Fragmentos de um programador

Artigos e insights da carreira de um profissional

Casa do Código

PHIL CALÇADO

Edição
Adriano Almeida
Vivian Matsui

Revisão e diagramação
Bianca Hubert
Vivian Matsui

[2016]
Casa do Código
Livro para o programador
Rua Vergueiro, 3185 – 8º andar
04101-300 – Vila Mariana – São Paulo – SP – Brasil
www.casadocodigo.com.br

ISBN

Impresso e PDF: 978-85-5519-222-7

EPUB: 978-85-5519-223-4

MOBI: 978-85-5519-224-1

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

QUEM É PHIL CALÇADO

Phil Calçado é diretor de engenharia na DigitalOcean em Nova Iorque. Em vidas passadas, ele foi diretor de engenharia na SoundCloud em Berlim, ajudou empresas a adotar SOA e entrega contínua na ThoughtWorks em Londres, Sidney e Melbourne. Antes de se tornar um nômade, ele trabalhou na equipe de vídeos da Globo.com no Rio de Janeiro.

PREFÁCIO

Eu me lembro até hoje de quando comecei a frequentar o GUJ, em 2006. Era uma comunidade pequena, muito longe do tamanho que tem hoje, mas vibrante. Praticamente todas as discussões que aconteciam por lá geravam longos debates, alguns até acalorados, porém todos cheios de aprendizados.

Era comum ver as opiniões firmes do Phillip por lá. Lembro-me até hoje de que, já naquela época, ele falava sobre granularidade de serviços e sua cruzada contra a hipercomplexidade de algumas plataformas da época.

Naquela época, eu engatinhava no mundo do desenvolvimento e acompanhava atentamente cada artigo que o Phillip publicava no seu antigo site, o Fragmental, que nem existe mais hoje. O que é uma pena, pois tinha **muito** conteúdo bom e que valerá até muitos anos ainda à frente.

Em uma conversa recente, falamos com o Phillip e conseguimos desenterrar seu blog. Fizemos uma coleta dos seus artigos publicados lá no passado no Fragmental, e que são leitura obrigatória até hoje para qualquer desenvolvedor de software de respeito.

Mesclamos também com conteúdos atuais que ele escreve em inglês, traduzidos para o português com a curadoria da Casa do Código. Você vai colher opiniões e vários insights sobre design de software, integração de sistemas, Orientação a Objetos, Domain-Driven Design, microsserviços e muito mais, tudo aqui dentro destas páginas.

Espero que você aproveite esse estudo e que valha a pena para você, assim como o Fragmental valeu a pena para mim lá no

passado e me fez olhar o mundo do desenvolvimento de software com olhos muito mais criteriosos.

Aproveite.

Adriano Almeida

Editor da Casa do Código

Sumário

Capítulo 1

INTRODUÇÃO

Minha introdução ao mundo da informática foi inusitada. Como ocorreu com tantas crianças da década de 80, meus pais ficaram muito preocupados com o tempo que eu gastava jogando videogame. Um dia, meu pai, que trabalhava em projetos de engenharia de petróleo, me disse que me daria um computador, o primeiro de toda a minha família. Meus olhos brilharam, imaginando todos aqueles jogos a que eu nunca tinha tido acesso no meu Mega Drive, mas sobre os quais havia lido em revistas.

Grande foi a minha decepção quando, em vez do 386 que eu havia imaginado, ganhei um livro. Eram umas cem páginas ensinando conceitos básicos de programação, linguagem de máquina, e diagramas de circuito descrevendo o computador. Este era o manual do TK-83, produzido por uma empresa chamada Microdigital. O computador era mais velho que eu, basicamente um clone brasileiro do britânico Sinclair ZX 81. Não havia interface com o usuário, e a única maneira de interagir com o computador era escrevendo programas em BASIC.

Meu pai disse que só me daria o computador em si depois que eu aprendesse a programar. Então, pelos próximos seis meses, passei muitas horas lendo aquele livro e escrevendo programas no papel. Após verificar meu progresso, enfim ganhei o tal computador dos meus pais.

Infelizmente, a pessoa que iria vender o TK-83 não o tinha mais

disponível, logo eles conseguiram um TK2000. Era um hardware mais moderno, mas em vez de um clone do ZX81, eu tinha em mãos um clone do Apple II+. Passei um bom tempo lutando com a diferença entre os modelos — pode-se dizer que comecei a ter apreço por portabilidade de código com onze anos!

Muitos anos, linguagens de programação e computadores depois, com 20 anos de idade, decidi largar a faculdade e a vida de freelancer e me dedicar ao meu trabalho como desenvolvedor em uma pequena agência web no centro do Rio de Janeiro. O trabalho era interessante no início, mas aos poucos fui percebendo que todo projeto se resumia a escrever o mesmo gerenciador de conteúdo (CMS).

Pensando em facilitar minha vida, passei um fim de semana escrevendo um CMS genérico, em ASP 3.0. Este foi o maior sistema que eu já tinha visto. E durante seu desenvolvimento e eventual customização para clientes, aprendi da maneira difícil sobre boas práticas em software.

Traumatizado com essas experiências, eu acabei atraído pela linguagem Java. Esta tecnologia prometia não só portabilidade, mas era também fortemente baseada em objetos. Todos os livros me diziam que esses tais objetos eram a bala de prata em software de fácil manutenção. O problema é que ninguém conseguia me dizer o que era um objeto.

Enquanto eu ainda estava na faculdade, um professor descreveu o paradigma da seguinte forma: "Orientação a Objetos é como ser um bebê. O bebê vê um chocalho, morde, é duro, joga fora. Ele vê uma chupeta, morde, vê que é mole, chupa". Até hoje eu ainda não tenho a menor ideia do que ele estava querendo dizer com isso, mas esse era o tipo de definição esotérica que havia no mercado.

Como bom nerd, eu passei a ler tudo o que conseguia achar

sobre esse assunto. Comecei a participar de diversas listas de discussão e me aventurar pelos grupos de usuário no Rio de Janeiro. Eventualmente, achei um fórum brasileiro em que essas ideias mais abstratas eram discutidas com frequência: http://guj.com.br.

Eu não tenho a menor ideia de quantas horas da minha vida passei no GUJ. Mas além da satisfação em discutir assuntos interessantes, lá também conheci quase todas as pessoas que são meus amigos mais próximos até hoje em dia.

As discussões no GUJ eram acaloradas, me faziam pensar e pesquisar. Eventualmente passei a escrever regularmente sobre minhas conclusões no meu blog pessoal, o Fragmental. Minha ideia era escrever sobre as dúvidas mais frequentes do fórum, para que da próxima vez que alguém perguntasse a mesma coisa, eu mandasse um link para o artigo em vez de ter de escrever tudo novamente.

Eventualmente o blog ficou popular, e eu comecei a ser convidado para palestrar e escrever uma coluna para revistas da indústria. Diversos artigos foram usados como referência bibliográfica em doutorados, livros e outros artigos.

Infelizmente, como às vezes ocorre na internet, o trabalho de manter o site no ar se tornou maior do que eu poderia investir. O blog já não existe fora do WebArchive, mas frequentemente recebo pedidos de estudantes e profissionais que querem acesso ao conteúdo. Então, em vez de continuar manualmente gerando PDFs e mandando para estas pessoas, a Casa do Código se propôs em publicar os artigos mais importantes em um livro.

Este é o volume que você tem em mãos — com a inclusão adicional de alguns artigos mais recentes traduzidos. Espero que esta obra sirva para tapar alguns 404 na literatura técnica brasileira.

Capítulo 2

GUIA DE GUERRA PARA FREELANCERS

— Artigo publicado em junho de 2005.

Esses dias tenho conversado via ICQ (sim, eu prefiro ICQ ao MSN) com um amigo meu, o Renato, sobre as desventuras dele enquanto freelancer. Tentei dar dicas que aprendi quando me embrenhava por essas terras distantes...

2.1 HISTÓRICO

Eu programo desde muito novo. Não vou me alongar aqui, mas por alguns anos fui ganhando uma graninha dando consultoria básica em ASP, VisualBasic (argh!) e C em listas de discussão. Coisa pouca que mal dava para comprar um livro de programação traduzido de 50 reais. Não demorou muito e passei a fazer programinhas como freelancer, como todo mundo do meio.

Cheguei a trabalhar "oficialmente" como freelancer por algum tempo, ganhando um salário razoável pela minha experiência formal, em casa, indo de duas a três vezes por semana para reuniões no escritório do cliente. Esse projeto acabou afundando no meio do curso; vou tentar explicar o porquê e tentar aplicar o que já aprendi até hoje nestes problemas.

2.2 ORGANIZAÇÃO DO AMBIENTE

Você quer trabalhar de freelancer? Organize-se. Não me importa se você é um moleque de quatorze anos que mal sabe fazer um loop (e devia estar estudando, não brincando de trabalhar!), ou um profissional tarimbado e conhecido que faz uns servicinhos de consultoria especializada: **você precisa se organizar!**

Para começar, onde você trabalha? Poucos freelas têm a disposição um escritório. A maioria trabalha mesmo é no quarto, de madrugada. Se você trabalha nas suas horas de folga, estipule o tempo de trabalho, **seja disciplinado com seus horários**. Defina as horas por dia que você vai trabalhar normalmente (ocasionalmente você terá de fazer "hora-extra", claro). Essas horas vão ser úteis também quando você calcular o preço dos seus serviços e suas entregas.

O ambiente de trabalho deve ser organizado. Não é *sua casa*, é **seu trabalho**! Claro que você tem muito mais liberdade que em um escritório formal, mas não deixe essa informalidade distraí-lo. A melhor coisa a fazer é providenciar uma **porta com tranca** para o lugar onde seu computador fica (se você não morar sozinho, claro). Este lugar deve ser silencioso.

Se você (como eu) mora em um lugar onde parece que os carros passam dentro do quarto, arrume uns fones de ouvido e coloque uma música para ouvir. Se você não gosta de trabalhar com música, é complicado, mas neste caso tente música instrumental — clássica é legal —, pois geralmente funciona e não nos distrai.

Desligue ou coloque em estado Away ou N/A todos os seus *Instant Messengers*. Se você possui um sócio ou outra pessoa que pode precisar falar com você imediatamente, configure isso (IMs de verdade como o ICQ permitiam fazer isso fácil fácil...), mas desative qualquer alerta que possa incomodá-lo.

Preferencialmente, não use seu e-mail pessoal para assuntos relacionados ao trabalho extra. Crie um e-mail próprio e deixe seu e-mail pessoal (cliente de e-mail ou Gmail *notifier* da vida) desativado.

Geralmente, o freelancer tem pouquíssimo tempo para trabalhar. Este tempo deve ser aproveitado o máximo possível. Bloglines (ou outro agregador RSS/derivados) e fóruns (http://www.guj.com.br/forums/newMessages.java), então, nem pensar!

2.3 METODOLOGIA DE TRABALHO

Em empresas (ou em qualquer lugar com mais de uma pessoa trabalhando, às vezes mesmo com uma pessoa só...), você geralmente vai ser obrigado a usar uma metodologia X, com processos definidos por outra pessoa. Muitas empresas pequenas definem apenas um conjunto de procedimentos padrão, mas mesmo assim a liberdade não costuma ser muito alta, mesmo porque você está em uma equipe.

O fato é que, se você está sozinho, não existe ninguém nem para lhe impor nada, nem para ajudar a decidir o que fazer e quando. Você vai ter de se virar para desenvolver um conjunto de processos e práticas, e aplicá-las.

Quando você não é freelancer profissional, quando geralmente só pega um *servicinho* aqui e outro ali, você costuma tratar cada projeto de modo diferente. Mesmo nesse caso, um mínimo de organização é esperado. Mesmo que você não tenha uma metodologia de trabalho definida, você deve parecer que tem uma se quiser manter clientes sérios e que acreditem no seu trabalho.

Use sempre boas ferramentas. Sim, eu sei que isso serve para

qualquer tipo de trabalho. Mas, para o freelancer, é fundamental você:

1. Ser rápido.
2. Ser produtivo.
3. Ter qualidade.

Ser rápido porque o cliente está em contato direto (veja a seguir) e vai lhe pedir muitas coisas. Se você for rápido o suficiente, pode criar protótipos que acabam lhe dando mais confiança do que o sistema que atende ao usuário, e pode ainda criar protótipos que resultem no fechamento de novos negócios, novas funcionalidades etc. Sempre é bom mostrar algo para o cliente em vez de ficar só conversando eternamente sobre o que poderia ser feito.

Ser produtivo é crucial para quem trabalha no tempo vago. Você não pode gastar muito tempo para desenvolver um módulo. Você tem poucas horas por dia e elas devem ser bem aproveitadas, com o que interessa. Minha dica aqui é o contrário do normal: fuja de ferramentas RAD de ponta a ponta, use apenas RAD para partes repetitivas do seu sistema, como layout de telas, relatórios e bases de dados. O que você precisa fazer para ser produtivo é ter conhecimento do seu código como um todo, e mantê-lo sempre limpinho, refatorando sempre. Assim, você consegue mudanças rápidas em vez de ter que contar com wizards bizarros e GUIs que não lhe dão controle nenhum.

Qualidade é fundamental para o freelancer, afinal, **quem vai manter esse código é você**! O cliente não quer saber quantos arquivos você vai alterar, ele quer que você implemente a funcionalidade rápido. E em um projeto desses, funcionalidades mudam a todo o instante. Faça tudo pensando que você vai ter de corrigir bugs o tempo todo nesse código, às 3 horas da manhã, tendo de ir trabalhar em seu emprego formal no outro dia!

Use ferramentas normais de uma empresa, um controle de versões é fundamental mesmo quando se trabalha sozinho. *Bugtracking* também ajuda, mas sem exageros (todos os bugs e tarefas vão estar assinalados para você no final).

2.4 SEJA COMPROMISSADO

Meio desnecessário dizer, mas não é porque esse não é seu trabalho formal que você pode fazer um serviço qualquer. Um freelancer deve atender seu cliente exatamente como uma empresa normal. A informalidade da coisa tem de ser uma vantagem para ser mais ágil, não uma desculpa esfarrapada para atender mal.

Qualquer freelancer de verdade sabe que seu nome e reputação valem mais do que ele mesmo pode pagar. Combine coisas verbalmente, mas formalize pelo menos com um e-mail para não ficar no *disse-me-disse*, discutindo com o cliente coisas que possivelmente tenham sido ditas. Tenha certeza de que vocês estão falando a mesma língua.

Se for atrasar, avise. Combine uma reunião periódica frequente para controle do cronograma. Pode ser um telefonema, mas tenha um ponto de referência. Se você não pode fazer reuniões com o cliente no horário do expediente, deixe isso claro já no início do projeto e proponha uma alternativa, como reuniões por telefone, logo após o expediente ou nos sábados, por exemplo.

2.5 CONHEÇA SEU CLIENTE

Como um profissional autônomo, você dificilmente vai pegar grandes projetos (quebrei suas ilusões?). Geralmente, quem solicita serviços de freelancers são empresas de micro e pequeno porte que precisam de algo altamente customizado e não têm dinheiro para contratar uma das *bigs* (mas, na verdade, era isso que eles queriam

fazer). Assim, você tem a grande oportunidade de conhecer todos os funcionários da empresa cliente (sim, todos os 3 funcionários), do dono do negócio à menina do caixa. Use isso.

Conhecer o dono do negócio é ótimo. Você fica sabendo das expectativas, das necessidades e do que ele pretende fazer (é claro, tudo muito exagerado), mas não se esqueça dos usuários (sim, todos os dois).

Certa vez, trabalhei na migração de uma rede de nove computadores Windows 98 para GNU/Linux. Tudo meticulosamente planejado, contratei um técnico especializado neste tipo de serviço. Mas como todo bom moleque de dezenove anos (não que eu seja muito menos moleque aos 22), esqueci que quem ia usar o sistema não era a pessoa com quem eu lidava, dono do negócio.

E lá fui eu para a primeira sessão de treinamento em OpenOffice.org (http://www.openoffice.org/), achando que estava tudo perfeito. Já tinha convertido todos os formulários-padrão para o novo sistema, instalado uma versão customizada e amigável de KDE, quando um dos funcionários me pergunta sobre as 30 e poucos macros VBA que ele fizera no Excel para acelerar um processo interno qualquer. O diretor falou que esqueceu de me avisar, e ninguém além daquele carinha tinha consciência de que aquelas macros eram superimportantes, só ele as utilizava! E lá vai o Phillip converter macros VBA para OO.org. Mas eu não entendia as macros, elas eram relativas ao negócio, isso seria praticamente desenvolver um sistema (o que não estava no contrato).

Ficou acertado que o funcionário usaria Excel em um Windows XP até que conseguíssemos solucionar este problema... Que não era o único. As pessoas da empresa trataram a migração como uma coisa imposta e não desejada. Afinal, para eles, era apenas a empresa economizando em licenças.

Considerando que a empresa ficava muito longe da minha casa, acho que, se ônibus tivesse programa de milhagem, eu poderia viajar para a Europa de graça de tantas vezes que tive de ir resolver problemas. E não, não eram problemas grandes, eram coisas do tipo:

> — Isso aqui (Mozilla) não funciona!
>
> — Não? O que aconteceu?
>
> — Eu tento entrar no site `www.abcd.com.br` e dá erro na conexão.
>
> — Uhm... Peraí. Esse aqui não é o cabo de rede da sua máquina?
>
> — Não sei, é?
>
> — Você arrancou?
>
> — Ah, acho que esbarrei aí sim... Mas deixa disso, esse tal Linux não funciona!
>
> — Por acaso o Windows tinha internet sem cabo de rede!?

Evite boicote: aproveite o fato de você ter apenas meia dúzia de usuários e converse com todos eles, tente agradar a todos (mesmo sabendo que quem paga é o dono do negócio apenas). Isso vai lhe economizar muita dor de cabeça.

2.6 SEJA ÁGIL

Metodologias ágeis são complicadas de aplicar em empresas grandes. Mas quando sua empresa é você, apenas elas podem ajudar

e muito.

Estude um pouco de XP (http://www.extremeprogramming.org/). Você não precisa aplicar todas as práticas (algumas nem vai conseguir, como *pair programming*), mas se aproveite da situação para aplicar o máximo destes conceitos de agilidade e foco no cliente. Não vou explicar para você os conceitos aqui — procure um livro ou algo do tipo —, mas algumas ideias básicas de como isso pode ajudá-lo são mostradas a seguir.

Pegue os requisitos diretamente com seu cliente. Não importa muito a técnica, mas sugiro que você dê uma olhada em *User Stories* (http://www.extremeprogramming.org/rules/userstories.html). O importante é que você defina os requisitos com a maior participação do cliente possível.

Como freelancer, você tem de estar pronto para mudanças. O cliente vai mudar tudo a toda hora, e como não tem muita burocracia, você precisa se preparar para isso (tirar proveito disso, até). Faça *Plano de Releases* pequenos, vá acrescentando funcionalidades ao sistema de pouco em pouco, a cada release. Conforme o cliente vai vendo o sistema, ele vai mudando (calma, não ache que vai trabalhar de graça, leia mais à frente sobre contratos, isso pode ser muito bom financeiramente) e você vai repriorizando suas entregas.

Tenha seu cliente sempre à mão, celular dele colado com post-it no monitor.

2.7 TENHA UM ADVOGADO

Isso é sério. Você não precisa ter um advogado contratado, mas tenha pelo menos uma boa indicação de um. Se for amigo ou

parente seu, melhor. Explore e use-o para revisar e redigir contratos (pagando apenas uns chopps).

O Luca, amigo de GUJ, me falou uma coisa sobre *profilers* que se aplica a advogados também: *você nunca aprende a usar um profiler antes de precisar usar um profiler, mas geralmente quando você for precisar, não vai ter tempo para aprender nada, tem é de consertar os problemas rápido!*

Então, já tenha pelo menos o telefone de alguém que vai poder ajudá-lo se as coisas derem muito erradas.

2.8 CONTRATOS ABERTOS

Isso parece meio louco, mas funciona muito bem, principalmente nesse caso.

Para cada módulo de seu sistema, defina um valor em horas (ou defina pontos, é mais XP). Defina um valor por ponto e se comprometa a entregar X pontos por mês, cobrando o equivalente.

Quando você entrega um módulo, o cliente pode querer alterar algo (ele vai querer!). Então, avalie a alteração em pontos, e repriorize suas entregas para o próximo mês (ou semana, ou duas semanas etc.).

Então, crie seu contrato de maneira que o escopo seja aberto. O cliente estaria pagando *R$Y* por *X* pontos por mês. **Não se iluda em terminar o projeto.** Um projeto de software só acaba quando o programa é deletado dos computadores definitivamente e sai de uso, o código-fonte é apagado e todos os programadores são mortos (frase do Ted Neward).

2.9 CONCLUSÃO E HISTÓRIA TRISTE

Bom, se eu tivesse seguido algumas dessas dicas, eu não teria passado por uma situação muito infeliz há alguns anos.

Trabalhava então *full-time* para uma empresa desenvolvendo um sistema de laudos. A ideia era simples: cinco módulos, pagamento por mês.

Os módulos eram:

- Gerência de estoque
- Gerência de laudos
- Gerência de usuários
- Agenda coorporativa
- Geração de um laudo especial

Baseado na dificuldade que eu via nas coisas, defini a entrega na ordem:

1. Gerência de usuários
2. Gerência de estoque
3. Agenda coorporativa
4. Geração de um laudo especial
5. Gerência de laudos

O gerenciador de laudos tinha umas customizações muito complicadas.

Começaram as complicações no desenvolver do projeto. No início, eu tinha muito tempo para isso, mas depois arrumei um emprego formal, e isso acabou com meu tempo (some, é claro, a faculdade). Simplesmente não conseguia arrumar tempo para trabalhar. Quando chegava em casa, às 23 horas, dormia. Nos fins de semana, eu tentava, mas sempre tinha alguém para me encher o saco (ou arrumar um lugar para sair e não trabalhar).

Em paralelo, começou uma dificuldade tecnológica. A ideia

inicial era desenvolver em Delphi (ugh!), mas o ambiente passou para GNU/Linux e eu tentei o Kylix. Depois de muito tempo xingando o bichinho (injustamente), passei para Java, e tive de me virar para aprender a programar com um JBuilder 5.

O usuário passou a me ligar diariamente cobrando a entrega. Propus mostrar uma apresentação sobre o sistema, fiz um bando de telas em um editor GTK WYSIWYG, e colei em um PowerPoint. Foi bom em parte para solucionar algumas dúvidas com a interface, mas o cliente reclamou, pois ele queria ver algo funcionando, mesmo que um protótipo.

Bem, enfim primeira iteração: entreguei o gerenciador de usuários. Enquanto o usuário descobria bugs no sistema, eu tentava preparar o outro módulo. Enquanto isso, ele me pedia para ver como estavam os módulos, e eu desconversava.

Quando entreguei os 3 primeiros módulos (atrasados, claro), o cliente começou a reclamar que estava gastando rios de dinheiro e não via retorno.

— É... A gente paga, paga, e nada. Temos de ver isso, hein.

— Como assim? Eu já entreguei três módulos!

— Que não fazem nada de útil para a minha empresa. Na verdade, eu tinha pedido esses módulos como complemento. Se eles não funcionarem, eu não vou ligar muito.

— Hum... É verdade, mas pelo menos temos alguma coisa.

— Que não me serve para nada.

Pois é. Eu fiz minha agenda de entregas baseado no que era bom

para mim, não para o cliente.

No final das contas, o módulo de laudos atrasou mais três meses. Eu já não tinha tempo nem de respirar, e precisava faltar ao trabalho constantemente para tentar trabalhar em casa, mas mesmo assim não adiantava. Ocasionalmente, meu computador tinha algum problema e eu perdia um ou dois dias de trabalho.

O final da triste história foi a contratação de uma empresa externa para fazer auditoria. Os caras viram que muita coisa havia sido entregue, e como eles foram tão técnicos quanto eu, acharam os atrasos normais. Acabou com o cliente cancelando o contrato e ficando com três módulos que não faziam nada de útil, além do prejuízo de um ano de desenvolvimento e mais um mês de auditoria que não deram em nada.

Imaturidade minha, falta de preparação. A minha sorte é que eu tinha meu emprego formal, porque minha moral ficou muito baixa. Sem falar no meu bolso vazio. Depois deste, já peguei diversos outros projetos como freelancer e acabei descobrindo práticas melhores para resolver o problema que tinha. Elas funcionam comigo, então acredito que outras pessoas possam se beneficiar também.

Hoje em dia, dificilmente pego um freela, falta de tempo e de paciência não deixam. Mas tirei essas dicas da minha experiência não tão grande, não tão pequena, que espero que ajudem.

Capítulo 3

CONTRATOS NULOS

— Artigo publicado em outubro de 2005.

Um post no blog de Todd Huss (2005) gerou causou algum burburinho recentemente. Todd reclama de ter de checar argumentos de métodos o tempo todo para ter certeza de que não são nulos e não ter uma linda `NullPointerException` .

Para quem não é de Java ou está começando na linguagem, uma `NullPointerException` (carinhosamente conhecida como NPE) acontece quando você chama um método em uma variável que aponta para `null` , por exemplo:

```
public static void fazerAlgo(String a){
   a.split(" ");
}
public static void main(String args[]){
   fazerAlgo(null);
}
```

Causaria uma NPE (chamando o método `split` em `null`). Existe todo um misticismo com uma NPE, basicamente por um motivo: **é um erro de programador**. Seu sistema não devia lançar NPE. O grande problema é que geralmente as pessoas fazem assim:

```
public static void fazerAlgo(String a){
   if(a==null)
      throw IssoNaoEhUmaNullPointerExceptionException("a deve ser
definido");

   a.split(" ");
}
```

Ou seja, fazem a mesma coisa que uma NPE faz, mas usando `IllegalArgumentException`, ou outra exceção qualquer.

Dos comentários que apareceram no blog, uns indicaram o projeto *nully*, no *java.net* (https://java.net/projects/nully/). O `Nully` é uma ferramenta para a IDE IntelliJ (http://www.jetbrains.com/) que basicamente adiciona uma anotação `@NotNull`, acionando *warnings* e erros no editor quando você tenta implementar uma operação que poderia resultar em um valor `null` para o parâmetro marcado com a tal anotação. Você pode ver mais detalhes no site do projeto, mas eu não gostei disso mesmo.

Uma outra alternativa dada foi votar no BUG 5030232 (http://bugs.java.com/bugdatabase/view_bug.do?bug_id=5030232) da Sun, uma tentativa de adicionar o tratamento que a linguagem Nice faz com `null`. Nice (http://nice.sourceforge.net/) é uma linguagem para a JVM (como Groovy ou BeanShell) e basicamente exige que você inicie as variáveis que podem ser nulas com um `?` (interrogação), e não deixa (na compilação) utilizá-las sem checar antes se a variável é `null`. Boa tentativa, mas ainda não é isso.

Adam Kruszewski comentou que no seu blog havia uma implementação com metadados e AspectJ. A implementação dele é bem simples. Você anota um método e, toda vez que este é chamado, seus argumentos são checados. Se algum for `null`, ele solta uma exceção. Simples, mas não foi desta vez.

Algumas linguagens mais dinâmicas (entre elas Ruby, mais em http://www.ruby-lang.org/en/) têm um objeto especial que representa o valor `null`. Nestas linguagens, não há como você receber uma NPE ou algo parecido, mas o objeto `null` (`nil` em Ruby) retorna sempre uma exceção quando algum método é chamado. É uma implementação de algo parecido com um `NullObject`, mas só funciona em linguagens altamente dinâmicas

(especialmente de tipagem dinâmica), o que Java não é.

A solução não sou eu quem vai dar, é um conceito muito antigo que surgiu junto com os primeiros pensamentos em objetos. Com o tempo, foi sendo cunhada até que Bertrand Meyer (https://archive.eiffel.com/general/people/meyer/) a publicou e deu um nome: **Design By Contract**.

O pensamento nessa linha é diferente. Você define o que sua classe (ou seu método) espera de entrada e o que ela garante de saída. Qualquer coisa além destes valores esperados é rejeitado, e isso vai incluir `null`. O ponto é que você não deve lutar contra o `null`, você deve lutar contra valores fora do que você considera válido. Estes parâmetros válidos são expressos na forma de um contrato.

Para explicar este conceito, vamos nos focar nos três pilares: **invariantes, pré-condições e pós-condições**. O conceito de *invariante*, entretanto, pede uma breve explicação de espaço-estado.

Espaço-estado

Você sabe que um objeto tem estados, não sabe? Pois bem. Cada componente do estado de um objeto (um atributo, por exemplo) define uma dimensão. Para simplificar, imagine um objeto da classe `Ponto`:

```
class Ponto{
   String rotulo=null;
   int x;
   int y;
}
```

Esse é um ponto cartesiano, com três atributos que definem seu estado. **O valor de `x` e o valor de `y` definem o estado atual do ponto, `x` e `y` são as dimensões do objeto.**

O modo como o ponto muda de estado é o seu *comportamento*. O objeto reage a estímulos; um objeto não muda de estado por si só. Em Java, implementamos o comportamento de objetos nos seus métodos. Vamos supor que nosso ponto só se mova de dez em dez espaços, na horizontal ou vertical, implementando assim:

```
class Ponto{
 int x;
 int y;

 public void moverX(){
  x=x+10;
 }

 public void moverY(){
  y=y+10;
 }
}
```

Sendo assim, os estados que um ponto pode assumir dependem do seu comportamento. No caso específico, o estado onde a dimensão `x` vale `2` é inválido, já que partindo do valor zero, `x` só poderia assumir o valor de 10. O espaço-estado do objeto seria:

```
x=0,10,20,30,40,50,60,70,80...
y=0,10,20,30,40,50,60,70,80...
```

Ou seja, **espaço-estado é o conjunto de valores de dimensões válidas para um objeto.** Acho que isso dá mais uma clara visão de por que atributos públicos são perigosos.

3.1 INVARIANTES

Existem algumas condições que são necessárias para que um objeto esteja em estado válido. Um triângulo tem sempre três ângulos, um carro em funcionamento tem quatro rodas, um usuário tem login e senha definidos etc.

Essas são invariantes. Invariantes porque **devem ser obedecidas**

em todos os momentos do ciclo de vida do objeto. Elas são restrições no espaço-estado do objeto.

Para alguns, é mais fácil pensar em invariantes como restrições de tabelas em um banco de dados. A tabela `A` tem uma chave estrangeira `A1`, que aponta para a chave primária de `B` e não deve ser nula.

Se, por exemplo, nosso ponto só puder se mover a uma distância máxima de 20 espaços de zero para `x` e 50 para `y`, a invariante seria o intervalo marcado em negrito a seguir:

*x=**0,10,20**,30,40,50,60,70,80...*

*y=**0,10,20,30,40,50**,60,70,80...*

É dever da implementação do objeto zelar que ele esteja sempre em estado válido. Sempre que fizer uma transição de estado, o objeto deve checar se está em um estado válido. Em Java, o que poderíamos fazer é:

```
public void moverX(){
x=x+10;
 checarEstado();
}

public void moverY(){
 y=y+10;
 checarEstado();
}

protected void checarEstado()(){
 if(!(x<20) || !(y<50)) throw new IllegalStateException();
}
```

Checar a invariante é apenas sanidade. **Sua invariante nunca deve ser quebrada, e essa é a responsabilidade das pré e pós-condições.**

3.2 PRÉ E PÓS-CONDIÇÕES

Toda operação (ou seja, método) define contratos. O contrato de uma operação é:

> Se quem chamar me garantir a pré-condição, eu garanto a pós-condição.

Ou seja, se o objeto que chama o método garantir que a pré-condição esteja cumprida, o método deve garantir que a pós-condição também esteja cumprida.

Para exemplificar, vamos criar um método que move nosso ponto um número de espaços qualquer, desde que seja no máximo 20 espaços por vez:

```
public void mover(int espacosX, int espacosY){
 x=+espacosX;
 y=+espacosy;
 checarEstado();
}
```

Para evitar que a invariante seja quebrada e nós tenhamos certeza de que o parâmetro é de até vinte espaços, vamos estabelecer uma pré-condição através da checagem de parâmetros enviados e uma pós-condição:

```
 public void mover(int espacosX, int espacosY){

  //checando pre condição
  if (!(espacosX > 15)) throw new IllegalArgumentException();
  if( !((x+espacosX) < 20)) throw new IllegalArgumentException();
  if( !((y+espacosy) < 50)) throw new IllegalArgumentException();

  int antigoX = x;
  int antigoY = y;

  x=+espacosX;
```

```
  y=+espacosy;

  //checando pos condições
  if(x!=(antigoX+espacoX)){
     x=antigoX;
      throw new IllegalStateException();
  }

  if(y!=(antigoY+espacoY)) {
      y=antigoY;
      throw new IllegalStateException();
  }

  checarEstado();

 }
```

> Você deve checar contra quebras de contrato, da sua parte ou de quem o chamou.

Como já foi dito antes, você não deve evitar `NullPointerExceptions` , mas sim evitar que seus métodos processem argumentos inválidos, estabelecendo e obedecendo a contratos.

Note que é uma excelente prática **checar se os valores não são válidos em vez de checar se os valores são inválidos.** Confuso? Partindo do ponto que seus valores válidos sejam inteiros de 1 a 10, você não deve fazer:

```
if(x<0 || x >10) throw...
```

Mas sim:

```
if(!(x>0 && x < 10)) throw...
```

Pode parecer a mesma coisa, mas é esta disciplina que garante a segurança e robustez do contrato. Você nunca sabe o que o usuário vai entrar em um programa, então teste para encontrar o que você

sabe que está certo. E obviamente o que não está certo, seja o que for, está errado.

A princípio, a pós-condição pode parecer inútil, mas lembre-se: você prometeu que ia entregar isso, não custa nada dar (teoricamente) uma segunda conferida. Ok, em um caso simples como esse, é superficial demais para valer sequer umas linhas de código.

Tenha atenção especial com contratos de classes de interface externa. Cuidado com dados que vêm, ou do/para o usuário, ou de outros subsistemas.

Isso me lembra de uma ocasião na qual tinha de gerar uma página HTML com no máximo 180 caracteres. O método que gerava o texto da página tinha no seu contrato a garantia de que geraria no máximo 180 caracteres. E como era um processo muito complexo, sempre que terminava, ele checava o texto que gerava. Isso garantiu que, mesmo que o usuário recebesse uma mensagem de erro, ele não receberia uma mensagem pela metade (dependendo do conteúdo, isso pode ser muito pior).

> Quando você não tem absoluta certeza de que vai conseguir cumprir seu contrato (ou se seu método depender de outros métodos que podem não obedecer contrato nenhum), cheque antes de retornar.

3.3 SUBCLASSES E CONTRATOS

Como uma subclasse estabelece seu contrato?

Um princípio muito conhecido da teoria de Orientação a

Objetos é o Princípio de Substituição de Liskov (LSP — *Liskov Substitution Principle*). Ele diz basicamente:

> Subtipos podem substituir seus supertipos.

Simples, óbvio... Até boçal alguém diria. Pode até ser, mais isto implica diretamente nos contratos dos objetos, e nem sempre vemos isso.

Quanto à sua invariante, uma subclasse pode ter uma definição do que é ser válida diferente da classe mãe, por isso ela pode definir uma invariante diferente. Vamos exemplificar criando uma classe-filha para nosso ponto:

```
class PontoLonge extends Ponto{
 protected void checarEstado()(){
     if( !(x < 1000) || !(y < 1000)) throw new IllegalStateExcepti
on();
 }
}
```

Nesse caso, a nova classe definiu que sua invariante permite valores maiores para `x` e `y`.

Como nosso ponto pode se mover para lugares muito mais longínquos, é trabalhoso ficar indo de pouco em pouco. Vamos redefinir os métodos de movimento para nos movermos mais rapidamente:

```
class PontoLonge extends Ponto{
 protected void checarEstado()(){
     if( !(x < 1000) || !(y < 1000)) throw new IllegalStateExcepti
on();
 }

 public void mover(int espacosX, int espacosY){

     //checando pre condição
```

```
     if (!(espacosX > 100) ||  !(espacosY > 100) )
         throw new IllegalArgumentException();
     if( !((x+espacosX) < 1000)) throw new IllegalArgumentExceptio
n();
     if( !((y+espacosY) < 1000)) throw new IllegalArgumentExceptio
n();

     int antigoX = x;
     int antigoY = y;

     x=+espacosX;
     y=+espacosy;

     //checando pos condições
     if(x!=(antigoX+espacoX)){
         x=antigoX;
         throw new IllegalStateException();
     }

     if(y!=(antigoY+espacoY)) {
     y=antigoY;
         throw new IllegalStateException();
}

     checarEstado();
 }
}
```

Parece legal, hein? Primeiro, uma sugestão é refatorar os métodos. Olhe só essa quantidade de coisas repetidas! Esse exercício fica para você, já que foge do escopo do texto.

Voltando à sobrescrita, você consegue imaginar um problema? Se não, veja o código seguinte, retirado de uma classe que usa os objetos `Ponto`:

```
public moverUmPoucoUmPonto(Ponto p){
 p.mover(11,21);
}
```

Que tal? Essa classe funciona com `Ponto`, mas não com `PontoLonge`. Parabéns para nós que acabamos de mandar todo o LSP para o espaço (provavelmente com a reusabilidade inteira do

nosso sistema), já que não podemos utilizar nossa classe derivada como usamos a classe mãe. Solução?

> A pré-condição de um método sobrescrito em uma classe derivada deve ser *igual ou menos restritiva* do que a pré-condição do método na classe base.

Isso significa que nós poderíamos aceitar coisas que a superclasse não aceita, mas **nós temos de aceitar tudo o que a superclasse aceitaria**. No código, nós poderíamos (*poderíamos*), por exemplo, dizer que o mínimo de espaços que podemos mover passa de 15 para 5, porque assim o código feito para a superclasse ainda funcionaria. Corrigindo o exemplo (ainda sem refatoração):

```
public void mover(int espacosX, int espacosY){
 //checando pre condição
 if (!(espacosX > 15)) throw new IllegalArgumentException();
 if( !((x+espacosX) < 1000))throw new IllegalArgumentException();
 if( !((y+espacosY) < 1000))throw new IllegalArgumentException();

 int antigoX = x;
 int antigoY = y;

 x=+espacosX;
 y=+espacosy;

 //checando pos condições
 if(x!=(antigoX+espacoX)){
     x=antigoX;
 throw new IllegalStateException();
 }

 if(y!=(antigoY+espacoY)) {
     y=antigoY;
     throw new IllegalStateException();
 }

checarEstado();
}
```

Não tem jeito de aumentar o valor mínimo, então voltamos ao original (e mais código repetido, anda logo com esse *refactoring* aí!).

Isso também explica porque você pode, em Java, aumentar a visibilidade de um método (alterar a visibilidade de um método que era `protected` para `public`), mas não diminuir (mudar um método `public` para `private`).

Para pós-condições, é bem parecido:

> Uma pós-condição de um método sobrescrito deve ser *igual ou mais restritiva* do que a pós-condição do método original.

Isso quer dizer que, se a pós-condição do seu método for que ele retorna um inteiro entre 1 e 100, os métodos que o sobrescreverem podem gerar um número entre 50 e 60, mas não um entre 1 e 500, por exemplo. Se uma classe estava esperando receber deste método entre 1 e 100 e recebe 50, não há problema. Mas se ela receber 500, aí sim o LSP deixa de funcionar como deveria (e pode ser que algo simplesmente exploda no seu sistema — talvez literalmente!).

3.4 CONTRATOS QUEBRADOS

Como qualquer contrato, o contrato de uma classe pode ser quebrado. Você já deve ter uma ideia de como reagir quando o cliente não cumpre sua parte, lançando uma exceção, mas lembre-se também de que você pode não cumprir o contrato. É aí que entra a pós-condição: se ela não foi obedecida, quem não cumpriu o contrato foi você.

Quase sempre (na prática, diria que sempre) é melhor você interromper o processamento com uma exceção do que retornar um

valor que não cumpre a pós-condição.

Você poderia ter outro comportamento nesse caso, como retornar uma flag, ou mesmo não fazer nada, mas exceções são a maneira padronizada de lidar com quebra de contratos. Com a pré-condição documentada, é obrigação do cliente provê-la. A checagem é só para evitar esforço desnecessário e identificar erros mais rapidamente.

3.5 DOCUMENTANDO

Um contrato só tem sentido se documentado. Muitas plataformas oferecem facilidades para documentar o contrato das classes, produzindo formas compactas para visualizar os contratos de uma classe e checando possíveis quebras de contrato nos editores. Infelizmente, Java não tem nada assim pronto. O que você pode fazer é usar o bom e velho JavaDoc.

Documente a invariante da classe na descrição dela. Você pode usar um pseudocódigo. Evite copiar e colar o código, porque assim você expõe sua implementação e geralmente não atualiza o JavaDoc toda vez que o código muda. Descreva os algoritmos.

O contrato dos métodos deve estar descrito na documentação destes. Use as tags de `@param`, `@throws` e `@return` para indicar o que você espera e o que provê.

Coloque na cabeça que checar contratos é documentação. Qualquer um sabe como usar corretamente seu método se você especificar seu contrato na documentação.

3.6 ISSO É TRABALHOSO DEMAIS!

Sim, eu sei. Infelizmente, Java não tem suporte nativo a *Design*

By Contract, ao contrário de algumas outras linguagens (geralmente, linguagens puramente OO). Existe o `assert` herdado do C, e existem abordagens em AOP, mas a linguagem por si só não tem esse recurso.

Uma linguagem com suporte a contratos vai dar-lhe um modo fácil de definir e se referenciar a pré e pós-condições, invariantes e guardar valores antigos (o nosso `antigoX` e `antigoY`). Vai fazer um `or` automático com uma pré-condição e a pré-condição do método que você sobrescrever, e um `and` com a pós-condição. Ainda deveria gerar um documento como um JavaDoc com espaço reservado para a pré e pós-condições (que, afinal, são públicas). Também deve ter um mecanismo que permita ligar ou desligar as checagens, se quisermos mantê-las só para depuração.

Enquanto não vemos isso em Java, nós temos de criar esses conceitos por nós mesmos. Isso geralmente implica em um *overhead* enorme, de uma maneira geral:

- Se uma classe possui estados inválidos e estes são atingíveis, providencie uma invariante e faça a checagem dela.
- Se seus métodos realizam um processamento e retorna um valor, cheque este valor se no meio do caminho você precisou usar métodos não confiáveis, como métodos de terceiros.
- Sempre estabeleça e documente um contrato, mesmo que você não faça checagem.
- Se uma classe precisa de muitos objetos para não estar fora da invariante e você não quer passar estes objetos no construtor, use uma Factory.
- Tente utilizar o máximo possível de boas práticas

estabelecendo checagens, e evite código duplicado ao máximo.

- Tenha testes unitários que testem todas as possibilidades que você consiga pensar de quebrar o contrato dos métodos.

Referências

HUSS, Todd. *Tired of checking for null arguments in the method body.* Maio 2005. Disponível em: http://jroller.com/thuss/entry/tired_of_checking_for_null#comments.

Capítulo 4

EVITANDO VOS E BOS

— Artigo publicado em junho de 2007.

Uma das grandes vantagens do desenvolvimento orientado a objetos é a possibilidade de representar o domínio que está sendo modelado de maneira mais próxima do mundo real. No entanto, esta prática não é usada com tanta frequência quanto esperado. A maioria dos sistemas construídos em linguagens OO, como Java ou C#, ainda é composta de estrutura de dados e funções como coisas separadas.

4.1 ORIGENS

Com o advento da programação estruturada, passou-se a separar um programa em partes menores (módulos), e estas partes em outras ainda menores (funções ou procedimentos). As funções e procedimentos continham a lógica de negócio necessária para manipular as estruturas de dados, que são um agrupamento de dados logicamente relacionados (leia mais sobre este paradigma em *5. Fantoches*).

Com a chegada das tecnologias orientadas a objetos, os desenvolvedores tiveram de aprender um novo paradigma. No ritmo em que as coisas funcionam em tecnologia, não é surpresa que muitos deles aprendiam sobre classes e objetos enquanto desenvolviam os próprios sistemas.

Infelizmente, esta pressa toda fez com que as linguagens OO funcionassem apenas como uma nova ferramenta para se fazer a mesma coisa. Quem programava em Pascal continuou programando em Pascal, só que utilizando este tal de Delphi. Quem programava em C continua escrevendo C, só que desta vez em Java. Existe um velho ditado em Ciência da Computação que diz: *"Você pode escrever FORTRAN utilizando qualquer linguagem"*.

4.2 QUANDO TUDO JÁ ESTÁ RUIM...

No mundo Java EE, esta situação foi piorada ao extremo pelos próprios fabricantes e evangelistas. Antes de Java EE se tornar toda uma plataforma de desenvolvimento, foi criado o conceito de Enterprise Java Beans, componentes distribuídos que talvez tenham no *curriculum* o maior número de enganos históricos da indústria.

EJBs foram criados como uma plataforma de componentes distribuídos que levasse Java a concorrer com big players da área, como C++ e Ada. Acontece que muitos não entenderam o recado, e EJBs foram tratados como a maneira de fazer aplicações corporativas em Java.

Um dos grandes problemas nesta abordagem é a maneira na qual EJBs se dividem. Existem basicamente três tipos de EJBs: Session Bean, Entity Beans e Message-Driven Beans. Os dois primeiros são os relevantes nesta discussão, podemos ignorar o último.

Entity Beans representam os dados em uma base de dados (quase sempre relacional). A tecnologia de mapeamento entre objetos e tabelas (Mapeamento Objeto-Relacional) do framework EJB até a versão 2.1 era precária, e basicamente você teria um Entity Bean por tabela no banco de dados. Os Session Beans são os responsáveis por manipular os dados, sejam representados por

Entity Beans ou não.

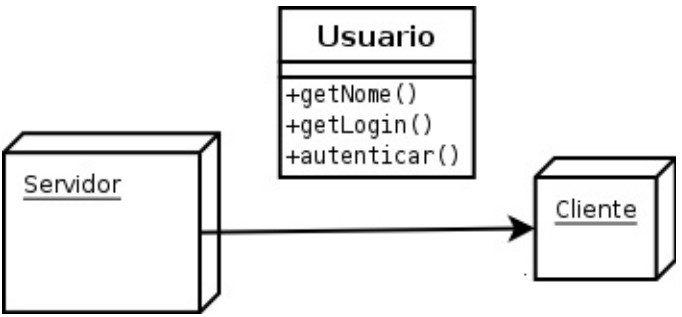
Para enfrentar as dezenas de problemas com esta arquitetura, a Sun lançou um catálogo de padrões chamados de Core J2EE Patterns. A maior parte deste catálogo se destina a lidar com problemas que EJBs causam, mas o nome "core" ajudou a difundir o mito de que Java EE era sinônimo de EJBs.

Eventualmente, a comunidade de desenvolvedores percebeu que EJBs não são a solução para qualquer problema — na verdade, são uma possível solução para uns poucos problemas —, e passou a desenvolver em um modelo mais simples, quase sempre usando apenas Servlets ou algum framework. O problema é que o modelo de programação criado para os EJBs, incluindo os padrões do catálogo da Sun, ainda são utilizados.

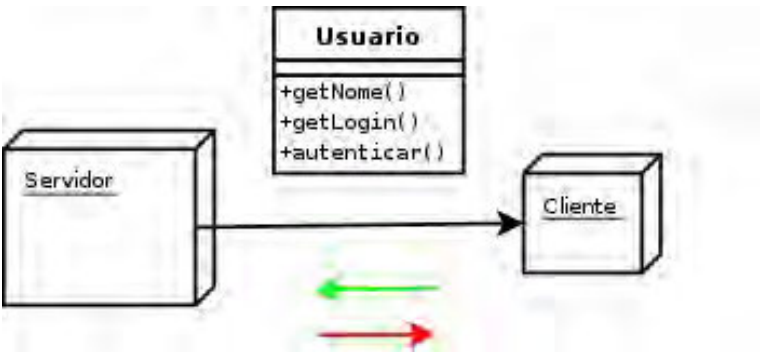
4.3 TRANSFER OBJECTS/VALUE OBJECTS

Um dos padrões deste catálogo era chamado de Value Object, ou VO. Posteriormente, este objeto foi renomeado para Transfer Object (TO) por haver um conflito de nomes dele e outro padrão (interessante notar que a documentação oficial ainda faz uso do nome antigo nos diagramas), e porque se trata de uma adaptação do padrão Data Transfer Object, também do Fowler.

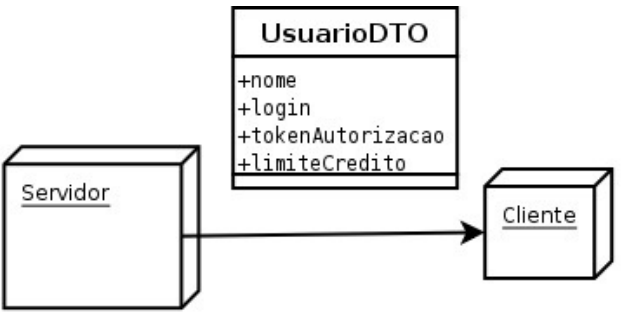
Um TO seria um objeto cujo objetivo é transportar dados entre um EJB e seu cliente, estando cada um em uma JVM diferente. Imagine que temos o cenário a seguir no qual uma aplicação cliente recebe um objeto remoto do EJB:



Toda vez que o cliente chamar um método no objeto vindo do EJB, é necessário fazer uma conexão na rede para pegar os dados dele do outro servidor:



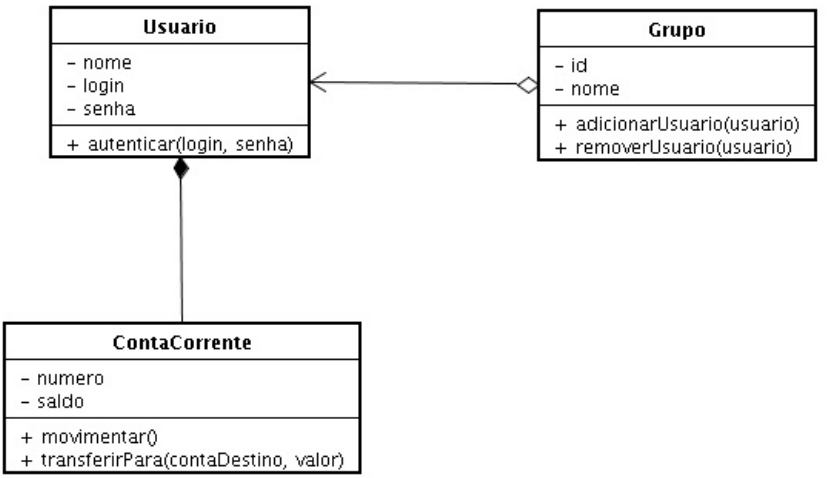
Estas conexões consomem recursos como Sockets, além de diminuir o desempenho da aplicação e aumentar o tráfego na rede. Para evitar este problema, o EJB pode transferir todos os dados que o cliente vai precisar de uma só vez, em um só objeto. Este objeto é o Transfer Object:



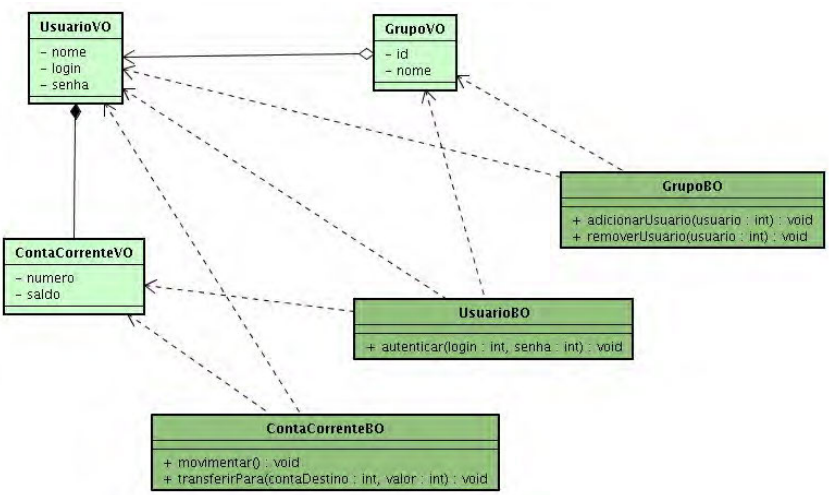
Como chamadas dentro de uma mesma JVM (que são o cenário mais comum) não possuem este custo agregado, não há porque utilizar TOs dentro de uma aplicação que não use chamadas remotas.

4.4 VOS E BOS: MODELO DE OBJETOS ANÊMICO

Em um modelo orientado a objetos, temos representações dos conceitos que estamos modelando na forma de classes, como mostrado a seguir.



Ao utilizar o modelo de programação inspirado em EJBs, devemos dividir nosso domínio em *classes de dados* e *classes de lógica*. As classes de dados recebem um nome terminando em VO (o correto seria utilizar o sufixo TO, mas vamos nos ater ao modo como é comumente utilizado), e as classes de lógica em BO. O diagrama seguinte mostra este modelo.



Este é o modelo que Martin Fowler (2003) batizou com o padrão *Anemic Domain Model* (Modelo de Domínio Anêmico). Em um modelo anêmico, existem objetos que parecem modelar a realidade, como os VOs do modelo anterior, mas na verdade não utilizam a característica básica de objetos, que é manter dados e comportamentos em um mesmo componente: **o tão falado objeto!**

Problemas da anemia

Um dos problemas principais deste modelo é, como Fowler aponta, que você paga o preço de ter um modelo rico (por exemplo, é mais difícil mapear esta estrutura para um banco de dados) sem ter as vantagens deste. Um modelo rico traduz o mundo real para software da maneira mais próxima possível. Ao utilizar classes de dados separadas das classes de lógica, esta vantagem é simplesmente jogada fora. No mundo real, não existe lógica de um lado e dados de outro, ambos combinados formam um conceito.

Também a manutenção de um sistema construído desta maneira é problemática. O módulo (o BO) possui acoplamento muito alto com a estrutura de dados (o VO), e a mudança em um afeta o outro drasticamente. As setas no diagrama anterior mostram como a dependência entre os módulos (já que não são objetos de verdade) aumenta drasticamente.

A coesão das classes de lógica (BOs) tende a ser baixa visto que agregam diversas funções relativas não só à sua própria estrutura de dados como muitas vezes de outras. Bom notar que, mesmo em Projeto Estruturado de Sistemas, esta é uma divisão pobre de módulos.

Aumentando este problema, dificilmente um VO será usado apenas por um BO (e se o for, temos um problema grande de reusabilidade) e os demais BOs vão igualmente ter conhecimento sobre a estrutura de dados (VO), já que não existe um

encapsulamento real do objeto. Assim, quando um destes VOs tiver sua implementação alterada, temos de verificar os possíveis impactos em diversos BOs espalhados pelo sistema.

A falta de encapsulamento também faz com que os VOs possam estar em estado inválido. Todo `Usuario` no exemplo dos diagramas deve ter um login e uma senha, além de estar autenticado. O que acontece se um BO altera o login do usuário para `null` no meio da execução do programa? Não há proteção quanto a isso. Caso o `Usuario` fosse um objeto de verdade, ele poderia se prevenir e não entrar em estado inválido.

4.5 CONCLUSÃO

Apenas deixar de utilizar EJBs não fez com que o uso de Orientação a Objetos se espalhasse na comunidade de desenvolvedores Java EE. A própria especificação EJB evolui e, em sua versão 3.0, traz recursos para que sejam criados modelos ricos. Ainda assim, a grande maioria das aplicações é construída de maneira procedural por pura ignorância, ou indução da documentação existente.

Desenvolver sistemas envolve, antes de mais nada, conhecer a tecnologia utilizada. Não se pode tirar proveito de Orientação a Objetos se não se distingue o que é OO e o que é procedural.

Referências

FOWLER, Martin. *AnemicDomainModel*. Nov. 2003. Disponível em: http://www.martinfowler.com/bliki/AnemicDomainModel.html.

Capítulo 5

MVC E CAMADAS

— Artigo publicado em junho de 2007.

A Arquitetura de Camadas é muito utilizada para separar responsabilidades em uma aplicação moderna. Apesar de a ideia da divisão de uma aplicação em camadas ter se popularizado nos anos 90, muitos desenvolvedores ainda não conhecem a técnica a fundo, boa parte por documentação ineficiente sobre este padrão arquitetural.

Junto com a popularização da Arquitetura de Camadas, ressurgiu o modelo MVC de desenvolvimento. Este modelo foi criado em Smalltalk, e traz simplicidade e coerência às interfaces.

Um problema com a popularização simultânea destes dois padrões arquiteturais é que eles passaram a ser eternamente confundidos. O objetivo deste breve capítulo é mostrar como MVC e Camadas são conceitos diferentes, que podem ser aplicados em conjunto ou não.

Nota

O termo *componente* é utilizado aqui para significar qualquer artefato de software. Pode ser substituído por classe, camada, objeto ou mesmo um componente no sentido dado por Component-Based Design (CBD).

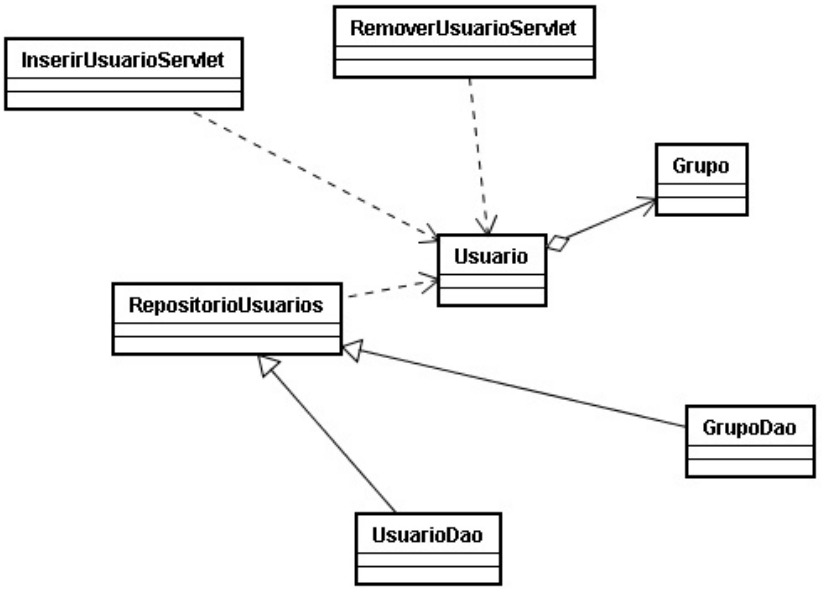
5.1 CAMADAS: SEPARAÇÃO ENTRE COMPONENTES

Para melhor organizar a manutenção dos componentes, é crucial que estes sejam separados por algum critério. Isolando-os em grupos, é possível diminuir o acoplamento entre eles, fazendo com que as mudanças em um grupo não impactem muito em outro.

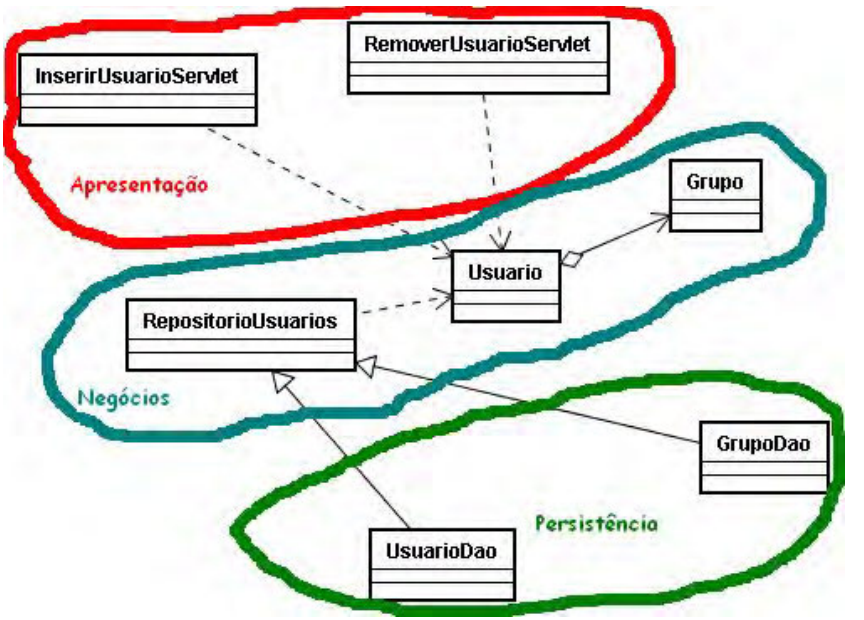
Entre as diversas formas possíveis de separar os componentes, temos a técnica de camadas. Ao separar componentes em grupos chamados *Camadas* (*Layers*, em inglês), o projetista agrupa componentes por responsabilidade em comum que possuam.

Uma aplicação Java EE tradicional, por exemplo, usa o modelo de 3 Camadas: Apresentação, Negócios e Persistência. Na Camada de Apresentação, estão todas as classes e demais artefatos (como páginas JSP) relacionados com a interação entre o usuário e a aplicação. A Camada de Negócios contém a modelagem do domínio do problema em classes de negócio, e a Camada de Persistência contém as classes com conhecimento sobre como persistir objetos no banco de dados (por exemplo, DAOs).

O diagrama a seguir mostra os objetos em uma aplicação simples. Neste caso não há separação lógica qualquer.



Aplicando a técnica de Camadas, vamos agrupar os objetos com responsabilidade semelhante.



Utilizando Camadas, o projetista tem um critério para usar quando decide agrupar duas ou mais classes, mas o uso de Camadas não diz nada sobre como estes grupos se comunicam. O máximo que diz é se estas são transparentes (abertas), ou opacas (fechadas). Camadas transparentes não evitam que uma superior enxergue a inferior, já as opacas, sim.

As Camadas promovem o encapsulamento (uma Camada não vai saber o que tem dentro da outra) e a coesão (componentes parecidos ficam no mesmo grupo).

5.2 MVC: INTERAÇÃO ENTRE COMPONENTES

Partindo do ponto que temos os nossos componentes separados de alguma forma, utilizando Camadas ou não, precisamos de um modelo de interação entre eles. A maneira mais natural de conduzir esta interação é não definir regra alguma.

Agindo desta forma, os componentes podem falar entre si sem qualquer limitação ou protocolo. Este modelo é bastante usado e relativamente eficiente, mas existe um aspecto de um sistema que normalmente não fica bem desta forma: interfaces gráficas.

Interfaces gráficas geralmente exibem o estado dos objetos de uma aplicação, e isso deve ser feito em 'tempo real' (com o perdão do mau uso do termo). Uma alteração no estado do objeto deve ser imediatamente refletida na tela visível ao usuário.

Outro aspecto é que a interface recebe os estímulos vindos do usuário e deve propagá-los para os objetos de negócio. Se cada componente em uma tela gráfica for responsável por estimular os objetos certos, o código tende a ficar repetitivo e difícil de manter.

Um exemplo clássico são interfaces gráficas construídas em tecnologias RAD, como Visual Basic e Delphi, em que diversas

regras e rotinas são contidas dentro dos botões e demais controles no formulário. Ao alterar uma destas rotinas, é necessário alterar o código do controle gráfico. E ao fazer isso, é preciso tocar no código da rotina, o que não é desejável, já que são aspectos diferentes de uma aplicação.

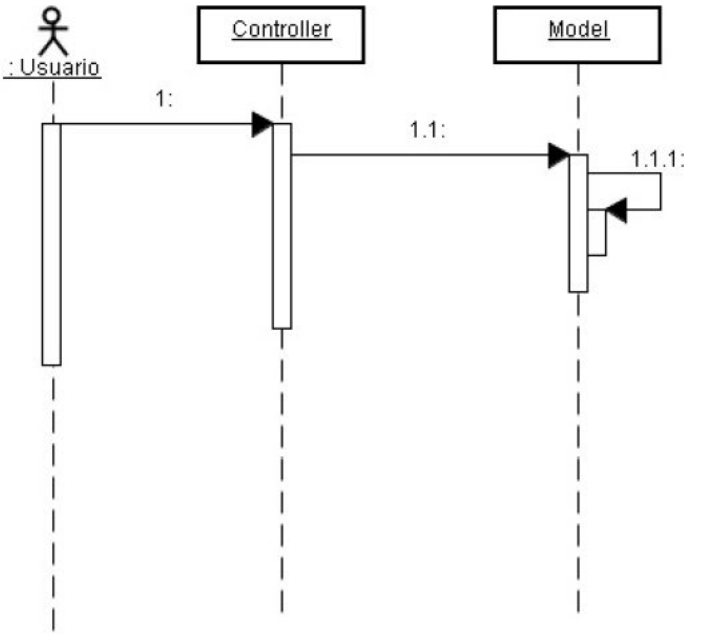
Uma solução criada nos anos 70 é o chamado Modelo MVC (*Model-View-Controller*). Ele consiste no bom uso integrado de alguns *Design Patterns* (Padrões de Projeto) clássicos, como *Observer* e *Strategy*.

Em um Modelo MVC, nossos componentes são divididos em 3, os já citados *View*, *Model* e *Controller*. A View é a parte exposta, o Controller é o controle sobre a comunicação que vem do usuário para o sistema, e o Model representa o estado do sistema.

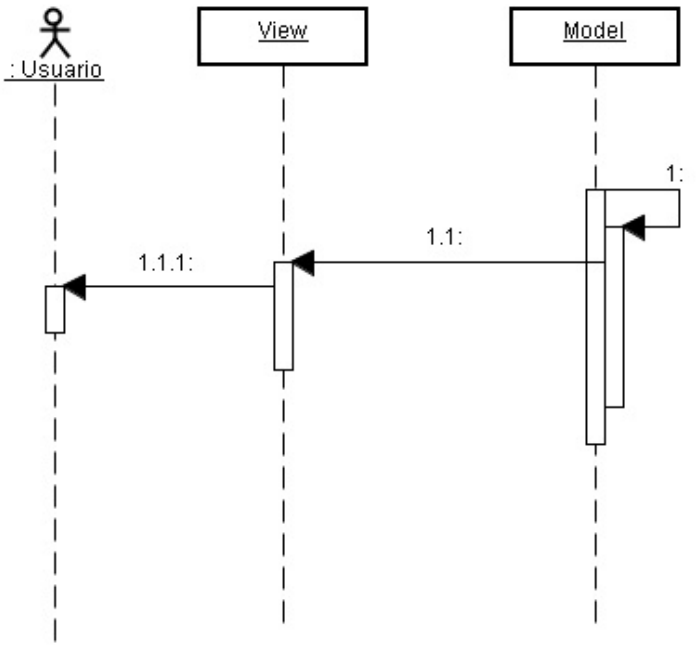
Note que o modelo MVC não precisa ser usado apenas em interfaces gráficas. Qualquer tipo de comunicação entre componentes, visuais ou não, pode ser regido por ele.

O MVC se baseia em dois princípios fortes.

- O Controller despacha as solicitações ao Model:



- A View observa o Model:



Estes princípios definem a comunicação. O estímulo vindo do usuário (ou de outro componente, se você está usando MVC fora de uma interface gráfica) vai para o Controller, que é o componente com inteligência o suficiente para saber qual operação invocar no Model. A operação invocada pode efetuar a mudança de estado no Model. Como a View o observa, assim que a mudança de estado for realizada, ela é atualizada. Então o MVC em si não traz mais do que duas regras básicas para a comunicação entre componentes, e não diz o que estes componentes contêm.

Como já mencionado, é possível, e muitas vezes desejável, integrar a técnica de Camadas com o modelo de interação provido pelo MVC, quando estamos lidando com aplicações monolíticas (não compostas por serviços — como em SOA —, ou por componentes — neste caso, componentes de negócio, como definido por CBD).

A partir do momento em que dividimos os nossos componentes em Camadas, podemos aplicar o MVC nelas. Geralmente, isto é feito definindo a Camada de Negócios como o Model, e a Apresentação como a View. O componente Controller exige um pouco mais de controle.

Na maioria dos casos, podemos definir o Controller dentro da Camada de Apresentação. Esta Camada ficaria responsável, então, por mostrar o estado do Model ao usuário e receber as requisições dele.

Algumas vezes, entretanto, é necessário que o Controller fique isolado dela. Este é o caso, por exemplo, quando possuímos mais de uma interface, como Swing e HTML. Neste caso, podemos usar uma Camada que quase sempre está implícita, a Camada de Aplicação.

5.3 CONCLUSÃO

Pela escassez de material disponível, é compreensível a confusão gerada pelos conceitos. De qualquer forma, é possível reduzir as diferenças a algo simples de lembrar:

> Camadas dizem como agrupar os componentes.
>
> MVC diz como os componentes interagem.

E lembre-se sempre de que, neste contexto, componentes referem-se a qualquer artefato de software: classes, páginas dinâmicas, scripts, objetos etc.

Mesmo sendo pouca a literatura sobre Camadas e MVC, é o suficiente para entender melhor como estes conceitos se relacionam.

Capítulo 6

FANTOCHES

— Artigo publicado em junho de 2007.

Eu gosto de Análise e Projeto Estruturados. Primeiro, porque me fizeram pensar em programas como mais que bits e bytes correndo por uma tela monocromática. Fizeram-me entender sistemas. Depois porque são uma bela técnica para resolver muitos tipos de problemas, principalmente os altamente matemáticos.

Mas o mundo real, ou pelo menos o mundo das aplicações mais comuns, não é feito por funções e subrotinas, mas por entidades colaborativas. A modelagem baseada em objetos é vendida como o meio ideal de se desenvolver um sistema nos dias de hoje.

A grande maioria das pessoas acredita nisso também, mas por muitas razões — entre elas o fato de aprenderem a desenvolver software de forma estruturada —, acabam utilizando tecnologias orientadas a objetos (linguagens, notações, bibliotecas, ferramentas etc.) para produzir software de forma estruturada. A ideia neste capítulo é tentar passar um pouco de como não produzir software de maneira estruturada, mas sim orientada a objetos.

6.1 MAS ESTRUTURAR NÃO ERA LEGAL?

Antes de continuar, deixa-me fazer uma definição que vai servir para o resto deste texto.

Paradigma Procedural é como eu chamo a programação na qual dados e funções estão separadas. Nesse modelo de desenvolvimento de software, as entidades são representadas por estruturas de dados simples que são manipuladas por funções independentes.

Eu não gosto do termo **Programação Estruturada** (apesar de usá-lo ocasionalmente). Este termo surgiu para separar a arte de destruir a sequência lógica de um sistema com instruções GOTO, comum em BASIC, da separação do fluxo em subrotinas, as funções e procedimentos. Dijkstra diz sobre isso tudo no seu paper clássico. Então, programação orientada a objetos é também estruturada no sentido em que separa as funcionalidades em módulos. Neste texto, chamaremos de Programação Procedural o modo de definir um programa como descrito anteriormente.

6.2 MODELANDO O MUNDO

A primeira coisa que se aprende em um curso básico de Análise de Sistemas (seja qual for o paradigma e metodologia usados) é que um sistema de informação modela um minimundo. Este mundo é uma versão abstraída do mundo normal, onde os únicos detalhes que importam são os relacionados com a aplicação, com o que o sistema faz.

Em um sistema procedural, geralmente se pensa em como as coisas acontecem, no fluxo de informação. Pensando proceduralmente, um simples cadastro é assim:

Cliente fornece endereço. Endereço é armazenado.

Como é perceptível, são apenas dados fluindo. Acho que todos imaginam que a função, chamada de Processo na nomenclatura de Análise Estruturada (aquelas bolhas do Yourdon), vai receber uma estrutura de dados `ENDEREÇO` e colocar no repositório (que pode ser um SGBD, um arquivo ou qualquer outra coisa).

Pensando de uma maneira OO, você tem:

Cliente tem um endereço.

Uhm? Sim. No seu minimundo OO, você vai mapear o que importa (o cliente e seu endereço) para um objeto. É claro que, dependendo da sua aplicação, o `Cliente` pode ser inútil, mas aí vai lhe restar o objeto `Endereço`.

Em um modelo OO, você está preocupado com o que existe no seu mundo real, como as coisas se relacionam nele.

6.3 IMPLEMENTAÇÕES PROCEDURAIS

Vamos pensar um cenário simples. Um estacionamento. Nosso sisteminha exemplo tem uma só funcionalidade: descobrir se existe vaga disponível para um carro no estacionamento e colocar esse carro na vaga.

Vamos ver o modo procedural de fazer as coisas. Vou usar Java para os dois casos. Como muita gente programa 100% procedural

nesta linguagem, não devemos ter muitos problemas no entendimento.

```
class Carro {
 private String placa;
 private Date dataEntrada;
 private int vaga = -1;

 public void setPlaca(String p) {
  this.placa = p;
 }

 public String getPlaca() {
  return placa;
 }

 public void setDataEntrada(Date data) {
  this.dataEntrada = data;
 }

 public Date getDataEntrada() {
  return dataEntrada;
 }

 public void setVaga(int vaga) {
  this.vaga = vaga;
 }

 public int getVaga() {
  return vaga;
 }
}

class Estacionamento {
 public final double VALOR_HORA = 0.5;
 private Carro[] vagas = new Carro[5];

 public boolean estacionar(Carro c) {
  int vagaLivre = vagaLivre();
  if (vagaLivre == -1)
   return false;

  c.setDataEntrada(new Date()); //coloca a data atual do sistema

  vagas[vagaLivre] = c;
  c.setVaga(vagaLivre);
  return true;
 }
```

```
 public double sair(Carro c) {
  int vagaOcupada = c.getVaga();
  Date dataEntrada = c.getDataEntrada();
  double conta = tempoEmHorasAteHoje(dataEntrada) * VALOR_HORA;

  c.setVaga(-1);
  vagas[vagaOcupada] = null;
  c.setDataEntrada(null);
  return conta;
 }
}
```

Aposto que muita gente vai estranhar eu rotular isso de procedural. Aqui nós temos a classe `Estacionamento` que cuida do processo de estacionar e deixar o estacionamento do `Carro`, representado por um *pseudoJavaBean* (um dia eu explico que é um pseudoJavaBean na minha concepção).

Mas, o que tem isso de procedural? São classes, não são?

Meu primeiro exercício é para quem tem background em linguagens procedurais. Tente reescrever isso utilizando pascal clássico (não Delphi), FORTRAN ou C (não C++). O código não vai ficar muito diferente, não é? Para quem não programa nessas linguagens, vou mostrar uma implementaçãozinha mixuruca em pseudocódigo baseado em C:

```
struct t_carro
{
 char * placa ; /* uma string */
 int vaga ;time_t data_entrada ; /* um dos tipos de data */
};

typedef struct t_carro carro ; /*para podermos usar a struct */

carro vazio;
carro vagas[5];

int estacionar(carro c)
{
 int vaga_livre = achar_vaga_livre();
```

```
 if(vaga_livre==-1) return 0; /* em C, 0 = false e 1 = true */

 c.data_entrada = data_do_sistema();
 vagas[vaga_livre] = c;
 c.vaga = vaga_livre;
 return 1;
}

double sair(carro c)
{
 int vaga_ocupada = c.vaga;
 time_t data_entrada = c.data_entrada;
 double conta = tempo_em_horas_ate_hoje(data_entrada) * VALOR_HORA
;
 c.vaga=-1;
 vagas[vaga_ocupada]=vazio;
 c.data_entrada=0;
 return conta;
}
```

Mesmo que você não saiba nada de C, dá para perceber que os códigos são muito parecidos. A maior diferença é a mais fútil, está nas convenções de nomenclatura. Se você colocar os atributos de `Carro` na primeira implementação como públicos, além de não perder nada (afinal, o que são `get` e `set` além de um jeito mais lento de fazer um atributo público?), o código ficará mais parecido.

Claro que a sintaxe das linguagens é muito parecida, e isso ajuda. Mas o que parece não é a sintaxe, é o meio como se programa. Esse é um programa altamente procedural. Note que não existe um só meio procedural de pensar e implementar, ou um só meio OO de fazer isso. Uns são melhores, outros piores, mas estes são apenas dois exemplos.

6.4 BAD SMELL: FORÇAS OCULTAS

O clássico livro *Refactoring*, de Martin Fowler (1999), introduz o conceito de *bad smells*, literalmente um fedor no código. Você olha, cheira, vê que tem algo errado, e precisa descobrir a causa.

Note que o sistema funciona! Você pode entregar para o seu cliente exatamente desta maneira, mas o que nos queremos aqui não é entregar algo rápido para gerar uma fatura. Estamos estudando a melhor maneira de implementar, e como objetos podem ajudar a tornar o desenvolvimento de software mais natural e mais barato em médio prazo.

Falando em natural, pense no código que criamos, em qualquer uma das implementações. Esse modelo se parece com o mundo real? Vejamos: temos um procedimento que recebe um carro, o joga no estacionamento... Como se na entrada do estacionamento houvesse um guindaste que carregasse os carros até suas vagas.

É assim que funciona? Não! O motorista ou o manobrista vai pegar o carro e dirigi-lo ate a vaga certa. O carro "sabe" estacionar, e essa é uma responsabilidade dele: dada uma vaga, estacione.

Claro que você pode efetivamente ter seu carro sendo movido desta forma (é só um sistema, não é o mundo real!), mas eu insisto: modele seu sistema o mais próximo que conseguir do mundo real. Isso o ajuda a conversar mais facilmente com seus usuários e manter a mesma linguagem que ele, podendo aproveitar os conceitos que ele aplica no dia a dia.

No caso específico, nosso método estacionar (`Carro c`) está com responsabilidades demais! Ele tem de coordenar o processo de encontrar uma vaga, modificar o carro, modificar a vaga etc. E olhe que isso é um exemplo simples, que não envolve acesso a recursos como bancos de dados. Vamos tirar o peso dos ombros do pobre método e colocar nosso carro para ser mais do que uma `struct` de luxo.

```
class Carro {
 private String placa;
 private Vaga vaga;

 public void estacionar(Vaga v) {
```

```
  this.vaga = v;
  v.ocupar(this);
 }

 public void sair() {
  vaga.setOcupante(null);
  this.vaga = null;
 }

 public Vaga getVaga() {
  return vaga;
 }
}

class Vaga {
 private Date dataEntrada;
 private Carro ocupante;

 public void ocupar(Carro c) {
  this.ocupante = c;
  this.dataEntrada = new Date();
 }

 public double getValor() {
  return (tempoEmHorasAteHoje(dataEntrada) * Estacionamento.VALOR_
HORA);
 }

 public long tempoEmHorasAteHoje(Date desde) {
  return 1;
 }

 public void setOcupante(Carro c) {
  this.ocupante = c;
 }
}

class Estacionamento {
  public static final double VALOR_HORA = 0.5;
  private Vaga[] vagas = new Vaga[5];

  public boolean estacionar(Carro c) {
   Vaga vagaLivre = vagaLivre();
   if (vagaLivre == null)
    return false;

   c.estacionar(vagaLivre);
```

```
    return true;
  }

  public double sair(Carro c) {

    Vaga vagaOcupada = c.getVaga();
    c.sair();
    return vagaOcupada.getValor();
 }
}
```

Perceba que nesse modelo nós explicitamos um conceito. O cliente, no caso, quer saber o preço do uso da vaga, não do carro, e para melhor modelar isso nós criamos um objetozinho com este conceito.

Observe os programas. Perceba como as responsabilidades foram divididas entre os objetos participantes, e como não existe mais um método (ou um conjunto de métodos) que manipula os dados. Geralmente cada objeto cuida de si próprio. Essa é a ideia.

De uma maneira geral:

> É um bad smell se seus objetos são guiados através de forças ocultas; se seus objetos se comportam como marionetes sendo manipuladas por uma função mestre, o mestre dos fantoches (qualquer semelhança com um famoso álbum é mera referência).

Isso é sintoma de um design altamente procedural, no qual a lógica de negócio está implementada em funções e em como estas manipulam estruturas de dados. Você geralmente acaba com uma série de classes (geralmente implementando o padrão `Command` induzido pelos Frameworks MVC web, mas por vezes são Façades) coordenadoras e um bando de objetos burros que nada mais são do

que um agrupamento de dados relacionados.

Neste cenário, é comum que simplesmente não se consiga reaproveitar lógica. É o tipo de sistema em que reusabilidade é mantida com `crtl+c/crtl+v` , já que seus métodos são "gordos" demais e são tão especializados que simplesmente não servem para nada além do que foram concebidos inicialmente. Outro sintoma deste tipo de design são métodos com muitas linhas de código.

Minha sugestão:

> Divida a responsabilidade do seu método faz-tudo por objetos que colaborem. Modele melhor seu sistema tentando aplicar conceitos e abstrações do mundo real.

6.5 ANATOMIA DE UM SISTEMA OO

Um sistema baseado em objetos e formado por componentes (os próprios objetos) que colaboram entre si. Isso não é tão diferente de um sistema procedural, no qual as funções e procedimentos devem trabalhar juntos para um baixo acoplamento e alta coesão.

Uma maneira prática de avaliar seu modelo OO contra fantoches e mestres é o diagrama de sequência, que serve exatamente para mostrar como os objetos interagem no sistema. Nesse caso, a interação é sempre iniciada pelo nosso mestre dos fantoches.

Tendo as responsabilidades distribuídas, os objetos agora colaboram entre si, não existe uma única unidade de controle. Se você precisar estender a funcionalidade do seu sistema, agora pode reutilizar os procedimentos que foram separados do método inicial.

O método `estacionar()` agora é apenas um estimulador dos outros objetos.

6.6 ENTRE FLEXIBILIDADE E PRODUTIVIDADE

Como sempre, o programador deve lutar entre ser flexível ou ser produtivo. Na verdade, essa é uma luta desnecessária. Quando você produz software de qualidade, você acaba sendo produtivo em longo e médio prazos.

A grande dificuldade é como saber quando sua arquitetura está flexível e quando está flexível demais (mais do que o necessário). Você pode perder dias esculpindo um framework em cima da sua aplicação, onde qualquer mudança será muito simples. Mas você precisa pensar que pode ser que uma mudança brusca nunca seja necessária, e não há na maioria das vezes como saber as necessidades futuras.

Um bom modelo de objetos permitirá que você consiga separar bem as coisas. Objetos mapeando significados do domínio, como o exemplo da `Vaga` , dão flexibilidade próxima da que seu cliente tem. Então, se ele resolver mudar seus conceitos, você pode seguir a mesma linha de raciocínio ao mudar os seus, incluindo estratégias parecidas de migração no mundo real e no sistema.

Refatorar seu código constantemente ajuda a manter algo simples e flexível. Siga esta prática.

6.7 CONCLUSÃO

Todo mundo aprende a construir algoritmos, e algoritmos são dados fluindo entre funções. A migração de paradigma é complexa, e a literatura especializada em plataformas altamente utilizadas,

especialmente Java e (principalmente) Java EE, não ajudam nem um pouco, pregando sistemas procedurais.

Existiu um motivo para a criação do paradigma de Orientação a Objetos. Este paradigma provê abstrações melhores e mais naturais, mais próximas do mundo. Antes de definir seu sistema, pense em como as coisas interagem no mundo. Pense que objetos não são dados+funções, eles são entidades por si só, e devem colaborar entre si. Não crie "donos da razão" no seu sistema, produza classes coesas e colaborativas.

Se você realmente quer utilizar objetos, fuja o quanto puder dos programas procedurais. Muitas vezes lidamos com coisas como bancos de dados relacionais e outros subsistemas que são intimamente ligados ao modo procedural de pensar. Entretanto, nestes casos você pode criar um mapeador (algo como o padrão DAO em Java EE) entre os mundos.

Referências

FOWLER, Martin. *Refactoring:* Improving the Design of Existing Code. Addison-Wesley Professional, 1999.
DIJKSTRA, E. W. *Structured programming*. London: Academic Press Ltd, 1972.

Capítulo 7

ARQUITETURA É SOBRE PESSOAS

— Artigo publicado em janeiro de 2008.

Todo livro sobre arquitetura de que consigo me lembrar claramente declara que a arquitetura é responsável por definir macroaspectos de um sistema baseado nos *inputs* dos seus stakeholders. Embora bastante clara, essa sentença é muito difícil de mapear nas nossas ações do dia a dia.

Provavelmente, um dos aspectos menos compreendidos do papel da arquitetura é que o time de desenvolvimento de software (**do qual o arquiteto faz parte**) é um stakeholder muito importante por si só. A arquitetura tem de preencher os requerimentos dos usuários do sistema, dos patrocinadores do projeto e dos desenvolvedores.

Isso significa que a arquitetura ideal para um dado projeto pode não ser uma que use as melhores ferramentas e técnicas.

Na maioria dos casos, o arquiteto terá de considerar que as qualidades do time definem uma das maiores razões para escolher uma dada ferramenta ou técnica. Imagine que você entrou para uma equipe de programadores, que programam há muito tempo em VB6, que está desenvolvendo um site. Apesar de você pensar que Ruby on Rails é a melhor escolha para essa tarefa, ninguém no time tem experiência com essa plataforma. O site precisa ser entregue em

um curto espaço de tempo, sem tempo para treinar. Você seguiria a melhor escolha técnica?

Ao mesmo tempo, um arquiteto precisa pensar no futuro. Talvez Rails não seja a melhor opção para esse release, mas se ele realmente agrega valor, o arquiteto deve incluir uma migração no roadmap do sistema. Tenho um pouco de experiência em introduzir Ruby em pequenos passos. A cada iteração, uma ou duas histórias seriam implementadas usando essa nova plataforma. Você pode começar utilizando Ruby, a linguagem, sempre que for necessário um pequeno programa ou um script, por exemplo, e introduzir lentamente o novo desenvolvimento da plataforma web. A coisa mais importante é que você deve ter uma visão clara de como o sistema deve aparentar no futuro e como fazer isso virar realidade.

Entretanto, antes de considerar usar Ruby ou qualquer outra nova tecnologia, o arquiteto deve apresentar suas ideias para o desenvolvedor. Assim como você deve perguntar aos seus usuários como a interface deveria ser, você também deve ter um *input* dos desenvolvedores. Claro que, assim como o usuário não pode ter sua interface ideal no estilo da Second Life, seus desenvolvedores vão sugerir ideias que não serão viáveis ou desejáveis. A coisa mais importante é você coletar o feedback e respeitar a opinião do grupo, que provavelmente será o mais afetado pelas suas decisões.

Eu lembro de trabalhar em um projeto altamente distribuído. Eu estava no Rio de Janeiro, e havia pessoas em Londres, Alemanha e Nova Iorque. Era quase impossível ter todo o time de desenvolvimento na mesma chamada de conferência devido aos fusos horários. O arquiteto do projeto estava em NY, talvez o menor escritório com apenas alguns desenvolvedores. Era muito interessante fazer um *svn up* e ver quais os novos brinquedos ele havia inserido no projeto.

Eu posso lembrar de que, ao atualizar meu workspace, tomei

consciência da introdução gradual do Spring (1.2, depois 2.0), Java 5, código C++ nativo acessado por JNI, PERL, Python e AWK (perfomando a mesma tarefa: substituições simples em arquivos de configuração durante o build). Nosso projeto era Java EE padrão, mas ninguém além dele tinha alguma ideia de como o sistema era estruturado de ponta a ponta. Quando ele foi questionado sobre isso por um gerente sênior, ele disse algo como *"Está tudo no wiki, esses desenvolvedores são preguiçosos!"*. Tenho certeza de que você pode imaginar que a qualidade de uma descrição de única página do wiki era uma bagunça total.

O problema nesse caso não é exatamente a terrível escolha técnica, mas o fato de que nós, desenvolvedores, não estaríamos cientes das mudanças até nós atualizarmos nossos arquivos de origem - nem preparados para resolver conflitos. Se o arquiteto não pode comunicar devidamente isso aos seus colegas na Europa ou na América do Sul, ele deveria ter adotado uma estratégia diferente do que simplesmente jogar mudanças em nós.

Distribuição de times, fisicamente ou não, é o maior problema da arquitetura. Digamos que temos um sistema que permite negociantes trocarem ações. Uma parte do sistema é sobre reunir dados do mercado, outra parte é sobre interação do usuário e outra é sobre submeter as transações de volta ao mercado. Por alguma razão, alguém decidiu que nós teríamos diferentes times trabalhando em cada módulo.

No final do dia, é um sistema único - qualquer um desses módulos isolado não teria utilidade - e poderia ser arquitetado assim. Isso acontece quando a estrutura da equipe influencia a arquitetura. Embora todos os módulos rodem no mesmo JVM, contexto de aplicação e contexto de web, eles estão sendo desenvolvidos em times diferentes. Quando o time responsável por reunir os dados precisa fazer um build completo, eles não

precisariam estar cientes de que o time responsável pelas transações está trabalhando em um release de manutenção.

Nesse cenário de exemplo, o arquiteto precisa planejar os módulos para que essas partes da mesma aplicação - elas não são componentes - estejam em caixas pretas (ou, pelo menos, em caixas cinzas) e os times possam trabalhar em um fluxo relativamente isolado.

Usar Rails, introduzir AWK e associados, e planejar o sistema como um módulo único e similares podem ser a melhor técnica de design. Mas arquitetura não é sobre técnica, pelo menos não apenas sobre isso. É o mesmo conceito que existe entre engenharia civil e arquitetura.

Quando você vai para uma inspeção no apartamento e percebe que o lugar perfeito para locação infelizmente recebe luz solar direta, em períodos indesejáveis, você está encarando exatamente os problemas que estamos discutindo aqui. Talvez colocar o prédio daquele jeito tenha sido a melhor escolha técnica (engenharia), mas a arquitetura deveria ter percebido que as pessoas eventualmente morariam lá e não ficariam felizes em acordar as 6 da manhã sendo queimadas pelo sol australiano.

Lembre-se: pessoas viverão no seu sistema. Fora e dentro dele.

Capítulo 8

DEIXE PARA LÁ O DDD

— Artigo publicado em março de 2010.

Nos últimos anos, realizei muitos workshops sobre *Domain-Driven Design* (DDD, ou em português, Projeto Orientado a Domínio). Nós tínhamos mais de 100 pessoas nessas sessões, e o feedback era frequentemente muito bom. Contudo, depois das sessões de 2009, eu disse ao meu parceiro de negócios que essa era a última vez que eu fazia isso.

Acho que *Domain-Driven Design* (EVANS, 2003) é um dos livros técnicos tradicionais mais importantes publicados na década passada. Eu desfrutei de várias atividades que realizamos durante os workshops, e também gostei de encontrar todas aquelas pessoas. Acho que estou apenas cansado de como nossa indústria não entende a diferença entre Domain-Driven Design e Orientação a Objetos.

Sendo mais específico, as razões para estar cansado são:

1. Parece ser extremamente difícil para as pessoas entenderem DDD.
2. As pessoas não precisam entender Domain-Driven Design para se beneficiarem com isso.

Deixe-me elaborar melhor.

8.1 PARECE SER EXTREMAMENTE DIFÍCIL PARA AS PESSOAS ENTENDEREM DDD

Vamos fazer uma busca rápida no Google:

Visitas no Google	Menções	Pergunta
11,500	34	+”domain driven design” +repository
5,800	20	+”domain driven design” +corruption
5,460	52	+”domain driven design” +ubiquitous

Esse é apenas um exemplo rápido de como as pessoas entendem DDD: há muito mais visitas no Google sobre *Repositório* e *Padrões da camada anticorrupção* do que sobre *Linguagem Ubíqua*.

Para entender melhor o problema que isso representa, vamos revisitar o livro *Domain-Driven Design* (EVANS, 2003). A Linguagem Ubíqua é definida na Parte I. De acordo com ele, a Parte I "apresenta os objetivos básicos de desenvolvimento orientado a domínio". Um pouco antes disso, em seu prefácio, Martin Fowler diz:

Eric também consolida várias das coisas que nós aprendemos ao longo dos anos. Primeiro, na modelagem do domínio, você não deve separar os conceitos da implementação. Um domínio modelador efetivo não pode somente usar um quadro branco com um contador, mas também escrever Java com um programador. Isso é parcialmente verdade porque você não pode construir um modelo conceitual útil sem considerar problemas de implementação. **MAS A RAZÃO PRIMÁRIA POR QUE CONCEITOS E IMPLEMENTAÇÃO PERTENCEM JUNTOS É ESTA: A MELHOR QUALIDADE DE UM MODELO DE DOMÍNIO É QUE ELE PROVÊ UMA LINGUAGEM UBÍQUA QUE CONECTA TECNÓLOGOS E EXPERTS EM DOMÍNIO JUNTOS.**

Parece bastante importante, não parece?

O Padrão de Repositório (*Repository Pattern*), por outro lado, foi catalogado por Fowler em seu livro *Patterns of Enterprise Application Architecture* (2002), e é apresentado por Evans na Parte II. O autor diz que a meta desta parte "é focar nos tipos de decisões que conservam o modelo e implementação alinhados um com outro, cada um reforçando a eficácia do outro".

O grande número de buscas por *Padrões da camada anticorrupção* é ainda mais preocupante. Isso é apresentado na Parte IV, e essa parte "permite que as metas da Parte I sejam realizadas em uma larga escala, para um grande sistema ou uma aplicação que se encaixa em uma rede empresarial altamente difundida".

Isso é estranho. A maioria das pessoas está falando sobre o trabalho de Eric Evans, e **eles ainda parecem estar mais interessados nos blocos de construção do que no núcleo real do Domain-Driven Design.**

Essa foi a razão de eu ter criado o workshop. Em vários fóruns e grupos de usuários, as pessoas estão falando sobre Domain-Driven Design como se ele fosse apenas uma técnica de modelagem e/ou um padrão de linguagem. Dado que os padrões e as técnicas que o livro menciona não são nada novos na Orientação a Objetos, preciso perguntar: **se Domain-Driven Design é apenas sobre como criar um objeto modelo, como ele é diferente do que foi desenvolvido em décadas de Orientação a Objetos?**

Repositórios, Value Objects, Entities, Especificações, nada sobre isso é novo. Se Evans tivesse escrito o livro apenas para catalogar esses padrões, ele apenas estaria repetindo o que outras pessoas já falaram anos atrás. É muito improvável que o trabalho dele tivesse qualquer relevância ou gerasse muito interesse.

Mas esse não é o caso. Domain-Driven Design usa os padrões, técnicas e estratégias para permitir a única coisa que essa técnica realmente traz: **manter o modelo e a implementação alinhados, e usar uma linguagem que reflita este modelo.** Citando o livro mais uma vez:

Em *Domain-Driven Design*, três usos básicos determinam a escolha de um modelo.

1. O modelo e o coração do projeto moldam um ao outro.
2. **O modelo é a espinha dorsal de uma linguagem usada por todos os membros do time. Por conta da ligação entre o modelo e a implementação, desenvolvedores podem falar sobre o problema nessa língua. Eles podem se comunicar com os *experts* de domínios sem traduções no meio. E por conta de a linguagem ser baseada no modelo, nossas habilidades linguísticas naturais podem se dirigir para aperfeiçoar nosso próprio modelo.**
3. O modelo é conhecimento destilado.

E esse livro atinge o objetivo de mostrar como vincular modelos, linguagem e implementação de um jeito brilhante. O problema é que as pessoas estão muito excitadas sobre os "novos" e brilhantes padrões para prestar a devida atenção ao que importa, mesmo que o autor repita várias vezes isso pelo livro!

Acredito que o problema aqui possa ser causado pelo fato de que a maioria das pessoas que conheço faz do livro *Domain-Driven Design* como seu primeiro livro de design do mundo real orientado a objetos. Não acho que seja um bom livro para isso. Na verdade, acho que é uma escolha terrível. O livro de Evans deveria ser sua terceira, talvez quarta, escolha de livros sobre este tópico. É muito melhor lê-lo depois de ter lutado contra modelos que não possuem uma língua unificadora.

8.2 AS PESSOAS NÃO PRECISAM ENTENDER

DOMAIN-DRIVEN DESIGN PARA SE BENEFICIAREM COM ISSO

E isso provavelmente é uma coisa boa. Como um consultor, estou sempre no lado do cliente, e sempre trabalhando em times de diferentes níveis. Frequentemente tenho um time que sabe apenas o básico do básico de Orientação a Objetos, e possivelmente não consegue digerir conceitos tais como aqueles apresentados por Evans.

Mas tudo bem, eles não precisam. Uma das melhores coisas sobre usar o jeito de pensar do Domain-Driven Design é que ele fornece um simples framework, para descobrir o que - e não como - fazer. Mas a melhor coisa sobre Domain-Driven Design na maioria dos cenários é que as pessoas não precisam saber o nome dessa técnica.

Nos meus workshops, explorei muito isso. Apenas após quatro horas de múltiplas atividades que focavam apenas em modelar um domínio, definir uma linguagem e refatorar, eu introduzi os blocos de construção — e a única razão para ter introduzido os padrões durante o workshop foi porque eu tinha pessoas que já haviam lido o livro e tinham questões para perguntar sobre ele.

E isso é o mesmo tipo de coisa que eu faço com os meus times. Eu tentarei nunca usar a palavra *Entity* ou *Value Objects*, mas explicarei os diferentes tipos que a identidade e o ciclo de vida dos objetos têm. Não direi que eles têm um Repositório de Usuários, mas modelarei para eles algumas classes que representam uma lista de coisas.

Minha experiência diz que o livro não é o suficiente. Minha experiência diz que o workshop também não é o suficiente. Minha experiência diz que as pessoas precisam ver o objetivo do Domain-Driven Design aplicando a linguagem ubíqua em um projeto real.

Referências

EVANS, Eric. *Domain-Driven Design:* Tackling Complexity in the Heart of Software. Addison-Wesley Professional, 2003.

EVANS, Eric. *Patterns of Enterprise Application Architecture.* Addison-Wesley Professional, 2002.

Capítulo 9

ARQUITETURA ÁGIL: 4 ESTRATÉGIAS COMUNS

— Artigo publicado em maio de 2010.

Na ThoughtWorks (https://www.thoughtworks.com/), nosso jeito preferido de começar um projeto era fazendo um conjunto de workshops e sessões com stakeholders por duas semanas. É o que chamamos de *Inception*. Depois da Inception, normalmente nós tínhamos um backlog do produto para o projeto e estávamos prontos para começar a escrever o código da produção.

Durante esse período, nós criávamos uma visão técnica para o sistema. É um artefato bastante incompleto, basicamente uma metáfora do sistema (http://c2.com/xp/SystemMetaphor.html), apenas bom o bastante para começar o projeto. Nessa etapa, sempre recebia olhares engraçados do time de desenvolvimento e arquitetura do cliente, e questões como: *"Então, nesse tal Agile, não temos arquitetura? Apenas começamos a codar sem pensar sobre como as coisas são estruturadas em um nível maior?"*.

Bem, não realmente. Nós resolvíamos problemas de arquitetura, mas de uma maneira bem diferente do que outras metodologias sugerem. Quando um projeto começa, eu não me importo em ter uma arquitetura sadia e completa. Eu me importo em ter uma estratégia sensata para lidar com as decisões sobre a arquitetura.

Em um projeto ágil, histórias do usuário contêm tanto valor de

negócio quanto investigação técnica/trabalho de arquitetura. O modo usual de lidar com isso é apenas acumular esses cards em um backlog de iteração, e distribuí-los em ordem prioritária. Isso funciona muito bem quando uma nova feature requere apenas mudanças de design, mas não é tão bom para decisões arquiteturais. Lembre-se, arquitetura pode ser definida como "coisas que as pessoas entendem como difícil de mudar" (FOWLER, 2003).

Como um Líder técnico/Gestor de iteração, experimentei e aprendi com várias diferentes estratégias para lidar com essas decisões "difíceis de mudar". Neste capítulo, quero discutir quatro delas.

9.1 ITERAÇÃO ZERO

Um jeito de lidar com problemas de arquitetura é buscar resolvê-los na primeira iteração para um projeto, que é frequentemente chamada de Iteração Zero. O time vai trabalhar em toda a investigação técnica e decisões arquiteturais que eles sabem que precisam ser endereçadas de uma vez só, antes de começar efetivamente a escrever algum código de produção.

Em teoria, essa estratégia tira proveito de certa economia de escopo (https://en.wikipedia.org/wiki/Economies_of_scope). Provavelmente é mais eficiente agrupar seus problemas técnicos e arquiteturais enquanto os resolvem. Histórias do usuário influenciariam a arquitetura definida na primeira iteração; elas deveriam conter 100% do valor de negócio, e nenhum risco técnico conhecido deveria ser endereçado enquanto trabalham com elas.

Entretanto, há problemas escondidos nisso:

- **Desperdício:** você gastará tempo planejando e arquitetando coisas para cartões que podem nunca

serem feitos.

- **Não entrega valor de negócio:** você está traduzindo métodos ágeis para um cliente. A primeira coisa que você quer dizer a eles é que, em toda iteração, eles terão algum valor de negócio. Depois de todo o trabalho convencendo-os, você começa o projeto e não entrega nada na primeira iteração. Isso pode se tornar um problema tão grande que gestores de conta da ThoughtWorks às vezes pedem para times que estão seguindo essa estratégia chamar isso de "Preparação", "Fase de Investigação Técnica" ou "qualquer coisa menos uma iteração!".

- **Você não sabe o que não sabe:** você pode tentar endereçar toda a investigação técnica agora, mas se o seu projeto é realmente ágil, o que você sabe agora é apenas uma fração do que você saberá em um período de suas semanas. Você não está fazendo compromissos no último momento responsável (POPPENDIECK; POPPENDIECK, 2006), e isso realmente reduz os benefícios dos métodos ágeis.

- **Não há tempo suficiente:** é muito comum na Iteração Zero quebrar o *timebox*, e continuar por várias semanas, até meses. A quantidade de investigação técnica e trabalho — que, lembre-se, pode ou não ser realmente necessária — não caberá em uma iteração regular.

Eu uso Iteração Zero, mas não uso como o ponto principal para o design e a investigação técnica. Minha Iteração Zero visa obter uma história pronta, de ponta a ponta. Esta precisa ser uma história importante, para que o trabalho realizado para entregar possa ser

reusado em outros projetos. Dado que nós não temos estrutura em vigor ainda, isso é frequentemente muito trabalho a ser realizado por um único desenvolvedor ou um par. O que eu normalmente faço é quebrar a história em tarefas, no estilo Scrum.

Desse jeito nós teremos alguma coisa para mostrar para o cliente no *showcase* e, de acordo com a *Regra do Segundo Cartão*, teremos feito design suficiente para reduzir risco para as próximas iterações.

9.2 HISTÓRIAS TÉCNICAS

Esse é o estilo sugerido por Philippe Kruchten. Nessa estratégia, trabalho de arquitetura e design têm seus próprios cartões que são priorizados e feitos juntos com cartões de valor de negócio. Chamamos esses cartões de *Tech Stories*, Histórias técnicas, ao contrário de Histórias do usuário. Teoricamente, a maioria das decisões arquiteturais serão feitas enquanto fazemos Histórias técnicas. O desenvolvedor, ao fazer a História do usuário, se beneficiará do fato de que todas as decisões já estão feitas.

O maior benefício dessa estratégia é que ela cria visibilidade sobre os riscos técnicos e o trabalho de design requerido para o projeto. Quando você mistura requerimentos técnicos e de negócio em uma única história, é muito difícil de entender o quanto de esforço lá é técnico e o quanto é relacionado ao negócio; você frequentemente tem de justificar para o negócio por que "adicionar um link na home page" é, na verdade, um grande trabalho, porque sua arquitetura não está pronta para isso.

É bom que não machuca a natureza *just-in-time* dos métodos ágeis. Você pode programar as Histórias técnicas para serem feitas em qualquer iteração, idealmente apenas antes de trabalhar uma História de usuário dependente.

Contudo, na minha experiência, isso não é tão bom:

- **Desperdício:** assim como a Iteração Zero, Histórias técnicas significam que você talvez gaste tempo investigando e planejando coisas que não serão úteis quando o requerimento mudar. E eles vão mudar.

- **A arquitetura torna-se uma baixa prioridade:** se esses cartões são colocados no backlog do produto, isso significa que eles precisarão ser priorizados pelo negócio, e é realmente difícil para eles entenderem o valor dessas tarefas técnicas. Desenvolvedores continuarão tendo os mesmos debates, tentando convencer que esses acrônimos têm valor real.

- **Cria uma cadeia de dependência:** a dependência entre Histórias técnicas e Histórias do usuário é realmente difícil de administrar. Se uma História técnica demorar mais do que o esperado, isso pode bloquear não uma, mas muitas Histórias do usuário!

- **Há ainda uma coisa faltando:** independentemente do tempo gasto fazendo investigações técnicas antecipadas, você não será capaz de cobrir todas as coisas possíveis para todas as Histórias do usuário. Design e investigação técnica ocorrerão dentro da História do usuário de qualquer forma.

Histórias técnicas acontecerão em qualquer projeto não trivial. Eu tento minimizar seus números, eu não quero meu cliente pagando adiantado por um esforço técnico de que ele pode precisar ou não. Eu quero que o cliente entenda que toda vez que adicionamos uma feature, há um custo, e que parte desse custo é derivado puramente de trabalho técnico.

Também odeio pedir ao cliente priorizar coisas que eles não entendem. Como posso explicar para um gestor de produto que “Refatorar a classe de usuário" é mais importante do que “Editar detalhes de usuário"? Muitos decisões arquiteturais importantes não podem ser diretamente mapeadas em um benefício tangível. Você pode tentar convencer seu cliente, mas você terá essa mesma conversa toda semana, pelo resto do projeto.

Quando trabalho em projeto no qual, por algum motivo, Histórias técnicas são a norma, eu geralmente aplico uma técnica chamada *technical budget*, ou Orçamento técnico (Krutchen chama de *“buffers”*). Nela, cada iteração tem uma capacidade fixa (normalmente expressa com a % da Velocidade) alocada na História técnica. Isso funciona, porém é bem ineficiente. O tamanho do orçamento sempre será muito grande ou muito pequeno; você muitas vezes acaba tendo de renegociar aumentos de buffer a cada iteração.

9.3 SPIKE

Fazer qualquer design ou investigação antecipadamente se trata principalmente da redução de riscos. Riscos só podem ser mitigados com geração de informação (veja o livro *Managing the Design Factory*, de Donald G. Reinertsen, para uma excelente discussão sobre isso). O modo canônico de gerar informação em projetos ágeis é chamado de *Spike* (http://www.extremeprogramming.org/rules/spike.html).

Um Spike é um experimento cujo objetivo é gerar informações. Pode ser algo como: "Nós podemos usar JSON.Net para serializar nossos dados em vez de fazer usando um *template engine*?" ou "nós devemos dividir esse *blob* em um sistema de três camadas?".

Um desenvolvedor ou um par fazendo um Spike tentará gerar a

informação requerida, muitas vezes escrevendo um pouco de código. Esse esforço tem um período e o Spike não acrescenta para a velocidade do time. Um Spike não removerá completamente a necessidade de uma investigação técnica e o risco das Histórias de usuário. Ele apenas reduzirá o esforço requerido ao fornecer alguma informação adquirida antes que o cartão seja trabalhado.

Meus problemas com essa técnica:

- **A arquitetura torna-se uma baixa prioridade:** assim como na Histórias técnicas, Spikes precisam ser priorizados pelo gerente do produto.

- **Alguns serão inúteis:** Spikes talvez acabem gerando nenhuma informação útil ou qualquer coisa do tipo. Claro que os desenvolvedores aprenderão alguma coisa, mas essa informação pode não ser nem um pouco relevante para o projeto.

- **É descartável:** o código escrito para um Spike deverá ser eliminado quando seu objetivo for atingido (ou seja, informação for gerada). Código derivado de Spike tende a ter uma baixa qualidade, e é possível que desenvolvedores apenas improvisem alguma solução simples para descobrir depois que não é aplicável para o código de produção.

Uso Spikes todo o tempo. Contudo, a maioria deles não é um código descartável. Eu frequentemente exijo que meu time tenha a mesma qualidade padrão com os Spikes como temos com o código de produção.

Como priorização, isso é um problema menor do que com Histórias técnicas. Histórias técnicas supostamente têm valor por elas mesmas, e isso é difícil de comunicar para as pessoas do

negócio. Entretanto, com Spikes, eu normalmente peço por duas estimativas: uma sendo "se esse cartão foi feito antes do Spike", e outra sendo "se esse cartão é feito depois do Spike". Em muitos projetos, não realizar o Spike antes da História do usuário significa uma estimativa 60% mais alta.

9.4 LINHA DE MONTAGEM

A última estratégia que quero discutir é o que chamamos de *Assembly Line*, ou Linha de montagem. Nela, a parte de desenvolvimento de uma História do usuário é dividida em duas ou mais seções, uma para investigação técnica/arquitetura, e outra para o desenvolvimento real (ou seja, cumprindo os critérios de aceitação)

Histórias de usuário primeiro são colocadas em alguma fila de investigação técnica e, depois que alguém faça a investigação e design necessários, eles movem-se para a fila real de desenvolvimento.

Isso é realmente útil quando há uma enorme diferença no trabalho e/ou na habilidade requerida para planejar uma solução e cumprir os critérios de aceitação.

Os problemas que vejo aqui são:

- **O que é "pronto"?:** é realmente complicado definir o que significa “pronto” para o trabalho realizado na fila de investigação. Espere muitas discussões intermináveis (e inúteis).

- **Cartões bloqueados para sempre:** quando usamos essa estratégia, muitas vezes eu tinha até dois pares de desenvolvimento e nenhum cartão livre para ser trabalhado. Eles estavam todos na fila de investigação

técnica, e ela normalmente tinha menos desenvolvedores do que a fila de desenvolvimento.

- **Desenvolvedores vs. "Arquitetos":** é muito fácil criar uma distinção entre pessoas que fazem investigação técnica e pessoas que apenas trabalham nos cartões. Os desenvolvedores que estão apenas fazendo os cartões normalmente acabam com muito pouco interesse e compromisso com o projeto. Quando encaram um problema técnico, eles provavelmente apenas movem o cartão de volta para a fila de investigação técnica em vez de trabalhar nele.

- **Jogue por cima do muro:** a separação do trabalho em dois passos tende criar uma cultura "jogue por cima do muro" (YIP, 2009).

Essa técnica é, às vezes, exatamente o oposto do que estamos tentando alcançar com os métodos ágeis. Vejo a Linha de montagem como uma estratégia que faz sentido só de vez em quando - recuperar um projeto é um bom exemplo - e somente por um período limitado. Impactos sobre a moral de um time que usa essa estratégia por mais tempo do que o necessário são enormes, e a divisão do trabalho cria todos os tipos de problemas relacionados com a cultura do compartilhamento de informação e culpa.

É possível que o seu projeto exija algum design antecipado pesado para alguns cartões, mas se isso é um cenário comum para você, provavelmente há algo errado. Design não deve ser independente da implementação (CALÇADO, 2009).

Referências

CALÇADO, Phillip. *Expressive Design (in 20 minutes).* Mar. 2009. Disponível em: http://www.slideshare.net/pcalcado/expressive-design-in-20-minutes.

FOWLER, Martin. *Who Needs an Architect?* IEEE Computer Society, 2003. Disponível em: http://martinfowler.com/ieeeSoftware/whoNeedsArchitect.pdf.

POPPENDIECK, Mary; POPPENDIECK, Tom. *Implementing Lean Software Development:* From Concept to Cash. Addison-Wesley Professional, 2006.

YIP, Jason. *Problems I Know You Have.* Out. 2009. Disponível em: http://www.slideshare.net/jchyip/problems-i-know-you-have.

Capítulo 10

COMO CRIAR UM REPOSITÓRIO

— Artigo publicado em dezembro de 2010.

Com exceção dos padrões catalogados por Eric Evans (capítulo *Deixe para lá o DDD*), o assunto repositório é provavelmente o mais popular.

Há algum tempo, persistência tem sido um tema popular quando falamos em desenvolvimento de software. O principal problema é que a abordagem mais popular para desenvolvimento de software nesses dias, Orientação a Objetos, não mapeia facilmente sistemas de armazenamento externo eficientes, como banco de dados relacionais ou mesmo NoSQL.

As limitações técnicas são, na maioria das vezes, resolvidas com alguma ferramenta fantástica de mapeamento de objetos, como o Hibernate, que tornam fáceis a consulta e persistência de objetos para a maioria dos cenários. Então, o problema é como integramos o ato de persistir e recuperar objetos com o nosso modelo de domínio e, mais importante, com nossa linguagem ubíqua.

A maioria das pessoas usa objetos especializados – DAOs e `Data Mappers` em geral – para converter objetos de negócio de/para seus equivalentes. Estes objetos são normalmente bons o bastante para a tarefa, mas eles pertencem à Camada de Infraestrutura, e não se integra à linguagem ubíqua de modo

transparente.

Um bom jeito de integrar as necessidades da persistência e da linguagem ubíqua é usando o que conhecemos como Repositórios. Em seu livro, Evans define o padrão Repositório como: *"Um mecanismo para encapsular armazenamento, recuperação e comportamento de busca que emula uma coleção de objetos"*. Esse conceito é facilmente assimilado pela linguagem ubíqua, e simples o suficiente para implementar e explicar para *experts* de domínio.

10.1 NOMEAÇÃO

O conceito de um repositório como uma lista de objetos não é tão difícil de entender, porém é muito comum para estas classes acabarem com métodos que não são relacionados com as listas de nenhum jeito.

Depois de treinar vários times na adoção de uma linguagem ubíqua e padrões relacionados, descobri que o melhor jeito de fazer as pessoas lembrarem de que repositórios não são DAOs (como classes) começa com a forma como você os nomeia.

Anos atrás, Rodrigo Yoshima (*@rodrigoy*) me contou sobre sua convenção quando ele nomeava repositórios. Em vez de um nome mais comum, como vemos a seguir:

```
class OrderRepository {
List<Order> getOrdersFor(Account a){...}
}
```

Ele defende isso:

```
class AllOrders {
List<Order> belongingTo(Account a){...}
}
```

Parece uma mudança pequena, porém ajuda bastante. Como um exemplo, vamos olhar dois repositórios que contêm métodos que

não pertencem a eles. Qual deles você acha mais fácil de identificar como problemático?

```
//classic naming style
class UserRepository{
 User retrieveUserByLogin(String login){...}
 void submitOrder(Order order){...}
}

//client code
User u = userRepository.retrieveUserByLogin(“pcalcado”);
userRepository.submitOrder(new Order());

//Yoshima’s naming style
class AllUsers{
 User withLogin(String login){...}
 void submitOrder(Order order){...}
}

//client code
User u = allusers.withLogin(“pcalcado”);
allusers.submitOrder(new Order());
```

Tenha em mente que a linguagem que você usa impacta em como você pensa (http://en.wikipedia.org/wiki/Linguistic_relativity). Métodos que começam com `retrieve` , lista de `get` são normalmente maus cheiros (*bad smells*).

10.2 EVITE UMA EXPLOSÃO DE MÉTODOS

Um bom repositório vai modelar os conceitos de domínio em sua interface. Como um exemplo, vamos imaginar que nós temos uma regra de negócio que diz que *todo pedido feito em um final de semana tem uma sobretaxa de 10% aplicada a ele.* Se nós queremos mostrar todos os pedidos nessa situação, nós podemos fazer algo como:

```
List<Order> surchargedOrders = allOrders.placed(user, IN_A_SATURDA
Y);
surchargedOrders.addAll(allOrders.placed(user, IN_A_SUNDAY));
return surchargedOrders;
```

Isso funciona bem, mas estão faltando abstrações aqui. O fato de que pedidos com sobretaxa são colocados em finais de semana não deveria estar exposto aos clientes desse jeito. Algo como isso seria melhor:

```
return allOrders.surchargedFor(user);
```

O problema com isso é que, para algumas entidade, você pode acabar tendo muitos métodos `querying` em um repositório. Existem diversas maneiras de se lidar com isso. Você pode parametrizar a chamada do método com uma `flag` ou uma especificação, por exemplo:

```
Specification surcharged = specifications.surcharged();
return allOrders.thatAre(user, surchanged);
```

> **Nota**
>
> Neste exemplo anterior, considero as especificações do objeto como um repositório de especificações.

Entretanto, há uma outra estratégia de que eu gosto: múltiplos repositórios. No nosso exemplo de pedidos, não há nenhuma razão para termos dois Repositórios: `AllOrders` e `SurchargedOrders`. `AllOrders` representa uma lista que contém todos os pedidos no sistema, e `SurchargedOrders` representa um subconjunto disso. Em nosso caso, nós acabaríamos tendo algo como:

```
//returns all orders
return allOrders.from(user);

//returns only orders with applied surcharge
return surchargedOrders.from(user);
```

O "conjunto de repositórios" será normalmente modelado como classes, mas ele pode ser apenas instâncias parametrizadas de um

repositório base. Por exemplo, nós podemos ter algo como:

```
//a base Repository
class Users {
 private User.Status desiredStatus = null;

 public Users(){
   this(User.Status.ANY);
 }

 public Users(User.Status desiredStatus){
   this.desiredStatus= desiredStatus;
 }
 //methods go here...
}

//instantiated somewhere as
private Users allUsers = new Users();
private Users activeUsers = new Users(User.Status.ACTIVE);
private Users inactiveUsers = new Users(User.Status.INACTIVE);
```

Obviamente, sempre existe o risco de nós substituirmos *explosão de métodos* com uma *explosão de classes*. Porém, minha experiência diz que se forem agrupados e modelados segundo a linguagem ubíqua, isso não se tornará um problema.

10.3 SOMENTE UM TIPO

Outro problema popular com repositórios acontece quando eles começam a se parecer mais uma mala cheia de coisas do que com uma coleção. A estratégia de nomeação, descrita anteriormente, pode ajudar a clarear esse problema, mas em muitos casos isso não parece ser uma grande coisa até que você acabe com um repositório com mil linhas de código e um acoplamento aferente tão alto que toda entrada inclui mudanças para essa classe.

Veja um exemplo de um sistema no qual eu já trabalhei:

```
public interface AllServices {

    List<Service> belongingTo(List<Account> accounts);
```

```
    Service withNumber(String serviceNumber);

    List<Service> relatedTo(Service otherService);

    List<Product> allActiveProductsBelongingTo(List<Account> accou
nts);

    List<Product> allProductsBelongingTo(List<Account> accounts);

    ContractDetails retrieveContractDetails(String serviceNumber);
}
```

Parece que o desenvolvedor começou aplicando a convenção de nomes do Yoshima, mas eventualmente começou a colocar todos os tipos de métodos relacionados no repositório. Isso não é mais modelar uma coleção, e o nome `AllServices` não faz sentido nenhum.

Um mau cheiro de design para se procurar quando criamos repositório é quando mais do que um tipo é retornado dos métodos. Tudo bem retornar tipos básicos, como strings e booleanos, mas se o seu repositório retornar mais do que um tipo de objeto de domínio, pode ser melhor você separar essa mala de coisas em coleções separadas:

```
public interface AllServices {

    List<Service> belongingTo(List<Account> accounts);

    Service withNumber(String serviceNumber);

    List<Service> relatedTo(Service otherService);
}

public interface AllProducts {

    List<Product> activeBelongingTo(List<Account> accounts);

    List<Product> belongingTo(List<Account> accounts);
}

public interface AllContractDetails {
   ContractDetails forServiceNumber(String serviceNumber);
```

```
}
```

Muitas vezes, repositórios acabam desse jeito porque uma dada classe tem acesso a tudo que é preciso para retornar mais do que um objeto e seria um desperdício criar `wrappers` para cada tipo. Neste caso, você provavelmente deverá ainda modelar seus repositórios como entidades diferentes, e implementá-los por meio de uma classe, desse jeito:

```
public class BillingSystemGateway implements AllServices, AllProdu
cts , AllContractDetails {

    List<Service> belongingTo(List<Account> accounts){...}

    Service withNumber(String serviceNumber) {...}

    List<Service> relatedTo(Service otherService) {...}

List<Product> activeBelongingTo(List<Account> accounts) {...}

List<Product> belongingTo(List<Account> accounts) {...}

ContractDetails forServiceNumber(String serviceNumber) {...}
```

Entretanto, isso não é muito comum. Se você está se deparando com esse cenário muito frequentemente, pode ser uma boa hora para revisitar seu código de integração.

10.4 NÃO APENAS PERSISTÊNCIA

Geralmente, repositórios são usados para modelar a persistência de objetos, mas esse não precisa ser o único caso. Eles são muito úteis para integração de sistemas (CALÇADO, 2010), coleções simples em memória e até mesmo para retornar `Value Objects` .

Tenha em mente que o principal benefício dos repositórios é ter explicitamente um lugar de onde objetos vêm, e fazer esse objeto parte de uma linguagem ubíqua. A real implementação de um repositório pode ter um grande impacto em como nós modelamos

sua interface, mas, no final do dia, nós devemos ter como objetivo tê-lo tão parecido quanto possível com uma lista de objetos de domínio.

Referências

CALÇADO, Phillip. *Everyday Tales: Anatomy of a Refactoring*. Fev. 2010. Disponível em: http://philcalcado.com/2010/02/24/everyday-tales-anatomy-of-a-refactoring/.

Capítulo 11

A UNIDADE DO TESTE DE UNIDADE É A UNIDADE DA MANUTENIBILIDADE

— Artigo publicado em junho de 2011.

Toda vez que eu falo com um cliente que passou por algum tipo de transformação Ágil, seja lá o que isso significa, eu frequentemente acabo tendo uma conversa interessante sobre testes com o time local de desenvolvimento.

Por alguma razão, muitos desenvolvedores acham que a combinação de teste de unidade + teste de integração + teste funcional é apenas para iniciantes. Essas pessoas passaram por muita coisa, como você sabe. Eles costumavam ter todos os tipos de problemas entregando software que fizesse qualquer sentido para o negócio, mas depois de anos e muitos dólares gastos em consultoria e recrutamento, eles agora já são *pro*.

E isso quer dizer andar de bicicletas sem rodinhas. E aparentemente isso nos leva a "escolher tendenciosamente os testes que fazem sentido para nós". É, não estou tão certo quanto a isto.

Os testes são especificações executáveis (CALÇADO, 2010), e o que estamos fazendo ao pular um nível do teste é deixá-lo como um comportamento indefinido. John Regehr (2010) tem vários posts em seu blog sobre como comportamentos indefinidos afetam a

linguagem de programação C, e acho essa citação particularmente importante para nossa discussão:

> Se qualquer passo na execução de um programa tem um comportamento indefinido, então a execução inteira não tem significado. Isso é importante: não é que a avaliação tem um resultado imprevisível, mas sim que uma execução inteira de um programa que avalia essa expressão não tem sentido algum. Além disso, não é que a execução é significativa a ponto de um comportamento indefinido acontecer: na verdade, os efeitos negativos podem preceder essa operação indefinida.

Agora, tudo bem você não se importar com um pedaço do software a ponto de escrever uma especificação apenas para o seu comportamento. Entretanto, o que acontece é que você acaba criando uma caixa preta em torno desta parte.

E tratar algo como uma caixa preta é principalmente um problema econômico.

Na década de 90, eu sabia muito sobre hardwares de computadores. Toda vez que eu tinha um problema com um hardware, eu abria minha máquina, realizava vários testes, identificava e substituía a peça responsável. Os upgrades também eram feitos dessa forma; eu comprava alguma placa de circuito, abria o computador e substituía a peça velha pela nova.

Isso era necessário porque os computadores eram monstruosamente caros. Hoje em dia, eu realmente não me chateio. Eu apenas vou a uma loja e compro um novo. A Apple também não se incomoda, ela apenas substitui o hardware com problema. De graça!

O problema aqui é que isso não é o que vemos na maioria dos casos de softwares.

Independente de quais peças que você escolha, o jeito que normalmente construímos sistemas é colocando abstrações, camada sobre camada, uma em cima da outra. Essas abstrações não são sem significado, elas desempenham um papel importante em gerenciar a complexidade e têm responsabilidades distintas.

Se uma abstração ou até mesmo um nível inteiro do seu sistema não faz valer a pena algum tipo de especificação automatizada, você provavelmente deveria se perguntar se aquela parte é realmente necessária. Existem alguns casos nos quais tudo bem acontecer isso - às vezes acontece com códigos gerados automaticamente, por exemplo - mas, na maioria das vezes, toda classe de um sistema é custosa o bastante para merecer sua própria especificação, para assim as pessoas entenderem como ela supostamente deveria funcionar.

E quando você precisa mudar alguma parte do código sem uma especificação, existem basicamente duas estratégias.

A primeira é tentar e ser extremamente cauteloso; mude apenas um punhado de linhas que você sabe que é absolutamente necessário para o que você quer fazer. Essa abordagem obviamente não é amigável com a refatoração, então eventualmente a classe acaba com um monte de comentários que dizem algo como:

```
// Não tenho a mínima ideia do que isso faz,
// mas não é seguro remover
```

A outra abordagem é fazer exatamente o que eu fiz com meu hardware do computador esses dias: apenas substitua a caixa preta inteira por outra coisa. Como você pode imaginar, isso é bem caro.

Se você se considera maduro o suficiente - seja lá o que isso

significa - para pular um nível de teste, não há nada que eu possa fazer além de implorar a você para prestar atenção nas caixas pretas (e campos minados)) que você pode acabar deixando para trás.

Referências

CALÇADO, Phillip. *Everyday Tales: We Call It Unit For A Reason.* Jun. 2010. Disponível em: http://philcalcado.com/2010/06/07/everyday_tales_we_call_it_unit_for_a_reason.html.

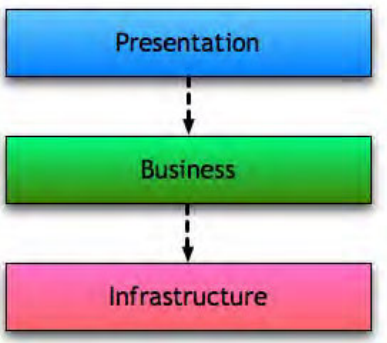
REGEHR, John. *A Guide to Undefined Behavior in C and C++, Part 1.* Jul. 2010. Disponível em: http://blog.regehr.org/archives/213>.

Capítulo 12

DATA TRANSFER OBJECTS INTERNOS

— Artigo publicado em agosto de 2011.

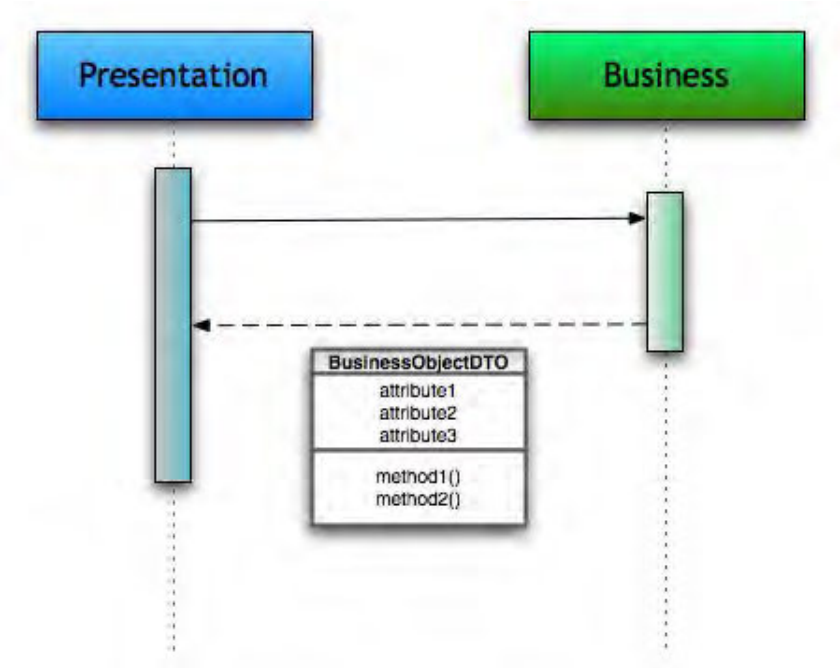
Uma questão recorrente quando tratamos sobre o uso de `Modelos de Domínio` no mundo real é como integrar a `Apresentação` e a `Camada de Negócio`. O *design pattern* Layers (*Camadas*) é uma abordagem disciplinada de como administrar dependências entre objetos. A questão levantada é como integrar as duas Camadas mais elevadas do diagrama a seguir:



Não sei por que existem tantas dúvidas sobre a relação entre essas duas Camadas e não sobre outras interações, como quando falamos sobre Persistência. Para mim, é exatamente a mesma coisa: a Camada de cima tem uma API e a de baixo usa esta API para

realizar sua tarefa. Os objetos recebidos e retornados pela API são apenas objetos padrões que abstraem detalhes da implementação sobre eles mesmos e sobre a Camada de onde eles vêm. A API frequentemente será implementada como uma *Façade*, e os objetos retornados dela serão adequados com estado e comportamento.

Mas nem todo mundo pensa desse jeito. Uma técnica comum é usar um *Data Transfer Object* (Objeto de Transferência de Dados - DTO) como uma comunicação intermediária entre essas Camadas.

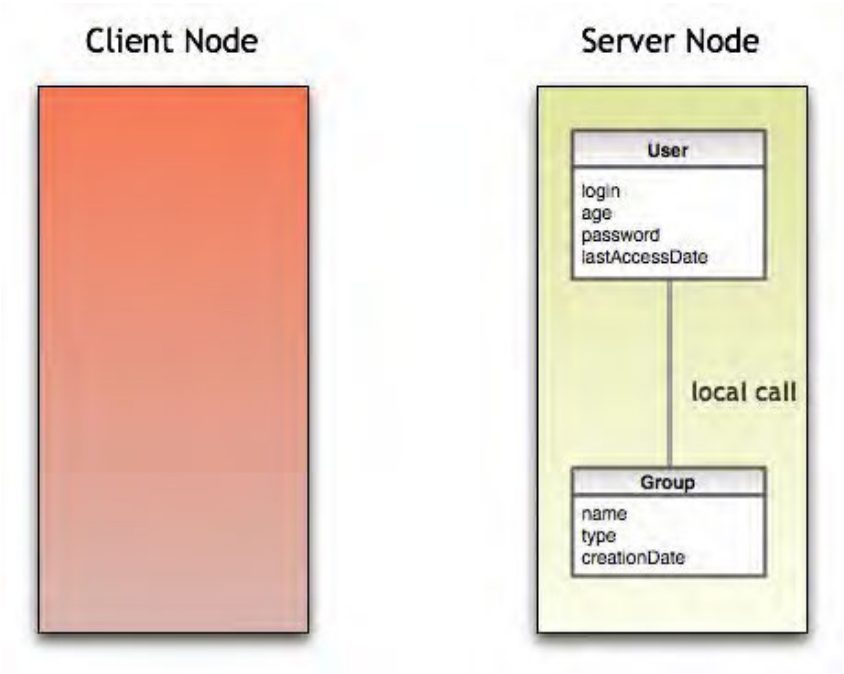


Acredito que isso não é só um exagero, como também origina algumas das arquiteturas mais confusas em sistemas aos quais já tive acesso.

12.1 UMA RÁPIDA INTRODUÇÃO A DTO

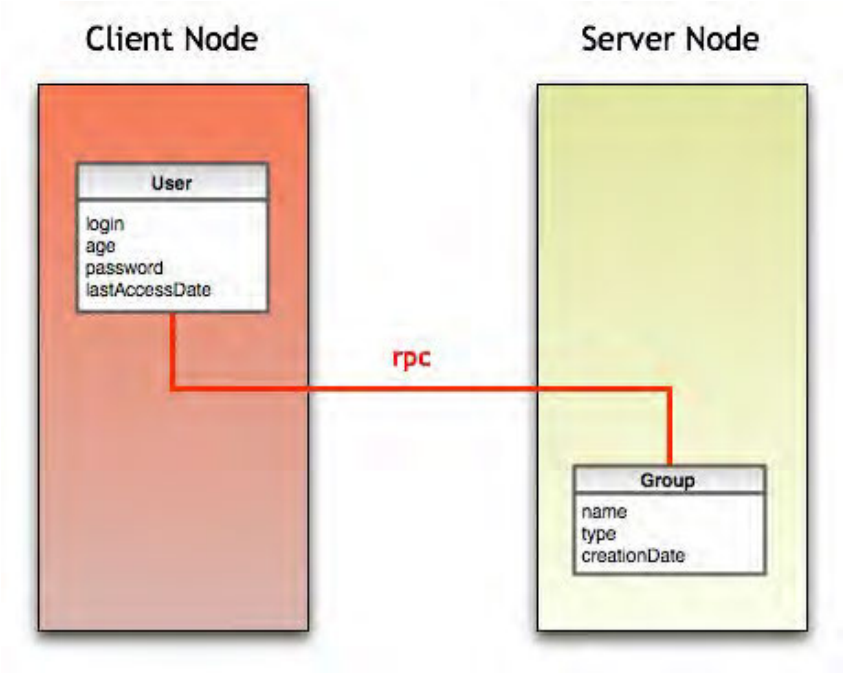
Antes de entrarmos em o que pode ter levado as pessoas pensarem que um *Data Transfer Object* é uma boa solução para esse "problema", vamos revisar o que é esse padrão.

Suponha que você tenha dois `nodes` (por exemplo, Máquinas Virtuais, processos, servidores, serviços web... aqui, "node" significa algo que tenha seu próprio lugar de endereço) e queira que eles compartilhem objetos entre eles.

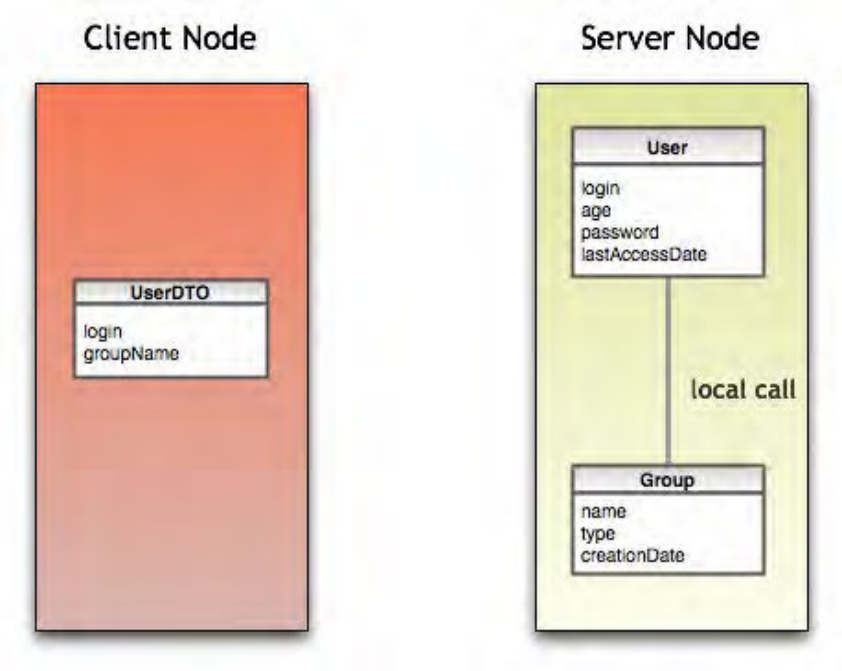


Um padrão comum é mandar um proxy ou a cópia do objeto para o cliente. O problema é que, se você seguiu as orientações da Orientação a Objetos de forma adequada, o objeto enviado dependerá de outro objeto para realizar sua tarefa. Eles podem não ter sido copiados para o novo servidor, logo, operações realizadas pelo proxy local (ou pela cópia) podem ficar sujeitas a RPC custosos ou a chamadas IPC para fazer as coisas mais absurdas, como chamar

um `toString()`.



O DTO, como catalogado por Martin Fowler, é um Padrão de Distribuição no qual você não pode enviar somente um proxy (ou uma cópia do objeto) para o cliente, mas um objeto de baixa granularidade que empacota basicamente qualquer coisa que ele precisará para realizar sua tarefa.



O objeto de baixa granularidade é otimizado para distribuição. Prefiro me referir a ele como uma versão `.tar.gz` do objeto.

Essa técnica tem alguns efeitos indesejáveis: de repente você precisa manter dois tipos de hierarquias de objetos diferentes, com um alto acoplamento entre elas, e precisa implementar um mapeamento entre uma hierarquia e outra. A hierarquia do DTO é de uma complexidade acidental, e não mapeia para nenhum conceito de negócio.

Nós usamos apenas DTO porque este é um dos diferentes jeitos que conhecemos que faz a computação distribuída funcionar. Sem ele, a maioria dos sistemas distribuídos seria extremamente lerda e ineficiente. DTOs são úteis para computação distribuída, mas é muito improvável que eles sejam necessários para comunicação local, assim como interação entre Camadas.

Já enfrentei alguns argumentos sobre DTOs internos antes. Falaremos sobre isso na próxima seção.

12.2 "PORQUE MVC PRECISA DISSO"

Há uma confusão generalizada sobre o padrão MVC em nossa indústria, especialmente sobre o que realmente significa Modelo. Vamos revisar o padrão MVC para ver se podemos encontrar alguma clareza sobre isso.

O artigo original sobre MVC (http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html) descreve Modelo como:

> Um Modelo é uma representação ativa de uma abstração na forma de dado em um sistema computacional.

Quando classifica o padrão MVC, Martin Fowler diz:

> Em sua forma orientada a objetos mais pura, é um objeto com um Modelo de Domínio (*Domain Model*). Você também pode pensar em um Transaction Script como o modelo, já que ele não contém mecanismos UX.

Em seu livro *Refactoring*, Fowler (1999) diz:

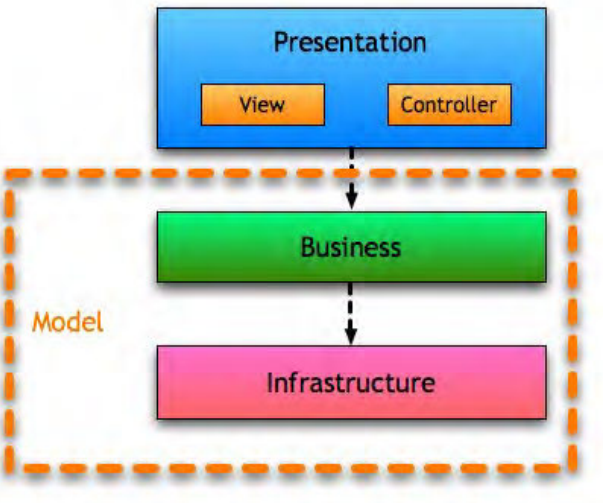
O conceito mais valioso do MVC é a separação entre o código da interface do usuário (a *Visão*, hoje em dia frequentemente chamada de *apresentação*) e a lógica do domínio (o *Modelo*). As classes de apresentação contêm apenas a lógica necessária para lidar com a interface do usuário. Objetos de domínio não contêm código visual mas sim toda a lógica de negócio. Isso separa duas partes complicadas do programa em partes que são mais fáceis de modificar. Isso também permite apresentações múltiplas da mesma lógica de negócio. As pessoas experientes no trabalho com objetos usam essa separação instintivamente, e ela provou seu valor.

Como podemos ver, não existe relação direta entre Camadas e MVC. Você pode usar um sem o outro.

Contudo, é proveitoso usar MVC com uma arquitetura com Camadas. Craig Larman (1998) explica como MVC pode ser usado para vincular as camadas:

Esse é um princípio-chave do padrão *Modelo-Visão-Controlador* (*Model-View-Controller* - MVC). MVC foi originalmente um padrão de pequena-escala de Smalltalk-80, e objetos de dados relacionados (Modelos), dispositivos da GUI (Visões), e manipuladores de eventos do mouse e teclado (Controladores). Recentemente, o termo *MVC* foi adotado pela comunidade de design também para o nível arquitetural de larga escala. O Modelo é a Camada de Domínio, a Visão é a Camada de UI e os Controladores são os objetos de fluxo de trabalho da Camada de Aplicação.

Então, é possível usar MVC para organizar as Camadas. Baseando-se no parágrafo anterior, provavelmente também é possível desenhar uma figura assim:



Logo, Camadas e MVC não são relacionados, e nosso Modelo de Domínio somado à infraestrutura inteira que suporta isso é o novo Modelo MVC. Vamos ver como o MVC original relaciona Modelo e Visão, e ver se eles têm algum jeito diferente de se comunicar entre si.

VIEW / VISÃO

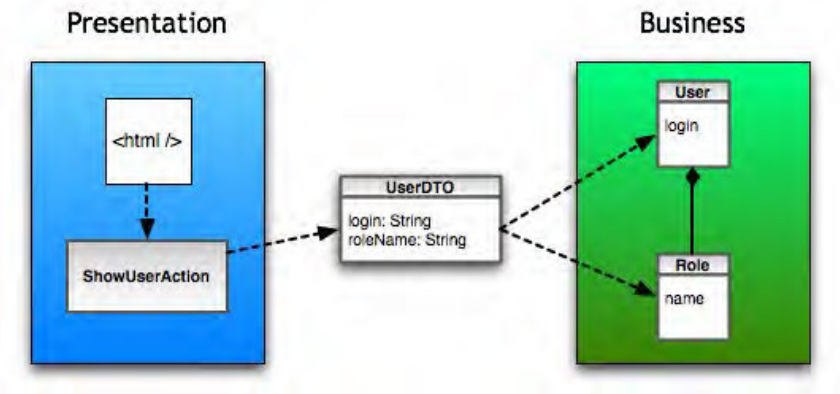
Definição:

Para qualquer dado Modelo, há em anexo uma ou mais Views, cada uma sendo capaz de mostrar uma ou mais representações do Modelo na tela e na cópia impressa. Uma View também é capaz de executar tais operações no Modelo razoavelmente associado a ela.

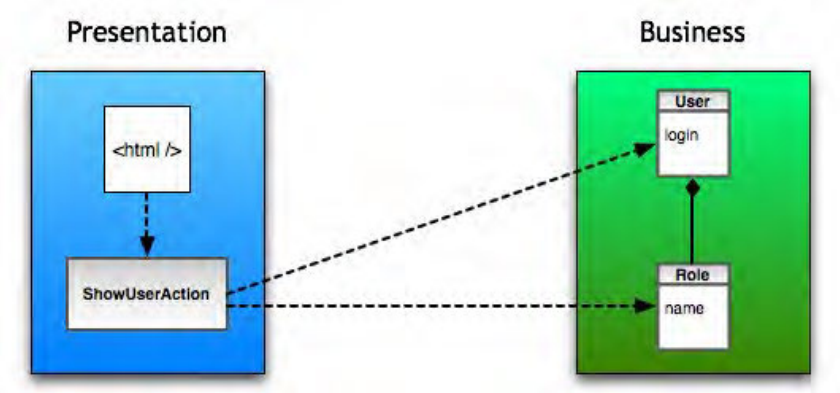
[...]

Uma View é uma representação (visual) de seu Modelo. Normalmente ela destacaria certos atributos do Modelo e suprimiria outros, agindo assim como um filtro de apresentação. Uma View é anexada ao seu Modelo (ou a parte dele) e pega dele os dados necessários para a apresentação através de perguntas. Ela também pode atualizar o Modelo ao mandar mensagens apropriadas. Todas essas perguntas e mensagens precisam estar na terminologia do Modelo, assim, a View precisa conhecer a semântica dos atributos do Modelo que representa. Por exemplo, ela pode pedir o identificador do Modelo e aguardar uma instância de `Text` ; ela pode não assumir que o modelo é da classe `Text` (REENSKAUG, 2007).

Então, a View não é só vinculada ao Modelo, como também vai filtrar seus dados e dispor somente o que for relevante. Isso é bem interessante porque algumas pessoas usam *Data Transfer Objects* para criar *viewpoints* diferentes.



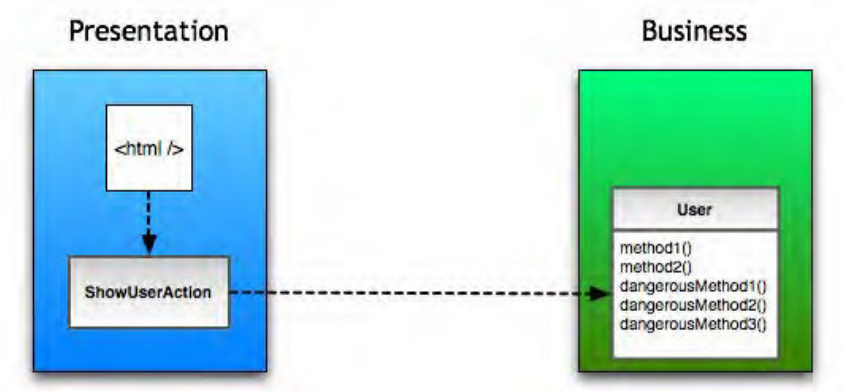
Porém, de acordo com a documentação do MVC, na verdade é esse o papel da View. Isso torna o DTO do modelo anterior completamente inútil.



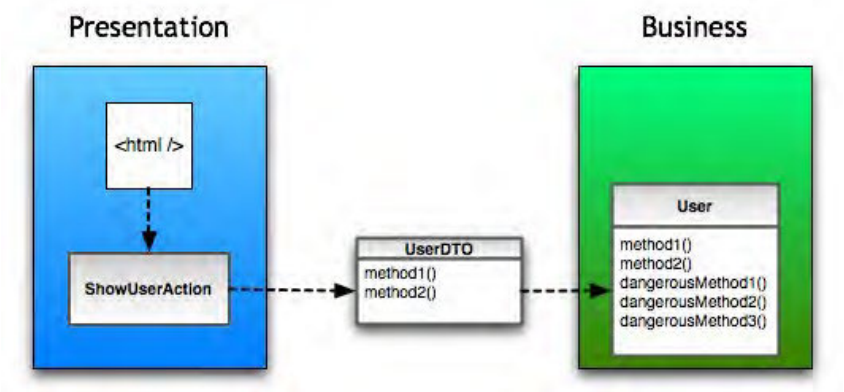
Desta forma, não há nada no padrão MVC que exija, por si só, você usar DTOs internos. A View é responsável por acessar o Model e extrair o que deveria estar disposto nele.

12.3 USANDO DTO PARA PROIBIR CHAMADAS A MÉTODOS PERIGOSOS

Outra razão comum dada para justificar a abordagem dos DTOs internos é para proibir código UI (ou seja, View e Controle do MVC) para chamar "métodos de negócio". Depois de algum tempo, descobri que, por "perigoso", na verdade as pessoas querem dizer métodos de negócios que causam efeitos colaterais.



Ao usar DTO, você pode esconder tais métodos e, teoricamente, as pessoas que estão desenvolvendo UI não serão capazes de chamá-los.

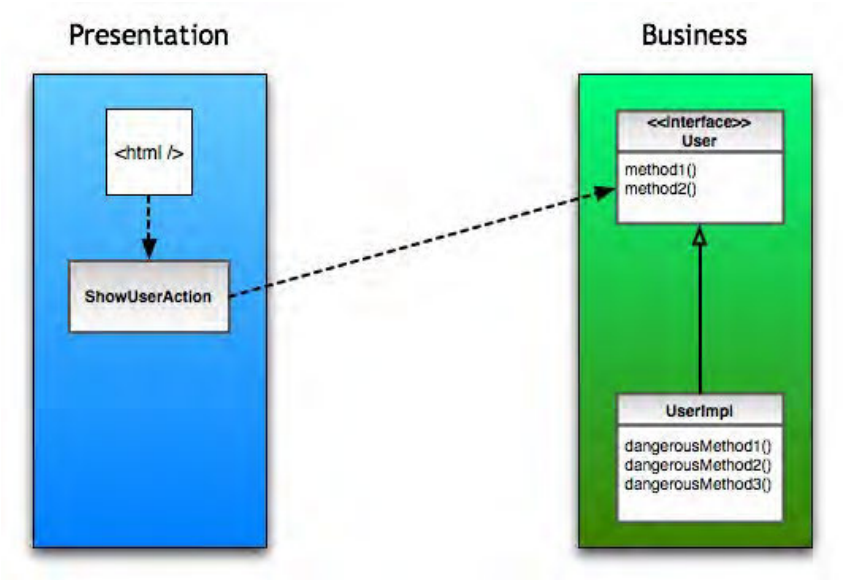


No início, isso parece sensato. As pessoas não deveriam chamar

esses métodos na Apresentação quando usam Camadas. Porém, minha solução para isso é: apenas não os chamem! Em um time responsável, não há necessidade disso; desenvolvedores não são crianças.

Mesmo se você não pode confiar no time de desenvolvimento por algum motivo, se eles quiserem, há sempre uma maneira de chamar tais métodos. Não importa quantas camadas de DTOs você use para esconder seus objetos de negócio, um desenvolvedor sempre consegue achar um jeito.

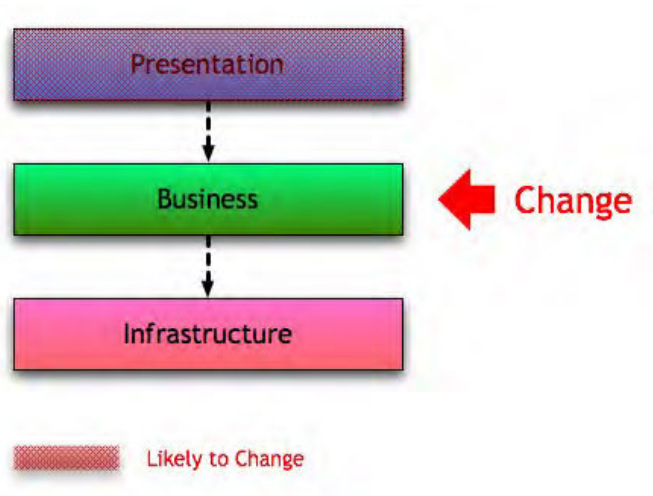
E se você realmente quiser fazer isso, existem outros modos. A solução mais simples que posso pensar é definir regras *checkstyle* (ou algo equivalente) que proíbam essas chamadas e quebrem o build. Se você quiser sofisticar, apenas defina uma interface que não tenha métodos "perigosos" e use algo como Macker (https://innig.net/macker/) para evitar chamadas à implementação.



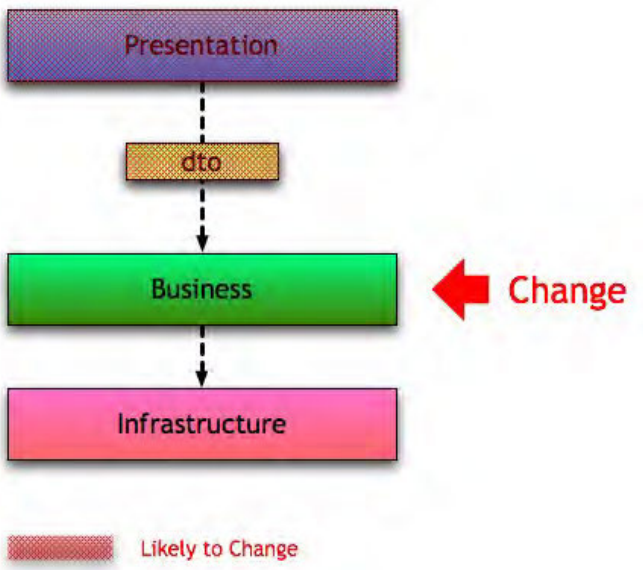
12.4 BAIXO ACOPLAMENTO

Uma dependência da Camada de Apresentação com a Camada de Negócios pode exigir código UI para ser alterada sempre que houver uma mudança nos Objetos de Negócio. Algumas pessoas acham que isso é um problema, e sugerem que DTOs entre essas duas camadas podem ajudar. Acredito que DTO aqui torne as coisas piores.

Sem o DTO, nós teríamos dois componentes, Negócio e View, e sempre que mudamos o Negócio, possivelmente temos que mudar também a View.



Mas ao adicionarmos um DTO, o problema fica pior. Agora temos três componentes: View, DTO e Negócio. Quando o Negócio mudar, é muito provável que se precise mudar o DTO, e potencialmente isso requererá uma mudança também na View.



Logo, isso não resolve o problema de acoplamento, apenas adiciona outro componente acoplado. Em vez de sincronizar dois componentes, você precisará sincronizar três.

A primeira coisa para saber é que você não pode evitar o acoplamento, independentemente da quantidade de referências indiretas que você use. Lembre-se do que David Wheeler disse:

> Qualquer problema na ciência da computação pode ser resolvido com outra camada de indireção. Entretanto, isso geralmente criará outro problema.

A chave para reduzir o acoplamento entre a Apresentação e o Negócio é definir uma boa API. Não deixe muitos detalhes sobre como a Camada de Negócio é implementada vazarem para a

Camada de Apresentação (SPOLSKY, 2002). Se sua Camada de Apresentação precisa ser extremamente desacoplada do Modelo de Domínio, pense em um Modelo de Apresentação (FOWLER, 2004).

12.5 CONCLUSÃO

Como dito anteriormente, não sei ao certo por que o padrão do Data Transfer Object é escolhido para integrar Camadas de Apresentação e de Negócio. Acho que existem dois condutores principais:

- Data Transfer Objects são muitas vezes utilizados incorretamente como registros (https://en.wikipedia.org/wiki/Record_(computer_science)). Mesmo depois de décadas fazendo programação orientada a objetos e usando ferramentas e linguagens OO, as pessoas ainda inconscientemente vão para o estilo Procedural (https://en.wikipedia.org/wiki/Procedural_programmin), em que o problema é resolvido por estruturas de dados bobas mais funções espertas. Não há nada de errado em usar esse estilo se você sabe o que está fazendo, mas esse nunca é o caso aqui.

- Sun evangeliza o uso do que chamamos de *Transfer Objects* (anteriormente chamados de *Value Objects*) em sua arquitetura EJB 2.x. Eles são DTOs internos ou remotos que costumavam resolver alguns problemas introduzidos por Entity Beans e tecnologia J2EE no geral. Em novas versões da especificação EJB e em aplicações que não usam essa tecnologia (em vez disso, usam Hibernate), o Padrão não apenas é desnecessário como também introduz novos problemas.

Os casos apresentados aqui para DTOs internos são os quais ouço com mais frequência. Tenho certeza de que existem vários outros, mas acho que todos eles são apenas exemplos de pessoas tentando usar Padrões para resolver problemas que podem nem mesmo existir. Claro que existem casos nos quais Data Transfer Objects internos podem ser valiosos. Não consigo pensar em nenhum, mas não posso negar que eles existem...

Referências

FOWLER, Martin. *Refactoring:* Improving the Design of Existing Code. Addison-Wesley Professional, 1999.

FOWLER, Martin. *Presentation Model.* Jul. 2004. Disponível em: http://martinfowler.com/eaaDev/PresentationModel.html.

LARMAN, Craig. *Applying UML and Patterns:* An Introduction to Object-Oriented Analysis and Design and Iterative Development. Prentice Hall, 1998.

REENSKAUG, Trygve. *The original MVC reports.* Fev. 2007. Disponível em: http://heim.ifi.uio.no/~trygver/2007/MVC_Originals.pdf.

SPOLSKY, Joel. *The Law of Leaky Abstractions.* Nov. 2002. Disponível em: http://www.joelonsoftware.com/articles/LeakyAbstractions.html.

Capítulo 13

COMO NÓS ACABAMOS TENDO MICROSSERVIÇOS

— Artigo publicado em setembro de 2015.

Microsserviços são uma moda nos dias de hoje.

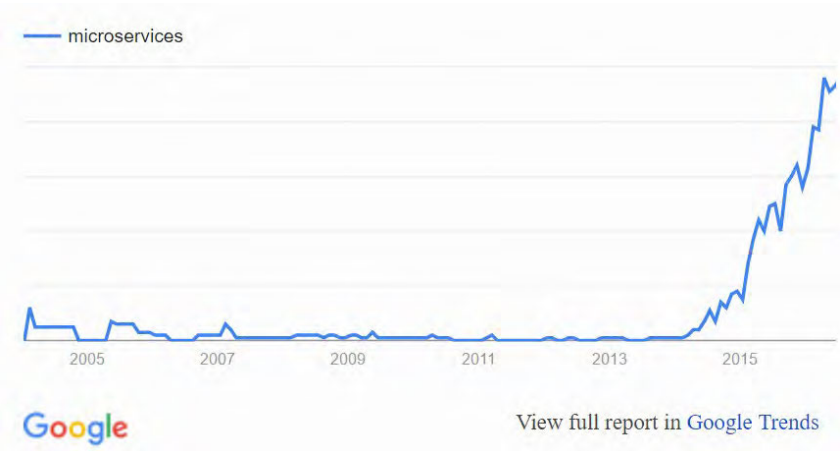


Figura 13.1: Interesse ao longo do tempo. Pesquisa online no mundo todo, de 2004 a 2016

Quando eu trabalhava na SoundCloud, era responsável pela migração de uma aplicação Ruby on Rails monolítica para uma constelação de microsserviços. Contei o lado técnico da história várias vezes, tanto em apresentações (CALÇADO, 2015) quanto em uma série de posts para o blog de engenharia da SoundCloud (CALÇADO, 2014). Essas partes sobre engenharia são aquelas que as pessoas estão mais interessadas em ouvir, mas recentemente

percebi que nunca expliquei para uma grande audiência como acabamos tendo microsserviços para começo de conversa.

Lamento decepcionar meus companheiros *techies*, mas a razão de migrarmos para microsserviços tinha muito mais a ver com produtividade do que puramente com questões técnicas. Vou explicar.

> **Nota**
>
> Este capítulo definitivamente tem um monte de *revisionismo* (https://en.wikipedia.org/wiki/Historical_revisionism), e ao tentar fazê-lo mais simples de se entender, simplifiquei uma cadeia bastante caótica de eventos em uma linha do tempo linear. Mesmo assim, acredito que retrata uma boa imagem dos meus primeiros anos na SoundCloud.

13.1 O PRÓXIMO PROJETO

Quando entrei na companhia, o projeto mais importante que tínhamos na jogada era o que chamávamos internamente de *v2*. Ele era uma completa repaginada do nosso site, e foi lançado sob a marca *The Next SoundCloud* (O Próximo SoundCloud, mais em https://blog.soundcloud.com/2012/05/09/next/).

Primeiro entrei para o time de back-end, o time de App. Nós éramos responsáveis pela nossa aplicação monolítica em Ruby on Rails. Naquele tempo, nós não chamávamos de legado, apenas chamávamos de *nave mãe*. O time de App dominava tudo no app Rails, incluindo a antiga interface do usuário. A seguir, seria uma aplicação web de única página em JavaScript (FISHER, 2012). Seguimos a prática padrão para o time (SELVITELLE, 2010) e

construímos esta página como um cliente regular para o nosso API público (CALÇADO, 2013), que foi implementado no monolito do Rails.

Esses dois times, o de App e de Web, estavam realmente isolados - nós até estávamos em prédios separados em Berlin. Nós basicamente nos víamos durante reuniões *all-hands*, e nossas principais ferramentas de comunicação eram rastreadores de problemas e IRC. Ainda, se tivéssemos de perguntar a alguém de qualquer outro time sobre como funcionava nosso processo de desenvolvimento, eles descreveriam algo como:

1. Alguém tem uma ideia para uma feature, eles escrevem alguns parágrafos e desenham uns mockups. Então, nós discutimos sobre isso como um time.
2. Os designers moldam a experiência do usuário.
3. Nós escrevemos o código.
4. Depois de alguns testes, nós fazemos o deploy.

Mas de alguma forma havia um monte de frustração no ar. Engenheiros e designers reclamavam que trabalhavam muito, mas ao mesmo tempo os gerentes de produto e companheiros reclamavam que eles nunca conseguiam fazer nada a tempo.

Por ser negócio de pequena clientela, nós realmente precisávamos ter certeza de que éramos parceiros de lançamento (você sabe, aqueles parceiros que a Apple e o Google mostram nos slides sempre que eles liberam um novo produto) para tantas empresas quanto pudéssemos, pois isso significava livre divulgação e crescimento. Nós também realmente precisávamos lançar o *The Next SoundCloud* em uma versão beta privada antes do natal; caso contrário, a temporada de feriados levaria todas as nossas iniciativas para o segundo trimestre do ano novo, já que não queríamos lançar nenhuma nova feature antes de o novo site estar no ar. Para levá-lo

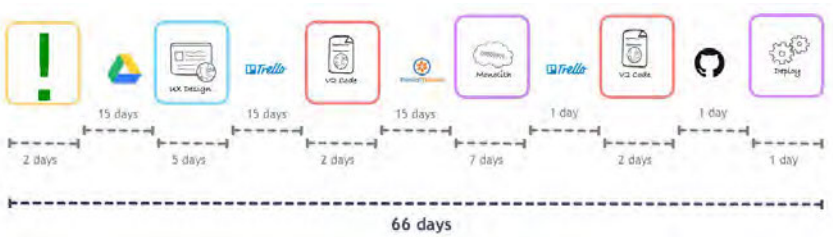
aos slides principais e ter certeza de que não desperdiçaríamos um trimestre inteiro no lançamento da feature, nós precisávamos começar a cumprir alguns prazos, para variar.

Foi nessa época que decidimos tentar entender exatamente o que nosso processo cultivado organicamente havia se tornado.

13.2 PROCESSO DE HACKING?

Antes de entrar na SoundCloud, passei muitos anos como um consultor, e uma das ferramentas mais valiosas que trouxe desse passado negro foi a ideia de criar um mapa de fluxo de valor. Não vou entediá-lo com os porquês dessa técnica, mas se o processo descrito a seguir soar interessante, pelo menos agora você sabe pelo o que googlar.

Com uma combinação de entrevistas informais com diferentes engenheiros e coleta de dados de nossos múltiplos sistemas automatizados, nós fomos capazes de desenhar um mapa para o nosso processo atual, em oposição ao processo que pensávamos que tínhamos. Não posso mostrar-lhe o documento real, mas não está tão longe do exemplo ficcional a seguir:



O fluxo real era algo parecido com:

1. Alguém apresenta uma ideia para uma feature. Então, eles escrevem uma especificação bastante breve, com alguns

mockups de tela, e guarda em um documento do Google Drive.

2. As especificações ficam nesse documento até que alguém do time comece realmente a trabalhar nisso.

3. O pequeno time de design pegaria as especificações e planejaria a experiência do usuário para a feature. Então, isso se tornaria um card no quadro do Trello do time de web.

4. O card ficaria no Trello por um tempo, pelo menos uma iteração de duas semanas, até que um engenheiro estivesse livre para pegá-lo.

5. O engenheiro começaria a trabalhar nele. Depois de converter o design em uma experiência apropriada usando dados falsos/estáticos, ele escreveria quais mudanças no Rails API ele precisaria para essa experiência funcionar. Isso iria para o *Pivotal Tracker*, a ferramenta escolhida pelo time de App.

6. O card ficaria no Pivotal até que alguém do time de App estivesse livre para olhá-lo, frequentemente demorando outra iteração de duas semanas.

7. O membro do time de App escreveria o código, faria testes de integração e qualquer coisa necessária para colocar essa API no ar. Então, ele atualizaria o card do Trello, fazendo com que o time de Web saiba que a parte dele está pronta.

8. O card do Trello atualizado ficaria no backlog por mais algum tempo, esperando pelo engenheiro do time de web terminar o que ele tinha começado enquanto esperava pelo término do back-end.

9. O desenvolvedor do time de Web faria seu código do lado do cliente combinar com todas as partes da implementação de

back-end, e daria sua aprovação para um deploy.

10. Como deploys são arriscados, lentos e doloridos, o time de App esperaria várias features chegarem à branch *master* antes de deployá-las para a produção. Isso significa que a feature ficaria na fonte de controle por alguns dias, e muito frequentemente ela era revertida devido a um problema em uma parte completamente não relacionada com o código.

11. Em algum ponto, a feature finalmente seria deployada para produção.

Haveria montes de vai e volta entre esses passos enquanto as pessoas precisassem de esclarecimentos ou viessem com ideias melhores, mas vamos ignorar isso por enquanto.

No total, uma feature demoraria dois meses para ir ao ar. Até pior: mais do que a metade desse período seria gasto em tempo de espera, ou seja, alguma parte do *Work In Progress* esperando para ser feito por um engenheiro.

Ferramentas como o mapa anterior facilitam encontrar etapas obviamente estranhas em um processo. Uma tarefa extremamente fácil que encontramos apenas ao olhar isso foi a ideia de que deveríamos adotar comboios de release para o monolito. Em vez de esperar até termos features suficientes para um deploy, nós começaríamos a deployar todo dia, logo após a reunião *standup*, independentemente de quantas features fossem para a *master*. Isso ainda estava longe de um deploy contínuo, mas aparou um pouco nosso ciclo:

Tarefas extremamente fáceis são grandes pilotos motivacionais, mas no nosso caso o principal elefante na sala foi claramente todo o vai e vem entre o desenvolvimento front-end e back-end.

O jeito como dividíamos o trabalho entre os times de Web e de App alienou completamente os desenvolvedores back-end do produto real. Eles ficavam frustrados e sentiam que eles não podiam opinar em nada do produto. A percepção era de que eles eram apenas paus mandados. Em um mercado no qual temos muito mais procura por desenvolvedores experientes do que provisão, tratar seu time desse jeito não é algo muito esperto.

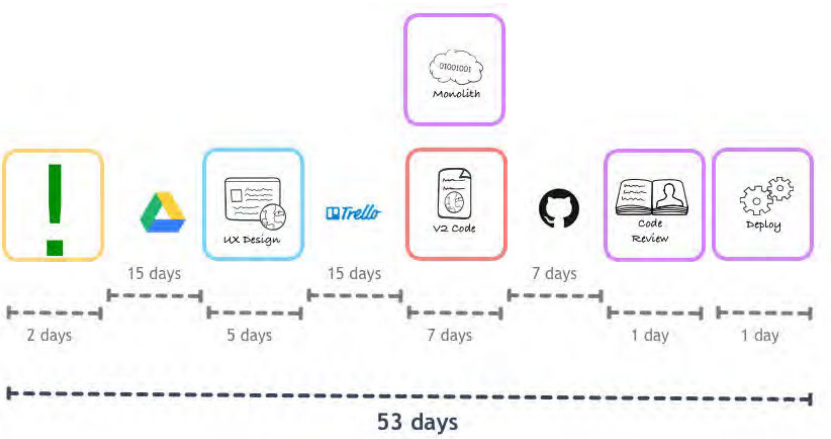
Porém, o problema que focaremos aqui é que dos 47 dias gastos na engenharia, apenas em 11 foi realmente feito algum trabalho. O restante era desperdiçado em filas e tempo de espera geral.

Há algo a ser dito sobre quanto tempo de espera foi devido à espera de uma nova iteração, mas mesmo quando mudamos para uma processo com menos iterações, como variações de Kanban, não ajudou a reduzir muito.

Então, nós decidimos fazer algo controverso: parear desenvolvedores back-end e front-end, e fazer este par ser totalmente dedicado a uma feature até ela estar completa. Nós tínhamos apenas 8 engenheiros back-end para 11 front-end, então a controvérsia em torno dessa estratégia foi devida à percepção de que precisávamos ter desenvolvedores front-end fazendo tanto trabalho

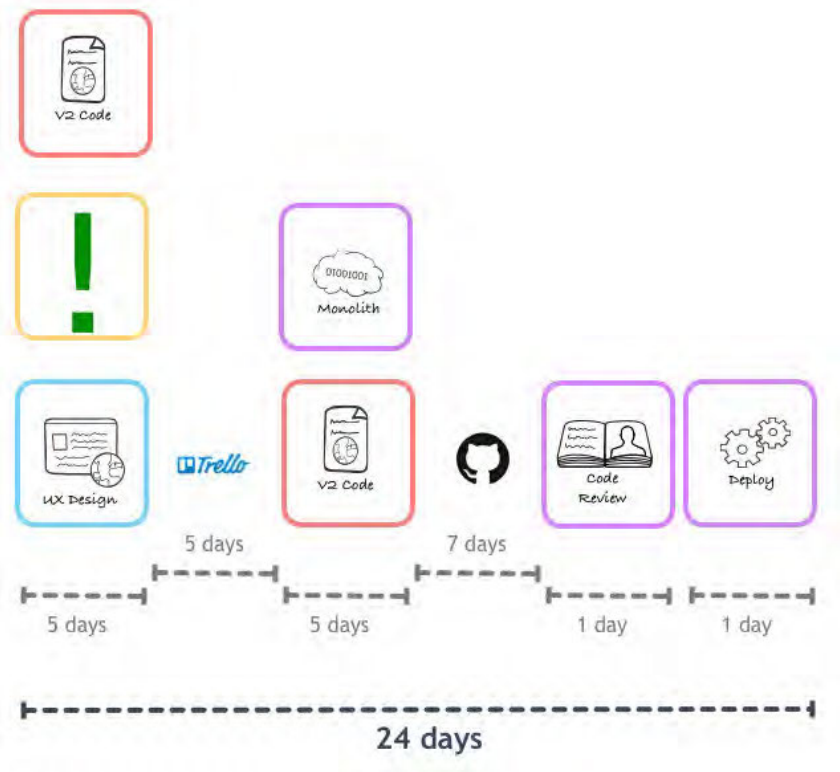
adiantado quanto possível, para que os desenvolvedores back-end gastassem o mínimo possível de tempo em cada feature. Essa configuração foi intuitiva, mas o processo de mapeamento nos mostrou que, na verdade, isso era bastante contraproducente. Mesmo descontando o vai e volta entre os desenvolvedores back-end e front-end, nós ainda tínhamos muito tempo de espera até que algo fosse realmente para a produção!

Nós decidimos tentar isso primeiro com um único par, estendendo para os outros ao longo do tempo. O novo fluxo era algo como:



Individualmente, cada pessoa acaba gastando mais tempo fazendo o trabalho por feature. Entretanto, isso foi irrelevante. Só porque eles estavam trabalhando ao mesmo tempo, eles foram capazes de terminar o código de ponta a ponta em muito menos tempo do que antes. A ressalva aqui é que, dado que o desenvolvedor back-end não era tão próximo do resto do time de App como antes, havia um processo de revisão de código obrigatória (*a.k.a.* Pull Request) antes de a mudança ir para a branch master do app Rails.

A redução foi uma façanha, e nós decidimos tentar fazer a mesma coisa com os outros passos no processo. Fizemos o designer, o gerente de produto e o desenvolvedor front-end trabalharem perto uns dos outros em todas as ideias de features, e o tempo do ciclo foi reduzido mais ainda:



Ótimas reduções! Com o workflow bem menor, nós podíamos facilmente lançar o primeiro release do *The Next SoundCloud* muitos antes do nosso prazo (BUTCHER, 2012). Mantivemos a iteração no pareamento de pessoas de diferentes áreas, eventualmente levando para a futura estrutura do time que o SoundCloud aplica hoje. Mas esse tópico é para outro dia, agora nós precisamos focar sobre o que estava acontecendo com aquela longa

fila de Pull Request.

13.3 DA NAVE MÃE PARA O LEGADO

Uma das coisas que me fizeram ficar mais excitado ao entrar na SoundCloud foi a cultura de engenharia. A maior parte dela parecia bastante similar ao que eu usava nos projetos da ThoughtWorks, mas havia um aspecto que era novo para mim: revisões de código obrigatórias.

Era 2011 e todas as startups estavam tentando replicar o modelo do GitHub, melhor resumido por Zach Holman, em sua palestra *How GitHub Uses GitHub to Build GitHub* (2011). Depois de tantos anos usando e defendendo desenvolvimento *trunk-based* (HAMMANT, 2013), eu mal podia esperar para ver como companhias de sucesso, como a SoundCloud e o GitHub, usavam essa diferente abordagem.

Naquela época, todos os engenheiros do time de App sentavam ao redor da mesma mesa, compartilhavam as mesmas tarefas do backlog, e eram geralmente muito próximos uns dos outros. O código base monolítico já estava velho, maduro e tedioso. Nós todos seguíamos os mesmos princípios e padrões em toda a base de código, commits apenas extrapolariam o design existente e não trariam nenhuma surpresa. Isso fez com que o processo de Pull Request fosse apenas uma formalidade, as pessoas deveriam gastar menos do que uma hora revisando a submissão.

Quanto mais e mais pessoas deixavam esse grupo unido para parear com pessoas do time de web, desenvolvendo as features do *The Next SoundCloud*, os canais informais de comunicação se quebraram. Deploys problemáticos, causados por mal entendidos sobre o que estava sendo deployado ou como uma feature era feita, tornaram-se frequentes. Como normalmente acontecem com seres

humanos, depois de muitos casos assim, nós decidimos que a solução seria aplicar um processo mais rigoroso em torno do *merge* das alterações. A partir de agora, antes de mandar algo para a branch master e deployar eventualmente, todas as mudanças deveriam ser formalmente aprovadas por um segundo engenheiro.

Como o mapa anterior mostra, isso acabou criando uma longa espera para os Pull Requests irem para a produção. Na tentativa de corrigir isso, o primeiro passo que tomamos foi torná-lo de tal forma que todo mundo gastaria ao menos uma hora por dia revisando Pull Requests que viessem de fora do time - ou seja, das pessoas que trabalhavam no *The Next SoundCloud*. Isso não reduziu muito a fila, e eventualmente nós percebemos que Pull Requests menores eram revisados por muitas pessoas enquanto os maiores (e os que vinham do projeto *Next* eram normalmente enormes) não eram revisados por ninguém até que o gestor de produtos gritasse com o time. Mudanças grandes requerem muito tempo de revisão e, devido ao nosso código Rails parecer um *spaghetti*, isso era bem arriscado. As pessoas evitavam esses Pull Requests grandes como se evitassem uma praga.

Nós nos juntamos e decidimos que os desenvolvedores que estavam trabalhando nas features do projeto *Next* dividiriam seus Pull Requests em pedaços menores, mais tratáveis. Isso funcionou bem no sentido de que cada Pull Requests seria revisado e mergeado rapidamente, mas ao mesmo tempo a repartição artificial de uma única feature em Pull Requests menores fez com que o revisor não conseguisse ver o cenário geral: às vezes, uma cadeia de bons commits escondia um perigoso erro arquitetural. Nós identificamos a necessidade de melhores histórias do usuário, mas treinar nossa equipe demoraria um pouco, e nós precisávamos de uma solução em curto prazo para sobreviver como um negócio.

A decisão foi aplicar o bom e velho truque: pareamento. Veja,

nossa condição era que o código deveria ser revisto por outro desenvolvedor. Com programação em par, nós teríamos uma revisão em tempo real todas as vezes, o que significava que cada commit era acompanhado automaticamente por outro. A maioria das pessoas ficou feliz com o pareamento, e para os que não ficaram foi dada a opção de continuar trabalhando sozinho, mas apenas em tarefas que não eram relacionadas ao projeto *Next*.

Nós começamos tentar isso com alguns pares, mas um problema interessante nos parou. Acontece que o código base monolítico que tínhamos era tão maciço e amplo que ninguém sabia-o por completo. As pessoas tinham desenvolvido suas próprias áreas de especialização e uma custódia em torno de submódulos da aplicação. Quando um par pegasse um card para fazer, era grande a probabilidade de que esse par não tivesse conhecimento suficiente sobre aquela parte do código base, então eles tinham de, ou esperar um expert ficar disponível e trocar de par, ou escolher outro cartão, muitas vezes de baixa prioridade. Ambas as opções eram ruins.

Um *meme* que rolava na companhia era "tudo é diversão e jogos até que alguém tenha de tocar no monolito".

13.4 A COMPLEXIDADE IRREDUTÍVEL DO MONOLITO

Para podar esses 8 dias de nosso prazo de entrega, nós precisaremos voltar um passo atrás e perguntar por que precisávamos de todos esses Pull Requests para começo de conversa. Conforme descobríamos um pouco mais sobre nosso próprio processo, nosso pensamento evoluía da seguinte forma:

1. Por que nós precisamos de Pull Requests? Porque sabemos, por anos de experiência, que frequentemente as pessoas fazem erros bobos, colocam a mudança no ar e tiram a plataforma

do ar por horas.

2. Por que as pessoas cometem erros tantas vezes? Porque o código base é bem complexo. É difícil manter tudo gravado na cabeça.
3. Por que o código base é tão complexo? Porque o SoundCloud começou como um site simples, mas com o passar do tempo cresceu para uma grande plataforma. Temos várias features, várias diferentes aplicações de clientes, muitos tipos de usuários, workflows sincronizados e não sincronizados, e uma alta escala. O código base implementa e reflete os vários componentes de uma plataforma que se tornou agora complexa.
4. Por que nós precisamos de um único código base para implementar tantos componentes? Por causa de economia de escopo. A nave mãe já tem um bom processo de deploy e ferramentas, tem uma arquitetura à prova de balas contra picos de performance e DDOS (*Distributed Denial of Service*), é fácil de escalar horizontalmente etc. Se nós construirmos novos sistemas, teremos de construir tudo isso para cada um.
5. Por que nós não podemos ter economias de escala para sistemas múltiplos e menores? *Hum...*

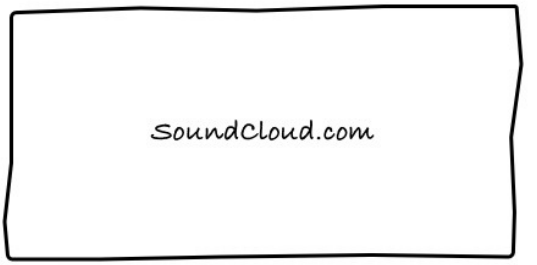
A quarta pergunta demorou um pouco para ser respondida. Nossa experiência coletiva e uma pesquisa com nossos colegas nos mostraram que havia duas alternativas possíveis:

1. *Por que nós não podemos ter economias de escala para sistemas múltiplos e menores?* Não é que não podemos, é que não seria tão eficiente quanto se mantivéssemos tudo junto em um único código base. Em vez disso, nós deveríamos construir melhores ferramentas e testes ao redor do monolito e desenvolver sua usabilidade. Este é o jeito como o Facebook e o Etsy fazem.
2. *Por que nós não podemos ter economias de escala para sistemas*

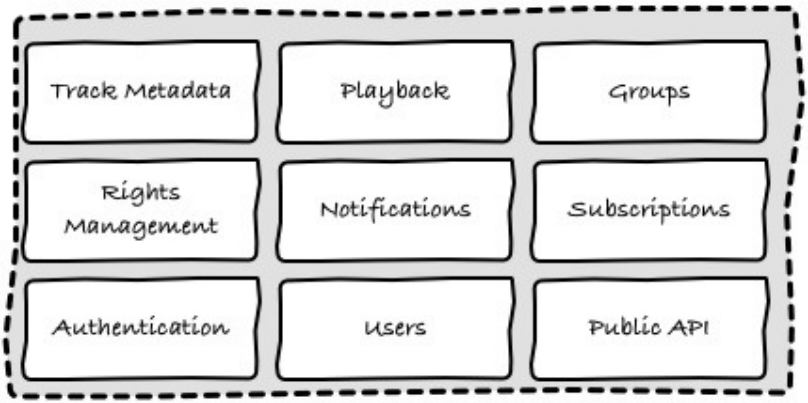
múltiplos e menores? Nós podemos. Vamos precisar fazer algumas experiências de quais ferramentas e suportes precisaremos. Além disso, dependendo de quantos sistemas separados vamos construir, vamos precisar pensar em economias de escala também, mas esse é o jeito como a Netflix, o Twitter e outros constroem seus sistemas.

Cada abordagem tinha suas propostas, e nenhuma parecia obviamente certa ou errada. As grandes dúvidas eram em torno de quanto esforço cada abordagem iria requerer. Dinheiro e recursos não eram um problema, mas não tínhamos pessoas ou tempo o suficiente para investir em nada monstruoso. Precisávamos de uma estratégia que pudéssemos implementar incrementalmente, mas começar entregando valor desde o começo.

Analisamos de novo o que tínhamos em mãos. Costumávamos pensar sempre no nosso sistema back-end de uma forma muito simples:



Essa mentalidade torna óbvio implementar este grande bloco como um único pedaço monolítico. Entretanto, como descobrimos no nosso autoquestionamento, as coisas não eram tão simples quanto a figura anterior nos levava a acreditar. Na verdade, se você abrisse essa caixa preta, você perceberia que nosso sistema era mais parecido com a figura (mais simplificada) a seguir:



Não tínhamos um único site, mas sim uma plataforma com múltiplos componentes. Cada um desses componentes tinha seus próprios donos e stakeholders, e ciclos de vida independentes.

Por exemplo, o módulo de assinaturas foi construído uma vez, e só seria modificado quando nossa porta de entrada de pagamento pedisse para nós mudarmos algo em nosso processo. Por outro lado, as notificações e os outros módulos relacionados ao crescimento e retenção sofreriam mudanças diárias enquanto nossa jovem startup tentasse adquirir mais usuários e conteúdo.

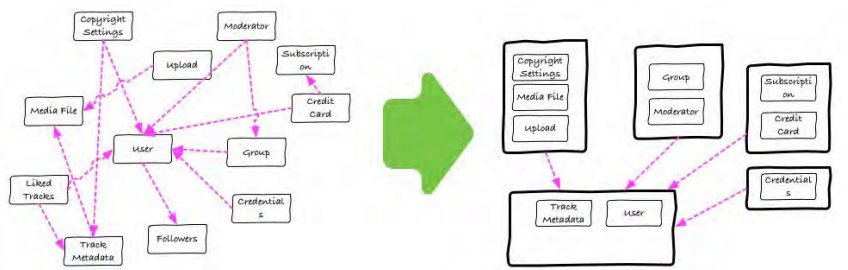
Eles também tinham diferentes perspectivas de níveis de serviço. Ninguém morreria se não tivéssemos notificações por uma hora, mas uma interrupção de cinco minutos no nosso módulo em andamento já era o suficiente para impactar fortemente nossas métricas.

Enquanto explorávamos a primeira opção, chegamos à conclusão de que a única maneira de fazer o monolito funcionar para nós seria tornar esses componentes explícitos, tanto no nosso código quanto na arquitetura de deploy.

No nível do código, nós precisávamos ter certeza de que uma

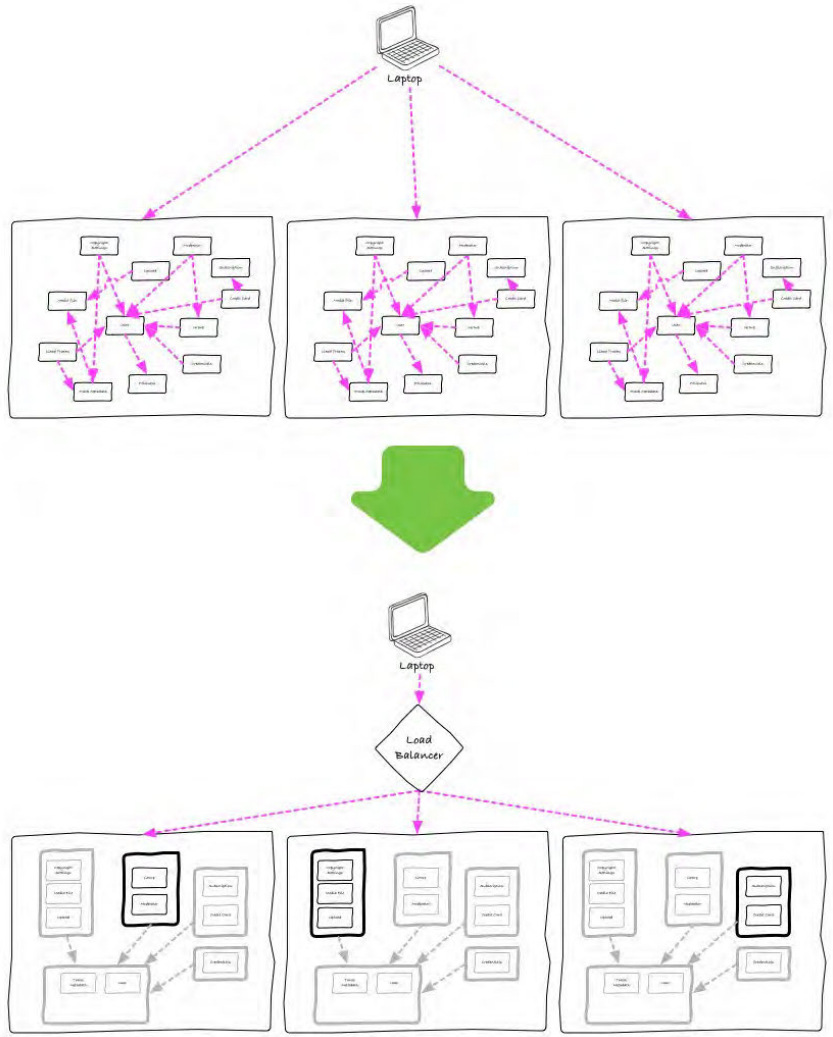
mudança feita para uma única feature pudesse ser desenvolvida em um isolamento relativo, não necessitando de que tocássemos em código de outros componentes. Precisávamos estar razoavelmente certos de que a mudança não introduzisse bugs ou mudasse o comportamento em tempo de execução em partes do sistema que não eram relacionadas a isso. Esse é um problema antigo desta indústria, e nós sabíamos que o que precisávamos fazer era tornar nossos componentes explícitos como contextos delimitados, ou *Bounded Contexts* (FOWLER, 2014), e ter certeza de que éramos cuidadosos sobre qual módulo podia depender do outro.

Discutimos usando mecanismos do Rails e várias outras ferramentas para implementar isso. Parecia algo assim:



No lado do deploy, precisaríamos ter certeza de que uma feature podia ser deployada em isolamento. Colocar uma mudança no módulo de produção não deveria requerer um novo deploy de módulos sem relação. E se isso desse errado e a produção quebrasse, a única feature impactada seria aquela que sofreu a mudança.

Para implementar isso, pensamos em ainda deployar o mesmo artefato para todos os servidores, mas usar balanceamento de carga (*load balancing*) para termos certeza de que um grupo de servidores era responsável apenas por uma única feature, isolando qualquer problema com essa feature e servidores de outros:

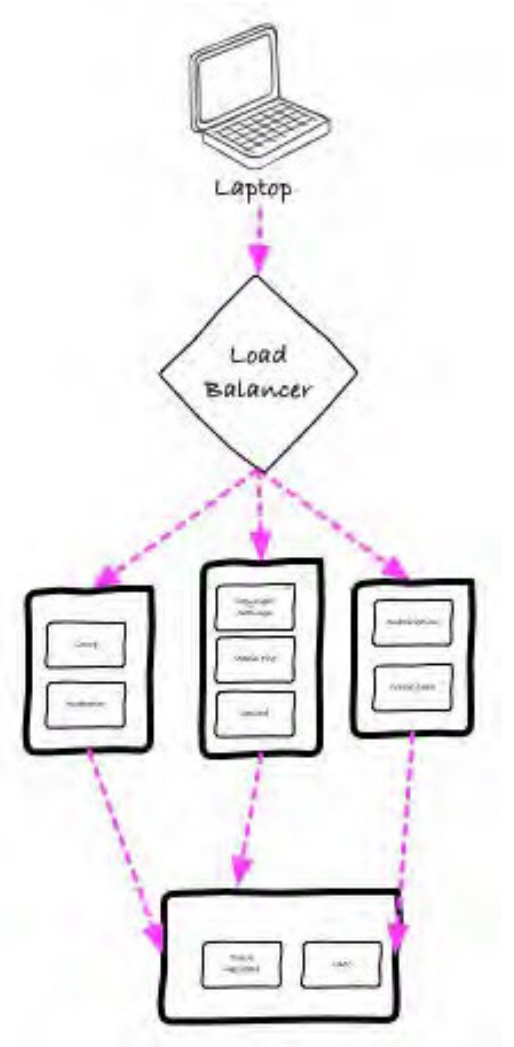


O trabalho de conseguir tudo isso pronto não seria trivial. Mesmo se não precisássemos de nenhum desvio da parte técnica e das ferramentas que estávamos usando, as mudanças traziam suas próprias questões e riscos.

Mas mesmo que tudo desse certo, nós sabíamos que o código atual para o monolito ainda precisaria ser refatorado de qualquer

forma. Ele tinha sofrido muitas mudanças durante os últimos anos, havia dívidas técnicas em todo o lugar. Além da confusão em que nos metemos, nós ainda tínhamos de atualizá-los de Rails 2.x para o 3, e isso já é um grande trabalho de migração por si só.

Essas considerações nos levaram a reconsiderar a segunda opção. Pensamos que não pareceria tão diferente:



Mas, pelo menos, nós seríamos capazes de nos beneficiar da abordagem desde o primeiro dia. Tudo novo que construiríamos se tornaria um projeto *greenfield*, e os atrasos introduzidos pelos Pull Requests não seriam necessários.

Decidimos dar uma chance, e eventualmente construímos tudo o que era necessário para nosso primeiro projeto monetizados como serviços, isolados do monolito. O projeto introduziu várias grandes features e uma completa renovação do nosso modelo de inscrição (PEREZ, 2013), e foi entregue antes do prazo por dois times compostos por dois engenheiros cada.

A experiência foi tão boa que decidimos continuar aplicando essa arquitetura para qualquer coisa nova que construíssemos. Nossos primeiros serviços foram construídos usando Clojure e JRuby, eventualmente mudando para Scala e Finagle.

13.5 REFERÊNCIA OBRIGATÓRIA À LEI DE CONWAY

É justo dizer que quase tudo o que construímos na SoundCloud desde 2013 usa serviços. Em algum ponto entre antes e agora, nós começamos a usar a palavra microsserviços para nos referir a eles, mas não tínhamos isso em mente na primeira vez que construímos usando essa arquitetura. A primeira vez que a palavra '*microsserviços*' foi usada em um e-mail na SoundCloud foi em 2013, e os primeiros serviços foram implementados em 2012.

Com o novo framework de arquitetura, fomos capazes de reduzir nosso tempo de espera para novas features para algo que, ainda longe dos bons velhos tempos, era muito mais aceitável para uma companhia que tentava participar de uma indústria musical altamente competitiva:

Até aqui tudo bem, porém isso era para novas features. Sempre que precisássemos evoluir uma feature já existente, executada no monolito, nós voltávamos para o ciclo antigo. Pior ainda, tantas pessoas estavam gastando mais tempo nesses novos microsserviços do que no monolítico, que o número de revisores disponíveis estava

caindo, e a fila de Pull Request ficando cada vez maior.

Toda vez que uma grande mudança era proposta, nós nos certificávamos de levar em consideração o tempo necessário para extrair o sistema antigo do monolito. Entretanto, de alguma forma isso nunca aconteceu. As pessoas ou ainda implementariam a mudança no antigo código base, ou criariam um híbrido estranho, em que as mudanças seriam implementadas em um microsserviço que era tão acoplado ao monolito que poderia muito bem ser o mesmo sistema.

Nessa fase, o time de App agia como um grupo de desenvolvedores back-end que era pareado com pessoas do time de Web, designers e gerentes de produto para trabalhar em uma feature para algum time. As pessoas estavam saltando de uma feature para outra toda hora, e percebemos que nós não tínhamos desenvolvido um senso de propriedade ou autonomia em relação a qualquer parte do sistema. Ninguém queria fazer um investimento arriscado de extrair alguma parte antiga do código se eles não se sentissem responsáveis por isso. É um exemplo do velho ditado: o que é responsabilidade de todo mundo não é responsabilidade de ninguém.

Pensamos em dividir o grupo em times menores, focados em áreas específicas. Depois de passar um bom tempo tentando achar uma boa lógica de separação para formar os times, não conseguimos concordar com nada. Era um exercício frustrante e, em algum ponto, eu apenas separei o grupo em times de 3-4 pessoas, e semialeatoriamente dei a eles responsabilidades sobre módulos.

Foi explicitamente dito a esses times que eles tinham total propriedade em cima desses módulos. Isso significava que eles estariam de plantão para qualquer interrupção relacionada a eles, mas também tinham a liberdade de evoluí-los para qualquer formato que eles achassem razoável. Se eles decidissem manter algo

no monolito, seria uma decisão deles. Afinal, eram eles que estavam mantendo aquele código mesmo.

Como você pode imaginar, vimos um êxodo do monolito. Mensagens, estatísticas e a maioria das features repaginadas que precisávamos para o novo app iOS foram extraídas do principal código base.

Tudo estava indo bem, mas meu modo semialeatório de dividir times tinha um grande problema: um único time era responsável pela maioria das features e objetos realmente fundamentais no ecossistema, coisas como trajetória e metadados dos usuários e o gráfico social. Esse time estava combatendo incêndios constantemente, e não tinha incentivo nenhum em migrar seus módulos para microsserviços já que isso introduziria ainda mais riscos e potenciais interrupções.

Este problema só foi abordado recentemente. Ainda mantemos um único time responsável por esses objetos, mas agora nossa arquitetura é muito mais estável, reduzindo o tempo necessário para combater problemas. Nós finalmente podemos arcar com essas pessoas lidando com a extração dos módulos do monolito como um projeto próprio.

Ainda hoje a SoundCloud tem o código monolítico vivo, mas sua importância diminui a cada dia. Ainda existe um caminho crítico para muitas features, mas devido a um sistema de estranguladores (FOWLER, 2004), já deixou de ser voltado à internet (*internet-facing*). Não tenho certeza de que ele algum dia irá embora, algumas features que ele provê são tão pequenas e estáveis que pode até ser mais barato mantê-lo lá para sempre, mas dou-lhe um ano para o monolito não estar em mais nenhum caminho crítico.

13.6 O FUTURO

Como mencionado no começo do capítulo, essa é uma versão supersimplificada da nossa jornada com os microsserviços.

Meus últimos 12 meses na companhia foram realmente focados nas economias de escopo e escala que queríamos explorar. Por mais que eu fique repetindo que o termo microsserviços não significa muito, uma coisa que você pode ter certeza quando alguém usa essa palavra para descrever sua própria arquitetura é de que haverá uma série de serviços. Enquanto as organizações crescem, elas precisam ser cuidadosas com o custo estabelecido de cada serviço.

Eu e meus times gastamos muito tempo pensando em como explorar nossas restrições e ter certeza de que operar essa arquitetura era menos caro e complexo do que operar o monolito. Esperançosamente, alguns dos trabalhos serão feitos de forma open-source, então certifique-se de se inscrever no blog de engenharia deles (https://developers.soundcloud.com/blog/).

Ao longo dos anos, nós aprendemos muito e, quando deixei a SoundCloud, estava bastante certo de que a estrutura geral e a organização de times (essas coisas vão juntas) fortaleceriam a empresa para alcançar altos voos ao longo dos próximos anos - talvez até permitam que os unikernels e os nanosserviços se tornem uma moda?

Referências

BUTCHER, Mike. *Soundcloud Revamps Site, Announces New Numbers:* Reaches 180M Users A Month and Counting. Dez. 2012. Disponível em: https://techcrunch.com/2012/12/04/soundcloud-revamps-site-announces-new-numbers-180m-users-and-counting/.

CALÇADO, Phillip. *APIs:* The Problems with Eating your Own Dog Food. Out. 2013.

CALÇADO, Phillip. *Building Products at SoundCloud—Part III:* Microservices in Scala and Finagle. Jun. 2014. Disponível em: https://developers.soundcloud.com/blog/category/scala.

CALÇADO, Phillip. *No Free Lunch, Indeed:* Three Years of Microservices at SoundCloud. Ago. 2015. Disponível em: https://www.infoq.com/presentations/soundcloud-microservices.

FISHER, Nick. *Building The Next SoundCloud.* Jun. 2012. Disponível em: https://developers.soundcloud.com/blog/building-the-next-soundcloud.

FOWLER, Martin. *StranglerApplication.* Jun. 2004. Disponível em: http://www.martinfowler.com/bliki/StranglerApplication.html.

FOWLER, Martin. *BoundedContext.* Jan. 2014. Disponível em: http://martinfowler.com/bliki/BoundedContext.html.

HAMMANT, Paul. *What is Trunk Based Development?* Abr. 2013. Disponível em: http://paulhammant.com/2013/04/05/what-is-trunk-based-development/.

HOLMAN, Zach. *How GitHub Uses GitHub to Build GitHub.* Set. 2011. Disponível em: https://zachholman.com/talk/how-github-uses-github-to-build-github/.

PEREZ, Sarah. *SoundCloud's New "Moving Sounds" Feature*

Holds Promise For Ads, Viral Content Like Tumblr's Gifs. Mar. 2013. Disponível em: https://techcrunch.com/2013/03/11/soundclouds-new-moving-sounds-feature-holds-promise-for-ads-viral-content-like-tumblrs-gifs/.

SELVITELLE, Britt. *The Tech Behind the New Twitter.com*. Set. 2010. Disponível em: https://blog.twitter.com/2010/the-tech-behind-the-new-twittercom.

Capítulo 14

PEDINDO AOS CANDIDATOS PARA CODAR

— Artigo publicado em março de 2016.

Estávamos fazendo uma mudança no processo de recrutamento de desenvolvedores back-end na DigitalOcean. Simplificamos a descrição do emprego e o processo de entrevista, e adicionamos uma etapa que pede para o candidato nos escrever um código. Este é um relato da minha experiência em processos de contratação e seu uso de revisões de código.

Por volta de 2005, fui entrevistado para uma posição de gerente na Globo.com, o braço da internet da maior conglomerada de mídia da América Latina. Eu estava voltando para o desenvolvimento de produtos a alguns anos como consultor, e superexcitado com a oportunidade.

Uma semana depois que comecei, meu novo chefe me mandou um e-mail com o que seria minha primeira tarefa: contratar quatro pessoas para se juntar com as outras quatro pessoas que já estavam no meu time. Era uma situação estranha; Eu nem saberia dizer que tipo de pessoa eu precisaria já que não havia começado a trabalhar ainda!

Comecei a pensar sobre que tipo de pessoas eu sabia que seriam

perfeitas para quase qualquer tipo de trabalho de engenharia, o tipo de pessoa que eu tentaria caçar. Então pensei quais traços e hábitos particulares essas pessoas tinham em comum. Logo, percebi um padrão comum: o tipo de pessoa que pensei ser ótima para qualquer tipo de trabalho de engenharia sempre estaria lendo livros. Não apenas livros de referências que todo mundo lê para o seu emprego diário, mas textos sobre softwares de arquitetura, design orientado a objetos, linguagens de programação misteriosas, e conselhos sobre como melhorar em sua profissão.

Eu tinha um blog sobre programação razoavelmente popular no Brasil, e escrevi um anúncio de emprego como um post de blog. Descrevi o emprego o melhor que pude, e pedi para as pessoas me mandarem seus currículos e uma lista com os três últimos livros que tinham lido.

A resposta foi excelente. O e-mail temporário que havia feito para isso estava cheio de currículos, e pude confirmar minha hipótese: os currículos mais interessantes vinham com uma lista que continha livros das categorias citadas anteriormente.

Na Globo.com, nós contratamos um grande time seguindo esse processo, e alguns de nós ainda usamos algo semelhante em nossas novas organizações.

Alguns anos se passaram e eu me candidatei para um emprego na ThoughtWorks. Eu havia trabalhado para companhias internacionais antes, nas quais precisava falar inglês quando interagia com outros escritórios, mas essa seria a primeira vez que eu precisaria falar inglês constantemente por mais de uma hora. E através do Skype. E tendo pessoas avaliando minhas habilidades em codar. Eu estava pirando.

Felizmente, a ThoughtWorks tinha um processo que era um pouco diferente dos padrões habituais das empresas daquela época.

Depois de uma rápida chamada de recrutamento, eles me mandaram três opções de um desafio de código, um pequeno problema que eu poderia resolver em qualquer linguagem de programação que quisesse. Então, a minha submissão do código seria usada durante as entrevistas seguintes, incluindo uma sessão de pareamento, na qual eu e um *ThoughtWorker* tentaríamos ampliar meu código adicionando uma nova feature.

Eu passei quatro anos na ThoughtWorks e vi esse processo produzir bons resultados constantemente e várias vezes. Também foi importante não ditar quais linguagens ou ferramentas um candidato teria de usar. Minha experiência com a ThoughtWorks foi que eu poderia optar por, ou um projeto interessante, ou uma linguagem de programação interessante (tenho escrito *web crawlers* de grande escala em Drupal e softwares de planilhas em F#), assim nós realmente praticamos uma contratação pela atitude, e não pelas habilidades (TAYLOR, 2011).

Depois de todos esses anos, era hora de seguir em frente. Tive uma boa experiência na ThoughtWorks, mas meu último projeto foi provavelmente meu pior também. Enquanto procurava por uma nova posição, receava ter de passar por sessões de codificação em um quadro branco, com um quebra-cabeça inútil - o tipo de coisa que meus amigos reportaram que tiveram de passar nas grandes empresas da internet.

Mais uma vez, fui sortudo e acabei me candidatando para uma posição na SoundCloud. Naquela época, era apenas um punhado de engenheiros, e assim como na ThoughtWorks, o processo começou com uma chamada de recrutamento e um desafio de código. Porém, diferentemente da ThoughtWorks, o desafio não era um quebra-cabeças, mas algo bem parecido como o trabalho que eu teria de fazer lá. A SoundCloud queria que eu construísse um *uploader*, desde a interface do usuário até qualquer camada de back-end que

eu achasse que seria necessária.

Havia uma parte peculiar desafiadora: implementar uma barra de progresso em tempo real. Por mais exausto que estava depois de 8 horas fazendo meu projeto, me encontrei pensando sobre possíveis soluções em um trem de East Croydon para Londres, e indo imediatamente para minha cafeteria favorita para trabalhar nisso durante a noite. Resolver o problema foi bem divertido, e se você quiser conferir o que estava na moda em Clojure em 2011 e como alguém pode fingir conhecer JavaScript ao escrever Scheme com colchetes, você pode ver o código em: https://github.com/pcalcado/UpCloud.

Umas semanas depois, consegui o trabalho e me mudei para Berlin. Como acontece em organizações pequenas, antes mesmo de terminar meu período de provação eu estava revisando código para novos candidatos. Infelizmente, a experiência foi bem desapontadora.

Acontece que a maioria das pessoas montava um app Rails com mais linhas no seu `Gemfile` do que linhas de códigos que elas realmente escreveram. Ainda pior, muito frequentemente alguém apenas pegava o uploader do Adobe Flash de um componente Flash de um site aleatório, e nem mesmo escrevia uma única linha de código. Tão pragmático quanto um plugin somado apenas com um código mínimo de ligação, isso não era um bom uso do meu tempo ou do tempo dos candidatos: eu precisava que eles escrevessem algum código que pudéssemos ler antes de convidá-los para mais entrevistas.

Então criamos um desafio diferente. Desta vez, nós seríamos um pouco mais rigorosos. Ainda não limitaríamos a linguagem de programação que o candidato usaria — assim como na ThoughtWorks, naquela época, em uma startup pequena, precisávamos mais do que nunca de pessoas *T-shaped* (BROWN,

2005) —, mas nós pediríamos para eles usarem apenas a biblioteca padrão da linguagem, e não libs ou frameworks de terceiros.

Para tornar essa tarefa mais fácil, não pedimos um aplicativo web, mas sim algo que toda plataforma oferecesse: uma interface *socket-server* e um simples protocolo de *string*. Para encaminhá-los mais para perto das necessidades de uma companhia que cresceu de 10 para 100 milhões de usuários em menos de um ano, nós também incluímos a necessidade de clientes, para lidar com centenas de clientes de uma só vez.

Mas também fizemos outra coisa. Algo que aprimoraria a experiência do candidato ao assegurá-los de que seu código preencheria os requisitos funcionais antes de expô-lo para nós. Algo que nos poupou muito tempo ao evitar ter de revisar código que obviamente nem mesmo funcionava.

Com a descrição do problema, mandamos ao candidato uma suíte de teste funcional, um binário que, quando começasse, tentaria se conectar com a implementação do servidor do candidato, abrir vários sockets, enviar várias mensagens, e verificar o resultado contra o que a descrição do problema especificava. O candidato era instruído apenas a mandar sua submissão uma vez que o código passasse no teste funcional localmente.

A melhor submissão definitivamente foi do Flávio Brasil (*@flaviowbrasil*), que agora trabalha no Twitter, que não só achou um bug na segurança de nosso teste, mas também decompilou o código (Scala!), escreveu um caminho e submeteu como parte da sua solução do desafio. Nós recebemos submissões escritas em diversas linguagens do mundo inteiro. Nós tínhamos tantos dados qualitativos, que eu até dei uma palestra sobre alguns problemas particulares que encontramos em submissões de códigos que usavam Node.js (veja em https://www.youtube.com/watch?v=kA4-b7hvWhg).

Para essa iteração do nosso processo de recrutamento na DigitalOcean, estava tentando usar como alavanca essas experiências passadas tanto quanto possível. No geral, o desafio de código é bastante similar com aquele desenvolvido na SoundCloud, mas existem algumas grandes diferenças.

A primeira maior mudança é que nós estamos tentando simplificar o processo de entrevista, ficando mais parecido com a ThoughtWorks. Antigamente, tínhamos algumas escritas formais no quadro branco ou sessões de pareamento. Desta vez, estamos dando ao candidato uma oportunidade de falar sobre o código que escreveu, em vez de alguma pergunta genérica já catalogada em um livro best-seller (MCDOWELL, 2011).

Outra mudança é que damos maior ênfase no *caminho para a produção*, ou em como a aplicação é testada, construída e executada. Independente de quão centrados no produto nós somos, a DigitalOcean é uma companhia de infraestrutura, e quebramos a quarta parede (https://en.wikipedia.org/wiki/Fourth_wall) o tempo todo durante o desenvolvimento de uma aplicação. Mostrar interesse no build e em ferramentas em tempo de execução é um bom indicador de um encaixe em nossa cultura.

Entretanto, a mudança mais significativa é em como o código é enviado para revisão. Em empregos anteriores, como um revisor, eu recebia um código e o currículo do candidato. Ao longo dos últimos anos, a indústria e a academia construíram evidências convincentes de que nosso preconceito influencia nossas decisões quando se trata de decidir o que é código bom ou ruim (NEWITZ, 2016).

Para manter nosso viés intacto, pedimos para os candidatos não adicionarem nenhuma informação de identificação pessoal na submissão. Permitimos aos revisores saber mais ou menos qual é o nível do candidato, e quantos anos de experiência ele tem, mas não divulgamos sexo, nome, nacionalidade, localização geográfica, ou

por quais companhias ou escolas eles passaram. Isto exigiu muito trabalho da nossa incrível e impressionante equipe de recrutamento, e muita paciência de nossos revisores, mas os resultados preliminares eram muito bons.

Estou superexcitado com a nossa primeira iteração do novo processo. E mais do que encontrar a pessoa certa para crescer nosso time, eu espero que tenhamos dados suficientes para compartilhar com a indústria em algum ponto de um futuro próximo.

Referências

BROWN, Tim. *Strategy by Design.* Jun. 2005.

MCDOWELL, Gayle Laakmann. *Cracking the Coding Interview:* 150 Programming Questions and Solutions. CareerCup, 2011.

NEWITZ, Annalee. *Data analysis of GitHub contributions reveals unexpected gender bias.* Fev. 2016. Disponível em: http://arstechnica.com/information-technology/2016/02/data-analysis-of-github-contributions-reveals-unexpected-gender-bias/.

TAYLOR, Bill. *Hire for Attitude, Train for Skill.* Fev. 2011. Disponível em: https://hbr.org/2011/02/hire-for-attitude-train-for-sk.

Capítulo 15

CONCLUSÃO

Espero que estas páginas pelas quais você passou tenham aberto sua mente para um mundo talvez diferente daquele com o qual você estava acostumado. Espero que o façam repensar sobre as arquiteturas que são tão comuns de encontrarmos por aí, e que muitas vezes são vendidas como arquiteturas de caixinha. O mundo ideal não deveria ser assim.

Continuo escrevendo em meu blog pessoal, no http://philcalcado.com/. Espero que você apareça por lá, deixe um comentário em algum artigo que o interessar e discuta as ideias.

Vamos tentar fazer um mundo melhor para o desenvolvimento de software.

Um abraço.