



Guilherme Moreira

(guilherme.moreira@caelum.com.br): Coordenador e instrutor da Caelum em Brasília, é um dos autores do livro "Arquitetura e Design de Software – Uma visão sobre a plataforma Java", trabalhando com Java desde 2005, com projetos em geral na parte web tanto para a Caelum quanto para os clientes da mesma. Possui as certificações SCJP e SCEA 5 e é responsável por alguns projetos open-source da Caelum, além de desenvolver cursos novos e material didático.

Guilherme Prado

(guilherme.prado@caelum.com.br): Formado como Técnico em Processamento de Dados pelo CTI/UNESP. Instrutor da Caelum em Brasília e Técnico de Informática do Conselho da Justiça Federal. Trabalha com desenvolvimento de software há 9 anos atuando principalmente com tecnologias Java, ASP, PHP e VB.

Domain-Driven Design

Além dos Patterns

O que se pode aprender do Livro de Eric Evans?

Domain-Driven Design é, como o próprio nome diz, observar o Design da aplicação pensando no domínio. O objetivo deste artigo é discutir algumas ideias citadas no livro de Eric Evans a respeito de como tratar o domínio de forma coerente entre a equipe técnica e a equipe que entende do negócio.

Desde o lançamento do livro Domain-Driven Design (de Eric Evans), a comunidade de desenvolvedores ganhou uma arma poderosíssima que reforça uma maneira já (muito) discutida de como desenvolver software com qualidade: as conversas entre equipe de desenvolvimento e quem entende do negócio.

Como se viu em recente post do Philip Calçado (link nas referências), podemos conferir que a busca por patterns do livro de Eric Evans (por exemplo repository, anticorruption layer) é muito maior que a busca por seus conceitos principais (por exemplo, Ubiquitous Language). Por que tanta procura por patterns? Onde se encaixa a Ubiquitous Language?

O objetivo deste artigo é discutir temas como linguagem ubíqua, design patterns e outras ideias do livro de Eric Evans, também avaliando um pouco das práticas do mercado em relação ao livro, além de claro a parte fundamental do livro, um pouco sobre a modelagem de domínio. No começo do artigo revisaremos alguns pontos fundamentais dos conceitos de DDD, passando pelas discussões sobre software versus conhecimento, juntando logo em seguida ao conceito de ubiquitous language, mostrando os pontos fortes e fracos dessa prática. Na segunda parte do artigo é discutido o papel de alguns Design Patterns em relação ao DDD e seus conceitos

Revisando o conceito de Domain-Driven Design

Softwares são feitos para melhorar ou ajudar algum processo que existe no mundo real. Se seguirmos essa linha de raciocínio, podemos pensar que os softwares partem de algo que já existe. Alguns softwares serão uma representação e outros vão auxiliar algo já existente. O conhecimento sobre o mundo real deve estar embutido no software. No processo de criação de um sistema temos dois papéis fundamentais: aquele que conhece o domínio muito bem e aquele que conhece a parte técnica muito bem. Esses dois papéis não podem de maneira nenhuma trabalhar separadamente. Quem entende do domínio precisa da equipe técnica para conseguir expressar seu conhecimento através de software e a equipe técnica precisa da equipe que entende do negócio para poder escrever um software útil e de acordo com a necessidade.

Onde entra o Domain-Driven Design nisso?

O Domain-Driven Design é um conjunto de princípios que auxiliam o desenvolvimento de software voltado para o domínio, ou seja, voltado para o processo que existe no mundo real, não ignorando o fato que esse domínio será expresso através de código. Domain-Driven Design prega que para grande maioria dos projetos o foco deve ser o domínio da aplicação e nas lógicas envolvidas nele e quando o domínio for complexo a base do projeto deve ser um modelo baseado no domínio. Esse foco ajuda projetos de domínio complexos a acelerarem o andamento do projeto, mantendo uma organização e uma base de discussão.

Conhecimento

Toda a base do Domain-Driven Design gira em torno do domínio e como desenvolver sendo ele a parte principal da aplicação. O domínio é o que dá valor à aplicação; é área de conhecimento, atividade ou influência na qual o software é aplicado. Este domínio não precisa, nem deve, ser representado com total fidelidade, ele deve ser refinado até chegar em um modelo que atenda às necessidades do cliente e que também seja implementável, isso é declaradamente uma tarefa árdua de se completar. Pode-se definir modelo como uma abstração que descreve aspectos selecionados do domínio que são relevantes à resolução de problemas relacionados a ele. O principal na hora de trabalhar com Domain-Driven Design é manter o foco no domínio, porém se este domínio é representado através de um modelo a importância do DDD é vista ao criar ao modelar o domínio. Nesta modelagem, o ponto fundamental é destilar o conhecimento, os domain-experts, aqueles que entendem do negócio, vão sentar junto com a equipe técnica, arquiteto, analista, programadores (às vezes nem todos os programadores) e montar um modelo que atenda às necessidades técnicas, que seja implementável e que esteja de acordo com o negócio.

O modelo

O modelo não precisa ser um diagrama UML, ele pode ser qualquer tipo de artefato que reproduza o conhecimento gerado pelas equipes durante as conversas, ele pode ser uma documentação escrita, um desenho ou um diagrama feito a mão com anotações sobre o que cada parte significa e como elas se juntam. A qualidade do **modelo** depende muito das duas equipes. Depende da equipe de domains-experts saber transmitir o conhecimento de forma clara e compreensível e depende da equipe técnica que dará veracidade técnica para ele. Os domains-experts devem entender as

necessidades técnicas do projeto e a equipe técnica deve se esforçar para entender do negócio. Sem esse esforço de ambas as partes não sairá um modelo útil. Esse modelo sofrerá diversas mudanças com o passar do tempo, pois ele não será perfeito na primeira tentativa e nem nas próximas. As equipes devem estar aptas às mudanças, tanto de regra, quanto de entendimento sobre o sistema. O modelo de qualidade deve ser algo maleável, que aceite evoluções e regressões caso seja necessário.

As duas equipes vão conseguir atingir um modelo de qualidade através de muita conversa, transmissão de conhecimento, nas duas vias. Mas não é qualquer conversa que será aproveitável. Uma conversa produtiva será aquela cujo resultado seja algo integrável ao código, sem nenhum intermediário. Os termos utilizados durante as reuniões devem ser amplamente aceitos, usados e entendidos, e a linguagem que os dois times usam deve ser a mesma.

A Ubiquitous Language

Não só nas conversas essa linguagem deve estar presente, como os termos empregados deverão aparecer também no código, nos e-mails trocados, nas reuniões e em possíveis documentações. Essa linguagem deve ser única e presente em todos os lugares. Nisso temos o conceito de **linguagem ubíqua** ou em inglês **ubiquitous language**. Em um cenário em que a linguagem é única, temos pontos muito fortes, como manutenção. Caso haja alguma mudança em alguma regra (e elas existirão) quando ela for discutida, a mudança será refletida nas conversas e quase mais importante ainda, refletirá no modelo. Consequentemente, o código deverá ser refatorado para atender a este novo modelo. Mas esse ganho não vem sem trabalho: o cenário descrito é a realidade de uma equipe madura e experiente com esse raciocínio.

Dificuldades ao usar a Ubiquitous Language

O maior impedimento da linguagem ubíqua é a diferença entre as duas equipes em relação aos termos empregados. É comum as equipes acabarem desistindo de usar essa linguagem única, pois uma equipe não entende os termos usados pela outra, ou na tentativa de usar a linguagem, acabarem criando traduções para os termos. Essas traduções são um erro gravíssimo, é perder o sentido da linguagem única. Essas traduções geram erros de interpretação que serão refletidos no código. A linguagem ubíqua deve ser consistente e livre de ambiguidades. Realmente, não adianta ter a equipe usando os mesmos termos quando cada pessoa entende o termo de maneira diferente (evitar ambiguidade), assim como apenas atrapalharia o andamento do projeto um conceito sendo representado por dois termos diferentes (evitar inconsistência).

Distância entre o modelo e o código

Ainda analisando mais sobre a linguagem ubíqua, outro erro comum cometido por causa de dificuldades no caminho é que mesmo que as conversas, os termos, tudo seja baseado no modelo, o código não seja uma representação próxima do modelo. Mesmo o modelo sendo útil para discutir novas funcionalidades, logo que o código começar a ser escrito ele ficará defasado e inconsistente, ele perderá sua principal função: ser a ponte entre o conhecimento das equipes e o código. Um modelo que não representa o código, segundo Evans é inútil.

• Encarando Design Patterns através dos conceitos

A linguagem ubíqua deve ser baseada no conhecimento gerado pelas equipes, trabalhada, até chegar em um modelo útil e de qualidade. Esse modelo deve ser legível pelas duas equipes.

Juntando a ideia de encapsulamento da orientação a objetos com este modelo, resulta que, ao olhar para o modelo, o programador não deve precisar ver sua implementação para descobrir o que ele faz, a sua função deve estar bem clara. Esta é a ideia do Supple Pattern **“Intention-revealing Interfaces”**, interfaces que revelam suas intenções sem olhar para a implementação. Falar sobre encapsulamento hoje em dia é uma ideia batida para a maioria dos desenvolvedores, mas podemos falar em dois níveis de encapsulamento: por código e por leitura. O encapsulamento por código seria mais o código que chama um método não depender da implementação dele, mas o desenvolvedor ainda sim seria obrigado a conhecer a implementação para saber o que fazer, e isso praticamente não é encapsulamento. **Intention-revealing Interfaces** está mais voltado para a leitura do código, nada mais prazeroso que olhar para uma interface e entendê-la sem nunca se preocupar com sua implementação. Esse é um ótimo exemplo de linguagem ubíqua pensada junto com código.

Durante o livro, Evans cita diversos exemplos de bons usos para a linguagem ubíqua, direta ou indiretamente. Tanto na Parte III (**Refactoring toward deeper Insight**), como na Parte II (**The Building Blocks of Model-Driven Design**). Mesmo quando são listados alguns Design Patterns, Evans sempre preza o Design do modelo com a implementação.

• Mais um exemplo de Pattern visto através da modelagem do negócio

Acredito que a palavra repository tenha um sentido mais comum para as pessoas que falam inglês, pois em português a palavra repositório não faz parte do vocabulário da maioria das pessoas (inclusive do meu), então como explicar para um domain-expert por que no modelo temos ContatoRepository? Não precisa, não importa o nome dado à classe, desde que através desse nome seja possível enxergar que aquela parte do sistema é responsável por guardar certos tipos de coisas (objetos) e recuperá-los quando necessário. Esses são os repositórios. Neles usamos termos conhecidos pela linguagem ubíqua e no modelo, não importa quais sejam, se precisarmos guardar alguma coisa é para isso que os repositórios aparecerão. Muitas vezes os repositórios serão apenas interfaces (caso você esteja trabalhando com Java), outras serão classes que delegam o trabalho para outras classes.

```
interface AgendaDeContatos{
    List<Contato> buscarContatosQueComecaCom(String começo);
    List<Contato> buscarContatosQueContenha(String trecho);
    Contato buscarContatoAtravesDeld(Long id);
}
```

Assim como o livro Design Patterns, o livro Domain-Driven Design também tem feito o mesmo sucesso, mas parece que ambos são lidos e tratados muitas vezes apenas como um catálogo de patterns. No livro do Gang of Four, onde os dois capítulos iniciais apresentam conceitos fundamentais até mais importantes que os próprios patterns, o DDD também deve ter seus conceitos muito bem explorados e os patterns devem ser enxergados como exemplificação destes.

Os patterns são consequências dos conceitos. DDD não é um catálogo de patterns, tanto que alguns patterns citados por Evans já existiam muito antes do livro (alguns deles no livro Patterns of Enterprise Application Architecture de Martin Fowler). Os patterns devem surgir com naturalidade.

Inclusive no livro Evans evita-se chamar Repository e Domain-Model de Patterns para evitar esteriótipos, porém os dois são padrões documentados por Fowler.

• Um pouco sobre DDD em larga escala

Em quais tipos de sistema que o Domain-Driven Design é vantajoso? Essa é uma pergunta bem difícil de responder. Em geral, sistemas onde o objetivo não é apenas guardar e mostrar dados, mas que envolvam algumas regras mais complexas e domínios ricos poderão tirar mais conselhos do livro. Porém, todo sistema pode tirar vantagem das práticas do DDD, nem que seja apenas como começar a conversa com o cliente e manter um modelo implementável. Aliás, todas as equipes de desenvolvimento podem tirar lições do livro, é um livro importante na vida dos desenvolvedores.

Nem todo sistema precisa ser colossal para se começar a pensar em DDD. Na verdade, para chegar a um sistema colossal, deveríamos começar com um sistema mínimo e expandi-lo para onde for necessário, em etapas (ou iterações, dependendo da metodologia que estiver trabalhando).

Depois que o sistema estiver maior, os tipos de problemas apresentados serão outros, e surgirão principalmente problemas de comunicação entre as equipes e os desenvolvedores. Nesse caso, provavelmente, muitos modelos estarão em jogo e é inevitável que este tipo de problema apareça.

Trabalhando com diversos modelos, pode-se ter cada parte da equipe trabalhando em cada um deles. Onde começa e termina o trabalho de uma equipe sem que atrase a outra? Não há uma fórmula mágica para evitar isso, o que pode ser feito é deixar bem claro e delimitado até onde cada um desses modelos vai trabalhar. Essa delimitação é o que no livro é chamado de bounded contexts (contextos delimitados).

O conceito de bounded contexts fica em um capítulo do livro chamado Maintaining Model Integrity, e serve para quando o sistema fica muito grande, porém o assunto se encaixa também no capítulo Large-Scale Structure. O capítulo Large-Scale Structure é mais focado em discutir ideias e padrões para sistemas muito grandes que vão precisar ir além de uma estrutura apenas maleável. Por exemplo, comenta-se a ideia de manter uma arquitetura que possa evoluir junto do sistema. Evans chama esse conceito de Evolving Order. A discussão neste capítulo gira em torno da função de uma arquitetura de software.

Por um lado, podemos ter uma arquitetura completamente restritiva, que obrigue o modelo a sofrer sérias consequências, como acabar por criar um modelo anêmico, sem inteligência nenhuma (ver referências do artigo), e que não fique livre para evoluir e acatar os novos insights (novos entendimentos sobre o domínio) que surgirem durante a produção do sistema, mas que resolva diversas questões técnicas como persistência, mensageria e rede. Porém, por outro lado, se tivermos uma arquitetura muito aberta onde cada desenvolvedor tem a chance de escolher coisas muito diferentes do resto do projeto, quanto mais o sistema crescer, mais o desenvolvimento ficará completamente desordenado e de baixa manutenibilidade, com diversas soluções para as mesmas questões espalhadas pelo código.

Um ótimo exemplo de arquitetura restritiva é o J2EE 1.4. Ao criar um session

bean era obrigatório implementar diversos métodos que não faziam o menor sentido para qualquer negócio.

```
public class VendaBean implements SessionBean{
    public void vender(Produto produto){
        //
    }
    //métodos de origem exclusivamente técnica
    public void setSessionContext(SessionContext ctx){}
    public void ejbActivate(){}
    public void ejbPassivate(){}
    public void ejbCreate(){}
    public void ejbRemove(){}
}
```

Uma proposta em **Evolving Order** é praticar uma arquitetura que não seja fixa, que ela possa evoluir com as necessidades que forem surgindo no caminho, mesmo que ela tenha que mudar completamente para se encaixar melhor no sistema. O arquiteto e o gerente do sistema devem estar aptos a conversar com a equipe e ponderar quando mudar a arquitetura e quando deixar que o modelo sofra as consequências. Percebam que nem sempre mudar a arquitetura é a melhor solução; essa é uma das principais funções dos arquitetos: ponderar os **trade-off's** que as mudanças causarão.

Não é fácil tomar decisão arquiteturais que sirvam para toda a vida do projeto. Por isso, a ideia de evoluir a arquitetura conforme a necessidade. Para que isso seja possível, é necessário que principalmente o domínio esteja desacoplado de qualquer questão técnica ou da comunicação com o usuário. Esse isolamento do domínio, aliado a separação de questões de infraestrutura e de comunicação com o usuário é outro conceito fundamental do DDD, a arquitetura de camadas ou a **Layered Architecture**.

Layered Architecture

Uma boa prática aplicada há muito tempo no Java é a divisão do sistema em camadas, por responsabilidade, aqui cabe lembrar do MVC, que é a divisão entre os objetos da camada do modelo, camada da visualização e camada de controle. Na DDD a arquitetura proposta muda um pouquinho do MVC, embora conceitualmente as duas divisões eles sejam bem parecidos, algumas questões técnicas fazem com que elas sejam diferentes.

A divisão proposta por Evans foca muito no conceito fundamental do DDD, o domínio. Essa arquitetura de camadas sugere isolar o domínio de qualquer outra camada, deixá-lo o mais puro possível, já que o próprio domínio já será suficientemente complexo para também ter que lidar com outras questões, como técnica ou da camada visual. A camada de domínio é onde o modelo é representado, as outras camadas são auxiliar desta. Não pense que só porque essas camadas são auxiliares elas não são importantes, sem elas a aplicação não rodaria, um modelo sem uma infraestrutura por baixo ou uma camada de apresentação para o usuário não faria sentido.

Ao todo o sistema fica dividido em quatro partes: **User Interface** (interface com o usuário), **Application** (Aplicação), **Domain** (Domínio) e **Infrastructure** (infraestrutura) ver figura 1.

A camada de interface com o usuário fica responsável, assim como no MVC, de se comunicar com o usuário, tanto para receber comando como mostrar resultados, o usuário não precisa necessariamente ser uma pessoa, pode até ser outro sistema (web-services, motores de indexação etc.).

Na camada aplicação ficam as operações de coordenação da aplicação e suporte a outras tarefas técnicas. Aqui é **disparada** a lógica de negócios que está em outra camada, ou é chamada alguma tarefa técnica de banco de dados, como o início de uma transação e seu respectivo final. Qualquer dado que a camada de domínio precisar para iniciar a operação é de responsabilidade da aplicação buscá-lo. A palavra-chave para entender a camada de aplicação é **quando** ela sabe quando disparar as coisas, mas não sabe como.

A camada de domínio é o coração do software, é aqui que está toda a regra discutida nas conversas, aqui que o modelo é representado em código usando a linguagem ubíqua. É esta camada que dá valor à aplicação, ela interage com todas as camadas da aplicação, ela recebe chamadas da camada de aplicação, devolve valores para a camada de interface com o usuário (esses valores podem passar pela camada de aplicação) e dispara requisições para a camada de infraestrutura.

Toda parte técnica do sistema, banco de dados, bibliotecas, e-mail, frameworks, renderização de componentes fazem parte da infraestrutura da aplicação. Esses detalhes técnicos ficam contidos na camada de infraestrutura, ela serve para isolar o resto da aplicação desses detalhes. Um caso interessante na camada de infraestrutura é que mesmo os componentes desenvolvidos internamente na empresa ficam nessa camada, permitindo uma maior reusabilidade dos componentes.

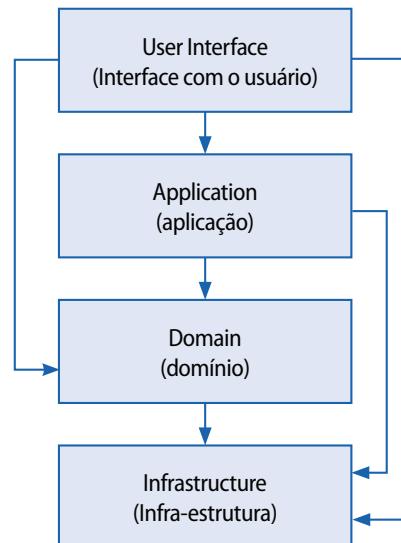


Figura 1. Divisão em camadas do sistema.

A importância da arquitetura de camadas não está só na reusabilidade dos componentes ou na opção de ter uma arquitetura que evolua junto da aplicação, sua importância é devido à manutenibilidade que ela oferece, é muito mais simples procurar um bug de lógica de negócios quando o domínio está isolado, do que procurar esse bug no meio de uma monte de código de tela junto com o código de negócios.

Assim como a maioria das arquiteturas de camada para que ela seja efetiva as camadas devem seguir algumas premissas: estar completamente desassociada uma da outra se comunicando sempre através de interfaces e a camada inferior nunca disparar operações na camada superior, imagine uma busca no banco de dados disparando operações de negócios, em um sistema muito grande achar quem dispara essa operação seria extremamente complexo.

Conclusão

Você não deixará de usar Domain-Driven Design se seu código não tiver a palavra repositório ou anti-corruption layer. Patterns mudam com o tempo; hoje eles são boas práticas, amanhã talvez nem tanto. O conceito principal DDD não são os patterns ou usar ou não todos os building blocks e sim pensar no domínio para chegar em um modelo rico de conhecimento e implementável, através de muita comunicação.

Não é por isso que devemos deixar de aprender design patterns. Eles são uma ótima fonte de ideias de como resolver problemas comuns do dia-a-dia, com o aval de muita gente que já usou aquela solução. Porém, os patterns não devem ser encarados como mais do que isso, mas sim, como artifícios que permitem que o modelo de domínio seja implementado de forma mais eficiente, assim como as outras camadas do estilo de arquitetura proposta por Eric Evans. **MU**

Para saber mais

Não deixe de conferir mais sobre os conceitos deste artigo no artigo do Sérgio Lopes na edição número 31 da **MundoJ**, no artigo "Criando Software mais Próximo do Cliente com Domain-Driven Design".

Referências

- Post no Blog do Philip Calçado (em inglês):
<http://fragmental.tw/2010/03/22/nevermind-domain-driven-design/>
- Post no Blog da Caelum sobre Design Patterns:
<http://blog.caelum.com.br/2006/12/17/design-patterns-um-mau-sinal/>
- Post no Blog de Martin Fowler sobre Modelo Anêmico (em inglês): <http://martinfowler.com/bliki/AnemicDomainModel.html>
- Eric Evans – *Domain-Driven Design: Tackling Complexity in the Heart of Software*
- Abel Avram e Floyd Marinescu – *Domain-Driven Design Quickly*
- Erich Gamma – *Design Patterns: Elements of Reusable Object-Oriented Software*

INSCREVA-SE JÁ

Palestrantes

Rebecca Wirfs-Brock
Wirfs-Brock Associates, USA

Fábio Kon
Universidade de São Paulo

Joe Yoder
U. Illinois / The Refactory, Inc, USA

realização



apoio



Realizado em conjunto com o
I Congresso Brasileiro de
Software: Teoria e Prática - CBSoft 2010
<http://cbsoft.dcc.ufba.br>

VIII Conferência
Latino-Americana
em Linguagens de Padrões
para Programação

De 23 a 26 de Setembro
Salvador - Bahia

<http://sugarloafplop.dcc.ufba.br/>

SugarLoafPLoP'2010