es muy importante para evitar tener problemas futuros y desarrollar un software exitoso, donde la comunicación sea el pilar para su obtención. En resumen, es hablar en el código como hablamos en el negocio.

Bounded context: ayuda a definir límites. Busca la autonomía real de las partes de la aplicación. Es un espacio delimitado donde un elemento del negocio tiene un significado perfectamente definido. Podemos indicar que el equivalente a los

subdominios es lo que se llama Bounded Context, mientras que el equivalente a Dominio es el modelo de dominio (esto para el DDD).

Beneficios e inconvenientes del DDD

Beneficios	Inconvenientes
Software con fuertes lazos con el dominio	Necesidad del experto de dominio
En ocasiones permite el testing de las distintas partes del dominio de manera aislada	Aprender a implementar buenas prácticas, patrones y nuevos procedimientos en el desarrollo, implica tiempo.
Lógica de negocio bien enfocada y dividida por contextos.	No se recomienda para un simple CRUD
A largo plazo es mantenible	Es costoso a nivel de tiempo determinar el modelo de dominio con un experto y el equipo de desarrollo
Se enfoca en el dominio y los subdominios a través del Bounded Contexts	
Información de primera mano. Los expertos y los desarrolladores se comunican directamente	

Lista de referencias

Evans, E., 2003. Domain-Driven Design Reference.