Práticas de código para devs eficientes

Alberto Souza

Conteúdo

Por que este ebook existe? **1**

Arquitetura x Design **2**

Um arquitetura para seu design **3**
Algumas inspirações já usadas para tentar suportar um bom design 3
Fazer software manutenível ainda é um desafio a ser vencido 4
Por que softwares ainda ficam muito complicados de serem mantidos? 5
Parece que as técnicas de design não estão sendo suficientes 6
Uma outra forma de lidar com a complexidade inerente do código 7
É necessário que você fique de mente aberta 7

Cognitive-Driven development **8**
Resumo da ideia 8
Introdução 8
Trabalhos relacionados 10
Como adotar Cognitive Driven Development 11

Cognitive-driven Development, DDD e o mundo real e moderno **15**
O cenário do DDD de hoje em dia 15
DDD da massa 15
Agora show me the code! 17
Conclusão 29

Conjunto de técnicas para seu dia a dia **30**
CDD em contextos já existentes 30
Outras técnicas de código para colocar no seu cinto de utilidades 30

A prioridade máxima é funcionar de acordo com o caso de uso. Beleza e formosura não dão pão nem fartura. **31**
A segunda coisa mais importante é que o código seja entendível 31
E agora, escolho que o código funcione ou que seja entendível? 31

Execute o seu código o mais rápido possível **33**
Comece pela entrada de dados do caso de uso 33

Exemplo de fluxo começando pela entrada de dados 33
Cada execução é uma chance de pegar um possível bug mais cedo 34

Protegemos as bordas do sistemas como se não houvesse amanhã. Principalmente a mais externa 35

Separamos as bordas externas do sistema do seu núcleo. 36
A fragilidade do contrato estabelecido 36
Genérico x Específico 37
Crie classes para recebimento de parâmetros 37
De quebra ainda estamos mais seguros 38
Meu código ficou mais complexo 38

Não serializamos objetos de domínio para respostas de API 39
A fragilidade do acoplamento 39
Criando uma representação para o endpoint 40

Toda indireção aumenta a complexidade 41
Um exemplo de código com quase nenhuma indireção 41
Começamos a jornada das indireções 42
A indireção para o tipo considerado de domínio 43
A indireção para persistir objetos 44
Código mais enxuto é menos complexo? 45
Analisando o código pela métrica derivada do CDD 45
Existe um limite de indireções? 46
Recado final sobre indireção 47

Usamos o construtor para criar o objeto no estado válido. 48
E o id, não vem pelo construtor? 49
E por que eu não vou usar os métodos de acesso? 49
Só que o framework precisa de um construtor sem argumentos 50

Deixe pistas onde não for possível resolver com compilação 51
Ouvi falar que deixar comentários ou pistas é ruim? 52
Essas validações de annotations vão ser executadas automaticamente? 52

A complicação do nosso é código é proporcional a complicação da nossa feature 53

Usamos tudo que conhecemos que está pronto. 54
O medo do acoplamento do negócio com libs e frameworks 55
Abrace o acoplamento com tudo que é maduro 56

Idealmente, todo código escrito deveria ser chamado por alguém. 57
Evite geração de código desenfreada 57
E quando fluxos forem modificados? 57

Só alteramos estado de referências que criamos. **58**
Alterar referências alheias complica todo acompanhamento do estado 59
Como solução alternativa, deixe dicas de que algo foi alterado 59
Também deixe dicas sobre alterações em sistemas externos 60
Explicite a dificuldade 61

A versão mais eficiente de um(a) dev desenvolve o que foi combinado **62**
Você precisa entender exatamente o que é necessário 62
Implemente o que foi combinado 63
Dev não é um artista? 63

Você precisa entender o que está usando e olhar sempre o lado negativo de cada decisão. **64**
Decidi adotar CDD como linha de design de código 64
Decidi utilizar uma inspiração arquitetural com muitas camadas 64
Decidi utilizar uma arquitetura distribuída 65
Conseguir olhar para os pontos de atenção é uma restrição 65
Quantos pontos de atenção eu devo levantar? 65

API's não democráticas **66**
O perigo da falta de estabilidade dos contratos 66
Precisamos de interfaces específicas e estáveis 67

Não usamos exception para controle de fluxo. **69**
Efeito colateral #1: Talvez custe performance 69
Efeito colateral #2: Ponto de tratamento do problema 70
Efeito colateral #3: Aumento de complexidade de entendimento 71
Efeito colateral #4: No mundo das exceptions de runtime, perdemos clareza 71
Possibilidade de utilizar retornos explícitos 72
Resumo da ópera 75

Regras de negócio devem ser declaradas de maneira explícita na nossa aplicação. **76**
Dando um passo na clareza do fluxo de negócio 77
O ponto de atenção do acoplamento extramente fraco 77
Explicitando o acoplamento 78

Favorecemos a coesão através do encapsulamento. **80**
Exemplo #1: Lógica de consulta de estado do objeto 80
Exemplo #2: Coesão através do encapsulamento para atualização de objeto 82
Uma entidade pode acessar um service? 83

Criamos testes automatizados para que ele nos ajude a revelar e consertar bugs na aplicação. **86**
A aposta potencialmente arriscada dos testes automatizadas 86
Por que você acha que o teste automatizado te dá segurança para alguma coisa? 87
Técnicas de teste para que ele seja mais revelador de bugs? 87

A distorção da cobertura de código 88
Testes inteligentes 90
O caminho para uma suíte de testes poderosa 91

Por que este ebook existe?

Este ebook existe para que eu(Alberto) tenha a chance de mostrar um pedaço do meu trabalho e para que você consiga ter uma prévia do que é trabalhado dentro da Jornada Dev Eficiente[1]. A jornada é um treinamento cujo foco é buscar alta performance para profissionais que querem criar, manter e refatorar código priorizando sempre o funcionamento e o entendimento. As três habilidades citadas serão treinadas de maneira intensa. Imagine que você vai sair como um(a) dev(a) atleta, preparado(a) para várias situações diferentes.

Dentro do ebook eu vou te contar como enxergo qualidade de código e porque acho que as técnicas utilizadas hoje em dia não são suficientes para que você produza código que funcione como deveria e que realmente seja entendível por outras pessoas no futuro.

Também vai ser encontrado aqui um conjunto de técnicas de código que utilizo no meu dia a dia e que me permitem buscar sempre por um código cuja complexidade seja proporcional a necessidade do problema que está sendo resolvido.

Para fechar, sou o criador do canal Dev Eficiente[2] onde falo sobre qualidade de código de maneira crítica e muito conectada com o que encontramos no mercado.

Espero que tudo escrito aqui seja útil! Caso você tenha acessado este ebook sem ter deixado seu email para mim, avalie o conteúdo e, se gostar, acesse o site Dev Eficiente[3] e se inscreva para saber da próxima Jornada Dev Eficiente!

[1] https://deveficiente.com/
[2] https://www.youtube.com/deveficiente
[3] https://deveficiente.com/

Arquitetura x Design

Muito se fala sobre arquitetura x design de código. Não parece interessante levar horas falando sobre isso, então podemos resumir da seguinte forma: arquitetura é transversal para todo o sistema e para múltiplos sistemas diferentes enquanto que o design de código, até hoje, sempre foi específico.

Durante muito tempo focamos muito na arquitetura, nos seus mais variados níveis:

- Qual é a infraestrutura que vamos utilizar aqui? Vai ser cloud native?
- Qual estilo de separação de camadas vamos usar aqui? Hexagonal, Clean arch, Onion etc? Percebe que MVC ficou tão padrão, que nem falamos mais.
- Como vamos lidar com a questão da segurança dos dados?

Todas essas preocupações são muito importantes, mas insuficientes. Falta um pedaço chave aí: o código específico que vai ser produzido. Cansamos de ver classes gigantes, com uma estratégia ineficiente de separação de interesses/responsabilidades dentro do chamado domínio da aplicação. Pouca importa o estilo arquitetural que você escolheu no tocante ao código em si, em algum lugar dele vai existir classes relativas ao problema real que está sendo resolvido.

Precisamos de uma arquitetura para nosso design :). Os livros e práticas que foram escritos até hoje não foram suficientes. E olha que já tivemos centenas, talvez milhares, de livros publicados sobre divisão de responsabilidades e organização de um código.

É necessároi buscar um linha mestra relativa ao design e que possa ser aplicável para a maioria dos sistemas que você for implementar. Como maximizar a chance que sistemas pequenos, médios e grandes sejam divididos de modo que a maioria das pessoas possa entender? Como tentar garantir essa régua mesmo quando você estiver na maior pressão da sua vida?

Vamos buscar por um design de código que seja sustentável no médio e longo prazo e também escalável quando pensamos em tamanhos de equipe.

Um arquitetura para seu design

Quando falamos de design do código, estamos querendo referenciar tudo que for relativo ao modo que as unidades específicas do seu sistema se juntam para resolver uma determinada funcionalidade. Alguns exemplos:

- Uma rede social precisa suportar criação de novos usuários;
- Um plataforma de um banco precisa suportar pagamento de conta;
- Um plataforma de aluguel de carros precisa suportar alugar o carro do lugar mais próximo;
- Uma plataforma de aulas precisa suportar que uma pessoa pergunte no fórum;

Perceba que para resolver cada das funcionalidades acima você vai precisar criar um código específico, pouco importa o estilo de separação de camadas que você decidiu. **Essa parte específica**, **que geralmente pode mudar de um projeto para outro**, **que você deve considerar como o design do seu código**.

Algumas inspirações já usadas para tentar suportar um bom design

Independente do estágio da carreira dentro da indústria de desenvolvimento, sempre ouvimos falar de alguns autores e livros que já podem até ser considerados clássicos;

- Design Patterns; escrito por Erich Gamma, Richard Helm , Ralph Johnson, John Vlissides;
- Clean Code; escrito por Robert Martin (uncle bob)
- Domain Driven Design; escrito por Eric Evans;
- Refactoring: Improving the Design of Existing Code; escrito por Martin Fowler
- Patterns of Enterprise Application Architecture; escrito por Martin Fowler
- Test Driven Development By Example; escrito por Kent Beck

Aqui são apenas alguns poucos livros que parecem influenciar demais a forma como o design dos códigos são produzidos mundo a fora.

É importante ressaltar que tais livros representam apenas a ponta do iceberg da busca por uma forma interessante de criar código que funcione como deveria, suporte as mudanças futuras e que ainda possa ser entendido pela versão futura de quem escreveu e pelas outras pessoas que vão passar por aquela base.

Quando olhamos para a academia, temos algumas publicações que influenciam muito tudo que fazemos hoje:

- No Silver Bullet Essence and Accidents of Software Engineering; escrito por Frederick P. Brooks, Jr
- Programming with abstract data types; escrito por Barbar Liskov e Stephen Zilles
- Applying Design by Contract; escrito por Bertrand Meyer
- On the Criteria To Be Used in Decomposing Systems into Modules; escrito por David Parnas
- A Complexity Measure (uma das várias métricas que nos guiam); escrito por THOMAS J. McCABE;

Poderíamos citar muitas outras publicações, principalmente quando falamos de métricas de código.

Fazer software manutenível ainda é um desafio a ser vencido

Com tanta literatura disponível influenciando a forma como fazemos código, como será que ainda somos capazes de produzir código que muitas vezes são muito complicados de serem mantidos? O custo de manutenção pode chegar a ser de 90%[4] relativo a tudo necessário para um software ser mantido. Dentro de tudo que pode ter a ver com manutenção, temos pontos específicos do software em si:

- Custo para alteração de funcionalidades (manutenção evolutiva);
- Custo para correção de bugs (manutenção corretiva);
- Custo para refatoração;
- Custo para criação de novas funcionalidades;
- Custo de pessoas para realizarem tais atividades;

Aqui[5] pode ser encontrado uma tentativa visual de mostrar diversos tipos de custos relacionados a manutenção, inclusive os relativos ao software em si.

Já temos também pesquisas que tentam explorar que tipo de unidade de código mais sofre intervenção durante a vida útil do software. No artigo On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation[6] foi explorado a relação entre necessidade de correção de bugs e classes que eram má avaliadas de acordo com métricas de software quando comparadas com as outras classes do próprio sistema.

Alguns resultados interessantes foram encontrados:

- Classes que apresentam avaliações ruins do ponto de vista das métricas tendem a apresentar mais bugs e sofrerem mais alterações;

[4] https://ocw.unican.es/pluginfile.php/1408/course/section/1805/SMCOSTS.pdf
[5] http://st.inf.tu-dresden.de/files/teaching/ws12/ring/03-Planning-Software-Evolution.pdf
[6] https://link.springer.com/article/10.1007/s10664-017-9535-z#Sec13

- Uma vez que uma classe apresentou um bug, ela tende a ser fonte de novos bugs no futuro. Mesmo depois que o bug é corrigido e ela refatorada;

O estudo analisou 390 releases de 30 projetos open source diferentes. Ou seja, foi uma análise profunda, justamente para tentar maximizar a chance de gerar dados relevantes. Estamos falando de projetos open source famosos e que são usados pelos sistemas desenvolvidos por todo santo negócio no planeta.

A primeira afirmação trazida pelo estudo parece até óbvia: Classes complexas apresentam mais bugs e são mais alteradas. Só que a segunda afirmação, é ainda mais preocupante. Uma vez que um pedaço de código apresentou bug, ele tende a ser fonte de bugs por muito tempo. Existem algumas possíveis causas:

- A tentativa de refatorar pode gerar novos bugs;
- A correção de um bug pode levar a outro;

Uma interpretação dessa análise pode ser a seguinte: **Precisamos cuidar do design de código para manter as unidades simples**. **Parece haver uma boa possibilidade de unidades mais simples apresentarem menos bugs e também de serem menos alteradas**. Correr atrás do leite derramado vai ser muito mais doloroso.

Por que softwares ainda ficam muito complicados de serem mantidos?

Como muito bem disse Frederick Brooks, existe uma dificuldade essencial no software que é a sua complexidade. No artigo *No Silver Bullets*, que é muito mais do que simplesmente "não tem bala de prata" existe uma seção cujo título é **Complexity** onde é explicado que software é um tipo de produto muito diferente e mais complexo do que somos acostumados(as) a lidar.

Diferente de uma geladeira, carro, prédio e hardware, para citar alguns poucos exemplos, software é um produto que é completamente alterado durante seu curso de vida. Não é que uma peça ou outra é trocada, o código é evoluído. Muitas vezes o que ele foi projetado para resolver aumenta muito de escala. Outras vezes o negócio vai mudando e o software precisa ser adaptado. É realmente tudo muito diferente. Por conta de toda essa combinação, vamos abrir espaço para citar um trecho do artigo de Brooks:

" *Many of the classical problems of developing software products derive from this essential complexity and its non-linear increases with size. From the complexity comes the difficulty of communication among team members, which leads to product flaws, cost overruns, schedule delays. From the complexity comes the difficulty of enumerating, much less understanding, all the possible states of the program, and from that comes the unreliability. From the complexity of the functions comes the difficulty of invoking those functions, which makes programs hard to use. From complexity of*

structure comes the difficulty of extending programs to new functions without creating side effects. From complexity of structure come the unvisualized states that constitute security trapdoors. “

Em tradução sugerida para português teríamos:

” *Muitos dos problemas clássicos de desenvolvimento de produtos de software derivam dessa complexidade essencial e seu aumento não linear com o tamanho. Da complexidade vem a dificuldade de comunicação entre os membros da equipe, o que leva a falhas no produto, estouro de custos, atrasos no cronograma. Da complexidade vem a dificuldade de enumerar, muito menos de compreender, todos os estados possíveis do programa, e daí vem a insegurança. Da complexidade das funções vem a dificuldade de invocar essas funções, o que torna os programas difíceis de usar. Da complexidade da estrutura vem a dificuldade de estender programas a novas funções sem criar efeitos colaterais. Da complexidade da estrutura vêm os estados não visualizados que constituem alçapões de segurança.* “

Sem dúvida ele viu na frente e acertou exatamente o que aconteceria. É importante lembrar que tudo isso foi escrito há mais de trinta anos.

Parece que as técnicas de design não estão sendo suficientes

Infelizmente, por mais que tenhamos literatura extensa, o desafio continua. Um exemplo de toda essa complexidade de código pode ser contada através da história da classe `EarlyAlertServiceImpl`, extraída de um projeto open source chamado *SSP* que está público no *Github*.

- Ela nasceu com 100 linhas de código e avaliada de forma ok pelas métricas;
- Quatro meses depois, ela tinha 500 linhas de código e já era xingada pelas métricas;
- Três anos e meio depois ela tinha 1000 linhas de código e estorou a cabeça de um grupo de pessoas que participou de um evento para tentar refatorar a classe monstra;

Essa é a história da vida real de muitas classes e funções espalhadas mundo a fora. História que cruza com a das pessoas que tiveram suas vidas impactadas por precisarem manter tais códigos.

Como que um código que nasceu com 100 linhas chegou a 1000? Numa conta básica, considerando complexidade apenas pelo número de linhas, temos um aumento de 10 vezes! E isso aumenta caso algumas métricas sejam combinadas! O que você conhece que aumenta 10x ou mais a complexidade de manutenção num intervalo de 3 anos? E lembre que, ainda considerando apenas o número de linhas, ela aumentou 5x de complexidade em 4 meses!

A complexidade do software é uma dificuldade essencial e precisamos experimentar novas formas de lidar com ela.

Uma outra forma de lidar com a complexidade inerente do código

Muito do que temos relativo a design de código foca na forma como os módulos do sistema se comunicam e como podemos ficar preparados para eventuais trocas de peças/engrenagens. E se for necessário outro framework MVC? E se for necessário outro mecanismo de persistência?

Livros como o Clean Code, Refactoring e Domain Driven Design, até tentam abordar o problema do ponto de vista de entendimento humano. Lá temos preocupações como funções longas, contextos bem definidos, nomes em comum entre negócio e engenharia etc. Por mais que sejam excelentes sugestões, ainda não foram capazes de resolver ou minimizar a níveis aceitáveis a dificuldade essencial da complexidade.

Se a complexidade é uma dificuldade inerente ao software, talvez a gente precise de algo também inerente ao código para segurar todo o aumento de tal complexidade com o tempo.

Um design focado no limite de entendimento humano

Em vez de olharmos para design de código como algo que visa facilitar a troca de peças do sistema no futuro, que tal olharmos para o design de código como algo que visa facilitar o entendimento daquele código no futuro pela perspectiva do ser humano? A premissa aqui é: **se for fácil de entender, vai ser mais fácil de manter**. Tudo isso vem dentro da ideia que a prioridade total é funcionar.

Se as pessoas tiverem muita dificuldade em entender o que está escrito, como que elas vão substituir algo com menor esforço? E pior, se elas tem dificuldade de entender o que está escrito, como que elas vão manter aquilo, mesmo sem precisa trocar engrenagens? Inclusive vemos isso falhando em sistemas que optaram por arquiteturas distribuídas, como a de microsserviços.

No próximo capítulo vamos falar sobre uma ideia de design onde o código será escrito respeitando um limite de complexidade de entendimento restritivo, inspirado por uma teoria chamada **Teoria da Carga Cognitiva**.

É necessário que você fique de mente aberta

Já faz muitos anos que códigos são escritos inspirados na literatura que já referenciamos aqui. Agora vamos partir para uma possível nova abordagem, que vai abrir novas possibilidades de implementação.

Claro que novos perigos devem surgir, já que tudo tem um lado negativo. Por outro lado, quem não arrisca não petisca e a indústria de software já está muito avançada para continuar produzindo código muito mais complexo do que deveria.

Cognitive-Driven development

Aqui você encontra a tradução do artigo onde é apresentada a ideia do CDD na conferência ICSME[7]. A teoria foi criada por mim(Alberto) e o trabalho publicado foi desenvolvido em conjunto com Victor Pinto. Você pode encontrar a versão original aqui[8].

Resumo da ideia

A separação do software em componentes é um reconhecimento de que o trabalho humano pode ser aprimorado com o foco em um conjunto limitado de dados.

A crescente complexidade do software sempre foi um desafio para a indústria. Várias abordagens foram propostas para apoiar o design de código com base em estilos de arquitetura e métricas de qualidade de código.

No entanto, a maioria das pesquisas envolvendo cognição humana em engenharia de software está focada na avaliação de programas e aprendizagem, ao invés de como o código-fonte poderia ser desenvolvido sob essa perspectiva.

Este artigo apresenta uma abordagem denominada Cognitive-Driven Development (CDD) que se baseia em medidas de complexidade cognitiva e na Teoria de Carga Cognitiva. Essa estratégia pode reduzir a sobrecarga cognitiva dos desenvolvedores por meio da limitação de pontos de complexidade intrínseca do código-fonte.

Algumas métricas de complexidade cognitiva foram estendidas e diretrizes são apresentadas para calcular os pontos de complexidade intrínseca e como seu limite pode ser adaptado sob certos critérios de qualidade.

Estudos experimentais estão sendo conduzidos para avaliar o CDD. Os resultados preliminares indicam que a abordagem pode reduzir o esforço futuro para manutenção e correção de falhas de software.

Introdução

A separação de interesses é um dos princípios-chave da engenharia de software 8[9], 14[10]. Desde a análise, os desenvolvedores precisam entender o

[7] https://icsme2020.github.io/
[8] https://github.com/asouza/pilares-design-codigo/blob/master/ICSME-2020-cognitive-driven-development.pdf
[9] http://cv.znu.ac.ir/afsharchim/lectures/p50-liskov.pdf
[10] https://apps.dtic.mil/sti/pdfs/AD0773837.pdf

problema.

Uma estratégia para isso é dividi-lo em blocos mais compreensíveis. Para desenvolver a solução, as práticas de codificação recomendadas e um padrão arquitetônico devem ser seguidos para atingir uma modularidade e coesão aceitáveis para as unidades de implementação. No entanto, a complexidade do software aumenta à medida que novos recursos são incorporados 20[11], [21 , 5[12] impactando sua manutenibilidade, um dos atributos de qualidade de software mais recompensadores 1[13].

Portanto, a separação da responsabilidade do componente deve considerar não apenas o domínio, mas também a complexidade cognitiva do software 18[14]. Na psicologia cognitiva, a carga cognitiva se refere à quantidade de informações que os recursos da memória de trabalho podem conter de uma vez.

A Teoria da Carga Cognitiva (CLT) 17[15], 2[16], 16[17] é geralmente discutida em relação à aprendizagem. Problemas que exigem um grande número de itens armazenados na memória de curto prazo podem contribuir para uma carga cognitiva excessiva. Segundo Sweller 17[18], alguns materiais são intrinsecamente difíceis de entender e isso está relacionado ao número de elementos que devem ser processados simultaneamente na memória de trabalho. Estudos experimentais realizados por Miller 12[19] sugeriram que os humanos geralmente são capazes de manter apenas sete mais ou menos duas unidades de informação na memória de curto prazo

Esse limite para unidades de informação pode ser aplicado para software uma vez que o código-fonte tenha uma carga intrínseca. Os desenvolvedores são frequentemente afetados por sobrecarga cognitiva quando precisam adicionar um recurso, corrigir um bug, melhorar o design ou otimizar o uso de recursos.

Normalmente, há muitas informações que eles não podem processar facilmente. SOLID Design Principles 9[20], Clean Architecture 10[21], Hexagonal Architecture 3[22] e outras práticas bem conhecidas são geralmente adotadas na indústria para tornar os projetos de software mais compreensíveis, flexíveis e sustentáveis. De acordo com as práticas de Domain-driven Design

[11] Acomplexitymetricforobject-orientedsoftware.
[12] https://www.researchgate.net/profile/Paul_Vossen/post/How_many_citations_does_it_take_to_indicate_an_academic_article_is_influential/attachment/59d62de0c49f478072e9eb89/AS%3A273568468799507%401442235213752/download/Fraser+et+al.+2007+%27No+Silver+Bullet%27+Reloaded+~+A+Retrospective+on+%27Essence+and+Accidents+of+Software+Engineering%27.pdf
[13] https://www.iso.org/standard/35733.html
[14] https://ieeexplore.ieee.org/abstract/document/4216416/
[15] https://psycnet.apa.org/record/2010-09374-002
[16] https://ro.uow.edu.au/cgi/viewcontent.cgi?article=1133&context=edupapers
[17] https://onlinelibrary.wiley.com/doi/pdf/10.1207/s15516709cog1202_4
[18] (https://psycnet.apa.org/record/2010-09374-002)
[19] https://pure.mpgde/rest/items/item_2364276_4/component/file_2364275/content
[20] https://fi.ort.edu.uy/innovaportal/file/2032/1/design_principles.pdf
[21] https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html
[22] https://alistair.cockburn.us/hexagonal-architecture/

(DDD) 4[23], a linguagem do código do software deve estar alinhada com o domínio do negócio. Embora nem todas as propostas sejam relacionadas a código e modelagem, o objetivo é mitigar a complexidade do software.

Como as melhorias são esperadas no ciclo de vida do desenvolvimento, o código necessita de baixa complexidade intrínseca, um atributo de software de qualidade 1[24]. No entanto, elementos com baixa coesão e separação ineficiente de responsabilidades continuam sendo produzidos e residindo nos lançamentos finais. A complexidade ciclomática de McCabe (CYC) 11[25] é uma métrica para o número máximo de caminhos linearmente independentes em um gráfico de controle do programa.

A ideia principal é identificar módulos de software que serão difíceis de testar ou manter. O CYC pode ser importante para limitar a complexidade do código e determinar o número de casos de teste necessários. No entanto, pode ser uma das métricas de código mais difíceis de entender o que torna difícil calcular.

Definir um ponto de complexidade intrínseca no software não é claro para os desenvolvedores, muito menos como estabelecer um limite de complexidade para cada unidade de implementação.

Este artigo apresenta uma abordagem denominada Cognitive-Driven Development (CDD). A abordagem proposta é baseada no CLT e na complexidade cognitiva do software. Para tanto, foi ampliado um conjunto de métricas para avaliar a complexidade cognitiva em aplicações orientadas a objetos, incluindo um método diferente para utilizá-las. São propostas diretrizes para definir um limite para pontos de complexidade intrínseca no código-fonte. Os resultados preliminares sugeriram que o CDD pode contribuir para reduzir o esforço necessário para manutenção e correção de falhas.

Trabalhos relacionados

Métricas de complexidade cognitiva orientadas a objetos foram propostas no trabalho de Shao e Wang em 15[26] e modificadas por Misra et al. 13[27]. Validação teórica e empírica foi conduzida para avaliar cada métrica com base nas propriedades de Weyuker 19[28].

[23] http://188.166.46.4/get/PDF/Eric%20Evans-Domain-Driven%20Design_%20Tackling%20Complexity%20in%20the%20Heart%20of%20Software_14.pdf

[24] https://www.iso.org/standard/35733.html

[25] https://www.academia.edu/download/48213691/tse.1976.23383720160821-12832-sniirk.pdf

[26] https://www.researchgate.net/profile/Yingxu_Wang/publication/3454763_A_new_measure_of_software_complexity_based_on_cognitive_weights/links/004635205604daa4f9000000.pdf

[27] https://ieeexplore.ieee.org/iel7/6287639/6514899/08253447.pdf

[28] https://www.researchgate.net/profile/Elaine_Weyuker/publication/3186968_Evaluating_software_complexity_measures_IEEE_Trans_Softw_Eng/links/0a85e533312f00e01f000000/Evaluating-software-complexity-measures-IEEE-Trans-Softw-Eng.pdf

Embora o CDD complemente essas métricas, seu principal objetivo é fornecer um método para conduzir o desenvolvimento que mantenha o limite de complexidade intrínseca. Gonçales et al. 6[29] fornecem uma taxonomia de carga cognitiva em engenharia de software. Com base nesta pesquisa, os avanços recentes estão relacionados às tarefas de programação, técnicas de aprendizado de máquina para identificar o nível de dificuldade do programador e sua compreensibilidade em nível de código. O CDD pode ser considerado como uma proposta de design complementar para mitigar o aumento da complexidade cognitiva independente do tamanho do software.

Como adotar Cognitive Driven Development

O desenvolvimento de software direcionado ao domínio de negócios, modularidade, reutilização de código, alta coesão, baixo acoplamento e independência de tecnologia são a essência da programação orientada a objetos (OO).

Porém, devido à contínua expansão da complexidade do software 21[30], a compreensibilidade não pode seguir na mesma proporção. O CDD é inspirado na psicologia cognitiva para o desenvolvimento de software, considerando um limite razoável para pontos de complexidade intrínseca 17[31].

A Tabela I apresenta algumas Estruturas de Controle Básico (ECB) e recursos, incluindo seu número correspondente para pontos de complexidade intrínseca. Eles são baseados em métricas para avaliar a complexidade cognitiva em aplicações orientadas a objetos 15[32]. “Acoplamento contextual e “acoplamento com tecnologias transversais” são novas categorias e a coluna Ponto de complexidade intrínseca (PCI) foi escolhida em vez de peso para esclarecer nosso método proposto. Os elementos descritos aqui não são limitados; os desenvolvedores são livres para incluir elementos adicionais que considerem interessantes.

Category	ECB e recursos	PCI
Branch	if/else/case/switch	1 ponto para cada instrução
Tratamento de exception	try/catch/finally	1 ponto para cada instrução
Acoplamento com tecnologias transversais	Acoplamento com tipos da linguagem base, frameworks escolhidos etc.	Não contamos ponto para isso

[29] https://kleinnerfarias.github.io/pdf/articles/icpc-2019.pdf
[30] http://adt.ivknet.de/download/papers/SCMM_rcomplexity.pdf
[31] https://psycnet.apa.org/record/2010-09374-002
[32] (https://www.researchgate.net/profile/Yingxu_Wang/publication/3454763_A_new_measure_of_software_complexity_based_on_cognitive_weights/links/004635205604daa4f9000000.pdf)

Para cada branch analisada, um ponto de complexidade é aumentado. Por exemplo, if-else tem 2 pontos e try-catch-finally, 3. Em relação ao acoplamento, se uma classe foi criada para lidar com uma responsabilidade específica dentro do mesmo projeto, 1 ponto é contabilizado. Funções que podem aceitar outras funções como argumentos, chamadas funções de ordem superior, requerem um certo nível de compreensão e, portanto, podem ser consideradas como um PCI. O código relacionado a tecnologias transversais não contam pontos. Embora tal código possa aumentar a complexidade cognitiva do software, sugerimos não considerá-los como PCI, porque eles, em geral, não são alterados dentro da construção da lógica de negócios.

```
24@RestController
25public class NewPaymentController {    8
26
27    @Autowired
28    private CCSoIsValidForOnlineCardValidator 1
29     ccSoIsValidForOnlineCardValidator;
30    @Autowired
31    private ValidPaymentForRestaurantUserValidator 1
32     validPaymentForRestaurantUserValidator;
33
34    @PersistenceContext
35    private EntityManager manager;
36    @Autowired
37    private AllPaymentProcessors allProcessors; 1
38    @Autowired
39    private UserRepository userRepository; 1
40
41    @InitBinder
42    public void init(WebDataBinder binder) {
43        binder.addValidators(ccSoIsValidForOnlineCardValidator,
44              validPaymentForRestaurantUserValidator);
45    }
46
47    @PostMapping(value = "/payments")                    1
48    public CompletableFuture<?> execute(@Valid NewPaymentForm form) {
49           PaymentAttempt paymentAttempt =
50                form.toModel(manager,userRepository);
51       CompletableFuture<TransactionPayment> res =      1
52             allProcessors.pay(paymentAttempt);
53
54        return res.thenApply(transactionPayment -> {    1
55            if(transactionPayment.isOk()) {             1
56                return ResponseEntity.ok().build();
57            }
58            return ResponseEntity.status(403).build();
59        });
60    }
61}
```

Fig. 1. Class NewPaymentController

Classe analisada

A figura acima apresenta um trecho de código de uma classe Java chamada *NewPaymentController*. No topo da figura é mostrado o número 8 correspondente ao total de PCIs. Os primeiros 6 pontos estão relacionados ao acoplamento contextual (linhas 28, 31, 37, 39, 48 e 51) e os pontos restantes referem-se à passagem de uma função como um argumento e à

instrução if (linhas 54 e 55).

Em experiências anteriores de treinamento de desenvolvedores de software, notamos que quando eles são instruídos a não exceder um limite de complexidade intrínseca durante a programação, o código resultante geralmente apresenta altos níveis de coesão e modularidade.

O CDD é altamente flexível para diferentes cenários de desenvolvimento. Em relação à definição de um limite para a complexidade cognitiva, temos algumas recomendações com base nas experiências citadas. No caso de aplicações web e equipes mistas em termos de experiência, o número máximo de pontos de complexidade intrínseca pode ser mais restritivo. A ideia é sempre ser guiado pelo limite de capacidade de entendimento sugerido pela Teoria da Carga Cognitiva.

Para aplicações e api's web, pode-se considerar um limite de sete mais ou menos dois pontos. Caso a classe tenha atributos de dependências, consideramos sete pontos como limite. Tais clases geralmente são aquelas conhecidas como *Controllers* e *Services*. Para classes com atributos de dados, podemos considerar nove pontos como limite. Estas classes geralmente são *entities*, *value objects* e classes de entrada e saída de dados.

Para frameworks e bibliotecas, pode-se ter um limite maior, talvez entre dez e doze pontos para cada unidade de implementação, considerando equipes
mais especializadas em uma determinada linguagem e contexto.

Cognitive-driven Development, DDD e o mundo real e moderno

O cenário do DDD de hoje em dia

As práticas de design sugeridas pelo DDD são agnósticas em relação a frameworks, linguagem e quaisquer outras tecnologias envolvidas. Por outro lado a maioria das aplicações são desenvolvidas em cima justamente de uma linguagem e com um conjunto de tecnologias estabelecidas que formam a fundação de qualquer sistema a ser desenvolvido naquele lugar.

Não podemos e seria até ingênuo ignorar os frameworks que facilitam nosso trabalho. Pelo menos para a maioria esmagadora das aplicações que são desenvolvidas. Precisamos é tirar proveito de tais tecnologias em cima de alguma sugestão de padrão, neste caso estamos falando do DDD.

DDD da massa

O DDD da massa é a sugestão de aplicação de junção do que é proposto por parte do livro DDD dentro do contexto design proposto pelo CDD especificamente para aplicações web desenvolvidas no mundo atual.

O que eu quero com isso? Possibilitar um padrão de desenvolvimento de aplicações web que façam o necessário, tenham design suficientemente flexível para suas necessidades e onde o esforço de entendimento da maior parte do projeto seja baixo. Alguns detalhes importantes:

- Focado em aplicações web;
- Pensado majoritariamente para aplicações que fazem uso de linguagens orientadas a objetos;
- Considera fortemente os frameworks que facilitam nossa vida;

Focado em aplicações web

Vamos levar em consideração o fluxo natural da maioria das requisições que chegam nas aplicações web. Lembrando que seu microservice é uma aplicação web.

1. Dados da requisição entram por um método do controller;
2. Tais dados são convertidos para os tipos específicos;
3. Se a conversão deu certo, eles são validados seguindo as regras da funcionalidade;

4. Em função de tais dados, algum objeto do model é criado/carregado;
5. Alguma lógica pode ser executada em cima do model;
6. Caso a lógica altere o estado do model, esse estado é refletido no seu banco de dados ou qualquer outro lugar;
7. Pode ser que essa alteração de estado dispare novos fluxos no sistema;
8. Um objeto de resposta é gerado;
9. Tal resposta é serialiazada num formato específico e enviada para a aplicação cliente;

Esse fluxo todo acontece dentro de uma requisição e, quando levamos em consideração apenas UMA requisição, não podemos dizer que tal fluxo faz uso intensivo de memória. Por outro lado é bem diferente de aplicações que precisam fazer realmente processamentos em memória a partir de qualquer fonte de dados. Processar dados textuais, imagens, buscas inteligentes etc.

É muito importante que você perceba a diferença desse tipo de software para outros, inclusive aqueles que você utiliza como tecnologia base. É muito diferente implementar um Tomcat versus uma aplicação web tradicional. O mesmo vale para seu framework, biblioteca etc.

Pensado majoritariamente para aplicações que fazem uso de linguagens orientadas a objetos

Desde cedo na minha carreira eu abracei as linguagens orientadas a objetos. Posso dizer que me considero um especialista em tal paradigma e sinto que tenho boas coisas para compartilhar e propor. Considero que as linguagens que suportam O.O, escolha a que quiser, fornecem os meios necessários para construirmos aplicações web suficientes para as necessidades do mercado.

Por outro lado não me aprofundei em outros paradigmas, um exemplo clássico é o funcional. Acho que faço bom uso de funções como cidadãs de primeiro nível em códigos com Javascript, Kotlin, Scala etc. Também tiro proveito dessas construções em linguagens como Java e C#. Sou também fã de imutabilidade e realmente acho que ela facilita o controle do estado da aplicação e o debug em cima de algum fluxo.

Daí a dizer que posso sugerir algo para quem usa clojure ou qualquer outra linguagem que faça uso mais forte do paradigma funcional seria muita ousadia :).

Consideramos fortemente os frameworks que facilitam nossa vida

Olhando para a sequência de passos, podemos notar que os frameworks atuam em algumas frentes facilitando nosso vida.

1. Encapsulam toda a lógica de receber a request, converter e validar boa parte dos dados;
2. Fornecem mecanismos para você criar validações customizadas que adicionem proteção as bordas;

3. Fornecem os meios para refletir o estado do model no seu banco de dados;
4. Podem fornecer(geralmente fornecem) os meios para disparar os eventos em função da alteração de estado;
5. Serializam a resposta no formato pedido pelo cliente;

Pensando em entendimento, não vejo motivos para criarmos aplicações que sejam desacopladas dos frameworks que formam a fundação de um código. Alguns exemplos práticos de combinação entre linguagem e framework.

- Java + Spring + ORM;
- Ruby + Rails + ORM;
- Python + Django/Flesk + ORM;
- Php + Laravel + ORM;
- Javasript + NestJs + ORM;

Uma tentativa de construir código que te deixe fortemente desacoplado dos frameworks citados vai elevar o número de linhas de código do seu projeto, trazer abstrações mais complicadas e consequentemente aumentar a carga intrínseca do sistema como um todo.

A sugestão vai abraçar o framework escolhido e deixar claro que está tudo bem seu código ficar acoplado a ele. O que você precisa é ter um olhar lógico sobre o quão preparado seu framework está para você decidir ficar amarrado com ele. Uma vez que decidimos, vamos tirar o máximo de proveito.

Alguns pontos que você pode olhar na hora que tiver se acoplando com uma tecnologia específica em algumas ou várias partes da sua aplicação. Vamos pegar para análise um pouco do Spring.

* Você recebeu como parâmetro de um tipo específico do Spring. Este tipo tem uma interface pública facilmente testável?
* Existem implementações de testes específicas para aquelas interfaces que você está se acoplando?
* Existe um suporte a realização de testes automatizados usando o contexto do framework?

Em geral, para tecnologias mais maduras, você vai ter boas respostas para todas ou parte das perguntas.

Agora show me the code!

É claro que vou show you the code. Especificamente neste texto eu vou trabalhar a distribuição da carga intrínseca pelo olhar de uma aplicação web escrita com o Spring, mas as práticas podem ser seguidas para qualquer combinação de linguagem O.O e um framework que realmente te abstraia o trabalho duro.

A carga intrínseca é contextual numa aplicação web

É importante que seja estabelecida uma régua de carga intrínseca que possa ser observada em toda a aplicação. Os projetos que analisam qualidade olham todo arquivo do mesmo jeito, entendo que esse é um jeito simplista e não condizente com o funcionamento de uma aplicação web.

Inclusive Aniche, no artigo Tailoring Code Metric Thresholds for Different Software Architectures(http://pure.tudelft.nl/ws/files/12555765/TUD_SERG_2016_023.pdf), traz essa observação. As medidas de coesão, acoplamento entre outras são diferentes em diferentes contextos da mesma aplicação.

Segue a minha sugestão de distribuição de carga intrínseca em função do contexto de uso de cada tipo de classe.

- Classes com atributos de dependência devem ter no máximo 7 pontos de complexidade. Alguns exemplos desses tipos de classes em um projeto:
  * Controllers e Domain Services(services comumente usados) não devem passar de 7 pontos de carga intrínseca. Elas lidam com fluxo de informações e isso deveria ser facilmente entendido. Imagino que exista espaço inclusive para ser ainda mais restritivo.
- Classes com atributos de dados, devem ter no máximo 9 pontos de complexidade. Temos alguns exemplos:

* *Value objects*. Elas em geral tem poucos métodos que operam sobre o estado.
* Classes com estado persistente Essas classes geralmente são as *entities*.

- Infrastructure Services podem ter a pontuação que você quiser. Elas geralmente são classes que acessam alguma coisa externa... Você quase não mexe e não tem problema gastar um pouco mais de tempo quando for dar manutenção.
- Classes de configuração também podem ter a carga intrínseca que quiser... O template delas é montado uma vez e depois a manutenção acontece de vez em nunca.
- Repository deve ter carga de no máximo 3 pontos. A depender do framework, seu repository pode até flertar com um ponto de carga intrínseca.

Todas as sugestões de código vão ser baseadas nas pontuações sugeridas acima.

Controllers 100% coesos

Lembre que um *Controller* é uma classe com atributos de dependência

A primeira coisa é entender o significado da palavra coesão. Existem algumas métricas disponíveis para avaliar coesão, a que eu utilizo como base aqui é uma chamada Tight and Loose Class Cohesion(https://www.aivosto.com/project/help/pm-oo-cohesion.html#TCC_LCC). Não dá para aplicá-la exatamente igual o sugerido, já que um controller não é bem uma classe no sentido puro da palavra.

Um controller é apenas um recipiente de rotas que representam os endpoints expostos nas aplicações. Ele não mantém estado da aplicação e seus atributos deveriam ser, na verdade, variáveis locais de seus métodos. Uma rota é uma função, que recebe uma entrada e gera uma saída. Pela definição do artigo Tipos Abstratos de Dados(https://dl.acm.org/doi/pdf/10.1145/942572.807045) ela é uma função abstrata. O problema é que linguagens como Java e C# nos obrigam a escrever classes para que seja possível declarar funções(métodos estáticos). E os frameworks se apoiam nisso. Em vez de declararmos variáveis locais, declaramos atributos e recebemos seus valores injetados pelo framework.

Dado esse cenário, a minha sugestão é que todo método de um controller use todos os atributos declarados. Além disso a carga cognitiva dele não deveria passar de 7 pontos(acesse aqui para entender o que aumenta a carga cognitiva). O resultado dessa combinação é que você vai ter controllers enxutos e que não ultrapassam o limite da memória de trabalho(https://medium.com/@albertosouza_-47783/um-outro-olhar-sobre-complexidade-de-c%C3%B3digo-16370f9f9c80).

O tradeoff é que você vai ter mais classes que representam controllers do que você tem atualmente. Na minha opinião isso não é um problema. Inclusive controllers inflados foi justamente um problema encontrado por Aniche(http://pure.tudelft.nl/ws/files/12555765/TUD_SERG_2016_023.pdf) numa pesquisa que ele fez em cima de várias aplicações web encontradas no github. É importante lembrar que mais arquivos só são um problema se eles não estiverem distribuindo de maneira equiibrada a carga cognitiva pelo sistema.

Por fim, um controller escrito usando um framework interessante, já abstrai toda infra http para você.

Agora você tem um cenário onde ele é enxuto e super específico. A soma disso é que na minha opinião você pode fundir em vários momentos a ideia de Application Service com Domain Service(Domain Service Controller?) e chegar em um código parecido com esse:

```
1  @RestController
2  public class RetornoPagamentoPagSeguroController {
3
4      @Autowired
5      private CompraRepository compraRepository;
6
7      @Autowired
8      private PagamentoRepository pagamentoRepository;
9
10     @Autowired
11     private ApplicationEventPublisher applicationEventPublisher;
12
13     @InitBinder
14     public void initBinder(WebDataBinder dataBinder) {
15         dataBinder.addValidators(new VerificaSeCompraJaEstaConcluidaValidator(compra\
16 Repository));
```

```
17      }
18
19      @PostMapping(value = {"/api/retorno-pagamento/{compraId}/pagseguro"})
20      @Transactional
21      public void processaRetorno(@PathVariable("compraId") Long compraId, @Valid Reto\
22  rnoPagamentoPagSeguroRequest retornoPagamentoPagSeguroRequest, UriComponentsBuilder \
23  uriComponentsBuilder) {
24
25          Compra compra = FindById.executa(compraId, compraRepository);
26
27          Pagamento pagamento = retornoPagamentoPagSeguroRequest.criaPagamento(compra);
28          pagamentoRepository.save(pagamento);
29
30          applicationEventPublisher.publishEvent(new NovoPagamentoEvent(this, pagament\
31  o, uriComponentsBuilder));
32      }
33  }
```

Dado a minha sugestão de que o único acoplamento que devemos levar em consideração é o feito com classes criadas no próprio sistema, temos a seguinte conta de carga intrínseca aqui:

- +1 CompraRepository;
- +1 PagamentoRepository;
- +1 VerificaSeCompraJaEstaConcluidaValidator;

+ +1 RetornoPagamentoPagSeguroRequest;

- +1 Compra;
- +1 FindById;
- +1 Pagamento;
- +1 NovoPagamentoEvent;

Temos 8 pontos de carga intrínseca. Ultrapassa em 1 ponto a sugestão para um Domain service controller :), mas continua dentro do limite aceitável. E perceba que você já fez tudo necessário para a execução da lógica de negócio também.

Eu sugiro a carga do Domain service controller ficar em 7 justamente porque ele está na borda mais externa da aplicação e, por ser um local onde as pessoas começam a olhar um código, deveria ser mais fácil de entender. Claro que você pode ser mais restritivo e baixar essa pontuação se achar interessante, experimente. Um detalhe legal é que só passamos de 7 pontos porque o foi decidido usar uma abstração chamada `FindBy` para isolar o tratamento do retorno Opcional da busca pelo id da `Compra`.

Você ganhou controllers coesos, com carga cognitiva baixa e que tem uma régua clara para review de código. Inclusive que pode ser automatizada. Se a carga cognitiva passar de 7, você tenta distribuir :). Vou dar um exemplo para esse de cima:

```
@RestController
public class RetornoPagamentoPagSeguroController {

    @PersistenceContext
    private EntityManager manager;

    @Autowired
    private ApplicationEventPublisher applicationEventPublisher;

    @InitBinder
    public void initBinder(WebDataBinder dataBinder) {
        dataBinder.addValidators(new VerificaSeCompraJaEstaConcluidaValidator(compra\
Repository));
    }

    @PostMapping(value = {"/api/retorno-pagamento/{compraId}/pagseguro"})
    @Transactional
    public void processaRetorno(@PathVariable("compraId") Long compraId, @Valid Reto\
rnoPagamentoPagSeguroRequest retornoPagamentoPagSeguroRequest, UriComponentsBuilder \
uriComponentsBuilder) {

        Compra compra = manager.find(compraId,Compra.class);

        Pagamento pagamento = retornoPagamentoPagSeguroRequest.criaPagamento(compra);
        manager.persist(pagamento);

        applicationEventPublisher.publishEvent(new NovoPagamentoEvent(this, pagament\
o, uriComponentsBuilder));
    }
}
```

Usei o `EntityManager` da JPA direto e ainda tirei uma linha de `save` que não precisava. Agora, se fizermos a mesma conta:

- +1 VerificaSeCompraJaEstaConcluidaValidator;

+ +1 RetornoPagamentoPagSeguroRequest;

- +1 Compra;
- +1 Pagamento;
- +1 NovoPagamentoEvent;

Saímos de 8 para 5 sem criar novos arquivos, códigos etc. Usei o `EntityManager` direto no nosso mais novo Domain service controller? Usei, e daí :)?

E se eu puder processar compras através de outras entradas do sistema? Adapte o código :). Quando um sistema cresce, pouco importa se a arquitetura é monolítica ou distribuída, você vai perdendo o controle do que está pronto ou não. No fim, você não precisa ter medo de mudança, basta que ela seja mais fácil de ser realizada. Você agora tem um fluxo com carga intrínseca baixa e que pode ser mais fatiado em caso de necessidade.

Form Value Objects (classes de formulário inteligentes)

Aqui é um caso clássico de fluxo de aplicações web. Você tem um formulário de cadastro ou de pesquisa na sua aplicação e precisa receber dados de modo que em algum momento eles possam ser convertidos em um objeto que faz parte do model, geralmente as entities. Durante um tempo, por conta da facilidade provida pelos frameworks, os objetos do model eram montados diretamente a partir dos parâmetros pelos próprios frameworks.

```
1          @PostMapping(value = "/usuarios")
2         @Transactional
3         //abre uma transacao
4         public void novo(@Valid Usuario novoUsuario) {
5                 manager.persist(novoUsuario);
6         }
```

A tragédia era anunciada, mas precisou acontecer um caso realmente impactante para que o mundo ficasse mais atento ao problema. Lá em 2012 o Github sofreu um hack((https://github.blog/2012-03-04-public-key-security-vulnerability-and-mitigation/).) através de uma técnica conhecida como mass assignment(https://en.wikipedia.org/wiki/Mass_assignment_vulnerability).

Um outro problema é específico do contrato. O parâmetro aqui é um contrato firmado com outra aplicação cliente. Neste exemplo, uma mudança no model pode acarretar uma mudança de contrato indesejada e o pior, que só seria percebida na hora do uso em si.

Essa combinação deixou bem claro que era necessário controlar tudo que vem do cliente. Inclusive o Sonar, software que avalia qualidade de código, possui uma regra que negativa seu código quando entities aparecem como argumentos. O engraçado é que isso já era algo trabalhado no Struts 1, muito antes do Github. Tudo fica de novo de novo.

A partir dali, o mesmo endpoint começou a ser escrito da seguinte forma:

```
        @PostMapping(value = "/usuarios")
        @Transactional
        //abre uma transacao
        public void novo(@Valid NovoUsuarioForm form) {

                Usuario novoUsuario = NovoUsuarioConverter.converte(form);
                manager.persist(novoUsuario);

        }
```

A classe que converte entrada de dados em objetos de domínio ficou tão famosa que até bibliotecas foram criadas em cima dela. Um exemplo disso é a ModelMapper(http://modelmapper.org/getting-started/). Precisamos sempre ter em mente que criamos novos arquivos para distribuir a carga intrínseca pelo sistema ou para utilizar uma funcionalidade da linguagem que casa com o novo arquivo. No caso acima, o mesmo código poderia ser escrito da seguinte forma:

```
        @PostMapping(value = "/usuarios")
        @Transactional
        //abre uma transacao
        public void novo(@Valid NovoUsuarioForm form) {
                Usuario novoUsuario = new Usuario(form.getNome(),form.getEmail())
                manager.persist(novoUsuario)
        }
```

Não aumentamos em nenhum ponto a carga intrínseca do código e atingimos o mesmo resultado. E se esse form fosse muito maior, com muitos dados, relacionamentos etc? Vamos pegar um outro exemplo.

```
@RestController
public class CrudPerformanceReviewController {

        @Autowired
        private GoalRepository goalRepository;
        @Autowired
        private PerformanceReviewRepository performanceReviewRepository;
        @Autowired
        private SystemUserRepository systemUserRepository;

        @PostMapping(value = "/api/performance/review")
        @Transactional
        public void save(@RequestBody @Valid NewPerformanceReviewForm form) {
                List<PerGoalPerformanceReviewRequiredInfo> reviews = form.getReviewsForm().stream(\
```

```
15  ).map(reviewForm -> {
16  return new PerGoalPerformanceReviewRequiredInfo() {
17
18                          @Override
19                          public String getPositivePoints() {
20                                  return positivePoints;
21                          }
22
23                          @Override
24                          public String getImprovingPoints() {
25                                  return improvingPoints;
26                          }
27
28                          @Override
29                          public NextGoalStep getNextGoalStep() {
30                                  return nextGoalStep;
31                          }
32
33                          @Override
34                          public Goal getGoal() {
35                                  return goalRepository.findById(goalId).get();
36                          }
37
38                  };
39
40                  }).collect(Collectors.toList());
41
42                  SystemUser employee = systemUserRepository.findById(employeeId).get();
43
44                  Optional<SalaryReviewInfo> salarayReviewInfo = form.isSalaryReview() ? Optional.of\
45  (salaryReviewForm.toModel()) : Optional.empty();
46                  PerformanceReview newReview = new PerformanceReview(employee,reviews,salaryReview,\
47  salarayReviewInfo);
48
49                  performanceReviewRepository.save(newReview);
50
51
52          }
53
54  }
```

Por conta do formulário com mais informações e com mais necessidades de transformação, naturalmente a carga intrínseca do código aumentou. Somando o acoplamento contextual e branches(map

+ if ternario) temos 10 pontos de carga intrínseca. Passou de 9? Vamos precisar refatorar. De novo o converter poderia entrar no jogo:

```
@RestController
public class CrudPerformanceReviewController {

        @Autowired
        private NewPerformanceReviewConverter newPerformanceReviewConverter;

        @PostMapping(value = "/api/performance/review")
        @Transactional
        public void save(@RequestBody @Valid NewPerformanceReviewForm form) {
        PerformanceReview newReview = newPerformanceReviewConverter.convert(form);
                performanceReviewRepository.save(newReview); //aqui poderia ser o EntityManager di\
reto :)
        }

}
```

Essa solução resolve? Sem dúvida. Distribui a carga instrínseca? Distribui também. Mas no fim ela aumenta a carga intrínseca do sistema como um todo em 1 ponto, já que temos uma nova classe. E se você pudesse distribuir a carga intrínseca sem necessariamente criar um arquivo novo?

```
@RestController
public class CrudPerformanceReviewController {

        @Autowired
        private PerformanceReviewRepository performanceReviewRepository;
@Autowired
        private GoalRepository goalRepository;
        @Autowired
        private SystemUserRepository systemUserRepository;

        @PostMapping(value = "/api/performance/review")
        @Transactional
        public void save(@RequestBody @Valid NewPerformanceReviewForm form) {
        PerformanceReview newReview = form.toModel(goalRepository,systemUserRepository);
                performanceReviewRepository.save(newReview);
        }

}
```

Perceba que mantemos a carga intrínseca do controller abaixo de 7, evitamos a criação de uma nova classe e conseguimos implementar a mesma funcionalidade. O que machuca os olhos é esse método

`toModel` combinado com argumentos que representam repositórios? O método `toModel` associa estado + comportamento combinando com parâmetros recebidos. Era justamente essa a proposta de Barbara Liskov no artigo Tipos Abstratos de Dados(https://dl.acm.org/doi/pdf/10.1145/942572.807045). Admito que desconheço melhor uso do paradigma. E você pode limitar o acesso aos métodos do repositório passando apenas a interface pública específica como argumento, caso ache necessário:

```
@RestController
public class CrudPerformanceReviewController {

        @Autowired
        private PerformanceReviewRepository performanceReviewRepository;
@Autowired
        private GoalRepository goalRepository;
        @Autowired
        private SystemUserRepository systemUserRepository;

        @PostMapping(value = "/api/performance/review")
        @Transactional
        public void save(@RequestBody @Valid NewPerformanceReviewForm form) {
        PerformanceReview newReview = form.toModel(goalRepository::findById, systemUserRepo\
sitory::findById);
                performanceReviewRepository.save(newReview);
        }

}
```

E você pode se perguntar, então por que eu não deixo literalmente toda a inteligência dentro das classes de model e forms da aplicação? Porque você estouraria a carga intrínseca máxima por arquivo. Não vai estourar? Experimente. Se liberte da ditadura da divisão de responsabilidade e camada. Sem experimentação, não tem questionamento e nem geração de conhecimento.

Para fechar, você pode olhar para a classe de formulário como uma extensão do Value Object do DDD. Ali ele cita que tais classes existem para guardar um estado que pode ser temporário ou persistente, mas que não necessariamente tem uma identidade. O formulário apenas não se encaixa no que ele chama de camada de model da aplicação, mas se relaciona com a explicação em si do pattern. Você pode chamar as classes de formulários de Form Value Objects :).

Maximize a manipulação de estado dentro de entities, value objects e suas variações.

Aqui é uma consequência das sugestões citadas e também reforçada pelo DDD. Quando restringimos a carga intrínseca das partes procedurais da nossa aplicação web, naturalmente vamos mover parte da inteligência para nossas entities e value objects. Só que precisamos de algumas restrições. Para

trabalharmos o exemplo, vamos pegar um código implementado para aceitar a participação de uma pessoa em um bolão entre amigos.

```
1     public ResultadoConfirmacao model(BolaoRepository bolaoRepository, ParticipanteR\
2 epository participanteRepository, UsuarioRepository usuarioRepository) {
3         final Bolao bolao = bolaoRepository.findById(this.idBolao).get();
4         final Participante participante = participanteRepository.findById(this.idPar\
5 ticipante).get();
6 
7         if (bolao.getDataExpiracao().isBefore(Instant.now())) {
8             return ResultadoConfirmacao.conviteExpirado();
9         }
10 
11        final Optional<Usuario> possivelUsuario = usuarioRepository.getByLogin(parti\
12 cipante.getEmail());
13        if (possivelUsuario.isEmpty()) {
14            return ResultadoConfirmacao.usuarioNaoExiste();
15        }
16 
17        final Usuario usuario = possivelUsuario.get();
18 
19        return new ResultadoConfirmacao(new Participacao(bolao, usuario));
20    }
```

Esse trecho de código em si tem 8 pontos de carga cognitiva e está em um método de transformação num Form Value Object. A métrica de carga intrínseca ia deixar passar, mas quando olhamos para a carga intrínseca da classe `Bolao` em si, percebemos que ele está super baixa. **Esse é um outro ponto de atenção**: se um ponto do código com carga intrínseca no limite usa uma entity com carga intrínseca super baixa, pode ser um sinal de má distribuição, lembre que você está em busca de equilíbrio. Pensando em orientação a objetos, pode ser que tenhamos entidades anêmicas.

Uma `Participacao` é resultado da combinação entre um usuário e um bolão específico. E é justamente o que o código tenta fazer. O problema é a falta de uso de restrições oferecidas pela própria linguagem. Inclusive ainda faltou uma verificação para analisar se o participante está no mesmo no conjunto de convites do bolão. Teria mais um if :). O mesmo código poderia ser escrito da seguinte forma:

```
    public ResultadoConfirmacao model(EntityManager manager, UsuarioRepository usuar\
ioRepository) {
        final Participante participante = manager.find(this.participanteId,Participa\
nte.class);
        final Optional<Usuario> possivelUsuario = usuarioRepository.findByEmail(part\
icipante.getEmail());

       if(!possivelUsuario.isPresent()){
         //exception
       }

        final Bolao bolao = manager.find(this.bolaoId,Bolao.class);

        return  bolao.aceita(usuario);
    }
```

E agora o método `aceita` da classe `Bolao` poderia ser assim:

```
  public ResultadoConfirmacao aceita(Usuario participante) {
       if (this.dataExpiracao.isBefore(Instant.now())) {
           return ResultadoConfirmacao.conviteExpirado();
        }

        if (!this.emails.contains(participante.getEmail())) {
           return ResultadoConfirmacao.naoConvidado();
        }

        return new ResultadoConfirmacao(new Participacao(this, usuario));
  }
```

O local de invocação do método `aceita` não está aderente ao sugerido na explicação do Form Value Object, mas isso é apenas um detalhe aqui. O importante é que concentramos as operações sobre o estado em quem possui o estado em vez deixar para quem tem acesso externo ao estado.

O resultado final é uma carga intrínseca ainda melhor distribuída entre as partes da aplicação fazendo uso dos recursos providos pela linguagem.

Domain services 100% coesos

Lembre que classes anotadas com `@Service` são também classes com atributos de dependência.

Pode ser que para determinado fluxo a carga intrínseca máxima de 7 pontos não seja suficiente no nosso Domain service controller. Neste momento você vai precisar pensar em como vai dividir a carga intrínseca. Algumas coisas que podem ser analisadas:

- Será que não estou fazendo código além do necessário para a funcionalidade?
- Será que não tem lógica sobre o estado da aplicação vazando das entities e variações de value objects?
- Será que não estou fazendo código que já foi resolvido pelo framework?

Dado que você analisou e entendeu que não tem como diminuir a carga intrínseca com alguma das técnicas citadas acima, você vai precisar simplesmente dividir para conquistar. Você vai criar uma nova classe para dividir a carga intrínseca daquele ponto do código. Provavelmente essa classe vai flertar com um domain service e você precisa usar a regra dos 100% de coesão nessa nova classe. Neste tipo de cenário você está simplesmente quebrando uma procedure em duas :).

Um segundo cenário, um pouco mais complicado de aparecer mas também factível, é se você tiver outra entrada de dados para executar a mesma funcionalidade. No caso disso acontecer no mesmo sistema, eu simplesmente tentaria trabalhar com o Domain service controller que eu já tenho :).

```
1  NewPerformanceReviewForm newReviewForm = converteEntradaManualParaForm();
2  crudPerformanceReviewController.save(newReviewForm);
```

O seu antigo controller, neste mundo de frameworks modernos, faz parte agora do seu domain model. Faça referência a interface pública dele e chame o método :).

Caso não goste da sugestão, leve o código do *Domain Service Controller* para uma classe intermediária e utilize no outro ponto. Fique sempre atento(a) com a pontuação máxima.

Conclusão

A proposta do DDD da massa é realmente trazer as práticas sugeridas por Eric Evans para um cenário mais específico. Como aplicamos tais práticas em um cenário de aplicações web construídas em cima de frameworks que abstraem boa parte do trabalho repetitivo?

Abraçamos o acoplamento inteligente com as tecnologias fundamentais para a aplicação e entendemos que a análise de carga cognitiva precisa ser feita de modo contextual. Inclusive sugerimos como podemos analisar de maneira lógica a carga intrínseca de cada ponto do nosso código.

Para fechar foi deixado algumas sugestões de práticas de código que podem facilitar a aplicação dos conceitos que foram trabalhados.

Você pode fazer o mesmo exercício para outras inspirações arquiteturais :).

Conjunto de técnicas para seu dia a dia

Bastante foi falado sobre controlar complexidade pela perspectiva humana. Agora como levar isso para seu contexto?

CDD em contextos já existentes

A primeira coisa que você deve pensar sobre o CDD é como adaptá-lo para seu cenário. Algumas sugestões:

- Para classes que já existem aceite o fato da complexidade que já está lá. Pontue e coloque objetivos de refatoração para diminuir a complexidade de entendimento em termos percentuais relativos ao total já acumulado.
- Para classes novas você pode estabelecer o limite restritivo;
- Para manutenção, corretiva ou evolutiva, já leve em consideração o limite e construa os novos os códigos com o limite em mente;

É importante que você considere complexidade como um fator de compilação. Código com sintaxe errada não compila, a mesma coisa deveria acontecer para código que não respeita o limite de entendimento humano.

Outras técnicas de código para colocar no seu cinto de utilidades

A partir do próximo capítulo você vai ser apresentado(a) a um conjunto de técnicas de código que são úteis para qualquer dev(a) que pretenda utilizar as opções disponíveis dentro da linguagem para construir códigos que respeitem o limite de complexidade sem ficar reinventando a roda.

A prioridade máxima é funcionar de acordo com o caso de uso. Beleza e formosura não dão pão nem fartura.

Aqui é a prática de código que eu considero mais importante. Antes de qualquer coisa o seu código deve executar de maneira que atenda ao caso de uso. Nada é mais importante do que código funcionando.

A segunda coisa mais importante é que o código seja entendível

Quando falamos que a prioridade máxima é funcionar, não estamos querendo dizer que ele deve funcionar de tal forma que apenas naquele momento do ciclo de vida do sistema o código deva ser entendido. **A segunda coisa mais importante é que possa ser compreendido por outras pessoas**.

Um código é um bicho diferente da maioria das coisas que são produzidas pela humanidade. Enquanto quase tudo que fazemos obedece a um ciclo de construção, entrega e uso com eventuais manutenções, um código vive num ciclo diferente. Ele nasce, é entregue e depois ele fica sendo alterado para sempre. Muitas vezes essas alterações são profundas e feitas com o sistema em funcinamento.

Por conta dessa natureza mutante, precisamos de códigos que possam ser entendidos por outras pessoas no futuro.

E agora, escolho que o código funcione ou que seja entendível?

Aqui não é uma questão de escolha. A pergunta que você deve fazer é: *como faço para implementar um código que funcione e que seja entendível por outras pessoas?* A resposta para isso é: Domine o máximo que puder do contexto que está inserido.

- Domine o negócio;
- Domine a linguagem de programação;
- Domine os frameworks e bibliotecas que sustentam o sistema;
- Domine as técnicas de teste que podem ajudar a revelar bugs;

- Domine qualquer outra coisa que te apoie no fluxo;

Você domina algo quando olha para aquilo e simplesmente sabe o que precisa fazer. É quase robótico :).

Execute o seu código o mais rápido possível

A prioridade básica de um software é funcionar e já falamos sobre isso no pilar "A prioridade máxima é funcionar de acordo com o caso de uso. Beleza e formosura não dão pão nem fartura". E como você pode abraçar essa ideia de fato enquanto desenvole?

Comece pela entrada de dados do caso de uso

Para cada funcionalidade necessária de ser implementada vai existir uma forma de entrada de dados. Aqui temos alguns exemplos:

- Via requisição HTTP;
- Via mensageria;
- Via processamento em batch;

A sugestão é que você comece seu código por aí. Dessa forma você vai ter a chance de executar o seu código o mais rápido possível e mais vezes durante a implementação.

Exemplo de fluxo começando pela entrada de dados

Suponha que você está implementando uma api que recebe dados via HTTP para suportar o funcionamento de criação de propostas para cartões crédito. Um fluxo de implementação que começa pela entrada de dados seria o seguinte:

- Já vai no Insomnia, Postman, curl ou qualquer cliente e mapeia a requisição que você quer;
  * Aqui, de maneira até ingênua, já podemos executar! Mesmo sabendo que vai dar 404;
- Cria agora o método que vai ser mapeiado para receber os dados da requisição;
  * aqui você já pode executar para verificar se a requsição está chegando;
- Cria a classe ou a abstração específica na linguagem escolhida para receber os dados que estão vindo na requisição;
  * já pode executar de novo para verificar se os dados estão chegando;
- Trabalha em cima das validações necessárias para proteger a borda do sistema;
  * mais uma chance de executar;

- Converte o objeto que representa a requisição com os dados da proposta para seu objeto de domínio que realmente tem os atributos que representam os dados necessários do sistema e que vão ser persistidos;
  * mais uma chance de executar o código e verificar se tudo está indo como deveria;
- Salva o objeto no banco de dados;
  * executa de novo;
- Manda uma mensagem para o sistema externo que precisa ser avisado da proposta;
  * executa de novo;

Perceba que no mínimo você executou seu código sete vezes antes de completar a implementação. E tudo isso em cima do fluxo real, ou seja, quando você acabar realmente vai estar com a implementação pronta.

Cada execução é uma chance de pegar um possível bug mais cedo

Cada vez que você executa você tem a chance de pegar o bug mais cedo e isso é muito positivo. **Quanto mais tempo você demora para executar o seu código, maior é a chance de ter um bug ali que você não está percebendo**.

Agora você pode criar seus testes automatizados para que eles tentem revelar ainda mais bugs :).

Protegemos as bordas do sistemas como se não houvesse amanhã. Principalmente a mais externa

Todo dado que entra em algum método do nosso sitema é potencialmente inválido. Por que você vai executar o código sem garantir que os parâmetros de entrada estão validos? Quando falamos da borda mais externa então, o cuidado é redobrado. Não controlamos nada do lado do cliente e não assumimos que nada está válido.

```
1         @PostMapping(value = "/propostas")
2         @Transactional
3         public ResponseEntity<?> cria(
4                         @RequestBody @Valid NovaPropostaRequest request) {
5                 //codigo que vai rodar em função dos parâmetros recebidos
6             }
7     }
8
9     public class NovaPropostaRequest {
10
11        private String email;
12        private String nome;
13        private String endereco;
14        private BigDecimal salario;
15        private String documento;
16
17        //construtor
18    }
```

Se você olhar para cima, como vai saber que o email é valido? E como vai saber que o nome não está em branco? O código que está dentro do método do controller só deveria rodar se as informações contidas no objeto que representa a request estiverem válidos. Fique sempre com isso na cabeça. Do mesmo jeito que você não aceita encomenda com caixa aberta, você não aceita parâmetros com valores inválidos.

Todos os frameworks maduros do mercado suportam algum mecanismo de validação integrado. O Spring não seria diferente. Ele tem um módulo chamado Spring Validation e que permite que você crie validações específicas e também se integre com a especifcação Bean Validation.

Separamos as bordas externas do sistema do seu núcleo.

Não ligamos parâmetros de requisição externa com objetos de domínio diretamente, assim como não serializamos objetos de domínio para respostas de API.

Todo framework web moderno permite que você receba os dados de uma requisição web utilizando uma classe do seu próprio domínio. Exemplo:

```
@PostMapping
public void novaProposta(Proposta proposta){

}
```

Essa classe proposta poderia um construtor com argumentos com os mesmos nomes dos parâmetros que vem na requisição.

```
public class Proposta{
    public Proposta(String email, String nome, String documento){
        ...
    }
}
```

Ou ela poderia ter um construtor sem argumentos e métodos setters para cada informação necessária. E talvez você esteja se perguntando: Isso é ótimo, qual é o problema?

A fragilidade do contrato estabelecido

No exemplo recebemos o documento(cpf/cnpj) como um argumento do construtor. Dessa forma, neste momento a aplicação cliente que manda dados para gente, deve estar enviando algo assim:

```
{
    "email":"email...".,
    "nome":"nome",
    "documento","..."
}
```

Agora imagine que você mudou o fluxo do recebimento do documento. Em vez de receber no construtor da Proposta você vai receber num segundo momento. Como movimento natural, você vai lá e tira aquele argumento do construtor. Neste exato momento você quebrou o cliente que consome qualquer endpoint que recebe uma proposta, pouco importa qual seja.

Genérico x Específico

A classe de domínio permeia a aplicação inteira. Ela nasce para ser utilizada em um ponto e naturalmente vai sendo exigida em outros lugares. Dada as necessidades tal classe pode ir sendo alterada. Como você vai minimizar a chance dessa alteração quebrar quem está consumindo algum endpoint?

Enquanto a classe de domínio nasce para ser usada na aplicação inteira, os parâmetros para um endpoint nascem para ser usados naquele local específico. Dois endpoints diferentes, por coincidência podem usar os mesmos parâmetros, mas eles ainda são diferentes. Os dados podem vir de formulários diferentes.

Crie classes para recebimento de parâmetros

A nossa sugestão é que você sempre crie classes específicas para receber os parâmetros. No nosso exemplo seu código poderia ser:

```
@PostMapping
public void novaProposta(NovaPropostaRequest request){
 ...
}
```

Existe uma bela chance da classe `NovaPropostaRequest` ter detalhes parecidos com a classe `Proposta` e está tudo bem. Não precisamos reaproveitar todo pedaço de código que existe.

Agora, se o código estivesse usando o construtor da classe `Proposta` e ele fosse alterado, seu código ia dar problema de compilação. E você pode pensar algo assim: “Ah, mas se eu alterar o construtor da classe `NovaPropostaRequest` vai quebrar também!”. Vai quebrar sim, mas vai quebrar apenas nesse endpoint :). Parar de funcionar é ruim, mas se a correção tiver sido facilitada o impacto disso é bem menor.

De quebra ainda estamos mais seguros

Criar classes específicas para receber os dados, ainda deixa nosso código mais seguro. Ficamos um pouco mais protegidos de um ataque conhecido como Mass Assignment[33].

Meu código ficou mais complexo

Ficou, é verdade. Dada a nossa ideia, esse código está mais complexo sim. Só que o tradeoff está claro, você aumenta a complexidade para diminuir a fragilidade da api e de quebra ainda ganha mais segurança. Claro que precisamos sempre ficar de olho no limite de complexidade inspirado no CDD e decidido pelo projeto.

[33] https://en.wikipedia.org/wiki/Mass_assignment_vulnerability

Não serializamos objetos de domínio para respostas de API

Essa daqui é uma prática simples de ser seguida e que tende a facilitar a manutenção dos contratos da sua api. Imagine que você precisa retornar uma lista de livros como JSON a partir de um endpoint. Abaixo temos um possível código.

```
    public List<Livro> todos(){
        return livroRepository.findAll();
    }

    public class Livro{
        private String nome;
        private BigDecimal preco;
        private Autor autor;
        private Categoria categoria;

        //getters para expor o que precisa ser exposto
    }
```

Não existe dúvida que esse código funciona.

A fragilidade do acoplamento

Por mais que do ponto de simplicidade o código esteja ótimo do ponto de vista do CDD, temos uma armadilha.

- E se você adicionar mais um getter na classe Livro?
- E se você remover um getter da classe Livro?
- E se você criar um getQualquerCoisa?

Qualquer uma das modificações acima vai modificar a resposta final gerada por aquele endpoint. E ela é muito possível de acontecer, justamente porque `Livro` é uma classe que faz parte do núcleo da aplicação e tem mais chances de sofrer modificações que cruzem a aplicação inteira.

Criando uma representação para o endpoint

Uma solução que adiciona um ponto de complexidde, dado a métrica que estamos trabalhando derivada do CDD, mas que diminui a fragilidade do contrato é a de criar uma classe específica para a saída de determinado endpoint.

```
1     public List<LivroListagemResponse> todos(){
2         return livroRepository.findAll().stream().map(livro -> {
3             return new LivroListagemResponse(livro);
4         });
5     }
6
7     public class LivroListagemResponse {
8         private String nome;
9         private BigDecimal preco;
10
11        public LivroListagemResponse(Livro livro){
12            this.nome = livro.getNome();
13            this.preco = livro.getPreco();
14        }
15    }
16
17    public class Livro{
18        private String nome;
19        private BigDecimal preco;
20        private Autor autor;
21        private Categoria categoria;
22
23        //getters para expor o que precisa ser exposto
24    }
```

Perceba que agora a classe `LivroListagemResponse` funciona como uma representante(Proxy) do livro para aquele endpoint. Vamos fazer as mesmas perguntas agora:

- E se você adicionar mais um getter na classe Livro?
- E se você remover um getter da classe Livro?
- E se você criar um getQualquerCoisa?

Para a primeira e terceira nada vai acontecer na nossa classe de saída. Já para a segunda vamos ter um erro de compilação, o que é ideal dentro de uma linguagem compilada. Mesmo se a linguagem for dinâmica, vamos tomar um erro em tempo de execução.

Importante ressaltar que antes aquele trecho de código tinha dois pontos de complexidade e passou a ter três, por conta da referência a nova classe.

Toda indireção aumenta a complexidade

Toda indireção aumenta a dificuldade de entendimento da aplicação como um todo, ela precisa merecer existir. Ou seja, precisa ajudar a distribuir a carga intrínseca pelo sistema.

Aqui estamos falando do caminho que o seu código precisa percorrer para que determinado fluxo de negócio seja executado. Tal caminho, em algum momento, também vai precisar ser percorrido pela pessoa que está buscando o entendimento do código como um todo.

Um exemplo de código com quase nenhuma indireção

```
    public class CadastraUsuarioController {
        public void cadastra(Map<String,String> request){
            String nome = request.get("nome");
            String idadeEmTexto = request.get("idade");
            String email = request.get("email");

            if(nome == null || nome.trim().equals("")){
                //retorna dizendo que nome é invalido
            }

            Integer idade = null;
            try{
                idade = Integer.parse(idadeEmTexto);
            } catch(ParseException exception){
                 //retorna dizendo que nome é invalido
            }

            // continua com validações
            try {
                PreparedStatement insert = driverDoBanco.prepareStatement("insert in\
to Usuario(...) values(?,?,?)");
                insert.setString(1,nome);
                insert.setString(2,idade);
                insert.setString(3,email);
            } catch(SQLException exception){
                //retorna erro 500 informando uma falha
```

```
27              }
28          }
29      }
```

O código usa construções básicas da linguagem, sem nenhuma abstração específica do próprio sistema. Teoricamente, neste código acima, caso a pessoa tenha familiaridade com a linguagem, não tem nada de novo. Por que não fazemos assim?

- Mesmo usando só coisa padrão, este código pode ficar com tanto if, try etc que pode dificultar a compreensão;
- Parte deste código é extremamente repetitivo entre funcionalidades, então queremos poupar tempo e usar algo pronto;
- A figura do usuário é importante para vários pontos da aplicação, então queremos constuir um tipo nosso para representá-lo e fazer referência em outros lugares;
- Por mais que o `Map` seja uma construção padrão da linguagem, a não ser que você tenha poderes mágicos,você não consegue saber as informações que ele tem lá dentro;

Começamos a jornada das indireções

```
1       public class CadastraUsuarioController {
2           public void cadastra(NovoUsuarioRequest request){
3               if(!request.taValido()){
4                   //retorna tudo que foi falha de validacao
5               }
6
7               try {
8                   PreparedStatement insert = driverDoBanco.prepareStatement("insert in\
9   to Usuario(...) values(?,?,?)");
10                  insert.setString(1,request.getNome());
11                  insert.setString(2,request.getIdade());
12                  insert.setString(3,request.getEmail());
13              } catch(SQLException exception){
14                  //retorna erro 500 informando uma falha
15              }
16          }
17      }
18
19      public class NovoUsuarioRequest {
20          private String nome;
21          private String email;
22          private int idade;
```

```
23
24          //métodos necessários
25
26          public boolean taValido(){
27              //verificacao aqui
28          }
29      }
```

Adicionamos a classe `NovoUsuarioRequest` para que o framework em questão consiga já fazer a conversão dos parâmetros que vieram na requisição para um tipo específico da aplicação.

Acabamos de colocar nossa primeira indireção e aumentamos a complexidade do sistema como um todo. A contrapartida é que diminuimos a complexidade deste ponto específico do código. Lembrando que aqui estamos sempre guiados(as) pela complexidade do ponto de vista cognitivo, sugerido pelo CDD.

Só que ainda temos uma abstração nossa para representar os dados da requisição, onde está a abstração para representar o usuário transversal ao sistema?

A indireção para o tipo considerado de domínio

```
1       public class CadastraUsuarioController {
2           public void cadastra(NovoUsuarioRequest request){
3               if(!request.taValido()){
4                   //retorna tudo que foi falha de validacao
5               }
6
7               Usuario novoUsuario = request.paraUsuario();
8               try {
9                   PreparedStatement insert = driverDoBanco.prepareStatement("insert in\
10  to Usuario(...) values(?,?,?)");
11                  insert.setString(1,novoUsuario.getNome());
12                  insert.setString(2,novoUsuario.getIdade());
13                  insert.setString(3,novoUsuario.getEmail());
14              } catch(SQLException exception){
15                  //retorna erro 500 informando uma falha
16              }
17          }
18      }
19
20      public class NovoUsuarioRequest {
21          private String nome;
22          private String email;
```

```
23          private int idade;
24
25          //métodos necessários
26
27          public boolean taValido(){
28              //verificacao aqui
29          }
30      }
31
32      public class Usuario {
33          private String nome;
34          private String email;
35          private int idade;
36
37          //o que for necessário
38
39      }
```

Agora temos mais uma indireção e a complexidade só aumenta :). Você pode até se pergunar, mas por qual motivo não recebemos o usuário direto ali? Lembre que **não ligamos parâmetros de entrada de dados com o objetos de domínio** pelos motivos já explicados no tópico referente a este pilar.

Especificamente neste caso não ganhmos nada em troca, apenas a indireção a mais. Ou seja, o código ficou apenas mais complexo na esperança que isso possa ser positivo para o sistema como um todo.

A indireção para persistir objetos

```
1       public class CadastraUsuarioController {
2           public void cadastra(NovoUsuarioRequest request){
3               if(!request.taValido()){
4                   //retorna tudo que foi falha de validacao
5               }
6
7               Usuario novoUsuario = request.paraUsuario();
8               orm.persist(novoUsuario);
9           }
10      }
11
12      public class NovoUsuarioRequest {
13          private String nome;
14          private String email;
15          private int idade;
```

```
16
17          //métodos necessários
18
19          public boolean taValido(){
20              //verificacao aqui
21          }
22      }
23
24      public class Usuario {
25          private String nome;
26          private String email;
27          private int idade;
28
29          //o que for necessário
30
31      }
```

Finalmente chegamos agora na última indireção desse fluxo. Queremos nos livrar do código necessário para trabalhar com seu banco de dados. Aumentamos mais um pouco a complexidade do sistema como um todo, mas aí ganhamos na diminuição da complexidade deste ponto específico de novo. Dado que não temos mais tratamento de exceptions.

Código mais enxuto é menos complexo?

Perceba que no final você tem um código com menos linhas. Ele é necessariamente menos complexo? Como comentamos, você pode encarar complexidade como a teia de necessidades que você precisa dominar para entender determinado material, nesse caso um pedaço de código.

No nosso exemplo final você tem algumas necessidades de entendimento:

- A classe `NovoUsuarioRequest`;
- A classe `Usuario`;
- A possível classe `ORM`;

Nada disso é padrão da linguagem. Duas são específicas do seu sistema e outra possivelmente é de um framework que pode ser utilizado em muitas aplicações diferentes. Ou seja, você precisa de outros entendimentos para que aquele pedaço de código fique realmente mais fácil.

Analisando o código pela métrica derivada do CDD

Dada a métrica atual sugerida pelo CDD, entendemos que aquele código fica sim mais fácil. Justamente porque você deveria saber a priori sobre o ORM escolhido e, quando você faz a troca dos

`ifs` e `trys` que estavam lá pelas indireções, o saldo fica positivo. Tudo vai depender da sua métrica de complexidade e do limite que você definiu.

Só para deixar claro, você pode ter um código enxuto e com várias construções que sejam penalizadas pela métrica atual do CDD. Lembre que agora você tem um jeito sistemático de analisar complexidade e completamente factível por qualquer pessoa, use!

Existe um limite de indireções?

A resposta para isso é não :). Não temos um limite de indireções, o que você precisa ter é clareza da troca que está fazendo. O código abaixo por exemplo:

```
    public class CadastraUsuarioController {
        public void cadastra(NovoUsuarioRequest request){
            if(!request.taValido()){
                //retorna tudo que foi falha de validacao
            }

            Usuario novoUsuario = request.paraUsuario();
            serviceNovoUsuario.cria(novoUsuario);
        }
    }

    public class ServiceNovoUsuario {
        public void cria(Usuario novoUsuario){
            orm.persist(novoUsuario);
        }
    }

    public class NovoUsuarioRequest {
        private String nome;
        private String email;
        private int idade;

        //métodos necessários

        public boolean taValido(){
            //verificacao aqui
        }
    }

    public class Usuario {
```

```
31          private String nome;
32          private String email;
33          private int idade;
34
35          //o que for necessário
36
37      }
```

Perceba que mais uma indireção foi criada e a complexidade do sistema como um todo aumentou, assim como a complexidade daquela unidade de código em específico. Isolando a análise em complexidade, só aumentou. O que você espera ganhar então?

O mercado vai dizer que você esperar ganhar flexibilidade para a troca do mecanismo de persistência. E é verdade! Acha que precisa, jogue duro e vá por este caminho. Tendo clareza do que está sendo feito, tudo é liberado.

Uma pergunta que você pode se fazer é? Será que seria tão complicado trocar os pontos de erro de compilação que vão aparecer por conta da troca do mecanismo de persistência?

Recado final sobre indireção

Realmente toda indireção aumenta complexidade. Quanto mais unidades você precisa entender, mais complexo é aquele ecossistema. Por outro lado ninguém busca entender o código de um sistema de uma vez só, geralmente caminhamos por unidades de código que se relacionam para completar alguma atividade. Até onde vai a caminhada, depende muito da necessidade que você tem de entendimento do contexto naquele momento.

A sugestão é que você sempre faça a conta da complexidade cognitiva respeitando o limite combinado pela equipe em função da métrica derivada do CDD que foi escolhida.

Usamos o construtor para criar o objeto no estado válido.

Em vários momentos do seu código você vai precisa instanciar um objeto. Geralmente isso vai acontecer no momento de criação dos seus objetos que têm atributos de estado e que tais estados vão ser persistidos num banco de dados por exemplo. Abaixo segue um fluxo de criação de um autor para um ecommerce de livros, por exemplo.

```
public class NovoAutorRequest {

        @NotBlank
        private String nome;
        @NotBlank
        @Email
        private String email;
        @NotBlank
        @Size(max = 400)
        private String descricao;

    //construtor para receber os dados

        public Autor toModel() {
        //ponto de criação de um autor em função de informações da request
                return new Autor(this.nome,this.email,this.descricao);
        }
}
```

Dado que as informações de nome, email e descrição são obrigatórias, forçamos ela através do construtor definido na classe.

```
    public Autor(String nome, String email,String descricao) {
                this.nome = nome;
                this.email = email;
                this.descricao = descricao;
        }
```

Isto é suportado pelas linguagens e deve ser usado. Você deve forçar o máximo de caminhos dentro do seu código.

E o id, não vem pelo construtor?

A pergunta que você deve se fazer é: O id é obrigatório na construção do objeto?

Sempre que você usar um id auto gerado pelo seu banco de dados ele não vai ser obrigatório pelo construtor. Informação obrigatória pelo construtor é aquela que é natural.

O id presente como atributo da sua classe só está lá por conta do banco de dados, no momento da criação do objeto você ainda não tem.

E por que eu não vou usar os métodos de acesso?

Você já deve ter visto o mesmo fluxo acima escrito da seguinte forma:

```
1         public Autor toModel() {
2         //ponto de criação de um autor em função de informações da request
3         Autor autor = new Autor();
4         autor.setNome(this.nome);
5         autor.setEmail(this.email);
6         autor.setDescricao(this.descricao);
7                 return autor;
8         }
```

E aí, lá na classe `Autor` está mais ou menos assim:

```
1     @Entity
2     public class Autor {
3
4         @Id
5         @GeneratedValue(strategy = GenerationType.IDENTITY)
6         private Long id;
7         private @NotBlank String nome;
8         private @NotBlank @Email String email;
9         private @NotBlank @Size(max = 400) String descricao;
10
11        public Autor() {
12
13        }
14
15        //getters e setters
```

Este código indica que nenhuma informação do autor é obrigatória na construção do objeto. A pessoa precisa navegar na classe, olhar as annotations e aí chamar os métodos corretos. Se você pode escrever um código que vai guiar a pessoa para fazer o que você quer, por que vai fazer um que não vai guiar?

Só crie métodos de acesso quando eles forem realmente necessários.

Só que o framework precisa de um construtor sem argumentos

Existem alguns frameworks como Hibernate que podem exigir que você tenha um construtor sem argumentos. E claro, a gente não precisa deixar de usar um facilitador deste tamanho só porque não queremos colocar um construtor sem argumentos. Olha o que você pode fazer:

```
1      @Deprecated
2          public Autor() {
3
4          }
5
6          public Autor(String nome, String email,
7                          String descricao) {
8                  this.nome = nome;
9                  this.email = email;
10                 this.descricao = descricao;
11         }
```

Você pode deixar uma dica informando que o construtor sem argumentos é depreciado. Isso envia um sinal de alerta para a pessoa que pensar em chamá-lo. Você pode inclusive acrescentar um comentário com a explicação.

Para você não ter que ficar escrevendo este construtor o tempo todo e colocando `@Deprecated` em cima, você pode também criar um atalho na sua ide.

Deixe pistas onde não for possível resolver com compilação

Deixamos pistas que facilitem o uso do código onde não conseguimos resolver com compilação.

Quando falamos do uso de construtor para forçar a passagem das informações naturais e obrigatórias[34], deixamos um exemplo de código. Vou lembrá-lo para você aqui:

```
    @Deprecated
        public Autor() {

        }

        public Autor(String nome, String email,
                        String descricao) {
                this.nome = nome;
                this.email = email;
                this.descricao = descricao;
        }
```

E aí, no outro ponto do sistema, tinha a chamada para o construtor com argumentos.

```
        public Autor toModel() {
                return new Autor(this.nome,this.email,this.descricao);
        }
```

Olhando só para ponto da chamada, como podemos saber se o nome ou descrição tem algum limite de tamanho, ou mesmo que o email tem o formato de email? Infelizmente não temos como fazer isso em tempo de compilação em nenhuma linguagem que seja amplamente utilizada no mercado, como Java ou C#.

Dado que não temos a melhor opção que é travar em tempo de compilação, podemos usar mecanismos da linguagem para deixar dicas.

[34] ./construtor-para-informacao-natural.md

```
    public Autor(@NotBlank String nome, @NotBlank @Email String email,
                        @NotBlank @Size(max = 400) String descricao) {
                this.nome = nome;
                this.email = email;
                this.descricao = descricao;
        }
```

Perceba que agora usamos algumas annotations da Bean Validation para deixar uma dica das restrições dos parâmetros necessários para a criação de um autor. Você poderia acrescentar comentários também.

Ouvi falar que deixar comentários ou pistas é ruim?

Talvez você já tenha ouvido falar algo assim: Se um código precisa de comentário então quer dizer que ele não está bem escrito. Então qual o motivo para projeto open source que sonha ser importante no planeta ter uma documentação muito boa? Inclusive os projetos open source da ZUP tem uma excelente documentação!

Deixar uma pista em forma de comentário, annotation ou alguma outra forma não significa que você vai explicar a implementação da coisa. Apenas você quer ajudar a próxima pessoa que pegar aquele código para que ela tenha mais facilidade de entendimento.

Essas validações de annotations vão ser executadas automaticamente?

Agora que eu deixei umas annotations de validação nos parâmetros do construtor, será que quando eu rodar o código abaixo a validação vai acontecer automaticamente?

```
        public Autor toModel() {
                return new Autor(this.nome,this.email,this.descricao);
        }
```

Deveria, mas infelizmente não. As annotations da Bean Validation não fazem parte do runtime padrão do Java e não existe nenhum hook do compilador para gerar o bytecode necessário para executar a validação. É realmente uma pena. Então, por agora, elas realmente servem para deixar dicas no código.

A complicação do nosso é código é proporcional a complicação da nossa feature

Todo software tem potencial para ficar complexo, é da natureza do software. Como bem descrito no excelente artigo No Silver Bullets, uma das dificuldades essenciais do desenvolvimento de software é que ele é um bicho mutante. Ao contrário da maioria dos produtos da terra, um software, n anos depois, é muito diferente da sua versão inicial. Agora pense: Quando você compra um carro, um computador etc o quão diferente eles são depois da anos na sua mão? Em geral, vai ser pouco diferente.

Como que um arquivo que nasce com 100 linhas, pode ter 500 depois de 4 meses de vida? Como que esse mesmo arquivo depois pode ir para 1000?

Inspirado eu nisso a sugestão deste pilar é que você mantenha o código o mais fácil de entender possível. Lembre que a nossa linha de design visa sempre diminuir a carga cognitiva e é importante que faça seu código viver sob regra.

Claro que a depender da necessidade de negócio, a versão final do seu código pode ter mais unidades, utilizar mais abstrações e qualquer outra técnica, mas ela precisa ser proporcional a complicação do requisito de negócio.

Usamos tudo que conhecemos que está pronto.

A versão completa dessa prática é: **Usamos tudo que conhecemos que está pronto. Só fazemos código do zero se for estritamente necessário.**

Aqui, muito melhor do que ficar explicando, é mostrar o código:

```
1     public class Pagamento {
2         //codigo omitido aqui
3
4         public void adicionaTransacoes(List<Transacao> transacoesGeradas) {
5             Assert.state(
6                     transacoes.stream()
7                             .noneMatch(transacao -> transacao
8                                     .temStatus(StatusTransacao.concluida)),
9                     "Não pode adicionar transacao quando já tem uma marcando que con\
10 cluiu");
11
12            Assert.state(this.transacoes.addAll(transacoesGeradas),
13                    "A transação sendo adicionada já existe no pagamento => "
14                            + transacoesGeradas);
15
16        }
17    }
```

A classe `Pagamento` é escrita em Java dentro de um projeto que utiliza o framework *Spring*. A classe `Assert` é do próprio *Spring* e está referenciada dentro do `Pagamento`, que faz parte do domínio da aplicação.

Uma outra versão deste mesmo código seria a seguinte:

```
    public class Pagamento {
        //codigo omitido aqui

        public void adicionaTransacoes(List<Transacao> transacoesGeradas) {
            if(transacoes.stream()
                            .anyMatch(transacao -> transacao
                                    .temStatus(StatusTransacao.concluida))){
                        throw new IllegalStateException("Não pode adicionar transaca\
o quando já tem uma marcando que concluiu");
                    }

            if(!this.transacoes.addAll(transacoesGeradas)){
                throw new IllegalStateException("A transação sendo adicionada já exi\
ste no pagamento => "
                            + transacoesGeradas);
            }

        }
    }
```

Os dois códigos fazem a mesma coisa e poderiam ser escritos, só que no primeiro temos as condicionais omitidas por um método pronto e no segundo temos elas feitas a mão. A depender da métrica de complexidade cognitiva derivada do CDD, você poderia ter algumas variações na pontuação.

- Considera que acoplamento com classes transversais conta ponto, então o uso do `Assert` conta 1 ponto de entendimento.
- Considera que acoplamento com classes transversais não conta ponto, então o uso do `Assert` não conta ponto.
- Considera que branch de código conta ponto, então cada `if` vai contar um ponto. No caso, teríamos 2 pontos a mais.

Vai ser complicado você achar uma métrica, independente se é derivada do CDD, que não leve em consideração o número de branches na verificação de complexidade.

Ou seja, por que eu vou optar por fazer um código mais complexo sendo que aquilo já está pronto e implementado dentro de uma base de código muito mais madura que a minha?

O medo do acoplamento do negócio com libs e frameworks

Geralmente o argumento para não acoplar nosso código de domínio com libs e frameworks é que queremos nos proteger de eventuais trocas desses componentes no futuro.

Caso o futuro reserve uma mudança, vai dar problema de compilação ou execução e trocamos. Será que precisamos ter medo da mudança mesmo? Como mencionamos é da natureza do software mudar de formas que não pensamos, então será que vale a pena mesmo ficar tentando colocar proteções?

Abrace o acoplamento com tudo que é maduro

Vários frameworks e bibliotecas são muito maduros. Tem tempo de estrada e equipes dedicas para a manutenção e evolução daquilo, é realmente uma frente de trabalho dentro de um negócio. Se aceitamos nos acoplar com frameworks caseiros, muito mais instáveis, por que não vamos nos acoplar com frameworks externos muito mais estáveis? Não parece uma decisão sensata.

Quanto menos você criar para buscar o objetivo, melhor. Quer dizer que você está chegando lá em cima de bases sólidas e só está investindo tempo no que realmente não existe.

Idealmente, todo código escrito deveria ser chamado por alguém.

A prática completa é escrita da seguinte forma: **Idealmente**, **todo código escrito deveria ser chamado por alguém**. **Se não tem ninguém chamando**, **ele não deveria existir**.

A ideia aqui é simples, não deveria existir classes, métodos, atributos, parâmetros e variáveis que não estão sendo utilizados. Tudo que você cria aumenta a dificulda de entendimento, então cada uma dessas coisas deveria merecer existir no sistema.

Evite geração de código desenfreada

Tanto pelas IDE's, quanto por bibliotecas, temos possibilidades muito interessantes de geração de código. E que fique claro: Você deve utilizar. O que você deveria evitar é simplesmente gerar porque deus quis e não refletir sobre aquilo.

Para mitigar código que não é utilizado por ninguém, você pode abraçar de vez a ideia de começar todo fluxo de código pelo ponto de entrada do dado. Fazendo isso, você maximiza a chance de só escrever o que foi necessário naquele fluxo.

E quando fluxos forem modificados?

Funcionalidades são criadas e extintas, então é normal acabarmos com código morto dentro do nosso sitema. É uma coisa certa, justamente pela natureza de mudança contínua do software.

A sugestão aqui é adotar como prática de desenvolvimento apagar código morto. Passou por algum lugar e percebeu que tal código não está invocado, apaga e segue a vida.

Só alteramos estado de referências que criamos.

A prática completa é: **Só alteramos estado de referências que criamos. Não mexemos nos objetos alheios. A não ser que esse objeto seja criado para isso, como é o caso de argumentos de métodos de borda mais externa. Estes são, geralmente, associados a frameworks.**

Esta prática tem muita influência do mundo funcional e tem a ver com imutabilidade. Enquanto que em linguagens funcionais todas as estruturas já nascem com a ideia de imutabilidade, nas outras linguagens a mutabilidade é cidadã de primeiro nível. Abaixo temos um exemplo do que queremos evitar:

```
1     public EarlyAlert create(@NotNull final EarlyAlert earlyAlert){
2         //mais codigo aqui para cima
3
4         final Person person = earlyAlert.getPerson();
5
6                 if (person.getCoach() == null
7                                 || assignedAdvisor.equals(person.getCoach().getId())) {
8                         person.setCoach(personService.get(assignedAdvisor));
9                 }
10
11        //mais codigo aqui para baixo
12    }
```

Este código foi extraído de um projeto open source, público no github.

O método recebe como argumento uma referência para um objeto do tipo `EarlyAlert`. Dentro do método, ele recupera uma outra referência para o tipo `Person` que, neste contexto, representa um estudante. Neste momento foque só no que o código faz e não se ele poderia ser escrito de outra forma :).

Perceba que caso a condição testada no `if` retorne `true`, a referência para o objeto do tipo `Person`, que foi extraída da objeto do tipo `EarlyAlert`, vai ser alterada.

E qual o problema que podemos ter aqui? Este trecho de código, é invocado a partir deste outro ponto aqui.

```
    private boolean createEarlyAlert(Person person, SuccessIndicator    successIndic\
ator) {
        //um novo EarlyAlert é criado
        EarlyAlert earlyAlert = new EarlyAlert();
        earlyAlert.setPerson(person);
        earlyAlert.setCampus(getCampus(person));
        earlyAlert.setCourseTermCode(getCurrentOrNextTerm());

        //chamou o método que pode alterar a Person que foi definida no
        //EarlyAlert
        earlyAlertService.create(earlyAlert);

        //você não tem garantia que a person retornada é a mesma de antes
        //da chamada do método e nem se as informações da Person são as //////mesmas.
        Person person2 = earlyAlert.getPerson()
    }
```

Alterar referências alheias complica todo acompanhamento do estado

Uma vez que o código do seu sistema aceita que referências de origem desconhecida sejam alteradas, você perde completamente a habilidade de "*trackear*" estado pela aplicação. Um objeto pode ter seu estado alterado em qualquer lugar e, a partir daí, seu **encapsulamento cai como um castelo de cartas**. O tempo inteiro você vai precisar entrar nos métodos que está chamando para verificar se alguém mexendo no objeto que você criou.

Você deve evitar com todas as suas forças alterar referências criadas em outros lugares. **Só quem mexe numa referência é o ponto do código que a criou**.

Como solução alternativa, deixe dicas de que algo foi alterado

O método `create`, que altera o estado do objeto do tipo `EarlyAlertService` poderia deixar uma dica para quem chama, informando que aquele objeto não é mais o mesmo.

```
    public Modified<EarlyAlert> create(@NotNull final EarlyAlert earlyAlert){
        //mais codigo aqui para cima

        final Person person = earlyAlert.getPerson();

                if (person.getCoach() == null
                                || assignedAdvisor.equals(person.getCoach().getId())) {
                        person.setCoach(personService.get(assignedAdvisor));
                }

        //mais codigo aqui para baixo
        return new Modified(earlyAlert);
    }
```

Agora o retorno do método deixa claro que ele mexe no que não é da conta dele. Dessa forma, o ponto do código que realiza a invocação já tem uma ideia do que está acontecendo. E quando você olha para o fluxo como um todo, também consegue ter uma ideia dos pontos que alteram estado de referências desconhecidas.

Também deixe dicas sobre alterações em sistemas externos

Agora que abraçamos a ideia de não alterar referências alheias inspirado pelo princípio da imutabilidade, podemos tentar levar a mesma ideia para alterações de sistemas externos, como banco de dados e outras aplicações que possamos usar.

O gargalo da imutabilidade pura para a maioria dos sistemas, é que em algum ponto ele é mutável. Por exemplo, o seu banco de dados. Todos os bancos de dados famosos do planeta são mutáveis e isso deveria ser levado em conta dentro da nossa prática. Pegue esse código como exemplo:

```
        public Pagamento executa(Long idPedido,
                        @Valid NovoPagamentoOnlineRequest request) throws BindException {

        Pagamento novoPagamento = request.toPagamento(idPedido, valor,
        manager);
                Pagamento novoPagamentoSalvo = executaTransacao.commi(novoPagamento);

        return novoPagamentoSalvo;
                }

        }
```

Este método é chamado daqui:

```
        @PostMapping(value = "/pagamento/online/{idPedido}")
        public void paga(@PathVariable("idPedido") Long idPedido,
                        @RequestBody @Valid NovoPagamentoOnlineRequest request)
                        throws Exception {
                Pagamento novoPagamento = iniciaPagamento.executa(idPedido,
                                request);
        }
```

Como podemos saber que o `Pagamento` retornado foi salvo no banco de dados? Ou salvo em qualquer outro lugar? Não tem como saber. Ou você entra no método para entender, ou corre o risco de até salvar duas vezes.

Aqui temos um ponto do código, diferente do ponto da entrada de dados, alterando sistemas externos. O contexto muda, mas a regra permanece: **só a entrada do dado deveria alterar sistemas externos**. Caso não seja possível abraçar a ideia, como o exemplo acima, você pode usar a ideia das dicas de código de novo.

```
        public Persisted<Pagamento> executa(Long idPedido,
                        @Valid NovoPagamentoOnlineRequest request) throws BindException {

        Pagamento novoPagamento = request.toPagamento(idPedido, valor,
        manager);
                Pagamento novoPagamentoSalvo = executaTransacao.commi(novoPagamento);

        return new Persisted(novoPagamentoSalvo);
                }

        }
```

Agora o ponto do sistema que invoca este método sabe que vai receber um objeto persistido. Com clareza sobre essa informação, você diminui as chances de refazer algo que já aconteceu.

Explicite a dificuldade

Nem sempre fácil manter tudo organizado e está tudo bem. O seu código precisa ser honesto em relação as dificulddes e deixar isso claro para a próxima pessoa. Lembre que ainda estamos na era que seres humanos mantém o código e, enquanto isso durar, precisamos construir sistemas que executem em máquinas e sejam possíveis de entender por pessoas.

A versão mais eficiente de um(a) dev desenvolve o que foi combinado

A prática completa é esta: **A versão mais eficiente de uma pessoa programando é aquela que entende, questiona e implementa estritamente o que foi combinado. Não inventamos coisas que não foram pedidas, não fazemos suposição de funcionalidade e nem caímos na armadilha de achar que entendemos mais do que a pessoa que solicitou a funcionalidade.**

Esse é um pilar mais comportamental do que técnico. Uma pessoa que desenvolve reune habilidades necessárias para automatizar desejos de clientes que gostariam de abandonar algum determinado fluxo manual.

Todo software é a automação de algum processo que poderia ser feito manualmente. E aqui você pode ir para os vários de tipos de projetos.

- Uma biblioteca automatiza um determinado fluxo de código que todo(a) dev teria que fazer repetidamente;
- Um framework automatiza diversos fluxos de código que todo(a) dev teria que fazer repetidamente;
- Um software para um cliente final automatiza um fluxo manual que deve ter deve ter ficado muito complicado ou que já nasce tão complicado que precisa de automação;
- Insira qualquer outro fluxo aqui :);

Você precisa entender exatamente o que é necessário

No livro Domain Driven Design, Eric Evans traz nos primeiros capítulos a importância de sermos ótimos "digestores" de domínio. Eu diria que precisamos também ser ótimos "digestores" de funcionalidades. Além de entender o contexto que você está inserido(a), você precisa entender o motivo de cada uma das funcionalidades que aparecerão na sua frente.

Quando pensamos neste entendimento, existem alguns fatores que podem ser levados em consideração e que podem ficar na sua mente:

- Por que vamos fazer aquilo? Aqui você pode tentar fazer uma análise de causa raiz[35] para verificar se existe realmente aquela necessidade.
- Em geral apenas um porque tende a ser superficial. Busque pelo menos o segundo :).

[35] http://www.ammainc.org/wp-content/uploads/2013/02/Root_Cause.pdf

- Dado que você sabe o motivo, busque saber quais são as restrições.
  * Tem requisito de performance?
  * Tem requisito de escalabilidade?
  * Quem pode acessar isso?
  * Preciso falar com alguém para liberar algo?
  * Quaisquer outras possíveis restrições que venham na sua cabeça.
- Qual é exatamente o resultado final esperado?
- Questione sempre que você discordar ou achar que tem algo que pode ser feito melhor em relação ao que está sendo proposto;

Como podemos automatizar um fluxo se a gente não tiver conseguido entender o desejo de automação direito? Simplesmente é um risco desnecessário.

Implemente o que foi combinado

Chegamos no cenário onde você entendeu exatamente o que foi pedido, dissecou a funcionalidade, questionou, debateu e, junto com as pessoas envolvidas, chegou a conclusão que precisa mesmo fazer aquilo.

Uma vez que tiver neste ponto lembre de implementar exatamente o que foi pedido. Utilize suas técnias de desenvolvimento de código para fazer um código suficiente para a funcionalidade, mantendo a dificuldade de entendimento baixa.

Não faça mais nem menos do que foi combinado. De novo, se você, no meio da implementação, perceber qualquer coisa que você acha que poderia ser melhor ou até que está errado, vai lá e fale com as pessoas envolvidas.

Devs eficientes não inventam fluxos, não ficam imaginam que o cliente vai mudar de ideia e nem fazem códigos baseados na esperança que aquilo um dia vai ser útil. Devs eficientes olham de maneira criteriosa para a funcionalidade, descobrem o valor daquilo para o negócio e fazem código que suporta aquele desejo.

Dev não é um artista?

Muito se fala que a profissão de desenvolvedor(a) é muito criativa e que pode ser comparada com pinturas, criar uma música etc. É importante que a gente se lembre que quadros e músicas também são encomendados. E quando um cliente encomenda um quadro ou uma música, a pessoa que vai produzir precisa entender exatamente o objetivo, os motivos e sair do outro lado com o resultado esperado.

Ser comparado(a) a um(a) artista não quer dizer que fazemos o que queremos, ser comparado(a) a um(a) artista significa que temos muitas maneiras de resolver um mesmo problema e podemos usar nosso repertório para chegar lá.

Você precisa entender o que está usando e olhar sempre o lado negativo de cada decisão.

Por mais clichê que isso seja, continua sendo verdade. Quase tudo na vida tem lado positivo e negativo. E aí você busca sempre tomar decisões que tenham mais coisas positivas do que negativas. É até óbvio, convenhamos.

Quando pensamos em desenvolvimento de software, tomamos várias decisões das mais variadas escalas. É mais do que necessário sermos capazes de analisar sempre que o estamos deixando na mesa para cada uma delas. Vamos tentar pegar alguns cenários.

Decidi adotar CDD como linha de design de código

A teoria proposta pelo CDD(Cognitive Driven Development) diz que você deve buscar dividir as responsabilidades do código pela linha do entendimento humano. Ela parece promissora e tudo mais, mas existem alguns pontos de atenção:

- Quantos projetos já foram desenvolvidos apoiados nela?
- Será que quando o sistema ficar maior, vamos mesmo conseguir controlar a crescente de complexidade por unidade?
- Dependendo da métrica, podemos ter muitas unidades de código gerada. Será que isso realmente vai facilitar?

Decidi utilizar uma inspiração arquitetural com muitas camadas

Como já falamos, um software pode ser escrito de diversas maneiras diferentes. Existem estratégias arquiteturais que buscam facilitar a troca de componentes do software que não são intimamente conectadas com o código que resolve o negócio em si. Algumas possíveis peças que você pode querer trocar:

- Framework web;
- Framework de acesso a banco de dados;

- Biblioteca que realiza requisições externas;
- Protocolo utilizado para entregar os dados de entrada da aplicação;
- Execução síncrona para assíncrona;

Você pode preparar seu código para qualquer uma das possíveis mudanças listadas acima. Será que conseguimos olhar alguns pontos de atenção?

- Toda indireção aumenta complexidade, como vou lidar com isso?
- E se eu nunca quiser trocar tais componentes?
- Será que viver sob esse conjunto de abstração não vai deixar minha produção de código mais lenta?

Decidi utilizar uma arquitetura distribuída

Vamos supor que sua equipe pretende criar ou evoluir um software agora apostando numa arquitetura distribuída, pouco importa qual. Quais são os possíveis pontos de atenção, será que conseguimos listar alguns?

- Como vou lidar com o fato de uma operação agora acontecer sequencialmente em vários sistemas diferentes?
- Vou me preocupar com conceitos de transação distribuída, preciso?
- Como vou lidar com a lentidão que pode acontecer em algum ponto da comunicação?
- Será que o ganho de flexibilidade que vou ter, supera a provável perda de performance por conta da comunicação distribuída?
- Como vou lidar com os contratos de borda?

Conseguir olhar para os pontos de atenção é uma restrição

Se você não for capaz de olhar para os pontos de atenção/negativos, talvez você não deva optar por aquele caminho naquele momento. Caso opte, é importante ter na mente que você está indo por um caminho que não sabe dos perigos e isso pode afetar sua visão de futuro, antecipação etc.

Quantos pontos de atenção eu devo levantar?

Pense aqui como se fosse uma análise de causa raiz. Quanto mais pontos de atenção, melhor. Quer dizer que você realmente tem uma visão crítica sobre aquela decisão e parece estar preparado(a) para lidar com um leque maior de situações que podem aparecer no futuro.

API's não democráticas

O pilar completo é este: **A sua api deve deixar claro o caminho que deve ser seguido pelo ponto do código que decide usá-la. Não espere que ninguém lembre de invocar nada. Faça de tudo para gerar obrigações. Quanto mais específico é seu código, menos democrático ele é.**

Aqui partimos do princípio que o código é um bicho mutante, que está em constante evolução e sendo mantido por pessoas diferentes ao longo do tempo.

No mundo ideal manter um código deveria ser tipo manter uma geladeira, um carro, um fogão ou algo do genêro. Todo mundo que entende do assunto é capaz de manter aquele produto sem muito estresse.

Só que a realidade é bem diferente. Enquanto uma geladeira tende a ter os componentes implementados do mesmo jeito depois de um certo tempo, um código, muitas vezes, nem tem mais aquele componente ou ele já mudou tanto que o conhecimento anterior passa sobre aquilo passa a ter menos peso.

O perigo da falta de estabilidade dos contratos

Vamos olhar para o seguinte exemplo de código. Tente prestar bastante atenção no uso do objeto do tipo `Person`.

```
1         public void ensureValidAlertedOnPersonStateOrFail(Person person)
2                         throws ObjectNotFoundException, ValidationException {
3
4                 if ( person.getObjectStatus() != ObjectStatus.ACTIVE ) {
5                         person.setObjectStatus(ObjectStatus.ACTIVE);
6                 }
7
8                 final ProgramStatus programStatus =  programStatusService.getActiveStatus();
9
10                if ( programStatus == null ) {
11                        throw new ObjectNotFoundException(
12                                        "Unable to find a ProgramStatus representing \"activeness\"
13                                        "ProgramStatus");
14                }
15
16                Set<PersonProgramStatus> programStatuses =
17                                person.getProgramStatuses();
```

```
18                  //1
19                  if ( programStatuses == null || programStatuses.isEmpty() ) {
20                          PersonProgramStatus personProgramStatus = new PersonProgramStatus();
21              //codigo omitido
22                  }
23
24                  //1
25                  if ( person.getStudentType() == null ) {
26              //codigo omitido
27                          person.setStudentType(studentType);
28                  }
```

O código acima usa 5 métodos públicos da classe `Person`, mas ela tem mais de 70 métodos públicos!

Enquanto que a classe `Person` é transversal ao sistema todo, este método é específico para um fluxo. Classes transversais e construídas dentro da própria aplicação tendem a sofrer mais alterações do que pontos específicos.

Além disso da possibilidade de sofrer mais alterações, tem o fato do método `ensureValidAlertedOnPersonStateOrFa` pedir um argumento com mais de 70 métodos públicos enquanto ele só usa 5. Existe a chance do outro ponto do código estar carregando um objeto `Person` para a memória, com todos seus relacionamentos e tudo mais.

Quando você for escrever um teste automatizado também vai precisar criar o objeto completo.

A única forma do ponto de código que invoca este método não carregar o objeto completo seria se ele conhecesse a implementação. E se a implementação mudar? Você já sabe que software muda, então não parece uma decisão sensata achar que aquilo não vai mudar.

Precisamos de interfaces específicas e estáveis

Uma outra versão, do mesmo código, poderia ser essa:

```
1           public void ensureValidAlertedOnPersonStateOrFail(PossibleValidAlertedPerson person)
2                   throws ObjectNotFoundException, ValidationException {
3
4                   if ( person.getObjectStatus() != ObjectStatus.ACTIVE ) {
5                           person.setObjectStatus(ObjectStatus.ACTIVE);
6                   }
7
8                   final ProgramStatus programStatus =  programStatusService.getActiveStatus();
9
10                  if ( programStatus == null ) {
11                          throw new ObjectNotFoundException(
```

```
12                                      "Unable to find a ProgramStatus representing \"activeness\"
13                                      "ProgramStatus");
14              }
15
16              Set<PersonProgramStatus> programStatuses =
17                              person.getProgramStatuses();
18              //1
19              if ( programStatuses == null || programStatuses.isEmpty() ) {
20                      PersonProgramStatus personProgramStatus = new PersonProgramStatus();
21          //codigo omitido
22              }
23
24              //1
25              if ( person.getStudentType() == null ) {
26          //codigo omitido
27                      person.setStudentType(studentType);
28              }
```

Perceba que só foi modificado o tipo do argumento de entrada, o resto ficou igual. `PossibleValidAlertedPerson` pode ser uma interface com apenas os 5 métodos necessários para aquele método. O que ganhamos?

- O ponto de invocação sabe exatamente o que precisa ser passado;
- A interface é específica para o método e, consequentemente, mais estável;
- Escrever testes automatizados tende a ser mais fácil, já que agora você precisa montar algo muito menor;

Aqui é uma técnica de orientação objetos que ficou famosa através do *SOLID*. Usamos a letra *I* que é o princípio da segregação pela interface. A ideia é criar contratos específicos para métodos, fazendo a aposta que essa espeficidade vai gerar um contrato mais estável e manutenível no futuro.

O ponto de atenção é que, pela métrica mais usada derivada do CDD, todo acoplamente específico conta ponto. Então você vai precisar usar a técnica com parcimônia.

Não usamos exception para controle de fluxo.

Exceptions foram criadas para que casos não imaginados possam ser tratados e um sinal enviado para o programa que ele deve interromper o fluxo de execução.

```
1         protected BeanWrapper createBeanWrapper() {
2                 if (this.target == null) {
3                         throw new IllegalStateException("Cannot access properties on null bean insta
4 " + getObjectName() + "'");
5                 }
6                 return PropertyAccessorFactory.forBeanPropertyAccess(this.target);
7         }
```

Acima temos um código retirado de uma versão do framework Spring, um método da classe `BeanPropertyBindingResult`. A classe tem um atributo que, no momento da invocação do método `createBeanWrapper` nunca deveria estar nulo. Mas é claro que existe uma diferença entre o que a gente quer e o que acontece de fato, então o código faz uso de técnicas de programação defensiva e verifica se o atributo `target` ainda está nulo. No caso da variável estar nula, temos uma situação estranha e não queremos que o código continue, por isso lançamos uma *exception*.

Perceba, literalmente queremos que aquele fluxo seja terminado, encontramos um **bug** e não vamos deixar nosso sistema operar sobre uma situação que pode gerar problemas inesperados.

Uma *exception* é como um *goto*, um hack da linguagem poderoso para você conseguir quebrar o fluxo de execução de qualquer ponto.

Efeito colateral #1: Talvez custe performance

Na linguagem Java, por exemplo, sempre que você cria uma instância de `Exception` olha o que acontece lá por baixo dos panos:

```
public class Throwable {
 public Throwable() {
    fillInStackTrace();
 }

 private native Throwable fillInStackTrace(int dummy);
}


public class Exception extends Throwable{
    public Exception(){
        //chama o construtor de Throwable
        super();
    }
}
```

Toda a pila de execução é colocada dentro da nova instância, o que pode ser custoso para sua aplicação caso a exception seja lançada múltiplas vezes por intervalo de tempo. Quando mais profundo é o ponto de lançamento da exception, maior vai ser a pilha de execução criada.

Claro que as máquinas virtuais e ambientes de execução em geral vão evoluindo muito com o tempo, mas com certeza é um ponto de atenção.

Efeito colateral #2: Ponto de tratamento do problema

Como a *exception* é um hack da linguagem, literalmente um *goto*, você pode tratar aquilo onde bem entender. Por exemplo, vários frameworks fornecem mecanimos para você tratar exceptions de maneira genérica.

```
@ExceptionHandler(BindException.class)
public ValidationErrorsOutputDto handleValidationError(BindException exception) {

    // codigo omitido
}
```

No Spring MVC você pode criar um método, adicionar a configuração dizendo que você quer capturar toda *exception* que seja igual ou filha de `BindException` e fazer o que quiser a partir dali. Ou seja, em qualquer ponto do seu código que tenha sido invocado em função da execução de um método de controller, você pode lançar essa exception e ela vai cair aqui.

E aí temos um caso clássico do mercado:

```
    @ExceptionHandler(BusinessException.class)
    public ValidationErrorsOutputDto handleBusinessException(BusinessException excep\
tion) {

        // codigo omitido
    }
```

Situações esperadas dentro do fluxo de negócio, que poderiam ser tratadas com retornos convencionais, lançam exceptions para pular de um ponto de código para outro.

Efeito colateral #3: Aumento de complexidade de entendimento

Cada classe de *exception* criada aumenta um ponto de complexidade no sistema como um todo e o seu uso aumenta um ponto de complexidade naquela unidade de código em específico.

Além disso, se tem exception, pode ter `try`, `catch` e talvez um `finally`, o que traz mais três branches de código para sua análise. O que também, dada a nossa métrica derivada do CDD, aumenta a complexidade de entendimento.

Efeito colateral #4: No mundo das exceptions de runtime, perdemos clareza

Hoje em dia, mesmo no Java, o uso de *exceptions* ficou restrito ao tipo de que não força um tratamento explícito no código. Dessa forma criamos várias situações onde os possíveis retornos não ficam claros. Abaixo, temos um exemplo clássico:

```
        public void abateEstoque(@Positive int quantidade) {
                //uma situação inesperada
                Assert.isTrue(quantidade > 0,
                                "Olha, não podemos abater quantidade menor ou igual a zero "
                                                + quantidade);

                //situacao esperada. Pq se não fosse esperada, não teria o if
                if(this.estoque < quantidade) {
                        throw new EstoqueException("Acabou o estoque");
                }
                this.estoque -= quantidade;
        }
```

O ponto do código que invoca o método acima não tem como saber que tal invocação pode lançar uma *exception*. Claro que podíamos ter deixado uma dica:

```
    /**
    ** @throws EstoqueException quando não tiver estoque suficiente
    **/
        public void abateEstoque(@Positive int quantidade) {
                //uma situação inesperada
                Assert.isTrue(quantidade > 0,
                                "Olha, não podemos abater quantidade menor ou igual a zero "
                                                + quantidade);

                //situacao esperada. Pq se não fosse esperada, não teria o if
                if(this.estoque < quantidade) {
                        throw new EstoqueException("Acabou o estoque");
                }
                this.estoque -= quantidade;
        }
```

A dica ajudaria, mas sempre que for possível tratar a situação com uma trava de compilação.

Especificamente na situação acima, seria até simples. Basta que o método retorne *boolean*.

```
    /**
    ** @return true se conseguir abater o estoque.
    **/
        public boolean abateEstoque(@Positive int quantidade) {
                //uma situação inesperada
                Assert.isTrue(quantidade > 0,
                                "Olha, não podemos abater quantidade menor ou igual a zero "
                                                + quantidade);

                //situacao esperada. Pq se não fosse esperada, não teria o if
                if(this.estoque < quantidade) {
                        return false;
                }
                this.estoque -= quantidade;
        return true;
        }
```

Não que já esteja 100% claro, dado que um *boolean* é um tipo padrão da linguagem Java e a semântic que ele carrega é de true ou false e isso, não necessariamente indica que o estoque foi abatido ou não. Só que aí juntamos com uma dica e fica um pouco melhor.

Possibilidade de utilizar retornos explícitos

Agora algumas situações hipóteticas para pensarmos:

- E se a gente também precisasse retornar um objeto representando a ideia de estoque abatido para indicar a hora exata da operação?
- E se a também gente precisasse ter um retorno indicando que o estoque zerou?
- E se a gente precisasse combinar os dois requisitos de cima?

Para estas necessidades o retorno booleano deixaria de ser suficiente. Agora temos uma situação onde precisamos retornar algo em caso positivo e outra coisa no caso negativo. Talvez cair para a *exception* esteja passando na sua mente. Tudo bem se tiver. Dessa vez vamos por um caminho diferente, vamos nos inspirar numa abstração da linguagem Scala chamada `Either` que, na verdade, é inspirada na abstração de mesmo nome da linguagem Haskell[36].

A ideia é que você a gente possa construir um objeto que tenha dois possíveis retornos, uma para sucesso e outro para erro. Em função disso, podemos chegar nesse código aqui:

```
1         public ResultadoAbateEstoque abateEstoque(@Positive int quantidade) {
2                 //uma situação inesperada
3                 Assert.isTrue(quantidade > 0,
4                                 "Olha, não podemos abater quantidade menor ou igual a zero "
5                                                 + quantidade);
6
7                 //situacao esperada. Pq se não fosse esperada, não teria o if
8                 if(this.estoque < quantidade) {
9                         return ResultadoAbateEstoque.vazio();
10                }
11                this.estoque -= quantidade;
12
13                if(this.estoque == 0) {
14                        return ResultadoAbateEstoque.estoqueNoLimite(this,quantidade);
15                }
16                return ResultadoAbateEstoque.estoqueAbatido(this,quantidade);
17        }
```

A classe `ResultadoEstoqueAbatido` fica assim:

[36] https://hackage.haskell.org/package/base-4.14.0.0/docs/Data-Either.html

```
1  public class ResultadoAbateEstoque {
2
3          private boolean vazio;
4          private Produto produto;
5          private @Positive int quantidade;
6          private boolean estoqueNoLimite;
7
8          private ResultadoAbateEstoque() {
9                  this.vazio = true;
10         }
11
12         private ResultadoAbateEstoque(Produto produto, @Positive int quantidade,boolean est\
13 oqueNoLimite) {
14                 this.produto = produto;
15                 this.quantidade = quantidade;
16                 this.estoqueNoLimite = estoqueNoLimite;
17
18         }
19
20         public static ResultadoAbateEstoque vazio() {
21                 return new ResultadoAbateEstoque();
22         }
23
24         public static ResultadoAbateEstoque estoqueNoLimite(Produto produto,
25                         @Positive int quantidade) {
26                 return new ResultadoAbateEstoque(produto,quantidade,true);
27         }
28
29         public static ResultadoAbateEstoque estoqueAbatido(Produto produto,
30                         @Positive int quantidade) {
31                 return new ResultadoAbateEstoque(produto,quantidade,false);
32         }
33
34         public boolean taVazio() {
35                 return vazio;
36         }
37
38         public boolean taNoLimite() {
39                 return this.estoqueNoLimite;
40         }
41
42         public EstoqueAbatido get() {
43                 Assert.isTrue(vazio,"Você não deveria buscar o estoque porque na verdade nao parec\
```

```
44  e ter estoque");
45                  return new EstoqueAbatido(produto, quantidade);
46          }
47
48  }
```

Agora, no ponto de invocação do método que abate o estoque você tem um retorno explícito e pode operar nele como quiser. A complexidade do código como um todo, pode até ficar parecida com o uso das exceptions, já que vamos ter controle de fluxo com o uso de `if`.

Só que agora ganhamos retorno explícito e temos usamos a compilação para indicar os possíveis caminhos do código. Entra aqui também a ideia de API's não democráticas.

Resumo da ópera

A parte boa do código é que podemos resolver o mesmo problema de N maneiras. A parte ruim do código é que podemos resolver o mesmo problema de N maneiras.

Neste exato momento você tem mais uma ferramenta para usar na busca por soluções. Sabendo sempre o lado positivo e negativo de cada solução, fica muito mais fácil de tomar uma decisão.

Regras de negócio devem ser declaradas de maneira explícita na nossa aplicação.

Aqui é um pilar mais simples, porém importante. Nas nossas aplicações temos requisitos que são funcionais e aqueles não funcionais. Os funcionais, em geral, estão relacionadas as lógicas que precisam ser automatizadas dentro da aplicação.

Imagine o cenário onde precisamos enviar emails entre outras coisas após o processamento de uma compra.

```
        private void processa(Long idCompra,RetornoGatewayPagamento retornoGatewayPagamento\
) {
                Compra compra = manager.find(Compra.class, idCompra);
                compra.adicionaTransacao(retornoGatewayPagamento);
                manager.merge(compra);

        }
```

A depender das tecnologias sendo usadas para suportar a construção da aplicação, você pode ter algumas maneiras de resolver isso. Por exemplo, o código acima é escrito em Java e utiliza o Hibernate e o Spring como frameworks de sustentação. Poderíamos ter o seguinte código:

```
    public class Compra {
        //atributos e outros métodos omitidos

        @PostUpdate
        public void postUpdate(){
            ApplicationContext ctx = //carrega o objeto do spring aqui
            ctx.publish(new EventoPagamentoProcessado(this));
        }
    }
```

O código exibido acima pode causar estranheza por conta do acesso a objetos do Spring dentro do contexto de domínio. Se você já leu o pilar relativo a **Usamos tudo que conhecemos que está pronto**, vai lembrar que não temos medo de nos acoplarmos com frameworks. Esta mesma prática é suportada em outras linguagens com outros frameworks, como Ruby e Rails e Typescript com NestJS :). **O problema é a falta de clareza no fluxo de negócio**.

Quando olhamos para o código de fluxo de novo, como vamos saber que existe um fluxo de processamento de pagamento?

```
        private void processa(Long idCompra,RetornoGatewayPagamento retornoGatewayPagamento\
) {
                Compra compra = manager.find(Compra.class, idCompra);
                compra.adicionaTransacao(retornoGatewayPagamento);
                manager.merge(compra);
        }
```

Geralmente aceitamos a falta de clareza em requisitos que não tem a ver com a regra de negócio em si.

- É necessário logar em modo de tracing toda entrada e saida de método;
- É necessário adicionar um correlation id para toda chamada num ecossistema de microserviços
- É necessário verificar o tempo todo se o usuário está logado para acessar tal ponto;
- É necessário fazer profile para pegar tempo de execução de métodos;

Dando um passo na clareza do fluxo de negócio

O mesmo fluxo de negócio citado poderia ser escrito da seguinte forma:

```
        private void processa(Long idCompra,RetornoGatewayPagamento retornoGatewayPagamento\
) {
                Compra compra = manager.find(Compra.class, idCompra);
                compra.adicionaTransacao(retornoGatewayPagamento);
                manager.merge(compra);
        applicationContext.publish(new EventoPagamentoProcessado(compra));
        }
```

Agora está claro que existe um processamento para compras que tiveram seus pagamentos processaos por algum gateway. A pergunta que fica é: quais processamentos?

O ponto de atenção do acoplamento extramente fraco

Sempre falamos de acoplamento fraco e coesão forte. Só que de vez em quando deixamos o acoplamento tão fraco que não conseguimos nem saber onde é que estão as coisas.

```
        private void processa(Long idCompra,RetornoGatewayPagamento retornoGatewayPagamento\
) {
                Compra compra = manager.find(Compra.class, idCompra);
                compra.adicionaTransacao(retornoGatewayPagamento);
                manager.merge(compra);
        applicationContext.publish(new EventoPagamentoProcessado(compra));
        }
```

Neste código, quem processa o `EventoPagamentoProcessado`? Um caso parecido acontece quando usamos brokers de mensageria. Só que ela estamos apostando em ganhar tratamento assíncrono, escalabilidade e a resiliência provida pelo broker. Aqui estamos apenas utilizando uma abstração simples do próprio framework, no caso o Spring.

Explicitando o acoplamento

Para este tipo de caso, pode ser útil deixar o código um pouco mais acoplado e criar sua própria abstração.

```
        private void processa(Long idCompra,RetornoGatewayPagamento retornoGatewayPagamento\
) {
                Compra compra = manager.find(Compra.class, idCompra);
                compra.adicionaTransacao(retornoGatewayPagamento);
                manager.merge(compra);
        eventosNovaCompra.processa(compra);
        }

    @Service
    public class EventosNovaCompra {

        @Autowired
        private Set<EventoCompraSucesso> eventosCompraSucesso;

        public void processa(Compra compra) {

            if(compra.processadaComSucesso()) {
                eventosCompraSucesso.forEach(evento -> evento.processa(compra));
            }
            else {
                eventosCompraFalha.forEach(evento -> evento.processa(compra));
            }
        }
```

```
24
25      }
```

No ponto do fluxo, mantemos a complexidade igual, já que trocamos a referência para `EventoPagamentoProcessado` por `EventosNovaCompra`. Só que agora temos um caminho claro que pode ser seguido para você a próxima pessoa saber exatamente onde a próxima lógica está acontecendo.

Deixando claro, nossa solução de eventos caseira, principalmente do ponto de vista de resiliência, é frágil também. Só que deixamos a complexidade similar no ponto de invocação e ganhamos clareza, o que pode ser interessante. Nenhuma das duas, porém, se compara em resiliência e escalabilidade com a utilização de um broker de mensageria mais robusto.

Favorecemos a coesão através do encapsulamento.

Operações sobre dados cujo tipo é padrão da linguagem devem ficar dentro das nossas classes.

Lá na raiz da orientação a objetos, no paper Programming with Abstract Data Types, é explicado que a ideia de uma classe nasce da necessidade de representar um tipo que não é suportado pela linguagem de programação em si. Esse tipo é construído em cima dos tipos que já existem ou talvez em cima de outros tipos criados dentro do sistema, o que conhecemos como atributos.

Além disso também é explicado que agora esses tipos abstratos de dados devem ter funções que manipulam os seus próprios atributos, o que conhecemos como métodos. Toda operação sobre os atributos deve residir dentro da própria classe. Neste ponto está sendo sustentado talvez o principal motivo de criarmos um tipo abstrato de dados (classes): Ter um local onde restringimos o uso daquelas variáveis para aquelas operações. Estamos falando de juntar estado e comportamento. Essa junção tende a promover a coesão.

Quando bem utilizado, o encapsulamento, além de promover a coesão, promove um jeito interessante de distribuir a dificuldade de entendimento entre várias das classes.

A pergunta que fica é: se unir é estado e comportamento é algo que pode ajudar no entendimento e também que está na base da criação da orientação a objetos, qual o motivo dos códigos não tirarem proveito ? A minha hipótese é que estamos esquecendo do jeito lógico de fazer a análise.

Exemplo #1: Lógica de consulta de estado do objeto

```
1       final Bolao bolao = bolaoRepository.findById(this.idBolao).get();
2       if (bolao.getDataExpiracao().isBefore(Instant.now())) {
3           return ResultadoConfirmacao.conviteExpirado();
4       }
```

Aqui temos uma lógica de verificação de data sobre um estado do objeto do tipo `Bolao`.

O mesmo código poderia ter sido escrito da seguinte forma:

```
final Bolao bolao = bolaoRepository.findById(this.idBolao).get();
    if (bolao.aceitaConvite()) {
        return ResultadoConfirmacao.conviteExpirado();
    }
```

Pode ser que seja necessário deixar o instante flexível. Para isso usamos os parâmetros.

```
    final Bolao bolao = bolaoRepository.findById(this.idBolao).get();
    if (bolao.aceitaConvite(Instant.now())) {
        return ResultadoConfirmacao.conviteExpirado();
    }

    public class Bolao {
        //codigo omititdo

        public boolean aceita(Instant data) {
            return this.dataExpiracao.before(data);
        }
    }
```

Quando você concentra toda lógica sobre um tipo padrão da linguagem que também representa o estado dentro da classe daquele objeto você aumenta as chances de distribuir a complexidade do código respeitando os limites de entendimento que você definiu derivado do CDD.

Por que você está falando sobre o tipo padrão da linguagem?

Porque essa é uma forma nítida de você ter um norte. No exemplo acima, DateTime é um tipo padrão do Java, então você concentra a lógica sobre ela dentro do seu tipo abstrato de dados(sua classe).

Só que como `DateTime` também é um tipo abstrato de dados, aplicamos a mesma regra de concentrar a lógica sobre tipos fundamentais da linguagem dentro da classe que é dona deles. Perceba que não foi acessada a informação que guarda o instante no tipo `DateTime` para analisar se era anterior ao instante atual. Delegamos a chamada para um método pronto na própria classe `DateTime`. Ou seja, a mesma regra foi aplicada.

Um outro exemplo onde podemos podemos fazer a mesma análise.

```
    public Set<PaymentMethod> filterDesiredPaymentMethods(User user) {
        return this.paymentMethods.stream()
            .filter(paymentMethod -> user.getDesiredPaymentMethods().contains(paymen\
tMethod))
            .collect(Collectors.toSet());
    }
```

Perceba que na linha `user.getDesiredPaymentMethods().contains(paymentMethod))` é acessado diretamente o tipo que referencia uma coleção e aí em seguida acessamos um método pronto. O problema é que esse tipo vive no `User`.

Aqui basta que seja aplicada a mesma regra: Operações sobre dados cujo tipo é padrão da linguagem devem ficar dentro das classes dos objetos. Neste caso, a coleção retornada pelo método `getDesiredPaymentMethods` é padrão da linguagem. Uma versão diferente do código pode ser a seguinte:

```
    public Set<PaymentMethod> filterDesiredPaymentMethods(User user) {
        return this.paymentMethods.stream()
            .filter(user :: acceptsPayment)
            .collect(Collectors.toSet());
    }
```

Agora você tem um método no `User` que opera sobre o estado dele. Lá dentro, para operar sobre estado da coleção, é invocado o método contains. E dessa forma você vai distribuindo as operações do sistema, deixando a inteligência da aplicação dividida de uma forma que respeite nosso limite natural de entendimento.

Por sinal, parece interessante ter um jeito lógico para aproximar os dados das operações, mas ainda não diminuimos de fato a dificuldade de entendimento em nenhum exemplo. Dada a métrica do CDD que considera branches e funções como argumento como pontos de complexidade, os exemplos acima ainda mantém suas pontuações, por mais que o código tenha ficado mais coeso.

Exemplo #2: Coesão através do encapsulamento para atualização de objeto

Considere o seguinte trecho de código:

```
                if (student.getCoach() == null
                        || student.getCoach().getId().equals(assignedAdvisor)) {
                    student.setCoach(personService.get(assignedAdvisor));
                }
```

Percebemos que o retorno do método `getCoach` é um tipo abstrato de dados. O código depois navega invoca o método `getId()` que retorna um tipo padrão da linguagem, no caso um `Long`. Usando a nossa lógica para deixar o código mais coeso, movemos a operação sobre o **id** para dentro do tipo da variável `student` e poderíamos chegar no seguinte:

```
1                   if (student.hasSameId(assignedAdvisor)) {
2                           student.setCoach(personService.get(assignedAdvisor));
3                   }
```

O código ficou mais coeso, porém o `if` continua no mesmo ponto de código e a dificuldade de entendimento ficou a mesma.

Mal cheiro nas múltiplas chamadas de método em cima do mesmo objeto

No código logo acima, utilizamos a variável `student` para invocar o método `hasSameId` e também é utilizada a mesma variável para invocar o método `setCoach`. O mesmo código poderia ser escrito da seguinte forma:

```
1       student.tryToChangeCoach(assignedAdvisor,personService);
```

Agora o código removeu um `if` de determinado ponto e moveu para outro lugar, literalmente distribuindo a dificuldade de entendimento pelo sistema. E ainda de quebra conseguimos fazer sem criar uma classe nova, apenas usamos um tipo que já existia. Claro que precisamos ficar de olho nos limites de entendimento que foram estabelecidos.

Uma entidade pode acessar um service?

Em primeiro lugar eu preciso que você preste bastante atenção na próxima frase: Você pode fazer o que quiser no código se souber analisar o lado positivo e negativo de cada decisão. Você precisa sempre ficar atento as duas coisas principais, respeitando a ordem de prioridde

1. A prioridade é funcionar
2. Vai ser bom se funcionar e se outras pessoas puderem entender

Dito isso, o método `tryToChanceCoach` ficaria da seguinte forma:

```
    public class Person {
        //atributos aqui

        public Person tryToChangeCoach(UUID assignedAdvisor,PersonService personServ\
ice) {
            if (this.coach == null
                    || this.coach.hasSameId(assignedAdvisor)) {
                this.coach = personService.get(assignedAdvisor);
            }

            return this.coach;
        }
    }
```

Esqueça qualquer coisa que uma classe terminada com o sufico *Service* possa dizer para você. Em geral você tem dois tipos de classes no código:

- As que tem atributos de dados
- As que tem atributos de depedência

Enquanto a `Person` é uma classe que tem atributos de dados, a `PersonService` é uma classe que tem atributos de dependência. Elas estão conversando. Um ponto de atenção é que a `PersonService` tem muitos métodos e é uma classe utilizada em muitos pontos do código.

Já conversamos sobre API's não democráticas então aqui é um momento que você pode combinar técnicas. Para que o método `tryToChangeCoach` não fique conectado com uma interface possivelmente instável, podemos criar uma nova, específica e muito mais estável.

```
    public interface GetPersonByUUID {
        Person get(UUID id);
    }

    public interface PersonService extends GetPersonByUUID {
        //não precisa mexer em nada aqui
    }

    public class Person {
        //atributos aqui

        public Person tryToChangeCoach(UUID assignedAdvisor,GetPersonByUUID getPerso\
nByUUID) {
            if (this.coach == null
                    || this.coach.hasSameId(assignedAdvisor)) {
```

```
16                  this.coach = getPersonByUUID.get(assignedAdvisor);
17              }
18
19              return this.coach;
20          }
21      }
```

E a mágica está feita. Seu método `tryToChangeCoach` está conectado com um tipo bem específico e talvez a sua cabeça não fique jogando contra você :).

Criamos testes automatizados para que ele nos ajude a revelar e consertar bugs na aplicação.

Quando olhamos para os testes automatizados construídos em diversos tipos de projetos e perguntamos o motivo daquilo, temos algumas respostas do pool abaixo:

- Segurança para refatorar;
- Quero garantir que o que eu fiz está certo;
- Preciso ter uma documentação executável;
- Segurança para alterar o código em função de um bug (manutenção corretiva);
- Segurança para alterar o código para acrescentar funcionalidades (manutenção evolutiva);
- Aumenta a qualidade do código;

Obviamente existem outras possibilidades de resposta aí nessa lista. Tirando a parte de ter uma documentação executável, muito se fala em segurança e qualidade do código do ponto de vista de confiabilidade de execução e manuntenção.

A aposta potencialmente arriscada dos testes automatizadas

Existem algumas coisas que são quase garantidas quando uma equipe decide que o software deve ser coberto por testes automatizados.

- Mais código será escrito;
- Bugs serão gerados no código de teste;
- O software, como um todo, vai ficar mais complexo pelo olhar de qualquer métrica que você queira;
- Mais código precisará ser mantido;
- Cada funcionalidade vai demorar mais tempo para ser escrita;

Claro que estamos evoluindo muito na área de pesquisa relativa a testes e inclusive já temos projetos que tentam gerar boa parte dos testes automatizados sozinhos. Mas, quando consideramos que é uma pessoa que vai escrever, a lista de cima apresenta uma bela tendência de ser verdade.

O potencial de retorno dessa aposta é o seguinte:

- Pode até demorar mais tempo, mas a qualidade do ponto de vista de confiabilidade de execução vai ser maior. Ou seja, vai apresentar menos falhas de execução;
- Pode até demorar mais tempo, mas a qualidade interna do código vai ser maior;
- Vai ser mais fácil alterar o código no futuro;

O ponto de atenção aqui é: E se você não obtiver este retorno? Como que o código relativo ao teste automatizado pode realmente facilitar a obtenção do retorno?

Do mesmo jeito que você deve se preparar para investir num mercado financeiro de alto risco, você também precisa preparar seus testes para maximixar o potencial de retorno dele.

Sempre que você tem o lado negativo garantido e o lado positivo em risco, é necessário muito mais esforço para sair do outro lado. É literalmente um objetivo muito desafiador.

Por que você acha que o teste automatizado te dá segurança para alguma coisa?

Aprofundando no tópico sobre segurança, precisamos fazer um questionamento. Como sabemos que o teste automatizado aumenta a segurança do código quando falamos de suportar refatoração, manutenções etc? Uma pergunta simples que você pode fazer é a seguinte: *Quantas vezes sua suíte de testes revelou um bug no sistema antes dele ter sido percebido em produção?*

Caso a sua suite de testes não seja reveladora de bugs é importante que você saiba que seu produto pode estar sendo prejudicado pelo seus testes automatizados. A expectativa de retorno que você tinha passadava pelo aumento de confiabilidade de execução, qualidade interna etc. Só que num cenário onde os testes não revelam bugs e você continua pegando tudo em produção, é como se eles não estivessem dando o retorno esperado.

Técnicas de teste para que ele seja mais revelador de bugs?

Talvez o primeiro o ponto que precisa ser analisado é: Os testes derivados para o código seguem algum tipo de técnica? Abaixo temos alguns exemplos de possíveis técnicas que podem ser combinadas.

- Specification-based testing;
- Boundary testing;
- Structural testing;
- Design-by-contract como técnica de self-testing;
- Property-based testing;

Cada uma dessas técnicas contribui para que você crie uma suíte de testes que tenha mais chances de revelar bugs antes que eles virem falhas em produção. Porque uma coisa é certa: O software está com bug, ele só não foi descoberto ainda.

A ideia é que a equipe responsável por desenvolver o código saiba das técnicas e seja capaz de utilizá-las de forma sistemática.

Em geral, o que se observa muito no mercado é o uso da técnica *Structural testing*, onde os testes são escritos para cobrir o que for possível das linhas de código geradas durante a implementação. Inclusive é daí que vem todo debate sobre cobertura de código e tudo mais.

Perceba que escrever os testes em função do código gerado é apenas uma das técnicas que pode ser utilizada para derivas bons casos, imagine se você combinar mais algumas?

A distorção da cobertura de código

Cobertura de código é uma métrica utilizada no mercado para analisar o número de linhas do código fonte que foram testadas pelo casos de testes automatizados. Alguns exemplos de como você pode olhar a cobertura:

- Por linhas de código no total;
- Por branches(if,else,loops);
- Por branches + condicionais;

Inclusive existem mais possibiliddes. Ainda existe muito sistema que o percentual de cobertura é definido em cima do critério mais básico, que é o de linhas de código no total.

```
1         public void validate(Object target, Errors errors) {
2                 TemCombinacaoUsuarioRestauranteFormaPagamento request = (TemCombinacaoUsuarioResta\
3 uranteFormaPagamento) target;
4                 Usuario usuario = manager.find(Usuario.class, request.getIdUsuario());
5                 Restaurante restaurante = manager.find(Restaurante.class,
6                                 request.getIdRestaurante());
7
8                 boolean podePagar = usuario.podePagar(restaurante,
9                                 request.getFormaPagamento(), regrasFraude);
10                if (!podePagar) {
11                        errors.reject(null,
12                                        "A combinacao entre usuario, restaurante e forma de pagament
13                }
14        }
```

Desconsiderando a indentação do código, temos 6 linhas de instruções. Se o critério de cobertura é apenas por linha e o percentual buscado for por exemplo de 80%, um teste pode ser criado verificando apenas o valor `true` para a variável `podePagar` e tudo vai estar certo. Isso considerando um cenário básico, imagine num sistema com muito mais fluxos?

Uma solução é obviamente você ir aumentando o percentual de cobertura para forçar que de um jeito ou de outro o teste passe pelas linhas. Só que, um percentual desejado de cobertura vira um objetivo, **e objetivo serve para direcionar esforço**. Agora será trazido um exemplo bem simples para demonstrar o que um objetivo mal setado pode fazer. Considere o seguinte código:

```
1  public class TesteCoberturaApplication {
2
3          public void test(boolean a, boolean b) {
4                  System.out.println("passando aqui 1");
5                  System.out.println("passando aqui 2");
6                  System.out.println("passando aqui 3");
7                  System.out.println("passando aqui 4");
8                  System.out.println("passando aqui 5");
9                  System.out.println("passando aqui 6");
10                 if(a || b) {
11                         System.out.println( "imprime");
12                 }
13         }
14
15 }
```

E agora considere os seguintes testes:

```
1  class TesteCoberturaApplicationTests {
2
3          @Test
4          @DisplayName("teste com a true")
5          void teste1() {
6                  TesteCoberturaApplication teste = new TesteCoberturaApplication();
7                  teste.test(true, false);
8          }
9
10         @Test
11         @DisplayName("teste com a false")
12         void teste2() {
13                 TesteCoberturaApplication teste = new TesteCoberturaApplication();
14                 teste.test(false, false);
15         }
```

```
16
17  }
```

Você acabou de ganhar 100% de código, caso você use instrução por linha. O primeiro teste faz com que o `if` avalie a expressão como verdadeira e execute o `sysout`. O segundo teste faz com que a segunda parte da condição seja exercitada, já que o primeiro parâmetro agora foi como `false`. O relatório vai dar 100% e o código não testou a combinação de primeiro argumento `false` e segundo argumento `true`.

Se a cobrança for por cobertura, vai ter cobertura. Agora se o critério usado for ruim, vai ter uma cobertura ruim. E esse resultado não é interessante para ninguém. O software continua frágil, sendo que mais esforço e tempo foram investidos.

O melhor critério é teste que revele bug. Aqui parece valer a pena uma repetição: **O critério mais poderoso para avaliar os testes é verificar quantos bugs foram revelados antes de irem para produção**. Buscando isso, naturalmente o código ganha os outros benefícios como segurança para refatorar etc.

Testes inteligentes

Para fechar, podemos falar de uma suite de testes inteligente. Alguns pontos de atenção:

- Será que se operadores de condição forem invertidos, os testes vão quebrar?
- Posso gerar valores contra meus métodos para tentar explorar mais possibilidades?
- Posso gerar automatizados de forma automática?

Criar suítes de testes reveladoras de bugs virou tema de pesquisa e tem muita gente investindo tempo para facilitar a vida do time de desenvolvimento. Para as perguntas acima, temos alguns tipos de testes que podem ser gerados de forma automática por ferramentas:

- Mutation testing;
- Fuzz testing;
- Search-based testing;

Já existem projetos voltados para essa área e com certeza vale a pena explorar o que por possível.

- PIT(pitest.org) instrumenta código e gera os mutantes. Será que algum teste vai quebrar? Deveria.
- JQF(https://github.com/rohanpadhye/JQF) é uma ferramenta para Fuzz Testing que tenta gerar valores interessantes buscando maximizar cobertura de código;
- Diffblue(https://www.diffblue.com/) é um software pago que tenta gerar testes automatizados automaticamente;

Todas podem exploradas visando a criação de um suíte de testes realmente poderosa.

O caminho para uma suíte de testes poderosa

Ter uma suíte de teste poderosa pode ser um caminho parecido com o seguido pelas empresas na busca de uma estratégia de integração, deploy e delivery contínuo. Não é fácil chegar lá, mas pode valer a pena.

O que adianta colocar software em produção rápido se ele volta sempre? Se o seu conjunto de testes não busca revelar bugs, você pode estar escrevendo a toa.