Resumo do livro "Domain-Driven Design"

Leonardo Leite

25 de julho de 2016

Atenção: este documento trata-se ainda de uma versão em desenvolvimento.

1 Introdução

Este documento é um resumo do livro "Domain-Driven Design, Atacando as Complexidades no Coração do Software", escrito por Eric Evans em 2003. Os objetivos deste resumo são:

- Em primeiro lugar, fazer um exercício de fixação e assimilação de minha leitura do livro.
- Servir de referência rápida para que eu tente aplicar os conceitos do DDD no desenvolvimento de software.
- Fornecer a outras pessoas uma ideia rápida sobre o que trata o livro Domain-Driven Design, assim como apresentar alguns de seus principais conceitos.

Uma consideração importante ao leitor: embora minha prática do desenvolvimento de software já tenha vários pontos de acordo com o DDD, este resumo foi escrito logo após minha leitura do livro. Ou seja, não se trata de um texto baseado em minha experiência com DDD, mas somente na leitura do livro.

Destaco também que o livro que eu li foi a primeira edição da tradução para o português (2009). Aliás, uma tradução não tão boa. O pior exemplo de tradução que achei foi a tradução de *graphs* (grafos) e *edges* (arestas) respectivamente por gráficos e bordas. Não sei dizer se a tradução em si foi melhorada na segunda edição da tradução (2011).

Minha impressão geral sobre o livro: gostei bastante da mentalidade do autor, bem condizente com princípios da Programação Extrema (XP) e ao mesmo tempo com uma boa dose de pragmatismo. No entanto o livro é um tanto quanto grande e verboso. Embora seja definitivamente um clássico, já é relativamente antigo (2003). Dessa forma, acho que talvez possam existir por aí livros mais resumidos que possam transmitir a essência do DDD sem tantas considerações e mais direto ao ponto considerando o atual estado-da-arte do desenvolvimento de software. Não que o livro tenha deixado de ser atual, mas algumas considerações menores poderiam hoje ser deixadas de lado. Ao mesmo tempo, já surgiram alguns paradigmas modernosos como *micro-serviços*, que são bem relacionados com o tema do livro. Portanto, sua decisão de ler ou não o livro vai depender bastante do quanto você deseja se aprofundar no assunto.

Outra consideração bem interessante é, pra mim, foi bem positivo ter demorado pra ler esse livro. Já tinha ouvido falar do livro há alguns anos, mas somente agora finalmente

o priorizei. E isso foi bom porque a experiência recente que eu tive desenvolvendo um sistema com um complicado modelo de domínio me ajudou bastante a ler o livro com melhor proveito. Outros sistemas em que trabalhara antes não tinham um modelo de domínio assim tão complexo. Ler o livro e comparar o que o autor descreve com suas próprias experiências é de grande valor para uma melhor leitura do livro. Dessa forma, dou a ousada sugestão de que se você ainda é um desenvolvedor iniciante leia apenas um livro básico (ou mesmo esse resumo) sobre o assunto e espere alguns anos até ler o livro Domain-Driven Design.

"A principal finalidade de um software é servir aos usuários. Mas, primeiro, esse mesmo software tem que servir aos desenvolvedores".

Obs: *"trechos nesta formatação são citações diretas do livro".*

Bom, chega de enrolação. Vamos ao que interessa...

2 O domínio de um software e seu modelo

Um modelo é uma simplificação da realidade. O modelo do domínio de um sistema é a forma como entendemos a realidade de um dado negócio. Esse modelo é algo abstrato, que tem impacto em diversos artefatos concretos, como requisitos, diagramas, o código-fonte e a estrutura do banco de dados. Todos esses artefatos devem estar alinhados com o modelo do domínio. E parte desse alinhamento diz respeito às palavras utilizadas.

* Normalmente o refinamento do modelo, do design e da implementação devem caminhar de mãos dadas em um processo de desenvolvimento iterativo.

* Normalmente à medida que o modelo está sendo refinado para suportar melhor o design, ele também deve ser refinado para refletir uma nova visão com relação ao domínio.

O modelo de domínio (expresso também no design) é a base do sistema.

"Um modelo profundo proporciona uma expressão lúcida das principais preocupações dos especialistas do domínio e de seu conhecimento mais relevante, descartando, ao mesmo tempo, os aspectos superficiais do domínio".

Conceitos presentes em versões iniciais do modelo podem ser descartadas conforme o modelo amadurece.

3 A linguagem onipresente

Evans defende extensivamente ao longo do livro a utilização de uma *linguagem onipresente* para falar do modelo e para se aplicar nos artefatos concretos. Assim, ocorre uma melhor assimilação dos desenvolvedores sobre o que dizem os especilistas dos negócios. Assim, fica evidente desvios que a arquitetura esteja tomando em relação ao negócio. A utilização de uma linguagem onipresente é uma ferramenta realmente poderosa.

Um dos lugares em que a linguagem onipresente deve estar presente são os diagramas do sistema. O autor enfatiza o como sessões em que desenvolvedores e analistas de negócios se juntam para rabiscar diagramas pode ser proveitosa. Nessa toada, uma coisa que o autor valoriza são os diagramas informais. Se prender às normas restritas da UML pode não ser o ideal para passar uma certa mensagem. Afinal, um bom diagrama é sobre comunicar sucintamente algo à equipe. Do meu ponto de vista, acho aborrecedor com as diversas ferramentas UML por aí que inserem restrições diversas (ex: homens palitos só podem ser inseridos em diagramas de caso de uso).

4 Camadas de um software

As camadas de um software geralmente se resumem a interface, aplicativo, domínio e infra.

Interface:

Aplicativo: não contém regras de negócio. Tende a ser bem magra. Essa camada utiliza a camada do domínio.

Domínio: é o coração do sistema, onde estão as regras do negócio. É a parte do sistema onde menos os *frameworks* e plataformas podem ajudar. É onde mais a criatividade e as capacidades de análise e design se fazem necessárias. E é nessa camada que o livro Domain-Driven Design é focado.

Infra: ecursos técnicos como mensagens, persistência etc.

Muitas pessoas exaltam muito a importância da infra-estrutura na definição de arquitetura de um software. Alguns acham que a escolha do Sistem Gerenciador de Bancos de Dados é a mais importante. Algumas outras se degladiam pela escolha do *framework* de desenvolvimento. Mas o Uncle Bob é bem feliz ao explicar como essas coisas na realidade importam muto pouco[1]. Assim, entendemos que o núcleo do sistema é composto pelas regras de negócio. Essas regras tendem a ser mais duradouras que as tecnologias empregadas. E são essas regras que, no fim, das contas, entregam valor ao negócio. Por isso a importância do estudo do DDD, que consiste no aprimoramento de nossa capacidade em melhor desenvolver essa cadama do domínio.

Uncle bob também esclare como a camada de domínio não deve depender da camada de infra. O negócio não pode depender da tecnologia. Isso fica evidente na apresentação da arquitetura hexagonal[2].

5 Elementos do domínio

Os elementos do domínio classificados por Evans são: entidades, objetos de valor e serviços.

Entidades são identificáveis e possuem um ciclo de vida.

Objetos de valor são intercambiáveis (não possuem identidade) e devem ser imutáveis.

Serviços acomodam operações que não cabem nas entidades. É preciso ter cuidado, porque existem serviços nas camadas de domínio, infra e aplicativo.

Módulos , também chamados de pacotes (no java), são agrupamentos de classes. Devem possuir significado no domínio, ou seja, a hierarquia de módulos deve refletir uma hierarquia do domínio.

* Separar pacotes por conceitos do domínio, e não pelas camadas técnicas. Ex ruim: entidades, valores, serviços. Ex bom: cliente, cobrança, transporte de cargas.

Pg 107 crítica à separação entity vs BC... modelo anêmico... objects vs data structures

[1] http://blog.cleancoder.com/uncle-bob/2016/01/04/ALittleArchitecture.html
[2] https://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html

6 Padrões do ciclo de vida de um objeto do domínio

agregados: possui uma entidade e vários objetos d valor. Membros externos ao agregado acessam a entidade e não podem modificar diretamente os objetos d valor. Entidade garante as invariantes do grupo. Ex: carro vs penu. Ex: laudo vs proprietario, veiculo vistoriado....

fábricas: A função da fábrica é instanciar um objeto potencialmente complexo a partir dos dados. Fábrica não cuida de persistência. A fábrica pode instanciar um novo objeto ou reconstituir um objeto já existente a partir de seu formato serializado.

repositórios: faz o CRUD do objeto. Interface do repositório é ligado ao modelo, e não à infra. O repositório usa um DAO para ter acesso a infra. Ou seja, no repositório ficam as regras de negócio de CRUD da entidade, não os mecanismos tecnológicos de persistência. Esses últimos ficam no DAO, que é da camada de infra.

Então um cenário típico de uso, para a criaçaõ de um novo objeto, fica assim:

1. Cliente pede pra fábrica criar objeto.
2. Cliente pede pra inserir objeto no repositório.
3. Repositório pede pro DAO persitir o objeto.

Já para a reconstituição de um objeto, podemos ter:

1. Cliente pede objeto para repositório.
2. Repositório acessa o banco via DAO para obter dados persistidos do objeto.
3. Repositório repassa dados persistidos do objeto a fábrica.
4. Fábrica reconsrói objeto, que é devolvido ao repositório, que é devolvido ao cliente para a edição.

A complexidade desses passos decorre do caso mais complicado no qual a reconstituição do objeto não se resume simplesmente a restaurar os dados persistidos no banco. O que é o banco guarda, pode ser apenas parte do que é objeto é em tempo de execução. Esse é um dos motivos pelo qual é muito arriscado permitir que o sistema de uma aplicação tenha acesso direto ao banco de dados de outra aplicação. Por mais que impor algumas restrições no modelo de dados do banco possa ser uma ideia em caráter de garantia extra, essas regras podem não dar conta de manter complexas restrições semânticas decorrentes do domínio.

Detalhe importante: mesmo a interface que o DAO fornece não deve conter detalhes da tecnologia. Assim, tudo o que o domínio sabe é que a entidade está sendo persistida em algum lugar, mas sem nenhuma dependência com alguma tecnologia específica. Alguns programadores leval isso mais a sério ao criar interfaces para os DAOs, fazerem o domínio depender da interface, e aí criar implementações acopladas ao banco, mas que são desconehcidas do domínio. Essa complicação faz sentido, mas eu acredito que esse trabalho possa ser evitado enquanto se mantenha primeira ideia do parágrafo: a interface do DAO (i.e. nome de classe e assinaturas dos métodos) não pode conter detalhes específicos de tecnologia.

7 Evolução do modelo do domínio

Refatoração constante.

Evoluções passo a passo são as vezes interrompidas por uma oportunidade de avanço onde as coisas mudam bastante de uma só vez. São as oportunidades de avanço:

* Busca por um modelo mais profundo.

* Evitar regras de casos particulares.

* As vezes há uma oportunidade de avanço para um modelo mais profundo que requer uma grande refatoração, daquelas que é difícil concluir sem deixar o código quebrado por um tempo. Mas vale a pena.

* Refatoração técnica (melhora o design, mas sem alterar o modelo do domínio) vs refatoração para refinamento do modelo. A primeira é algo mecânico, a segunda é mais subjetiva.

8 Outras dicas de design

* Associações bidirecionais (e referências circulares) são problemáticas para o design (mas as vezes necessárias).

* A direção da travessia muitas vezes capta uma visão aprofundada do domínio, aprofundando o próprio modelo.

* Explicite os conceitos implícitos.

* É preciso muita iteração e conversa com os especialistas do domínio. Rabiscar diagramas no quadro branco ajuda bastante.

* Acertar o design de primeira não existe.

* Ler livros sobre o domínio pode ajudar também.

* Outros conceitos além de substantivos/verbos para fazerem parte do modelo: restrições, processos de negócios e especificações.

Uso de especificações...

* Interface reveladora de intenções. Diminuir esforço cognitivo do desenvolvedor. Se a mente do desenvolvedor está transbordando de detalhes internos sobre o objeto utilizado, sua mente não está limpa para resolver o problema. Se é preciso considerar a implementação de um componente para utilizá-lo, o valor do encapsulamento é perdido. Utilizar bons nomes é um caminho para resolver esses problemas. TDD (testes de unidade antes da implementação) também ajuda.

* Funções (devolvem valores) vs comandos (têm efeitos colaterais). Um método deve ser ou uma função ou um comando, não misture os dois. Prefira funções. Comandos devem ser bem simples. Cálculos complexos ficam melhor em objetos de valor (não em entidades).

* Se esforce para reduzir as dependência entre classes e módulos (diminuir o acoplamento). Se possível, crie classes autônomas, que podem ser estudadas por si só. Isso diminui o esforço cognitivo em compreender/usar/testar os elementos do domínio. Obs: dependências visíveis na interface são piores que dependências internas; dependências com elementos fora do módulo são piores que dependência com elementos dentro do módulo.

* Se possível, crie operação *fechadas* sobre elas mesmas. Ou seja, a operação retorna um objeto do mesmo tipo que o objeto que contém a operação (x.inverse()). Também ajuda: retorno do mesmo tipo que o argumento.

* O autor vê com certa reserva abordagens como design declarativos, sistemas baseados em regras e linguagens declarativas. São interessantes, mas há limitações. O autor

encoraja a utilização dos padrões apresentados ao longo do capítulo para que o código de objetos tenha um estilo mais declarativo. Essa discussão é bem parecido com a questão se fazer BDD com ou sem a linguagem de alto nível, mais próxima da linguagem natural.

* Como estilo declarativo o autor dá alguns exemplos de combinações de especificações usando operadores lógicos. Assim chegamos a códigos como "Spec both = ventilated.and(armored)", onde ventilated e armored são Spec. No geral, creio que padrões de "linguagens fluentes"ajudem nesse propósito. Em geral esses padrões são construídos que retornam o próprio objeto invocado (ex: design pattern Builder). Pode-se até chegar ao ponto de verificar se uma spec contém outra spec: manSpec.inclui(mortalSpec).

Depois de uma discussão sobre alterar o modelo, pode ser legal esperar alguns dias pra discutir novamente, com a ideia mais madura, antes de de fato por a mão na massa e alterar o design ("dormir com o problema"ajuda).

Refatore quando: * O design não expressa o entendimento atual da equipe sobre o domínio. * Conceitos importantes estão implícitos no design. * Você vê uma oportunidade de tornar mais flexível alguma parte importante do design.

9 Arquitetura de larga escala

10 Objetos não são estruturas de dados

Modelo anêmico (post caelumn)