Notas sobre o livro "Domain-Driven Design"

Leonardo Leite

15 de agosto de 2016

Resumo

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" [Evans, 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".

Antes de chegar ao que interessa, mais uma consideração. A abordagem de DDD, tal como apresentada no livro, é intimamente ligada à orientação a objetos (OO). Caso você ainda não domine a OO em profundamente, não se preocupe, estudar DDD é até uma boa maneira de reforçar o aprendizado em OO. Mas um alerta se faz necessário. Muitas pessoas ainda vendem a ideia de que o foco principal da OO consiste no mapeamento de entidades do mundo real para o código e a utilização de encapsulamento, herança e polimorfismo. Embora todas essas coisas estejam inclusas na OO, é mais útil considerar que o foco do OO é a decomposição do código de um sistema em unidades coesas e fracamente acopladas. Essas unidade são tanto as pequena unidades (classes), quanto as grandes unidades (pacotes). Uma consequência almejada dessa decomposição é que a manutenção do sistema fica facilitada, pois é possível encontrar rapidamente o ponto a ser alterado e a mesma alteração não precisa ser repetida em diversos pontos do código. Um ótimo livro que ajuda a passar essa ideia sobre a OO é o "Orientação a Objetos e SOLID para Ninjas" [Aniche, 2015].

Obs: *"trechos nesta formatação são citações diretas do livro"*. Conceitos chaves do livro em *itálico*. Não há separação rígida entre oq o livro diz e oq eu digo, mas maioria das afirmativas são baseadas no livro. Explicitar as duas coisas quebraria a fluidez do texto. Este não é um texto acadêmico.

Obs: citações não são tão e somente pra justificar sentenças, mas mais pra indicar mais leituras ao leitor...

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

2 O modelo do domínio de um software

O *domínio* de um software é o assunto relacionado às atividades desempenhadas pelos usuários desse software. Exemplos de domínios são compra de passagens aéreas, gestão de pessoas, tradução de texto para outras línguas, o processo legislativo, segurança veicular, previsão do tempo etc.

Para desenvolvermos um software que auxilie as atividades humanas em um dado domínio, precisamos de um *modelo* desse domínio que seja útil à criação desse software. *"Cada modelo representa algum aspecto da realidade com uma ideia que seja de interesse. Um modelo é uma simplificação. Ele é uma interpretação da realidade que destaca os aspectos relevantes para resolver o problema que se tem em mãos ignorando os detalhes estranhos"*. Ao ignorar seletivamente esses detalhes, modelos nos ajudam a tratar a sobrecarga de volume e complexidade das informações relacionadas ao domínio.

Diagramas não são o modelo. O modelo é uma abstração que se reflete em diversos artefatos concretos, como requisitos, diagramas, código-fonte e estrutura do banco de dados. Todos esses artefatos devem estar alinhados com o modelo do domínio. Esse alinhamento

diz respeito até mesmo às palavras utilizadas na comunicação entre os membros do time (incluindo desenvolvedores, analistas de negócio e cliente).

O modelo do domínio não pode ser facilmente definido antes da construção do software. O processo de desenvolvimento traz um grande aprendizado para a equipe sobre o domínio. Esse aprendizado deve se refletir em evoluções incrementais do modelo ao longo do desenvolvimento. E enquanto o modelo evolui, os demais artefatos (diagramas, código etc.) também devem evoluir para acompanhar a evolução do modelo.

3 A linguagem onipresente

A situação típica é de que o time de desenvolvimento não esteja habituada ao domínio do software a ser construído. Especialistas do domínio possuem jargões que os desenvolvedores não conhecem. Assim surgem complicações na comunicação entre desenvolvedores e especialistas do domínio. Até mesmo entre os próprios desenvolvedores pode haver confusão, pois cada um pode criar um entendimento diferente sobre o domínio em sua cabeça.

Para fortalecer a comunicação dos envolvidos em um projeto de software e garantir a precisão do modelo, Evans defende extensivamente ao longo do livro a utilização de uma *linguagem onipresente*. Essa linguagem onipresente deve ser utilizada na comunicação oral e escrita do time, além de ser utilizada nos diagramas, código-fonte etc. Assim, a linguagem onipresente faz a ligação entre a comunicação do time e a implementação do software.

Lacunas na linguagem onipresente evidenciam a falta de um conceito no modelo. Palavras estranhas podem evidenciar imprecisões do modelo ou divergências entre a implementação e os objetivos do negócio. Mudanças na linguagem onipresente são mudanças no modelo. Assim ela deve ser mantida e zelada por todo o time. Especialistas do domínio devem impedir a existência de termos inapropriados para o domínio, enquanto que desenvolvedores devem impedir ambiguidades e inconsistências.

Na comunicação diária, tanto especialistas quando desenvolvedores ainda utilizarão termos fora da linguagem onipresente. Esses termos podem ser, por exemplo, termos técnicos ligados à implementação do software, ou conceitos mais avançados do negócio, não necessários à implementação do software. Tais termos devem ser complementares à linguagem onipresente, e não conflitantes com a linguagem onipresente.

Embora seja rápido e fácil falar sobre a linguagem onipresente, ela é uma ideia central ao longo do livro. Em praticamente todos os capítulos, o autor aproveita para reforçar o assunto, aplicando-o nos mais diferentes contextos, seja na produção de código, seja na conversa com os especialistas.

4 Camadas de um software

O primeiro passo para um design dirigido pelo domínio é o isolamento da lógica do negócio de outras características que o software deve implementar. Isso é feito com a separação do software em camadas. Assim temos uma camada do domínio que fica desacoplada do restante do sistema. Isso facilita a manutenção, pois podemos criar designs mais coesivos e identificar rapidamente os pontos de mudança no software. Além disso, o isolamento das regras de negócio potencializa o reuso de código, pois a mesma regra de negócio pode ser reaproveitada em um aplicativo móvel ou por um web service, por exemplo.

As camadas podem ser classificadas em interface, aplicativo, domínio e infra.

Interface: interage com o usuário, coletando entradas e exibindo saídas. Esse usuário pode ser tanto uma pessoa quanto outro sistema.

Aplicativo: implementa as funcionalidades disponíveis para o usuário. Essa camada tende a ser bem magra e apenas dirige a camada de domínio para que a funcionalidade seja entregue.

Domínio: é o coração do sistema, onde estão os conceitos, regras e estado 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: recursos técnicos como persistência, troca de mensagens etc. É aqui que ficam os chamados DAOs (Data Access Objects), que fazem o mapeamento entre objetos e registros de um banco de dados.

"O princípio essencial é de que qualquer elemento de uma camada depende somente dos outros elementos da mesma camada ou dos elementos das camadas "abaixo" dela. A comunicação para cima deve passar por algum tipo de mecanismo indireto". Ou seja, o importante aqui é que a camada de domínio seja auto-contida e não dependa de nenhuma das outras camadas do software.

A aplicação do princípio da inversão de dependência para possibilitar que a camada de domínio não dependa de outras camadas é descrito nesses dois posts do Uncle Bob: "The Clean Architecture"[1] e "A Little Architecture"[2]. Citando um trechinho: *"A web é um detalhe. O banco é um detalhe. Nós deixamos essas coisas do lado de fora, onde elas não podem trazer muito dano."*

Muitas pessoas exaltam 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 post "A Little Architecture" esclarece como essas coisas na realidade importam muto pouco. As regras de negócio tendem a ser mais duradouras que as tecnologias empregadas. E queremos ser capazes de facilmente substitui tecnologias obsoletas se preciso. Entendemos que as regras de negócio são a parte mais valiosa do sistema. Por isso a importância do estudo do DDD, que consiste no aprimoramento de nossa capacidade em escrever a camada do domínio.

5 Elementos do domínio

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

Entidades representam algo com identidade e com uma continuidade rastreável por diferentes estados e até diferentes implementações. A definição de um objeto como esse não está em seus valores, pois objetos diferentes com valores iguais ainda devem

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

ser distinguíveis. A definição da identidade é a principal característica de uma entidade. Um exemplo de entidade é um veículo que deve ser identificável em diferentes partes do sistema como correspondente a um determinado veículo único existente no mundo real. Dois veículos Fox, da Volkswagen, ambos de cor preta e demais características de fábrica idênticas ainda são veículos distintos.

Objetos de valor descrevem o estado de alguma coisa. Por exemplo, um veículo pode ter uma cor, que é descrita por uma combinação numérica RGB. Dois veículos podem ter cores idênticas, mas não precisamos nos preocupar com que os dois objetos veículos apontem exatamente para o mesmo objeto cor. Ou seja, nesse caso a cor não possui uma identidade. Ela é meramente um valor. Para reforçar: enquanto uma entidade define *quem* ela é, um objeto de valor define *o que* ele é. Objetos de valor devem ser imutáveis. Assim, um veículo pode mudar de cor, mas não é a cor do veículo que se altera. Ou seja, o objeto veículo passa a apontar para um novo objeto cor e o antigo objeto cor é esquecido. A imutabilidade traz várias vantagens ao design de software [Bloch, 2008b], e podemos aproveitar essas vantagens nos objetos de valor.

Serviços são operações oferecidas como interfaces, isoladas no modelo e que não possuem estado. Acomodam operações que possuem significado no domínio, mas que não se encaixam em entidades ou objetos de valor. Serviços devem ser criados com parcimônia, pois a utilização de serviços deixa o software menos orientado a objeto e mais procedural. Mas às vezes são convenientes para que regras complicadas de orquestração dos elementos do domínio não vazem para a camada de aplicação. Um exemplo seria a transferência de dinheiro entre contas bancárias, caso a operação "transferência" não se encaixe adequadamente no objeto "conta". Serviços existem em todas as camadas do software. Um exemplo de serviço de infra é o envio de e-mail. Mas é preciso ter os serviços de cada camada bem caracterizados e separados.

Módulos, chamados de pacotes na linguagem Java, são agrupamentos de classes. Devem possuir significado no domínio. A hierarquia de módulos deve refletir uma hierarquia do domínio. O nome de um módulo deve ser expressivo e pertencer à linguagem onipresente. A coesão e acoplamento dos módulos também deve ser controlada. A separação dos módulos deve isolar conceitos para facilitar o entendimento dos mesmos. A refatoração da estrutura de módulos é mais complicada do que a refatoração de classes, mas não menos importante.

6 Entidades e suas regras de negócio

Uma prática adotada por muitos desenvolvedores é a separação das definições de dados e comportamentos de uma entidade em objetos separados. Em algumas nomenclaturas seriam as *entities* e os *BCs* (*business classes*). A página 107 do livro inicia uma discussão importante ao criticar essa separação. Quando possível, deixe o comportamento de uma entidade no mesmo objeto que define seus dados. Assim, Evans sugere que as regras de negócio que ficam isoladas nos BCs passem a integrar as entidades. Além disso, regras do BC que sejam na verdade regras que manipulem o ciclo de vida da entidade (criação, alteração e deleção) podem se acomodar melhor em fábricas ou repositórios (ver próxima seção).

Essa separação de entidade com dados de um lado e BCs com regras de negócio do outro gera o que é chamado por aí de modelo anêmico. Alguns textos de rápida leitura que corroboram com o desencorajamento do modelo anêmico: "O que é Modelo Anêmico? E por que fugir dele?"[3] e "AnemicDomainModel"[4].

Mas a adoção dessa prática deve possuir alguns cuidados, como não criar acoplamentos indesejados. Exemplo: *empresa.possuiFuncionarioProcessado()* parece uma interessante adesão a um modelo não anêmico. Mas se a princípio o modelo não exige que o objeto "empresa" tenha conhecimento de objetos de "processos", esse método pode não ser uma boa ideia. Nesse caso, melhor talvez seria algo como *processoRepositorio.possuiFuncionarioProcessado(empresa)*.

Contudo, ainda há controvérsias... No livro Clean Code, Uncle Bob faz uma distinção entre objetos e estruturas de dados [Martin, 2008]. Ele considera que no padrão *active record*[5], os modelos (classes que fazem a ponte com o banco de dados para a persistência das entidades) devem ser simples estruturas de dados. Ele critica a colocação de regras de negócio nessas estruturas de dados e termina concluindo que *"a solução é, claro, tratar o Active Record como uma estrutura de dados e criar objetos separados que contenham as regras de negócio e que escondam seus dados (que são provavelmente apenas instâncias do Active Record)"*. Alguns trechos a mais desse capítulo do Clean Code podem ser encontrados no post "Clean Code: objetos não são estruturas de dados!"[6]

Essa solução no fundo vai de encontro à recomendação geral de não expor publicamente todos os dados de uma entidade. Mas forçar essa separação da entidade em classes diferentes parece um tanto burocrático e cair no caso criticado por Evans, no qual há uma separação da entidade conceitual em diferentes objetos, sendo um para manter a estrutura de dados e o outro as regras de negócio da entidade.

Para concluir a controvérsia, minha posição neste momento é de que a postura mais equilibrada parece ser implementar suas entidades como classes contendo os dados da entidade nos atributos e as regras de negócio em métodos públicos, mas evitando a publicização desnecessária dos dados da entidade. Em Java isso seria não criar *getters* e *setters* desnecessariamente. Uma posição similar pode ser depreendida do post "Como não aprender Java e Orientação a Objetos: getters e setters"[7]. Mas essa é uma discussão pra lá de complicada!

7 Padrões do ciclo de vida de um objeto do domínio

Nesta seção apresentamos os padrões que auxiliam no controle do ciclo de vida de objetos: como eles nascem, são persistidos, recuperados, excluídos e como a integridade dos objetos e de seus relacionamentos é mantida.

Agregados possuem uma entidade raiz e vários objetos de valor. Um exemplo é uma entidade carro que possui quatro objetos pneus. Embora os pneus sejam objetos separados, o conjunto do carro mais os quatro objetos pneus formam uma unidade coesa que sempre permanece junta, à o que chamamos de agregado. Como no exemplo os pneus não tem razão de ser fora do carro, esses pneus podem ser meros

[3] `http://blog.caelum.com.br/o-que-e-modelo-anemico-e-por-que-fugir-dele/`
[4] `http://www.martinfowler.com/bliki/AnemicDomainModel.html`
[5] `https://pt.wikipedia.org/wiki/Active_record`
[6] `http://polignu.org/artigo/clean-code-objetos-n%C3%A3o-s%C3%A3o-estruturas-de-dados`
[7] `http://blog.caelum.com.br/nao-aprender-oo-getters-e-setters/`

objetos de valor. Membros externos ao agregado acessam somente a entidade raiz e não podem modificar diretamente os objetos de valor. Assim, a entidade raiz garante as invariantes do grupo. No nosso exemplo, uma invariante do agregado a ser observada é a existência de exatamente quatro pneus. Outro exemplo de agregado: uma empresa pode ser uma entidade. Mas para a completa descrição da empresa podemos necessitar de um objeto endereço, que seria um objeto de valor fazendo parte do agregado da empresa.

Fábricas encapsulam processos complexos de criação de objetos a partir de um conjunto de dados. A fábrica não cuida de persistência. A fábrica pode instanciar um novo objeto ou reconstituir um objeto já existente a partir de seu formato serializado. É comum que a fábrica crie todo um agregado completo. Existem diversos padrões de projetos sobre como criar fábricas. O padrão *builder* é um exemplo bem legal [Bloch, 2008c]. Algumas vantagens da utilização de fábricas podem ser encontradas no livro Effective Java [Bloch, 2008a].

Repositórios oferecem a abstração de coleção de um conjunto de objetos ou agregados do mesmo tipo. Assim, clientes[8] podem inserir, editar, excluir e consultar objetos de uma determinada coleção. Repositórios devem ser construídos apenas para as raízes dos agregados. As consultas podem ser feitas na medida para evitar o vazamento de encapsulamento de detalhes para o cliente, o que seria mais propenso a acontecer caso o cliente acessasse diretamente o banco de dados (via DAO). Condições de negócio para a inserção/alteração/exclusão do objeto devem ser garantidas pelo repositório. A implementação do repositório é uma implementação de negócio, não devendo se acoplar com a tecnologia de persistência (ex: banco de dados). Por isso, em última instância, o repositório delega uma tarefa ao DAO. Exemplo: uma empresa com alguma irregularidade não pode ser inserida na coleção de empresas participantes de uma licitação. Essa é uma regra do repositório, e não do DAO. O DAO deve se preocupar somente com a tecnologia da infraestrutura.

Utilizando os padrões apresentados, eis um cenário típico de criação de um novo objeto:

1. Cliente pede à fábrica para criar um objeto.
2. Cliente recebe o objeto criado pela fábrica.
3. Cliente pede ao repositório para salvar objeto criado.
4. Repositório pede ao DAO para persistir dados do objeto no banco de dados.

Já para a recuperação de um objeto podemos ter:

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

[8]Quando um objeto A utiliza outro objeto B, dizemos que o objeto A é o *cliente* dessa relação.

A complexidade desses passos vale a pena no caso mais complicado, no qual a reconstituição do objeto não se resume simplesmente a restaurar os dados persistidos no banco. O que o banco guarda pode ser apenas parte do que o objeto é em tempo de execução. Esse é um dos motivos pelo qual é muito arriscado permitir que um sistema tenha acesso direto ao banco de dados de outro sistema. Impor restrições no modelo de dados do banco pode nos dar algumas garantias. Porém, essas regras no banco podem não ser suficientes para manter complexas restrições semânticas decorrentes de regras do domínio.

O DAO é o objeto focado na tecnologia de banco de dados, mas isso é um aspecto apenas de sua implementação. A interface do DAO não deve conter detalhes da tecnologia. Tudo o que o domínio sabe é que o objeto está sendo persistido em algum lugar. O domínio permanece totalmente ignorante quanto a tecnologia utilizada. Ou seja, a interface do DAO deve ser definida do ponto de vista do negócio. Lembre-se: o domínio não deve depender de detalhes (o banco é um detalhe). São são os detalhes (DAOs) que devem depender do domínio. Idealmente temos então a interface do DAO sendo definida dentro do domínio e sua implementação na camada de infra. Em Java podemos separar esses dois artefatos (interface e implementação) nessas diferentes camadas. Mas como na maioria dos casos o DAO sempre vai ter uma única implementação, uma abordagem mais pragmática é fazer uma única classe DAO na camada de infra, sem o artefato de interface. Contudo, as assinaturas dos métodos e o nome do DAO continuam sendo sua interface, e essa interface continua devendo ser definida em termos do domínio.

8 Evolução do modelo do domínio

O modelo do domínio não nasce pronto. Conforme os desenvolvedores avançam na implementação, eles aprendem e assimilam muito conhecimento sobre o domínio. E esse conhecimento deve ser utilizado para incrementar o domínio. Assim, Evans defende constantemente a refatoração contínua do software. Não só a refatoração técnica que melhora o design das classes, mas a refatoração para que o modelo melhor expresse conceitos e soluções no domínio.

Por vezes os desenvolvedores podem notar indícios de que o modelo deve ser refinado. Algumas situações: pessoas diferentes utilizam nomes diferentes para a mesma coisa (talvez conceitos devam ser uniformizados); a descrição de algum relacionamento é muito complicada (talvez esteja faltando um novo conceito); novos conceitos que surgem não se encaixam com um dos conceitos já estabelecidos (talvez algum conceito velho deva ser descartado); percebe-se que o design não expressa o entendimento atual da equipe sobre o domínio (conceitos do software não coincidem com os conceitos mentais das pessoas).

Um bom momento para detectar refinamentos necessários ao modelo é durante sessões de discussão entre desenvolvedores e pessoas do negócio quando ambos podem interagir por meio de diagramas rabiscados em uma lousa. Nesse sentido, Evans valoriza bastante a utilização de diagramas informais. O ponto central sobre diagramas é comunicação. A UML é um bom ponto de partida, mas se prender às suas regras restritas traz mais prejuízo que benefício. Eu particularmente me aborreço com ferramentas que restringem como você pode desenhar seus diagramas só para não desrespeitar a UML.

Pequenas evoluções incrementais são as vezes interrompidas por uma *oportunidade de avanço*, que leva a um *modelo mais profundo* por meio de alterações abruptas que se propagam por várias partes do sistema. Embora esse tipo de refatoração seja complicada, um modelo mais profundo pode trazer vantagens ao aumentar sua versatilidade e seu poder explicativo.

Um modelo mais profundo consegue resolver situações particulares com regras gerais. Quando um sistema tem muito código dedicado a casos particulares, isso é um sinal de que o modelo não tem força suficiente. Um modelo bem desenvolvido oferece ao software a propriedade da *emergência*. A emergência significa que comportamentos complexos benéficos emergem (de forma não antecipada) a partir de regras simples contidas nas unidades do sistema [Fried et al., 2006]. A busca por conceitos elegantes que evitem a utilização de regras complexas é um dos benefícios esperados das oportunidades de avanço. Porém, uma oportunidade de avanço não é algo planejado, mas um evento que ocorre de tempos em tempos. E é preciso estar atento para quando o momento chegar.

Mas antes de arregaçar as mangas e se aventurar em alterações radicais de seu design, mais um conselho. Após uma discussão sobre uma oportunidade de avanço, espere alguns dias e discuta novamente a proposta. Esse tempo ajuda a deixar a ideia mais madura e dar confiança ao time de que esse é o melhor caminho. "Dormir com o problema" ajuda.

9 Dicas de design

- Cada caso é um caso. O próprio autor em diversos momentos toma uma postura bem pragmática, incentivando que a equipe tome as decisões considerando o contexto e ponderando os compromissos.
- Associações bidirecionais e referências circulares são problemáticas para o design. Ás vezes são necessárias, mas tente evitá-las.
- A direção da associação muitas vezes capta uma visão aprofundada do domínio. Exemplo: é o veículo que possui as rodas, e não o contrário.
- 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 também pode ajudar.
- Outros conceitos além de substantivos e verbos (classes e métodos) podem fazer parte do modelo: restrições, processos de negócios e especificações.
- Objetos e agregados devem manter invariantes, que são *restrições* sobre seus atributos. Explicite as restrições extraindo a lógica de restrição para um método dedicado a isso. Há também casos para extrair a restrição para uma nova classe: a lógica de restrição é muito complexa; a lógica de restrição apresente dependências estranhas ao objeto original; a mesma restrição aparece em classes diferentes. Um exemplo de restrição modelada em uma classe especializada é uma `política de overbooking`, que é utilizada para manter uma restrição entre as associações de um objeto de `viagem` para vários objetos de `carga` considerando os atributos `viagem.capacidade` e `carga.tamanho`.
- Serviços do domínio podem encapsular procedimentos que façam sentido do ponto de vista do negócio. Além disso, existem *processos* mais complexos, no qual uma

determinada entidade passa por etapas de um longo processo para atingir um objetivo. Um exemplo comum é a emissão de documentos, que exige etapas como fornecimento de dados, pagamento e coleta do documento. Nesses casos, modelos baseados em máquinas de estado são úteis. Um padrão de projeto que pode ajudar é o *state* [Freeman et al., 2004].

- *Especificações* utilizam regras complexas para avaliar um objeto e retornam um booleano. Com regras simples poderíamos determinar se uma `fatura` está vencida com o método `fatura.vencida()`. Mas caso estas regras se tornem muito complexas ou criem dependências estranhas à classe `Fatura`, podemos extrair a lógica de avaliação do vencimento para uma nova classe. Para avaliar o vencimento da fatura teríamos então `especificacaoDeFatura.estaVencida(fatura)`, sendo que é `EspecificacaoDeFatura` que depende de `Fatura`, e não ao contrário. A especificação é um mero objeto de valor que é criado e utilizado quando necessário, sendo logo depois descartado.

- Para obter uma coleção de objetos que respeite uma determinada especificação, a princípio seria interessante desacoplar a especificação do repositório. Mas está aí um caso no qual o desempenho pode ser um fator decisivo para que o repositório conheça as regras da especificação, de forma a termos uma consulta no banco que já recupera somente os objetos de interesse.

- Quando aplicável, utilize *asserções* para garantir o estado do objeto após determinada operação. Na prática, essas asserções são implementadas nos testes de unidade.

- Use *interfaces reveladoras de intenções* para diminuir esforço cognitivo do desenvolvedor. Para utilizar um objeto não deve ser necessário entender sua implementação, mas apenas sua interface. Utilizar bons nomes é um caminho para resolver esses problemas. Desenvolvimento orientado por testes (TDD) também ajuda a definir interfaces mais claras e focadas no cliente.

- *Funções* devolvem valores, enquanto *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 em vez de entidades.

- Se possível, crie funções *fechadas* sobre elas mesmas. Ou seja, a função retorna um objeto do mesmo tipo que o objeto que contém a função. Exemplo: `x.inverse()` retorna um número real a partir de outro número real. Se o retorno for do mesmo tipo que o argumento da função também já ajuda. Exemplo: `Math.inverse(x)`.

- 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. Dependências na interface são piores que dependências internas; dependências com elementos fora do módulo são piores que dependência com elementos dentro do módulo.

- Evans 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 próprio código orientado a objetos tenha um estilo mais declarativo.

- Uma disputa similar sobre paradigmas exóticos ocorre sobre a utilização de linguagem natural para a especificação de testes de software. Alguns autores defendem a utilização da própria linguagem de propósito geral (ex: Java) com sentenças cuidadosamente criadas em um estilo mais declarativo para expressar os requisitos do software em testes automatizados [Freeman and Pryce, 2009].

- Especificações combinadas com operadores lógicos podem dar um estilo declarativo ao código. Sendo *Spec* uma especificação de um contêiner de transporte, podemos ter: `Spec both = ventilated.and(armored)`, sendo os objetos envolvidos do tipo `Spec`. Podemos também testar se `umaSpec.inclui(outraSpec)`.

- Um estilo declarativo bem popular hoje em dia é a chamada *interface fluente*[9]. Podemos encontrar um exemplo na biblioteca Mockito, utilizada para a criação de mocks em testes de unidade: `when(object.do()).thenReturn(someThing)`. O próprio padrão *builder* é um exemplo de interface fluente.

- Lembre-se: softwares não são somente para usuários. Eles também são para desenvolvedores, que devem manter o código.

10 Modelando sistemas de grande escala

Os capítulos finais do livro *Domain Driven Design* são voltados para a modelagem de grandes sistemas, que envolvem integrações diversas. As discussões apresentadas nesta seção podem não se aplicar a sistemas mais simples, mas tratam de uma realidade comum em grandes empresas. Não só para esse conteúdo, mas para qualquer recomendação de design, saiba avaliar o que se aplica à sua situação específica.

A unificação total do modelo do domínio para um sistema grande pode não ser factível ou econômica. Divide-se assim o modelo em *contextos delimitados*. Um contexto delimitado é algo maior que os módulos, correspondendo normalmente a sub-sistemas ou até a sistemas diferentes que possuem alguma relação entre si. A visão global da relação entre esses contextos delimitados é o *mapa do contexto*. Os nomes presentes no mapa do contexto devem fazer parte da linguagem onipresente. Essa delimitação entre contextos nos ajuda a evitar problemas como conceitos duplicados e falsos cognatos. A separação entre contextos pode ter relação com a divisão de trabalho entre equipes.

Há várias abordagens para tratar a passagem conceitos de um contexto para outro contexto. Essa passagem pode requerer a *tradução* de conceitos. Pode-se evitar a passagem de conceitos por meio da criação de um *núcleo compartilhado*, que é um contexto compartilhado por diferentes sub-sistemas. Muitas vezes o núcleo compartilhado corresponde ao *domínio principal* da aplicação.

Um caso comum é que dois contextos delimitados tenham a relação de cliente/fornecedor, na qual o fluxo de controle flui em apenas um sentido entre os contextos. O cliente pode adotar uma abordagem *conformista*, aceitando um contexto imposto pelo fornecedor sem o isolamento de uma camada de tradução. Caso opte-se pela tradução, mas o contexto fornecedor seja muito complicado e até mal desenhado, o contexto cliente pode tomar uma atitude defensiva utilizando uma *camada anticorrupção* para que o modelo do cliente mantenha-se limpo. A camada anticorrupção normalmente é implementada

[9] `http://martinfowler.com/bliki/FluentInterface.html`

como um conjunto de serviços. Padrões de projetos usados na camada anticorrupção são a *fachada* e *adaptadores*.

O fornecedor pode expor uma *linguagem publicada*, que é um contexto desenhado especialmente para ser utilizado por outros sistemas. Esse é o caso da exposição de APIs REST ou Web Services, bastante utilizados hoje em dia.

Mas a integração entre sistemas é sempre cara, e por isso as vezes é melhor manter *caminhos separados* e duplicar um pouco de código para manter as coisas simples e até facilitar a evolução de sistemas que, apesar de alguma semelhança, são independentes.

Subdomínios coesos que não são centrais para seu domínio principal e que podem servir de apoio para outros domínios podem ser segregados em *subdomínios genéricos*. Exemplo: princípios gerais de contabilidade, conversão de fuso-horários, etc. Não se preocupe antecipadamente com a reutilização do subdomínio genérico por outro sistema. Faça o mínimo possível para resolver o seu problema. Mantenha o domínio segregado sem dependências para seu domínio principal. Subdomínios genéricos ajudam a diminuir a sobrecarga no domínio principal. Ou seja, fica mais simples de entender o domínio principal. Subdomínios genéricos podem também ter sua implementação terceirizada mais facilmente do que partes do domínios principal.

Seja qual for a estratégia para gerenciar múltiplos contextos delimitados, a *integração contínua* ajuda a manter a harmonia entre os diferentes contextos. Testes automatizados executados continuamente em um ambiente neutro fortalecem a garantia de consistência entre os contextos. Assim, a integração contínua funciona como uma forma de comunicação eficiente para alertar caso alterações que uma equipe faça em um contexto tenha impactos negativos em algum contexto mantido por outra equipe.

Pode-se também dividir um sistema em *camadas*. A camada A utiliza a camada B, enquanto que a camada B nem conhece a camada A. O livro fornece um exemplo genérico de divisão em camadas que pode ser adotado em diferentes sistemas. A camada de *potencial* define políticas e especificações, que são as coisas como elas deveriam ser. Exemplo: especificação da rota de um navio cargueiro. A camada de *operações* corresponde ao estado das coisas como elas realmente estão. Exemplo: a rota já definida para um navio cargueiro. E a camada de *apoio a decisões* possui algoritmos inteligentes que usam como insumo as camadas de potencial e de operações. Exemplo: geração de relatórios sobre custos baseados nas rotas de uma frota de cargueiros.

TODO relação entre camadas e contextos?

O autor enfatiza que a estrutura em grande escala também deve evoluir conforme o entendimento sobre o domínio é refinado. A refatoração do modelo de grande escala é bem mais complicada, mas o autor julga ser um investimento necessário para que o modelo possa continuar evoluindo de forma saudável.

11 Conclusão

O desenvolvimento de software deve utilizar uma abordagem voltada para sistemas complexos. Ou seja, componentes fracamente acoplados interagem para que um comportamento alinhado com os objetivos do negócio possa emergir. O estudo do DDD nos orienta na modelagem do software seguindo a mentalidade de sistemas complexos, desacoplando partes do domínio e desacoplando o próprio domínio de outros detalhes do sistema.

Quando se fala em arquitetura de software, muitos logo pensam na topologia de máquinas servidores, sistema de virtualização, servidores http e de aplicação, *frameowkrs* e nas conexões com o banco de dados. Tudo isso faz parte da arquitetura do sistema. Mas

também faz parte da arquitetura o *modelo do domínio* e sua organização. Essa modelagem requer um profundo entendimento do domínio. Por isso, o conhecimento de negócio assimilado pelos desenvolvedores deve ser valorizado.

É comum que desenvolvedores mais talentosos sejam atraídos para o desenho da arquitetura de infraestrutura ou utilização de *frameworks*. Essas áreas costumam empolgar os desenvolvedores pois são áreas que apresentam oportunidades para a utilização de novas tecnologias. No entanto, transformar um domínio complexo em um design de software compreensível e útil é uma atividade complexa e extremamente desafiante. Por isso, a capacitação na modelagem e implementação do domínio merece toda a atenção durante o desenvolvimento de um sistema. Afinal, é o modelo do domínio o verdadeiro coração do software, aquilo que dá valor ao sistema e o torna único.

Referências

[Aniche, 2015] Aniche, M. (2015). *Orientação a Objetos e SOLID para Ninjas*. Casa do Código.

[Bloch, 2008a] Bloch, J. (2008a). Item 1: Consider static factory methods instead of constructors. In *Effective Java*, chapter 1 Creating and Destroying Objects. Addison-Wesley, 2nd edition.

[Bloch, 2008b] Bloch, J. (2008b). Item 15: Minimize mutability. In *Effective Java*, chapter 4 Classes and Interfaces. Addison-Wesley, 2nd edition.

[Bloch, 2008c] Bloch, J. (2008c). Item 2: Consider a builder when faced with many constructor parameters. In *Effective Java*, chapter 1 Creating and Destroying Objects. Addison-Wesley, 2nd edition.

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

[Freeman et al., 2004] Freeman, E., Robson, E., Bates, B., and Sierra, K. (2004). Chapter 10. The State Pattern: The State of Things. In *Head First Design Patterns*. O'Reilly Media.

[Freeman and Pryce, 2009] Freeman, S. and Pryce, N. (2009). *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley.

[Fried et al., 2006] Fried, J., Hansson, H. D., and Linderman, M. (2006). Lower your cost of change. In *Getting Real, The smarter, faster, easier way to build a successful web application*, chapter 3 Stay Lean. 37signals.

[Martin, 2008] Martin, R. C. (2008). Chapter 6. Objects and Data Structures. In *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall.