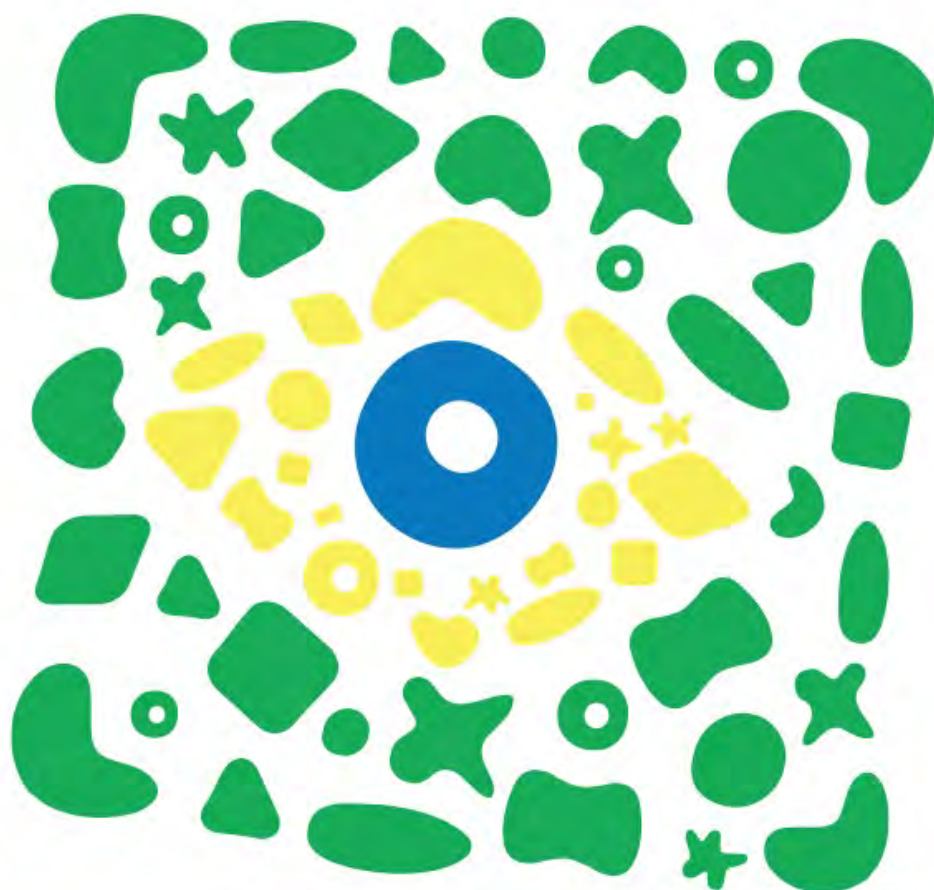


ThoughtWorks®

Antologia Brasil

Histórias de aprendizado e inovação



 Casa do
Código

EDITADO POR PAULO CAROLI
PREFÁCIO POR LUCA BASTOS

© Casa do Código

Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.

Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos, mecânicos, gravação ou quaisquer outros.

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

Prefácio

por Luca Bastos

Quando o Paulo Caroli me pediu este prefácio, a ideia era falar um pouco sobre cada capítulo. Comecei a escrever sobre cada um, mas hesitei. Em vez disso, optei por explicar o raciocínio que responde a uma antiga curiosidade que eu tinha antes de conhecer detalhes desta empresa. Antes de entrar na ThoughtWorks, sempre me intrigou como os thoughtworkers têm tempo e condições para avaliar e usar tantas práticas e tecnologias, e ainda conseguir compartilhar conhecimento em diferentes meios, como no Thoughtworks Insights [74], em blogs pessoais, livros, palestras e workshops, participação em projetos de Software Livre, além do já famoso e aguardado Tech Radar [103].

Certo dia, presenciei uma pergunta feita ao Danilo Sato por um aluno de mestrado em uma palestra no IME-USP, Instituto de Matemática e Estatística da USP.

- Danilo, sei que o conceito de BDD, *Behavior Driven Development*, foi criado na ThoughtWorks. Todo mundo lá usa esse tal de BDD?

A resposta dele surpreendeu professores, mestrandos e doutorandos presentes:

- Por acaso, meu time atualmente está usando BDD, mas há tempos não usava.

Uma das minhas primeiras descobertas na ThoughtWorks foi saber que não se adota a mesma coisa em todos os projetos. Acontece até com as técnicas criadas ou aperfeiçoadas aqui. Por exemplo, os projetos de software livre não são empregados apenas por serem desenvolvidos por outros colegas da mesma empresa.

Descobri que, em vez de unificar suas práticas e tecnologias, de buscar uma solução de referência para todos os projetos, padronizando bancos de

dados, linguagens, ferramentas e processos, esta empresa promove, experimenta e amadurece práticas, ferramenta e processos. E é isso que você, meu caro leitor, pode esperar desta antologia.

Buscar uniformidade por conveniência dificulta encontrar o que é mais adequado, e isto é terrível para criação, evolução ou a inovação de ideias e formas de trabalhar. Na ThoughtWorks, cada caso é analisado em busca de agregar experiências passadas e tendências futuras às exigências atuais do projeto. Escolhas são feitas com a visão mais ampla possível. Tudo dentro do nosso conceito de sucesso, que implica na satisfação do cliente junto com o impacto positivo na sociedade. Somos atualmente uma empresa de quase três mil pessoas, com mais de trinta escritórios em 13 países. Em cada escritório, uma média de 100 pessoas, das quais 30% são de algum outro escritório. Uma troca constante de experiência e ideias.

Temos a paixão por compartilhar conhecimento, interna ou externamente. São comuns as apresentações na hora do almoço ou depois do expediente abordando temas novos ou mostrando como determinado problema foi resolvido. Muitas dessas palestras ou workshops são feitos para todos os escritórios no Brasil, e algumas vezes assistimos a apresentações de escritórios de outros países. Tudo de acordo com o que disse o autor, editor e consultor Peter Seibel: “Como ser um engenheiro 10X: ajude 10 outros serem duas vezes melhores”.

Essa exposição ao que está acontecendo em muitos projetos aqui e lá fora é somada às condições de discutir e experimentar novas práticas e tecnologias. Foi assim que entendi como compartilham tanto conhecimento, incluindo escrever antologias como esta.

Espero que aproveitem a leitura e conheçam mais um pouco do que usamos ou fazemos. Tenho certeza de que a experiência aqui compartilhada servirá para que mais gente nos ajude na missão de melhorar a humanidade por meio do software, criando um ecossistema socialmente responsável e economicamente justo, além de mudar o papel da tecnologia na sociedade.

Autores

Disse Aristóteles: “Se vi ao longe é porque estava nos ombros dos gigantes”. Assim me sinto como editor desta obra. Foi com muito orgulho que participei da fundação da ThoughtWorks Brasil em 2009, e é com muita honra que apresento esta antologia da Thoughtworks Brasil.

Este livro contém 16 capítulos independentes, mas complementares, com histórias, ensinamentos e práticas dos meus amigos e colegas de trabalho, gigantes do desenvolvimento de software.

Paulo Caroli, editor

Segue, em ordem alfabética, a lista de autores desta antologia.

Adriano Bonat

Possui mais de uma década de experiência trabalhando com projetos de desenvolvimento de software para grandes empresas que atuam em diversas áreas de excelência como saúde, educação, e-commerce de conteúdo digital e leilões virtuais, entregando projetos com sucesso e mudando a maneira com que o software é colocado em produção. Contribui em diversos projetos open source e trabalha como consultor na ThoughtWorks, onde mentora times nas melhores práticas para construir software de qualidade.

Alexandre Klaser

Alexandre Klaser é um evangelista de métodos ágeis e entusiasta de gestão moderna e melhoria contínua. Possui mais de 15 anos de experiência trabalhando com desenvolvimento de software e atualmente é analista de Negócios na ThoughtWorks Brasil. Seu objetivo atual é aproximar clientes e equipes de

desenvolvimento, fazendo com que partilhem uma visão comum sobre como o desenvolvimento de software pode ser ao mesmo tempo divertido e lucrativo, sempre colocando as pessoas em primeiro lugar.

Alexey A. Villas Boas

Alexey é bacharel e mestre em Ciência da Computação pelo IME-USP e apaixonado por todos os aspectos de projetos de software, desde tecnologia até as questões humanas e de formação de times. Foi professor universitário em cursos de graduação em Ciência e Engenharia da Computação e trabalha com projetos de TI há cerca de 15 anos nos mais diversos papéis: desenvolvedor, analista de negócios, gerente de projetos, coach e gerente de área. Em 2012, juntou-se à ThoughtWorks e atualmente é consultor principal e gerente geral da ThoughtWorks São Paulo.

Carlos Lopes

Carlos Lopes já atuou em diversas empresas dos mais variados ramos, desde instituições de ensino até grandes portais, envolvido nos mais diferentes tipos de projetos e com as mais variadas tecnologias. Atualmente, é consultor da ThoughtWorks, onde aplica princípios e práticas ágeis no seu dia a dia.

Claudia Melo

Claudia Melo é Diretora de Tecnologia da ThoughtWorks Brasil e pesquisadora associada ao IME-USP. Doutora em Ciência da Computação pelo IME-USP, pesquisou produtividade de times ágeis no Brasil e trabalhou em conjunto com a Norwegian University of Science and Technology. Nos últimos 15 anos, Claudia se uniu a diferentes empresas brasileiras e multinacionais com foco em desenvolvimento de software. Também facilitou o aprendizado de alunos de graduação e pós-graduação por 10 anos. É participante ativa da comunidade ágil nacional e internacional, com diversas publicações e palestras no Brasil, Estados Unidos, Europa, Oriente Médio e Escandinávia.

Daniel Amorim

Agile QA Consultant @ ThoughtWorks. Trabalha no mercado de TI desde 2007, buscando as melhores práticas de testes ágeis para contribuir com o seu time – não apenas automatizar testes, mas também construir uma aplicação de alta qualidade através de trabalho colaborativo do time ágil. Ele trabalha em pesquisas focadas em automação de testes, testes não funcionais, práticas de devops, comunicação efetiva com o cliente. Palestrante e facilitador de testing dojos. Fomentador da comunidade de QAs no Brasil.

Êmerson Hernandez

Êmerson Hernandez é consultor da Thoughtworks Brasil, trabalhando principalmente com gerência de projetos. Mestre em Ciência da Computação pela Universidade Federal do Rio Grande do Sul, é apaixonado por resolução de problemas através de software e crescimento humano via ensino. Em sala de aula ou em ambientes de desenvolvimento de software, tenta ampliar o poder coletivo através da aplicação de valores ágeis.

Fabio Pereira

Fabio Pereira trabalha como consultor lead pela ThoughtWorks Austrália desde 2008. Pioneiro em metodologias ágeis, desde 2001 ele vem ajudando empresas a desenvolver software atuando em diversas áreas e papéis como arquiteto, gerente de projetos e coach. Como palestrante já participou de diversas conferências incluindo QCon e AgileAustralia. Apaixonado por fotografia e psicologia, ele acredita que o entendimento do que nos motiva e de como tomamos decisões pode ser extremamente útil para o sucesso do trabalho em equipe.

Francisco Trindade

Francisco Trindade é um desenvolvedor de software com certo interesse em inovação e empreendedorismo, e atualmente trabalha na empresa Your-Grocer, sua segunda startup, em Melbourne, na Austrália. Francisco trabalhou como consultor da ThoughtWorks por seis anos em diferentes países, passando por Inglaterra, China, Brasil e Austrália. Seus interesses passam

por tecnologia, software, pensamento sistêmico e eficiência de equipes e organizações.

Glauber Ramos

Glauber Ramos trabalha como designer da Experiência do Usuário (UX) e front-end developer na ThoughtWorks Brasil. Gosta de trabalhar em ambientes colaborativos onde possa construir a melhor experiência com foco no usuário. Tem experiência em diversas áreas como saúde, bancos, educação e e-commerce. Apaixonado pela web e a colaboração que ela propõe, ele acredita que a constante validação e refinamento é que fazem um produto de sucesso.

Guilherme Froes

Guilherme Froes é desenvolvedor de software há quase 10 anos e atualmente é consultor na ThoughtWorks Brasil, onde ajuda clientes das mais variadas indústrias a entregar valor a seus clientes através de práticas ágeis. Tem particular interesse por desenvolvimento orientado a testes (TDD) e está tentando responder a pergunta: como times ou organizações tradicionais podem se tornar ágeis?

Hugo Corbucci

Hugo Corbucci é mestre em Ciências da Computação do IME/USP no tema “Aplicação de Métodos Ágeis ao Desenvolvimento de Software Livre”. Ele é fundador e coordenador do projeto Archimedes – The Open CAD (em 2005) e fundador do Coding Dojo São Paulo (em 2007). Foi professor nos cursos de verão do IME/USP (de 2007 a 2010), onde também atuou como assistente de ensino no curso de Programação Extrema da graduação. Também já ministrou cursos sobre métodos ágeis no ICMC e foi palestrante em conferências nacionais e internacionais. Já foi desenvolvedor e assessor em métodos ágeis na Maps Risk Management Solution no período de adoção de Scrum da empresa (em 2006). É sócio-fundador da Agilbits e atuou como programador e líder de projetos, desenvolvendo sistemas desktop com Java, usando a plataforma Eclipse RCP e sistemas web com Ruby usando Rails de

2008 a 2011. Atualmente, trabalha como consultor na ThoughtWorks em Chicago, ajudando clientes a evoluir seus sistemas legados. É apaixonado por programação e trabalho em equipe além de ser um assíduo escalador.

Juraci Vieira

Juraci Vieira é um consultor senior na Thoughtworks onde trabalha com desenvolvimento de software com foco em qualidade. Tendo iniciado sua carreira na área de Qualidade de Software, sempre manteve interesse em expandir suas habilidades técnicas relacionadas a automação de testes e provisionamento de ambientes de integração contínua. Possui interesse em propagar a qualidade no desenvolvimento do software utilizando praticas como Specification by Example, BDD e DevOps.

Luca Bastos

Luca Bastos é um *principal consultant* na ThoughtWorks. Engenheiro, empreendedor, desenvolvedor do tempo da Carochinha e eterno aprendiz.

Luiza de Souza

Luiza de Souza é formada em Ciência da Computação pela Universidade Federal do Rio Grande do Sul e atualmente trabalha como desenvolvedora na ThoughtWorks Brasil. Durante a sua carreira teve a oportunidade de trabalhar em países como Alemanha, Brasil, Índia e Estados Unidos em projetos desafiadores na área da educação, saúde e e-commerce. Apaixonada pelo que faz, busca, por meio de seu trabalho, trazer mais qualidade de vida para as pessoas.

Marcos Brizenno

Marcos Brizenno é Cientista da Computação pela Universidade Estadual do Ceará e, atualmente, consultor desenvolvedor na ThoughtWorks Recife. Trabalha com desenvolvimento web e também executa atividades de pesquisa sobre novas tecnologias e tendências no desenvolvimento de software. Apaixonado por Engenharia de Software e Metodologias Ágeis.

Mário Areias

Mário Areias trabalha com TI há mais de sete anos e é desenvolvedor/consultor na ThoughtWorks Brasil. Já trabalhou em países como Brasil, EUA e Índia e também com diversos clientes, desde bancos, hospitais, e-commerce até segurança na internet. Não apenas desenvolve como também é facilitador e coach de metodologias ágeis. É participante ativo das listas de discussões do OpenMRS e é apaixonado por tecnologias open source, testes e metodologias ágeis.

Maurício Silveira Sanches

Mauricio Silveira Sanches trabalha como consultor na ThoughtWorks desde 2010 e é formado em Ciências da Computação pela Universidade Federal do Rio Grande do Sul. Desempenhou diversos papéis em sua carreira, mas acima de tudo é apaixonado por desenvolvimento e metodologia ágeis.

Natalia Arsand

Natalia Arsand é designer da Experiência do Usuário (UX) na ThoughtWorks Brasil. Gosta de desafios de usabilidade, questionamentos inteligentes, sessões de rabisco, interações intuitivas, harmonia das cores e preza por experiências incríveis. Na ThoughtWorks há 2 anos, já passou por projetos desafiadores em setores como o da saúde, educação e segurança na internet, tendo Lean UX e Design Thinking como seus aliados.

Nicholas Pufal

Nicholas Pufal atua como desenvolvedor na ThoughtWorks Brasil. Começou a programar em cima de iniciativas open source e já trabalhou em startups na área de aprimoramento de metodologias e técnicas aplicadas na definição de escopo e implementação de MVPs. Compartilha de um profundo interesse por temas como design patterns, boas práticas de desenvolvimento e em formas de aprimorar a comunicação/qualidade de entrega de times ágeis um assunto extenso que inclui ideias como BDD e Specification by Example.

Paulo Caroli

Paulo Caroli é M.S. em Engenharia de Software pela PUC-Rio, e atualmente trabalha como consultor de transformação ágil para grandes corporações pela ThoughtWorks Brazil. Possui mais de duas décadas de experiência em desenvolvimento de software com passagem em várias corporações no Brasil, Índia e EUA. Desde 2000 tem focado sua experiência em processos e práticas de Gestão e Desenvolvimento Ágil usando Lean, XP, Scrum e Kanban. É sempre palestrante convidado em vários eventos grandes no Brasil e no exterior e também autor de um sem número de artigos, e mais recentemente livros sobre métodos ágeis. Foi o idealizador, um dos criadores e palestrante desde 2009 do Agile Brazil, um dos maiores eventos de desenvolvimento de software da América Latina. Em 2006, ingressou na ThoughtWorks USA como desenvolvedor, em 2008 foi transferido para a ThoughtWorks Índia como treinador e gerente de projetos e em 2010 regressou para o Brasil para fundar a ThoughtWorks Brasil, onde desde então tem atuado como mentor e coach ágil.

Roger Almeida

Pai, esposo, autodidata, desenvolvedor trabalhando na área desde 2002. Músico nas horas vagas. Começou a trabalhar com métodos ágeis em 2006 e desde então é um aficionado pela busca do desenvolvimento perfeito.

Tainã Caetano

Atualmente desenvolvedor na ThoughtWorks Porto Alegre, Tainã tem experiência como analista de testes, gerente de projetos e coach em múltiplas equipes distribuídas entre Brasil, Estados Unidos e Índia. Coautor do livro Fun Retrospectives, regularmente ministra palestras e workshops sobre práticas ágeis e melhoria de processo. Trabalhou como coach e professor na ThoughtWorks University, um programa global para thoughtworkers recém-contratados.

Foreword

Ao leitor.

Depois de algumas décadas em um determinado campo, não é incomum para um praticante achar que já viu tudo, que as histórias, os problemas e as próprias soluções se repetem. Um dos meus grandes prazeres no campo da tecnologia foi poder escapar desse cinismo existencial. Mesmo que os problemas e as histórias se repitam, o universo tecnológico está em constante evolução e esses problemas e histórias sempre apresentam crescentes nuances que tornam a busca de soluções diferentes e inovadoras algo continuamente estimulante.

Com os anos, adquire-se um módico de entendimento e percebe-se que essa visão não é uma constante no campo. Ela exige uma mentalidade deliberada e um cultivo próprio para florescer. Como organização, a ThoughtWorks considera isso mais do que uma mentalidade - considera uma missão. Para nós, inovar, buscar novas soluções, ajudar a revolucionar a indústria de tecnologia é parte do que somos e carrega consigo também um senso de responsabilidade para com o mercado e a sociedade em balancear tecnologia com as necessidades humanas.

É com um prazer especial, então, que recebo essa antologia. Nos ensaios dispostos aqui, eu vejo essa mentalidade e esse cultivo, esse senso de maravilhamento e busca constante de contar ou renovar histórias, de qualificar problemas novos e antigos, de buscar soluções que definam e redefinam práticas e empurrem o campo em novas direções. Minha esperança é que esses ensaios forneçam a você essas percepções e vislumbres e que tragam consigo esse senso de renovação. E, acima de tudo, que iniciem o próximo ciclo.

Ronaldo Ferraz, Diretor-presidente da ThoughtWorks Brasil

Você pode discutir sobre este livro no Fórum da Casa do Código: <http://forum.casadocodigo.com.br/>.

Sumário

1	Inceptions de uma semana	1
1.1	Inception	1
1.2	Inception de uma semana	2
1.3	Estrutura de uma inception de uma semana	2
1.4	Quebrando o gelo	6
1.5	Visão do produto	7
1.6	Objetivos	10
1.7	Personas	11
1.8	Features	13
1.9	Histórias	14
1.10	Refinamento das histórias	16
1.11	O mapeamento das histórias	20
1.12	Conclusão	22
2	Produtividade e adaptabilidade de times ágeis: conceitos e paradoxos	25
2.1	Ágil é produtivo?	26
2.2	Conceitos de produtividade	27
2.3	Agilidade é sobre mudança de paradigma	31
2.4	O que significa produtividade de times ágeis?	32
2.5	Paradoxos envolvendo eficiência e flexibilidade e seu impacto na produtividade de times ágeis	35
2.6	Conclusão	37

3	Peopleware: entendendo ambientes corporativos	39
3.1	Cargos e estruturas	40
3.2	Empatia	41
3.3	Motivadores	43
3.3.1	Motivadores extrínsecos	43
3.3.2	Motivadores intrínsecos	44
3.4	Algumas ferramentas	44
3.4.1	A escala de necessidades de Maslow	45
3.4.2	Os níveis lógicos de pensamento	46
3.4.3	Os 10 desejos intrínsecos	48
3.4.4	Mapas de empatia	49
3.5	Como usar tudo isso: seu círculo de influência	51
4	Psicologia cognitiva explicando Ágil	55
4.1	Não somos racionais Ilusões cognitivas e um cérebro que pensa de forma relativa	56
4.2	Ilusões de óptica e ilusões cognitivas	59
4.3	O que as ilusões cognitivas têm a ver com Desenvolvimento Ágil?	61
4.4	Síndrome do Estudante explicando iterações	62
4.5	Cegueira não intencional	64
4.6	Geladeira ou radiador?	65
4.7	Um desmaio na rua, a difusão de responsabilidade e os itens da retrospectiva	67
4.8	Valorizo, pois também sou dono	68
4.9	Princípio da prioridade relativa	69
4.10	Um taco, uma bola, um teste	72
4.11	Resumo das correlações	73
4.12	A importância das correlações	74
4.13	Psicólogo por paixão	74

5	Queremos inovar!	75
5.1	Uma história da (falta de) inovação	76
5.2	Mudar é necessário	78
5.3	Conheça a Lean Startup	79
5.4	Entendendo as dificuldades	81
5.4.1	Inovação não é só entregar software	81
5.4.2	Sucesso não é apenas fazer o que você já faz bem	82
5.4.3	Uma pessoa com uma ideia, a unidade de inovação	82
5.4.4	A regra é fracassar	83
5.5	Quebrando a uniformidade	83
5.6	As fases da inovação	84
5.6.1	Geração de insights: como fazer as ideias fluírem?	85
5.6.2	Modelando ideias: transformando pensamentos ale- atórios em modelos de negócios	86
5.6.3	Executando modelos de negócios: transformando ideias em atividades lucrativas	88
5.7	Reflexões	90
6	Mirebalais Melhorando a sociedade com tecnologia	91
6.1	Contexto	92
6.2	O foco diferenciado	94
6.2.1	Parte técnica	95
6.2.2	Produto Mínimo Viável	97
6.2.3	Experiência de usuário	98
6.3	Inovação	103
6.4	Conclusão	105
7	Formando novos profissionais: um relato de parceria entre em- presa e universidade	107
7.1	Motivação	108
7.2	Idealizando o programa	109
7.3	O framework de capacitação	111
7.3.1	Captação de participantes: divulgação e recrutamento	112

7.3.2	Estrutura: papéis e práticas	116
7.4	Práticas ágeis adotadas	117
7.5	Conclusão	124
7.6	Agradecimentos	125
8	Programação em par	127
8.1	Definição	128
8.2	Quando pairar	129
8.3	Ilhas de conhecimento	130
8.4	Ferramental e pareamento remoto	132
8.5	Técnicas	133
8.6	Conclusão	139
9	Vários projetos, objetivos diferentes, mas o código em comum	141
9.1	Problemas com <i>branching</i>	142
9.1.1	Merge hell	143
9.1.2	Merge monkey	144
9.1.3	Integração promíscua	146
9.2	<i>Trunk Based Development</i>	147
9.2.1	Tipos de trunk	148
9.2.2	Feature toggles	153
9.3	<i>Branches</i> são do mal?	154
9.4	Conclusão	155
10	Dissecando <i>feature toggles</i>	157
10.1	Feature toggles na prática	158
10.2	Precauções	159
10.3	Tipos de <i>feature toggles</i>	160
10.3.1	Toggle de desenvolvimento	161
10.3.2	Toggle de operações	161
10.3.3	Toggle de negócio	162
10.4	Arquitetura da aplicação	163
10.4.1	Conteúdo estático	164
10.5	Quando remover feature toggles	165
10.6	Conclusão	168

11	Estudo de caso: automação como primeiro passo para entrega contínua	169
11.1	Em busca da entrega contínua	170
11.2	Contexto	172
11.3	A iniciativa de melhoria de implantações	173
11.3.1	Fase 1 – Automação	174
11.3.2	Fase 2 Melhorando a confiança nos scripts	177
11.4	Avaliando o resultado	179
11.5	Conclusão	180
12	Explorando padrões de implementação em Ruby	181
12.1	Padrão <i>State</i>	182
12.2	Padrão <i>Active Record</i>	188
12.3	Padrões construtores	192
12.4	Padrão Map/Reduce	195
12.5	Conclusão	198
13	Testando JavaScript com o Protractor: um exemplo de solução integrada	201
13.1	Press start	202
13.2	Entendendo o Protractor mais a fundo	203
13.3	Por que usar Protractor?	211
14	Melhore seus testes	213
14.1	Trate código de teste como código de produção	213
14.1.1	Repetir ou não se repetir?	214
14.1.2	Legibilidade vs. repetição	215
14.1.3	Qualidade do código de testes	216
14.2	Utilize padrões de testes	217
14.2.1	Padrões de organização de código	217
14.2.2	Padrões de banco de dados	219
14.2.3	Padrões de verificação de resultado	220
14.2.4	Vários outros tipos de padrões	221

14.3	Evite testes instáveis	221
14.3.1	Testes não determinísticos	222
14.3.2	Possíveis causas de instabilidade	222
14.3.3	Melhorando a instabilidade	224
14.4	Teste no nível apropriado	225
14.4.1	Custos e níveis de testes	225
14.4.2	Pirâmide de testes	226
14.4.3	Antipadrões de balanceamento	228
14.4.4	Como diminuir o problema	228
14.5	Conclusão	229
15	Entendendo e utilizando dublês de teste	231
15.1	Stub	232
15.2	Fake	234
15.3	Dummy	236
15.4	Mocks	238
15.5	Spy	239
15.6	Dublês de teste e TDD	240
15.7	Utilizar dublês, sim	241
16	Desmistificando o BDD	243
16.1	BDD é geralmente mal compreendido	244
16.2	Os fundamentos de BDD	244
16.3	O BDD aprimorando o TDD	245
16.4	Os três princípios do BDD	248
16.5	Principais reclamações sobre BDD	248
16.5.1	O cliente não se importa com testes, logo, não lhe interessa se o time aplica BDD ou não	248
16.5.2	O cliente não quer escrever as especificações na ferramenta destinada a isso	249
16.6	Comunicação e o fluxo de trabalho	252
16.7	Sessão de 3 amigos	254
16.8	Um caso de sucesso usando BDD	257
16.9	Considerações finais	258

Bibliografia

269

CAPÍTULO 1

Inceptions de uma semana

por Maurício Silveira Sanches, Paulo Caroli e Tainã Caetano Coimbra

A execução de um projeto ágil envolve duas etapas principais: *discovery* e *delivery*. É natural que aconteçam descobertas durante a execução de um projeto dessa natureza. Entretanto, no início do planejamento, é muito importante que haja consenso entre os membros da equipe. Alcançar esse consenso é um dos principais objetivos de uma inception. Neste capítulo, compartilhamos uma estrutura para a execução de inceptions de uma semana.

1.1 INCEPTION

Uma *inception* marca o início de um projeto e se caracteriza pela reunião das pessoas envolvidas em um mesmo local. Ela é uma ferramenta que tem como

objetivo fazer com que a equipe descubra e entenda coletivamente o escopo do que será desenvolvido. Ao seu final, o time deve estar mais entrosado e com uma visão mais clara do caminho a seguir.

1.2 INCEPTION DE UMA SEMANA

O processo de descoberta pode durar de alguns dias até várias semanas. Geralmente, quanto maior o projeto, mais tempo é necessário para compreender suas nuances. Entretanto, nem sempre se consegue todo o tempo necessário para a realização de uma inception completa e detalhada.

Após enfrentar repetidas restrições de tempo, começamos a enxugar a inception, reduzindo sua duração. O mais enxuto que conseguimos foram inceptions de uma semana, que deram resultados excelentes. Tentativas com menor tempo de duração não foram efetivas, e necessitaram dias extras para alcançar o conjunto mínimo de artefatos indispensáveis para que o projeto fosse executado.

1.3 ESTRUTURA DE UMA INCEPTION DE UMA SEMANA

Por dispor de um tempo tão reduzido, a inception deve ser bem focada. Por essa razão, criamos uma estrutura específica e visual para garantir que a equipe obtenha tudo o que necessita para iniciar o projeto.

A inception de uma semana segue uma cadência bem definida, que descobre e detalha o escopo ao mesmo tempo em que refina o plano. É importante que haja foco o suficiente para que os artefatos essenciais sejam gerados. A seguir, apresentamos a representação visual da estrutura que utilizamos:

A hand-drawn agenda burn-up chart on a grid. The vertical axis (Y-axis) lists tasks in Portuguese: 'USER STORY MAPPING', 'DETALHAR ESTÓRIAS', 'DESCOBRIR ESTÓRIAS', 'FEATURES', 'OBJETIVOS', and 'VISÃO'. The horizontal axis (X-axis) lists days: 'DIA 1', 'DIA 2', 'DIA 3', 'DIA 4', and 'DIA 5'. Red 'P' marks indicate planned tasks: 'OBJETIVOS' on DIA 1, 'VISÃO' on DIA 1, 'DESCOBRIR ESTÓRIAS' on DIA 2 and DIA 3, 'FEATURES' on DIA 2, 'DETALHAR ESTÓRIAS' on DIA 3 and DIA 4, and 'USER STORY MAPPING' on DIA 5.

	DIA 1	DIA 2	DIA 3	DIA 4	DIA 5
USER STORY MAPPING					P
DETALHAR ESTÓRIAS			P	P	
DESCOBRIR ESTÓRIAS		P	P		
FEATURES		P			
OBJETIVOS	P				
VISÃO	P				

Fig. 1.1: A agenda burn-up

AGENDA BURN-UP

Esta estrutura é a *agenda burn-up*, onde estão representados os tópicos no eixo Y e o tempo no eixo X. Manter a agenda burn-up visível para todos ajuda a gerenciar o tempo e a focar no resultado. Uma estrutura simples mas extremamente eficaz para o sucesso da inception.

No início da inception, deve-se descobrir qual é a necessidade de negócio que motiva o projeto e qual é o objetivo da equipe. Essa tarefa, idealmente, é realizada por quem mais compreende as expectativas com relação ao time em Scrum seria o *product owner*.

No decorrer da inception, o objetivo deve ser refinado, de modo que possa

ser alcançado pouco a pouco. Para que isso aconteça, devem-se identificar as diferentes *features* necessárias para que a meta seja atingida.

Uma vez que se entendam as features, é preciso identificar quais pedaços de trabalho são necessários para alcançá-las. Em projetos ágeis, essas partes menores são chamadas de *user stories*, que são descrições em alto nível de um requisito, normalmente do ponto de vista do usuário daquela funcionalidade; estas são breves e claras o suficiente para que o time possa implementá-las.

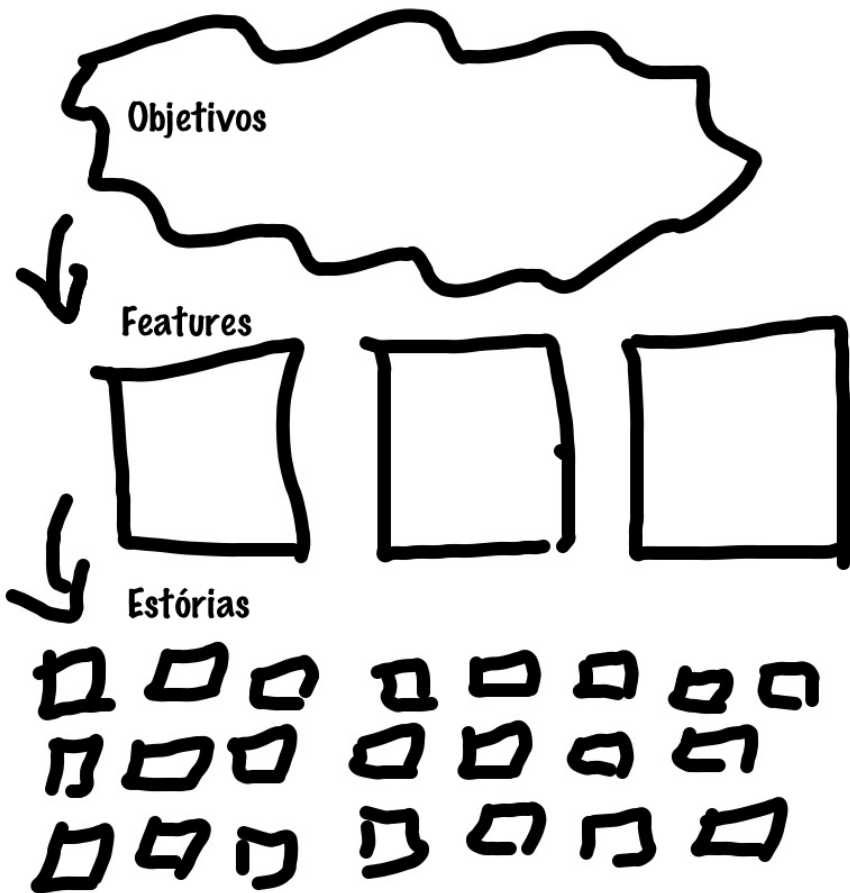


Fig. 1.2: Detalhando os objetivos

Enquanto as ideias são refinadas e o trabalho é dividido em *stories*, dependências e prioridades são identificadas. Essa é uma informação essencial

para a construção do plano de entrega do projeto.

Ao final da inception, a equipe deve ter um conjunto de artefatos com informação suficiente para que o plano possa ser traçado e acompanhado durante a execução do projeto. A seguir, detalhamos o processo que seguimos para construir esses artefatos.

1.4 QUEBRANDO O GELO

Para que a inception seja bem sucedida, o time deve estar entrosado. Assim, para o primeiro encontro da equipe, sugerimos realizar uma atividade para quebrar o gelo. Segue um exemplo de atividade:

Atividade: Zip Zap Zoom



Fig. 1.3: Zip, Zap, Zoom

Carlos: *Zip* Mauricio Mauricio: *Zip* Thiago Thiago: *Zoom* Rodrigo

- 1) Solicite à equipe que forme um círculo e que cada pessoa feche as mãos com os dedos indicadores apontando (foto apresentada);
- 2) O primeiro participante deve executar um comando verbal, apontando para um receptor. O receptor deve repetir o processo e, assim, sucessivamente.

O comando verbal é composto por *Zip*, *Zap* ou *Zoom*:

- *Zip*: Apontar para a pessoa imediatamente ao lado, mantendo a direção do movimento anterior;
- *Zap*: Apontar para a pessoa imediatamente ao lado, alterando a direção do movimento anterior;
- *Zoom*: Apontar para qualquer um do círculo, mencionando seu nome. O receptor vai decidir o próximo comando e sua direção.

Caso um participante execute um comando incorretamente (por exemplo, um *zip* para a direção errada), ele é retirado do círculo. Esta atividade não só energiza o grupo, como também exige foco e atenção dos participantes. O comando *zoom* ajuda os membros da equipe a lembrarem dos nomes uns dos outros.

1.5 VISÃO DO PRODUTO

Para o primeiro dia da inception, é interessante reunir não só a equipe, mas também outros colaboradores interessados na execução do projeto. Nesse encontro inicial, devem ser apresentadas as ideias que motivam o projeto e o impacto que o time pode causar. Normalmente, é feito um discurso que tem como objetivo instigar um entendimento inicial do projeto em cada membro da equipe, assim como motivar o time.

O entendimento da necessidade do projeto guiará as atividades da inception. Os conceitos apresentados inicialmente são apenas uma visão, visto que

a ideia será refinada durante o processo. Sugerimos a seguinte atividade para que a equipe atinja uma visão compartilhada:

Atividade: Visão do produto

- 1) Escreva o seguinte *template* em um quadro branco ou *flipchart*, de forma que seja visível para toda a equipe:

Para [cliente final] Cujo [problema que precisa ser resolvido], O [nome do produto] É um [categoria do produto] Que [benefício chave, razão para adquiri-lo]. Diferentemente da [alternativa da concorrência], O nosso produto [diferença chave].

- 2) Divida a equipe em grupos menores e solicite que cada um deles preencha uma lacuna separadamente (ou mais, dependendo do tamanho da equipe);
- 3) Reúna o resultado de cada grupo, formando uma única frase.

Nessa atividade, é muito comum que o resultado seja uma frase desconexa. Logo, após a execução do terceiro passo, é importante que a equipe trabalhe em conjunto para formar uma frase homogênea, utilizando e alterando os resultados anteriores conforme necessário. Veja mais no livro *Crossing the Chasm* de Geoffrey A. Moore.

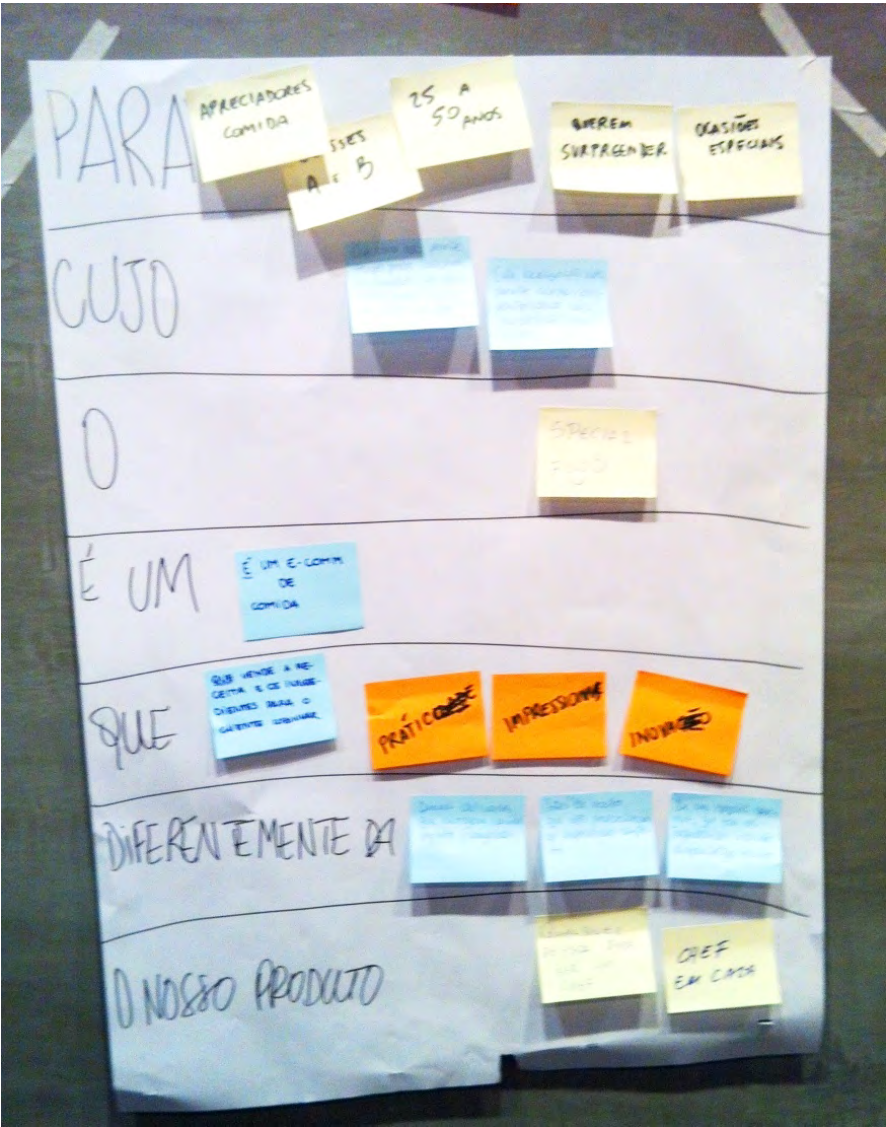


Fig. 1.4: exemplo de resultado; a visão do produto

1.6 OBJETIVOS

Cada membro da equipe deve compartilhar o que entende sobre os objetivos do projeto, e os vários pontos de vista devem ser discutidos para que o time alcance um consenso sobre o que é realmente importante.

Para evitar uma falsa sensação de consenso, sugerimos a seguinte atividade:

Atividade: Esclarecendo objetivos

- 1) Solicite a cada membro da equipe que escreva, individualmente, três respostas para a seguinte pergunta: “Se você tiver que resumir este projeto em três objetivos, quais são eles?”
- 2) Solicite aos participantes que compartilhem os objetivos que escreveram em um canvas comum, agrupando-os por similaridade;
- 3) Solicite à equipe que reescreva os objetivos, agora coletivamente. Nesse momento, ficará claro que alguns dos elementos listados não são realmente objetivos do projeto devendo, portanto, ser descartados. Com isso, ficará nítido para a equipe qual o foco do projeto.

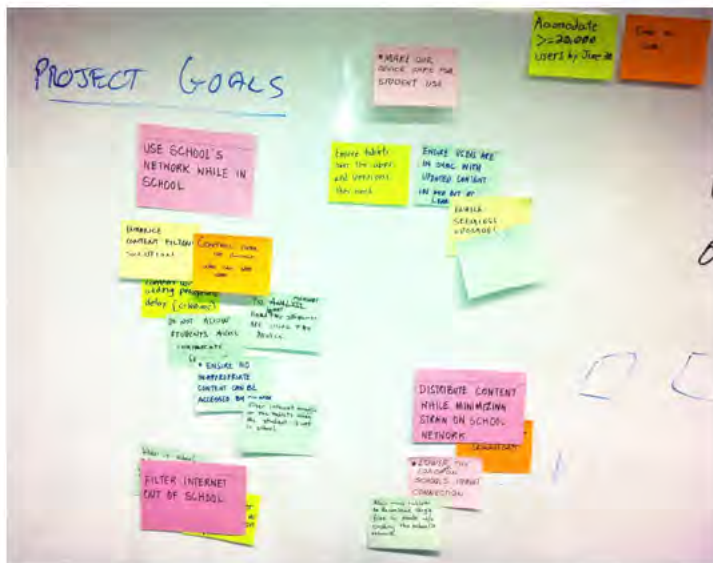


Fig. 1.5: exemplo de objetivos agrupados com post-it

Nota-se, nessa foto, o resultado final da atividade **Esclarecendo objetivos**. Nessa imagem, pode-se perceber o agrupamento de objetivos (escritos individualmente) em três objetivos (escritos pelo grupo, em cartões rosa). Observa-se também que alguns dos objetivos escritos individualmente não serão levados adiante (canto superior direito da foto).

Na execução da atividade em questão, deve-se evitar o uso de materiais previamente elaborados sobre o projeto, como diagramas e tabelas. A equipe deve ter à disposição apenas caneta e papel (preferencialmente *post-its*).

1.7 PERSONAS

Para identificar efetivamente as funcionalidades de um sistema, consideramos importante ter em mente os usuários e seus objetivos. Os usuários normalmente são representados por *personas*.

Uma *persona*, ao representar um usuário do sistema, descreve não só o seu papel, mas também suas necessidades específicas. Tal fato cria uma representação realística de usuários, auxiliando o time a descrever funcionalidades

do ponto de vista de quem interagirá com o produto final.

Enquanto cada persona é criada, duas perguntas devem ser respondidas: quem ela é e do que precisa. Dessa forma, a equipe cria empatia pelo seu usuário e ao mesmo tempo mantém o foco no que é mais prioritário para cada persona.

Atividade: Identificando personas

- 1) Solicite ao time que se divida em pares ou trios e entregue o seguinte template de personas para cada grupo:

<p>[Persona's Name]</p> <p>[A tag line for the persona]</p> <p>About [Name]:</p> <ul style="list-style-type: none">• Who are they?• What is their background?• What is their context?• What's important to them?• What are their pain points and frustrations?	 <p><i>"A quote the persona might say"</i></p> <p>Key Goals & Needs:</p> <ul style="list-style-type: none">• Goals• Motivations• Drivers• Needs
--	---

Fig. 1.6: template para persona

- 2) Solicite a cada grupo que crie uma persona, utilizando as perguntas do template como referência;
- 3) Solicite aos participantes que apresentem suas personas a todo o time;
- 4) Solicite ao time que mude os grupos e repita os passos de 1 a 3.

Ao final da atividade, um conjunto de personas deve ter sido criado e os diferentes tipos de usuários do sistema devem estar descritos. Os *stakeholders* que conhecem os objetivos do projeto devem participar ativamente da atividade, auxiliando a equipe na criação das personas e sugerindo alterações em suas descrições conforme necessário.

Nas inceptions que tivemos, algumas dessas personas acabaram sendo menos usadas do que outras, entretanto todas foram úteis no processo de criação e descoberta de funcionalidades. Nossa sugestão é que o grupo não se prenda a nenhuma persona específica, já que algumas vão naturalmente desaparecer.

1.8 FEATURES

Feature é um agrupamento de funcionalidades afins. Tal agrupamento ajuda a compreender o requisito como um todo, bem como as suas partes menores e complementares. O entendimento de feature varia de time para time, o que é importante é que ele faça sentido para aquela equipe.

Sugerimos a seguinte atividade para descoberta de features, utilizando o conhecimento e os artefatos adquiridos nas atividades anteriores:

Atividade: Descobrindo as features

- 1) Solicite que a equipe coloque os objetivos em um canvas comum, em ordem de prioridade, da esquerda para direita, como títulos de colunas;
- 2) Solicite que a equipe coloque as personas no mesmo canvas, em ordem de prioridade, de cima para baixo, como títulos de linhas;
- 3) Promova um *brainstorm* de features. A discussão deve ser guiada para que se descubram quais features são necessárias para atender objetivos e personas. Uma pergunta para ajudar nesse processo é: “O que precisa ter no sistema para que tal persona alcance tal objetivo?”

Sugerimos que o time se guie na direção do canvas do topo esquerdo para o canto inferior direito (conforme figura a seguir). Dessa forma, as features de maior prioridade surgirão primeiro.

Aconselhamos que as demais features sejam esclarecidas (e documentadas), mas que a equipe não entre em detalhes, mantendo o foco nos itens prioritários.



Fig. 1.7: descobrindo as features

Embora o canvas seja semelhante a uma matriz, não necessariamente haverá uma feature para cada interseção. Pode haver múltiplas features para uma persona alcançar um objetivo específico, assim como é possível haver personas que não necessitam de uma feature para determinado objetivo.

Caso sejam identificados objetivos e features que não atendem as necessidades de nenhuma pessoa, estes devem ser descartados ou repensados, pois o seu valor não está claramente associado a um usuário.

1.9 HISTÓRIAS

O desenvolvimento ágil promove uma abordagem incremental, cujo foco é uma pequena parte dos requisitos a cada momento. Histórias são esses pequenos pedaços, uma definição informal de requisito que pode ser escrita em

uma ou duas sentenças. Incrementalmente, elas devem ser discutidas e refinadas.

Durante a sua descoberta, as histórias devem ser identificadas e capturadas. Em uma inception, interessa o que deve ser feito. É preciso detalhar apenas o mínimo necessário para que o time se sinta confortável, tanto no aspecto técnico quanto no de negócio. A seguir, propõe-se uma atividade que visa descobrir histórias:

Atividade: Descoberta de histórias

- 1) Reúna a equipe ao redor de uma mesa limpa. Cada membro deve possuir *index cards*, *post-its* de cores diferentes e canetas;
- 2) Solicite a um membro da equipe que leia uma feature em voz alta;
- 3) Discuta quais são os pedaços de trabalho necessários para implementar a feature;
- 4) Solicite à equipe que anote qualquer informação relevante em *post-its*. Estes devem ser agrupados e anexados a um *index card*, que serve como um canvas comum para os detalhes da história.



Fig. 1.8: exemplo de features e histórias

Ao final da atividade, a equipe terá alcançado um conjunto inicial de histórias.

1.10 REFINAMENTO DAS HISTÓRIAS

As atividades seguintes auxiliarão a equipe a adicionar mais detalhes ao esboço inicial, tornando as histórias mais completas.



Fig. 1.9: exemplo de features e histórias

Uma delas tem o objetivo de discutir como está o entendimento da equipe sobre os requisitos da história. Através da discussão, novas notas são capturadas e a história se torna mais detalhada, além de discordâncias e dúvidas ficarem mais aparentes.

Atividade: Entendimento técnico X entendimento de negócio

- 1) Crie, em um canvas comum, um gráfico, cujo eixo X representa entendimento técnico (como fazer) e o eixo Y representa entendimento sobre o requisito de negócio (o que fazer). No eixo X, o objetivo é verificar o entendimento da equipe com relação aos desafios técnicos, às dependências e aos requisitos de infraestrutura. No eixo Y, a proposta é verificar a clareza sobre o objetivo da história, o benefício para o negócio e o que deve ser feito.

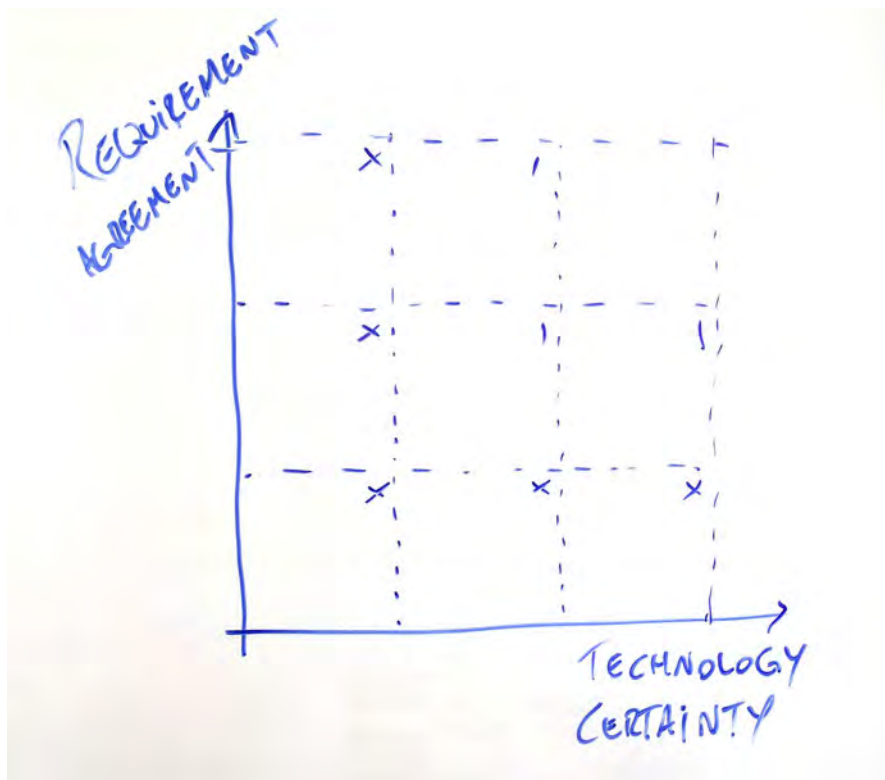


Fig. 1.10: o gráfico: Entendimento técnico X entendimento de negócio

- 2) Solicite a um membro da equipe que leia a história em voz alta e a posicione no gráfico de acordo com o seu entendimento sobre ela (entendimento técnico e de negócio);
- 3) Questione a equipe se todos compartilham aquela opinião. Se alguém não concordar, a equipe deve discutir os requisitos e a tecnologia envolvida de forma que haja um consenso sobre a história. Tudo o que for mencionado e que ajude a alcançar uma melhor compreensão deve ser anotado e anexado à história;
- 4) Anote, na história, o nível de entendimento. Por exemplo, um X no canto superior direito do index card representa baixo entendimento técnico e de

negócio.

Para cada história capturada anteriormente, repita os passos de 2 a 4.

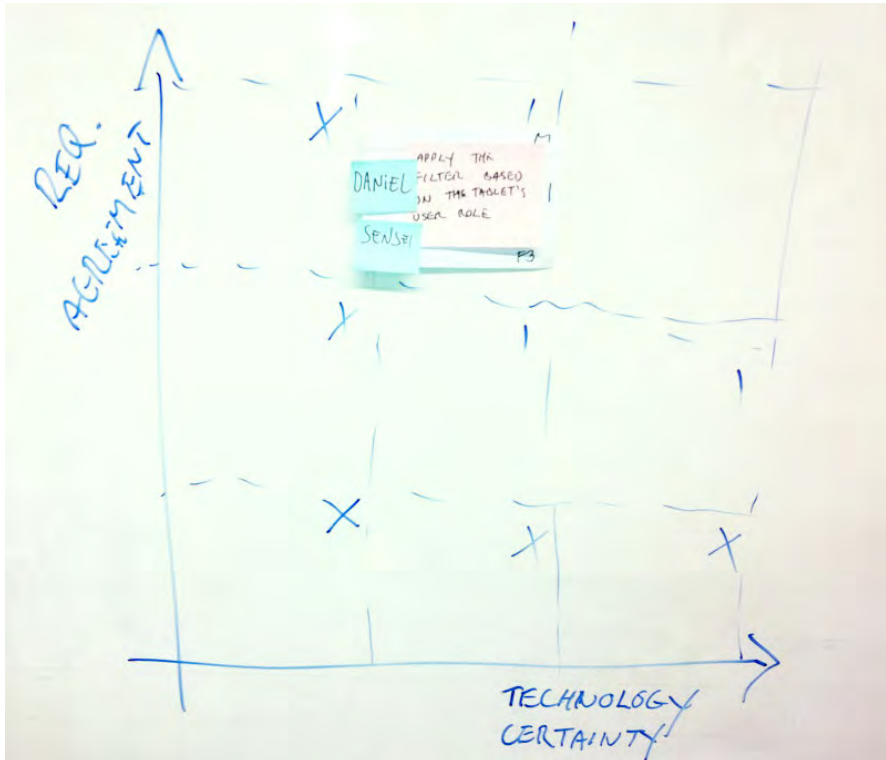


Fig. 1.11: uma história no gráfico

Ao final da atividade, as histórias marcadas com um X representam riscos para o projeto. Normalmente, a equipe decide dividi-las em pedaços menores de trabalho. Caso isso não aconteça, a marcação deixa visível o risco da história para o plano.

1.11 O MAPEAMENTO DAS HISTÓRIAS

A equipe criou coletivamente um conjunto de histórias e, a partir desses elementos, é preciso elaborar um plano de execução. Para isso, adicionamos

dados de capacidade da equipe, sequenciamento lógico, prioridade e tempo.

O mapeamento das histórias é uma representação visual deste plano. A seguir, descrevemos uma atividade que motiva a sua criação.

Atividade: Criação do mapeamento das histórias

- 1) Solicite à equipe que organize as features em ordem de prioridade, da esquerda para a direita. Elas devem ser colocadas no topo de um canvas comum;
- 2) Peça ao time que agrupe histórias relacionadas a uma feature abaixo de cada uma delas;

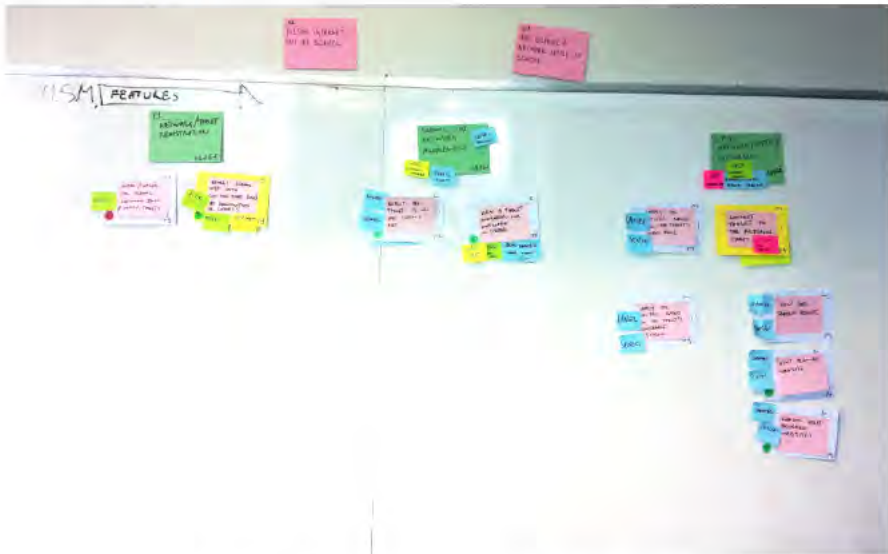


Fig. 1.12: Preparando o user Story Mapping

- 3) Crie uma linha do tempo vertical e múltiplas faixas horizontais;
- 4) Solicite à equipe que identifique a quantidade de histórias que podem ser jogadas simultaneamente. Por exemplo, um time de seis pessoas que sempre parecia será capaz de jogar três histórias ao mesmo tempo;

- 5) Peça para a equipe distribuir as histórias na linha do tempo, respeitando o limite de cada faixa e dependências entre histórias.



Fig. 1.13: User Story Mapping; exemplo de resultado

Ao final da atividade, a equipe possuirá um entendimento coletivo sobre o plano de execução do projeto. Independentemente do *framework* utilizado (seja Scrum, Kanban ou qualquer outro), a ordem de execução das histórias, baseada em dependências e prioridade, estará definida no *user story mapping*.

1.12 CONCLUSÃO

Ao final da inception, a equipe está alinhada quanto ao trabalho que deve ser realizado durante o projeto, no que diz respeito à sua prioridade, ao sequenciamento e ao esforço. Além da obtenção de artefatos bem definidos (as

histórias e o plano), outros benefícios não tangíveis são alcançados (como o aumento da confiança entre os membros do time, por exemplo).

Embora esse modelo de inceptions de uma semana apresentado resulte frequentemente em restrições de tempo e/ou budget, ainda assim ele tem se mostrado efetivo. Em um período de dois anos, mais de 20 inceptions foram realizadas seguindo tal modelo, o qual foi refinado através de tentativas, mostrando resultados excelentes. Uma *inception* bem sucedida não garante o sucesso do projeto, porém, quando mal realizada, geralmente resulta em falhas de execução.

CAPÍTULO 2

Produtividade e adaptabilidade de times ágeis: conceitos e paradoxos

por Claudia Melo

Produtividade, um atributo desejado por organizações e conceito ainda pouco compreendido. Este capítulo discutirá conceitos fundamentais de produtividade em geral e produtividade de times ágeis de software em particular. Vamos mostrar a relação entre métodos de desenvolvimento tradicionais e produtividade, fazendo um contraste com o movimento de métodos ágeis. Abordaremos também o paradoxo entre eficiência e flexibilidade e como esse paradoxo pode ser uma vantagem útil rumo à inovação em software.

2.1 ÁGIL É PRODUTIVO?

A produtividade de software tem sido estudada intensivamente ao longo das últimas décadas, geralmente com o intuito de reduzir custos de desenvolvimento e melhorar o tempo gasto até o lançamento de produtos e serviços (*time-to-market*). Diversos métodos, ferramentas e técnicas foram desenvolvidos para melhorar a produtividade de software, desde a produtividade individual até a de times e organizações.

Métodos ágeis de desenvolvimento de software vêm ganhando crescente popularidade desde o início da década de 2000 e podem oferecer melhores resultados para os projetos de software quando comparados às abordagens mais tradicionais em algumas circunstâncias. Eles são regidos pelo Manifesto Ágil, conjunto de valores e princípios criados por 17 desenvolvedores experientes, consultores e líderes da comunidade de desenvolvimento de software. Exemplos de métodos ágeis são a Programação Extrema, Scrum, Crystal e Lean Software Development.

Métodos ágeis prometem produzir software de alta qualidade com alta produtividade, o que atrai a atenção das empresas, que demandam cada vez mais velocidade e qualidade em seus produtos. Resultados recentes de pesquisas em nível mundial (VersionOne, 2013) e nacional mostram que um dos maiores motivadores para a adoção de métodos ágeis por organizações é a expectativa de aumento da produtividade.

A Figura 2.1 ilustra os resultados da pesquisa feita no Brasil a respeito de adoção de métodos ágeis e a busca do aumento de produtividade. Em 2011, 91% dos 471 respondentes afirmaram que produtividade foi a razão mais importante para adoção de métodos ágeis no seu time ou organização.

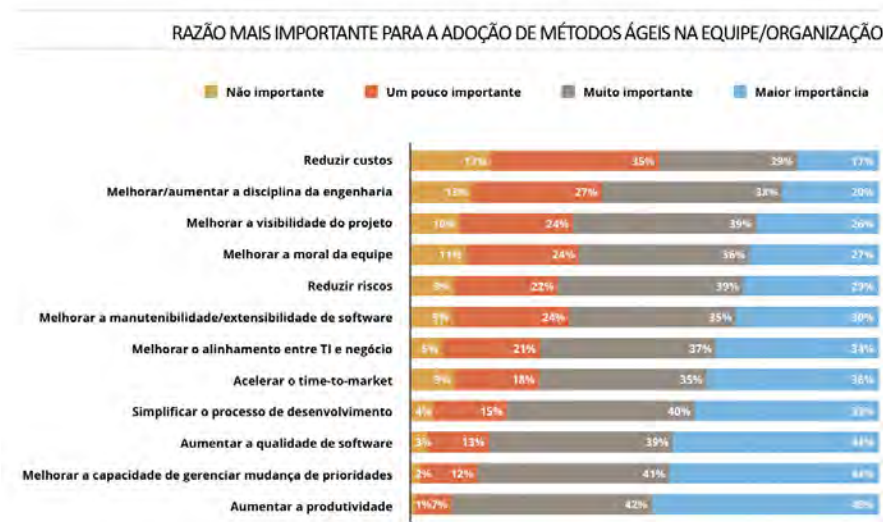


Fig. 2.1: Produtividade como principal motivo para adoção de Métodos Ágeis

No entanto, produtividade é um conceito vago, o que gera confusão e má definição do termo por times e organizações. Diante de tantas definições, como avaliar a produtividade de times ágeis? Como identificar se os times têm apresentado maior produtividade, com a finalidade de mantê-la, ou mesmo identificar melhorias? Para respondermos essas perguntas, devemos partir do ponto básico: definir produtividade.

2.2 CONCEITOS DE PRODUTIVIDADE

Apesar de produtividade ter sido um tema intensamente estudado ao longos dos anos, ainda é uma questão controversa. Em primeiro lugar, existem diversos conceitos relacionados à produtividade, como efetividade, eficiência e desempenho, o que gera mal entendidos e sobrecarrega o termo *produtividade*.

PRODUTIVIDADE

É a capacidade de produzir saídas com qualidade com o melhor aproveitamento dos recursos de entrada (como o tempo). É uma clara relação entre efetividade e eficiência.

Peter Drucker, pai da administração moderna, cunhou a expressão trabalhador do conhecimento (*knowledge worker*) para descrever as pessoas que aplicam conhecimento teórico e analítico adquirido, por educação formal e experiência, para desenvolver novos produtos e serviços.

Eles frequentemente experimentam novas tecnologias e seu estilo de trabalho é sem rotina, lidando com complexidade, autonomia e imprevisibilidade. Drucker já previa que essa seria a maior parte do trabalho do século 21: o trabalho relacionado ao conhecimento. Essa foi uma transição importante entre abordagens de trabalho. Nos séculos 18 e 19, com a Revolução Industrial e o Taylorismo, a ênfase do trabalho estava na otimização, em processos e mecanismos, na estabilidade e previsibilidade, na especialização e trabalho individual. O foco da produtividade estava na eficiência.

No século 20, ainda mantivemos, em nossas estruturas de trabalho, o paradigma centrado em otimização e eficiência. Com a chegada da globalização e o aumento significativo da concorrência e da necessidade de personalização, o resultado esperado do nosso trabalho mudou. Espera-se agora um trabalho criativo, para resolver problemas cada vez mais complexos em um cenário mais turbulento e competitivo. A Figura 2.2 ilustra a passagem para o paradigma de adaptação, em que os times e seu conhecimento são necessários para gerar resultados diferenciados.



Fig. 2.2: Focos da produtividade, da otimização para a adaptação.

Times de desenvolvimento de software são um ótimo exemplo para as ideias de Drucker. Eles encapsulam conhecimento sobre o negócio, criatividade, inovações e os transformam em algo tangível: software funcionando. O sucesso de uma organização no século 21, de acordo com Drucker, depende da sua vantagem relativa sobre outras organizações em ter a maior produtividade dos trabalhadores do conhecimento. Se no século 21 a natureza do trabalho mudou, certamente a noção de produtividade também. Descrevo algumas das principais dimensões relacionadas à produtividade de trabalhadores de conhecimento na tabela da Figura 2.3.

Dimensões de produtividade	Descrição
Quantidade	Considera quantidade de saídas produzidas
Lucratividade	Considera lucratividade e custos provenientes do produto sendo criado
Em tempo oportuno	Grau em que a solução foi entregue em tempo hábil. Considera o cumprimento dos prazos estabelecidos e outras questões temporais
Autonomia	Grau de independência e quantidade de coisas que o trabalhador faz por vez
Eficiência	Conceito transversal, que refere-se a fazer as coisas de maneira correta, eficiente, atendendo às especificações, mesmo as não importantes para o trabalho
Qualidade	Grau do quão bom o trabalho é
Efetividade	Conceito transversal, que refere-se a fazer as coisas certas, ou seja, apenas as tarefas que são importantes para o trabalho, mesmo que sejam concluídas sem cumprimento das normas de qualidade, tempo etc.
Satisfação do cliente	Explica o fato de que o produto precisa adicionar valor ao cliente
Sucesso do projeto	Considera o resultado global do trabalho, considerando-se a tomada de decisões, interação da equipe, comunicação, previsibilidade, gestão de crises, documentação, transferência de trabalho etc.
Grau de inovação / Criatividade	Representa a capacidade de criação de soluções novas, além de ideias para melhorar a produtividade.
Responsabilidade	Grau em que há bom desempenho em momentos críticos
Aprendizado para uma vida	Trabalho do conhecimento requer aprendizagem e ensino contínuos

Fig. 2.3: Dimensões da produtividade do trabalhador do conhecimento

O que o conceito de produtividade de trabalhadores de conhecimento traz de novo em termos de compreensão do que é produtividade? Ela adiciona novos aspectos importantes a serem considerados, como grau de autonomia, responsabilidade, aprendizado e criatividade. Trabalhadores do conhecimento também são mais produtivos quando desenvolvem trabalhos com maior autonomia (menor dependência), quando são capazes de assumir responsabilidades, quando aprendem e quando geram soluções criativas. Todas elas são sinais de produtividade de times de desenvolvimento de software.

2.3 AGILIDADE É SOBRE MUDANÇA DE PARADIGMA

Depois de 10 anos da publicação do Manifesto Ágil, ainda existem questionamentos sobre quais as reais diferenças entre a abordagem ágil e a abordagem tradicional de desenvolvimento de software. Agilidade é um paradigma, não apenas um conjunto de processos a serem inseridos em times. Ao tentar tratar métodos ágeis como uma sequência de passos e técnicas, perde-se o valor real que a abordagem propõe.

Para entender produtividade no contexto ágil, é importante ter a proposta de métodos ágeis clara e, com isso, estabelecer as dimensões de produtividade que de fato importam. A tabela da Figura 2.4, adaptada do artigo de Nerur e Baliyepally, apresenta os principais princípios de métodos ágeis, em contraste aos métodos tradicionais.

	Métodos tradicionais	Métodos ágeis
Suposições fundamentais	Sistemas são totalmente especificáveis, previsíveis e podem ser construídos através de meticuloso e extenso planejamento	Software flexível e de alta qualidade pode ser desenvolvido por pequenas equipes usando os princípios de melhoria contínua e teste do design, com base no feedback rápido e mudanças.
Controle	Centrado em processo	Centrado em pessoas
Objetivo	Otimização	Adaptação, flexibilidade e capacidade de resposta
Características chave	Controle e direção; evita o conflito; formaliza a inovação; o gerente é controlador e o maior responsável; projeto precede a implementação	Colaboração e comunicação; abraça conflito e dialética; encoraja a exploração e a criatividade é oportunista; gerente é facilitador e time é responsável; projeto e implementação são inseparáveis e evoluem de forma iterativa
Abordagem de resolução de problemas	Seleção dos melhores meios para chegar em um determinado fim através de atividades bem planejadas e formalizadas	Aprendizagem por meio de experimentação e introspecção, constantemente reformulando o problema e a solução

Fig. 2.4: Foco dos métodos tradicionais versus Métodos Ágeis

A agilidade de software tem claros atributos de flexibilidade, velocidade, leveza (*leanness*), responsividade e aprendizado.

Flexibilidade é a habilidade ou o comportamento que habilita uma pessoa/time/organização a se adaptar a mudanças quando necessário. Em desenvolvimento de software, é a habilidade de criar ou abraçar mudanças proativamente ou reativamente em tempo hábil.

Velocidade é caracterizada por um comportamento rápido para chegar ao destino desejado ou simplesmente alcançar objetivos. Um método pode ajudar a mostrar resultados mais rapidamente usando alguma técnica específica.

Leveza (*leanness*) refere-se ao grau de compactação e organização. Um método leve gera uma saída com a qualidade desejada, economicamente, no menor tempo possível, aplicando meios simples de produção. É a maximização do valor por meio de economia, qualidade e simplicidade.

Aprendizado refere-se ao conhecimento e melhoria, sendo indispensável para qualquer indivíduo, time ou organização. Ele é obtido primariamente no uso de conhecimento atualizado e experiência ganha ao longo do tempo. Um método que prima pelo aprendizado e mostra melhoria contínua ao longo do tempo.

Responsividade refere-se à sensibilidade e à reatividade. Um método responsivo não fica silencioso quando uma resposta é requerida em diferentes situações. A mudança não é apenas aceitável e fácil de realizar, mas também visível.

2.4 O QUE SIGNIFICA PRODUTIVIDADE DE TIMES ÁGEIS?

Quando as mudanças cruciais que métodos ágeis propõem são compreendidas, fica mais fácil entender como a produtividade se dá em um contexto ágil. A essa altura, o leitor já deve ter percebido as similaridades entre métodos ágeis e as ideias de trabalho do conhecimento de Drucker.

Times ágeis são formados por trabalhadores do conhecimento trabalhando em conjunto para alcançarem um objetivo em comum. Nesses grupos, a produtividade individual não é tão importante quanto a produtividade de time. Sendo assim, o foco de produtividade em ambientes ágeis deve ser muito mais nos times do que nos membros em si.

Uma outra mudança de foco pode ser percebida no próprio Manifesto Ágil, que traz como primeiro princípio:

“Nossa maior prioridade é satisfazer o cliente através da entrega cedo e contínua de software com valor agregado.”

– primeiro princípio do Manifesto Ágil

Dessa forma, métodos ágeis encorajam que o esforço de desenvolvimento de software seja associado ao valor que ele propicia. Os times são responsáveis por entregar software funcionando e que gere valor constantemente. Sendo assim, a produtividade de um time ágil está muito relacionada à capacidade de entregar funcionalidades de alto valor para o cliente o mais cedo possível.

É possível compreender melhor a produtividade no contexto ágil cruzando os princípios ágeis e as definições de produtividade já apresentadas. A tabela da Figura 2.5 apresenta essa relação, na qual fica claro que diversas dimensões da produtividade do trabalhador do conhecimento são adequadas para times ágeis. Ela traz referências do Manifesto ágil, de Cooke (2010) e dos Poppendieck para ilustrar a importância das dimensões de produtividade para um time ágil.

Dimensões de produtividade	Relevância	Descrição
Quantidade	Baixa	Medir a quantidade de saídas em times ágeis não é tão prioritário, uma vez que a "arte de maximizar a quantidade de trabalho não realizado é essencial". No entanto, a métrica pode ser utilizada caso necessário no cálculo de outras métricas de time.
Lucratividade	Alta	"Nossa maior prioridade é satisfazer o cliente através de entrega contínua e adiantada de software com valor agregado." Observar que essa lucratividade é referente ao cliente, não ao time. No entanto, o time também precisa cuidar da sua própria lucratividade.
Em tempo oportuno	Alta	Times ágeis devem fazer entregas contínuas, gerando valor em tempo oportuno. Prazos não devem ser modificados (respeito ao timebox), pelo contrário, devem ser iminentes para serem levadas a sério.
Autonomia	Alta	Autonomia do time é essencial para o estabelecimento da auto organização e estabelecimento da confiança. "Dê a eles o ambiente e o suporte necessário e confie neles para fazer o trabalho."
Eficiência	Média	Apesar de eficiência ser importante, o foco dos métodos ágeis está na adaptação, flexibilidade e capacidade de resposta. Portanto, seu uso deve ser dosado ao compor um indicador.
Qualidade	Alta	Aspectos de qualidade interna e externa são muito importantes ao considerarmos a produtividade de times ágeis: "contínua atenção à excelência técnica e bom design aumenta a agilidade" e "nossa maior prioridade é satisfazer o cliente através de entrega contínua e adiantada de software com valor agregado".
Efetividade	Alta	"Nossa maior prioridade é satisfazer o cliente" e "a arte de maximizar a quantidade de trabalho não realizado é essencial".
Satisfação do cliente	Alta	"Nossa maior prioridade é satisfazer o cliente."
Sucesso do projeto	Alta	Desenvolvimento de pessoas (tomada de decisão, interação da equipe e comunicação) são de extrema importância para times ágeis.
Grau de inovação / Criatividade	Alta	Métodos ágeis encorajam a exploração e a criatividade é oportunista. "Em intervalos regulares, a equipe reflete sobre como se tornar mais eficaz e então refina e ajusta seu comportamento de acordo."
Responsabilidade	Alta	"Dê a eles o ambiente e o suporte necessário e confie neles para realizar o trabalho."
Aprendizado para uma vida	Alta	"Em intervalos regulares, a equipe reflete sobre como se tornar mais eficaz e então refina e ajusta seu comportamento de acordo."

Fig. 2.5: Dimensões de produtividade de times ágeis

Praticamente todas as dimensões de produtividade do trabalhador do conhecimento se aplicam a times ágeis. Definir quais delas serão usadas e em

que momento, assim como o peso de cada uma, cabe a cada time ou organização. A partir disso, torna-se viável avaliar a produtividade dos times visando a melhoria contínua, um objetivo intrinsecamente ágil.

2.5 PARADOXOS ENVOLVENDO EFICIÊNCIA E FLEXIBILIDADE E SEU IMPACTO NA PRODUTIVIDADE DE TIMES ÁGEIS

O paradoxo entre eficiência e flexibilidade é um dos mais conhecidos e centrais em teorias sobre organizações. Uma organização precisa ter eficiência para manter suas operações competitivas, com baixo custo. Por outro lado, também precisam ser inovadoras para serem competitivas. No final dos anos 80, diversas organizações e pesquisadores começaram a se perguntar se era possível ser eficiente e inovador ao mesmo tempo.

Por definição, os paradoxos podem conter afirmações aparentemente impossíveis, mas que no final são verdadeiras ou possíveis. Uma visão paradoxal enfatiza a coexistência entre dois temas aparentemente contraditórios e faz uso dessa tensão gerada em vez de tentar eliminá-la.

Existem diversas definições paradoxais em métodos ágeis, por exemplo nas atividades de planejamento. Jim Highsmith, um dos signatários do Manifesto Ágil, descreveu o planejamento ágil como um paradoxo, pois os times estão planejando algo em um ambiente adaptativo em que resultados são naturalmente imprevisíveis.

A visibilidade de um projeto ágil também é paradoxal, pois, de um lado, há tanta visibilidade que o cliente poderia cancelar o projeto a qualquer momento se o progresso não estiver suficiente. Por outro lado, o progresso é tão visível, a habilidade de tomar decisões sobre o que fazer em seguida é tão completa, que os projetos ágeis costumam entregar mais do que é necessário, com menos estresse e pressão.

O mesmo paradoxo entre eficiência e flexibilidade existe em times ágeis: ao mesmo tempo em que se espera um aumento de eficiência em não se produzir software inútil e em manter-se a disciplina, é esperado que eles sejam suficientemente flexíveis para abraçar constantes mudanças e adaptar-se a turbulências do mercado. Mas como isso de fato funciona?

Em processos/atividades extremamente maduros, a eficiência e a previsibilidade são altos, mas há poucas oportunidades naturais de aprendizado. Essas atividades já são consideradas automáticas pelas pessoas, com pouco espaço para interrupção. Se não há experimentação, não há erro, portanto não há aprendizado.

Um exemplo disso é o uso de arquiteturas padrão para o desenvolvimento de todos os sistemas de uma organização. A arquitetura padrão é conhecida, eficiente, mas é tão madura que há pouca chance de times encontrarem soluções disruptivas para os problemas que os cercam. A organização é eficiente no curto prazo, mas não é inovadora. Com o tempo, passará também a ser ineficiente, pois sua arquitetura será tão ultrapassada (assim como seus times), que o custo de mudança será muito maior. No longo prazo, essa organização não terá sido produtiva.

Para reativar o aprendizado, essencial para o trabalhador do conhecimento e para inovação, é preciso ter espaço para a introdução de mudanças. Tomando nosso exemplo, há uma série de possibilidades para aumentar a flexibilidade e manter a eficiência. Um exemplo seria tornar a arquitetura flexível para acomodar mudanças e evolução. Essas mudanças podem ser intencionalmente introduzidas ao longo do tempo – o que é natural em times ágeis, pois eles abraçam a mudança.

Obviamente essa não seria a única mudança para garantir a adaptabilidade e inovação, pois elas dependem das condições do sistema complexo em que estão inseridas. Inovação e adaptabilidade são conceitos diretamente relacionados à imprevisibilidade.

Um dos casos mais interessantes de inovação pela acomodação do paradoxo entre eficiência e flexibilidade é o da Toyota. Após seis anos de pesquisa científica ao redor do fenômeno do sistema de produção Toyota (*lean*) e dos resultados da empresa em épocas de profunda crise nacional e global, Osono e seus pares encontraram o que parece ser o segredo de sucesso da Toyota: abraçar ativamente paradoxos e contradições.

Sendo assim, a produtividade ágil é inerentemente ligada à capacidade de indivíduos, times e organizações de compreenderem o paradoxo entre eficiência e flexibilidade. Não apenas compreender, mas tirar proveito dessa tensão para gerar um resultado maior, a produtividade do trabalhador do conhe-

cimento e a inovação.

2.6 CONCLUSÃO

A inovação é um dos segredos de prosperidade das nações para atender necessidades existentes, melhorar as soluções existentes ou mesmo articular necessidades antes não vistas. A inovação em software depende de muitos fatores, um dos quais é a produtividade do trabalhador do conhecimento.

Neste capítulo, refletimos sobre o conceito de produtividade e sua evolução diante das necessidades e paradigmas de trabalho. Os métodos ágeis vieram como uma mudança significativa no desenvolvimento de software em resposta à transição para a era da complexidade. Discutimos a relação entre produtividade e times ágeis como uma forma de apoiar os times a compreenderem mais sobre suas diversas facetas produtivas. Por fim, também abordamos os paradoxos inerentes aos times ágeis, especialmente entre flexibilidade e eficiência. Ambos tão importantes para que a inovação aconteça e seja sustentável.

CAPÍTULO 3

Peopleware: entendendo ambientes corporativos

por Alexey A. Villas Bôas

Projetos de software nunca foram fáceis. De fato, projetos desse tipo sempre tiveram uma taxa de falha bastante desanimadora e, ao longo de muito tempo, buscaram-se explicações para isso. Novas soluções em hardware vieram, bem como o desenvolvimento de uma série de ferramentas e métodos para construção de software: técnicas de design, ferramentas de desenvolvimento, linguagens, processos. Mas o aspecto humano continua a ser determinante no sucesso de projetos.

Quem já não viu situações em que uma pessoa bem-humorada, bem relacionada e sempre preocupada com os outros e com o bem-estar geral faz

uma tremenda diferença? Na realidade, em muitos casos, ninguém percebe o que essa pessoa faz (e nem que ela faz alguma coisa de fato) até que ela sai do projeto e as coisas ficam estranhamente mais difíceis. A verdade é que muitos indivíduos com esse perfil atuam como catalisadores, ligando as pessoas, obtendo consenso, criando espaço para que todas as ideias sejam consideradas e permitindo que todos funcionem melhor em conjunto: atuam justamente no “*peopleware*” de um projeto.

Essa constatação traz uma nova responsabilidade para os profissionais de tecnologia e um novo conjunto de habilidades a ser desenvolvido. Se para haver projetos de software bem-sucedidos é necessário repensar questões humanas da equipe, então é preciso refinar e trabalhar as competências nesse sentido. E, no geral, não se pode dizer que essa atitude é exatamente confortável para a maioria dos desenvolvedores. Na realidade, quantos desenvolvedores já não passaram pela frustração de não conseguirem defender uma ideia utilizando argumentos puramente racionais? Ou já tiveram aquela desgostosa sensação de que as coisas não andam por “razões políticas”, sem entender direito o porquê? O ponto, porém, é perceber exatamente o que está acontecendo, quais interesses pessoais estão em jogo, como as pessoas são motivadas e quais os seus objetivos. Isso é fundamental para que o projeto avance como um todo. Neste capítulo, fala-se de motivação e de ferramentas que auxiliam a compreender o comportamento de pessoas que atuam isoladas ou como membros de um time.

3.1 CARGOS E ESTRUTURAS

Naturalmente, os diferentes papéis em um projeto requerem diferentes níveis de maturidade no uso de ferramentas para motivação. Gerentes de projetos são um exemplo disso: dificilmente uma pessoa terá tal cargo se não tiver seus *soft-skills* bem desenvolvidos. Entretanto, gerentes não são os únicos a se beneficiarem desses atributos: qualquer um que precise obter consenso, conciliar interesses ou pontos de vista, defender uma ideia ou provocar mudança em um projeto precisa fazer uso de algumas dessas competências, em maior ou menor escala. Situações nas quais elas são necessárias são certamente muito comuns a desenvolvedores, líderes técnicos, arquitetos ou ana-

listas de negócio.

De fato, Niels Pflaeging comenta que toda organização possui duas estruturas internas: a bem conhecida estrutura hierárquica de poder e a estrutura social. A primeira reflete a distribuição de poder advindo da disposição de cargos e chefias. A segunda representa as relações individuais entre as pessoas, o poder de persuasão e influência que elas têm entre si e o efeito em rede que isso traz. Normalmente, a capacidade que a estrutura social tem de produzir mudanças em uma empresa (especialmente quando aspectos culturais estão em jogo) supera a da estrutura hierárquica, fato que é, muitas vezes, negligenciado.

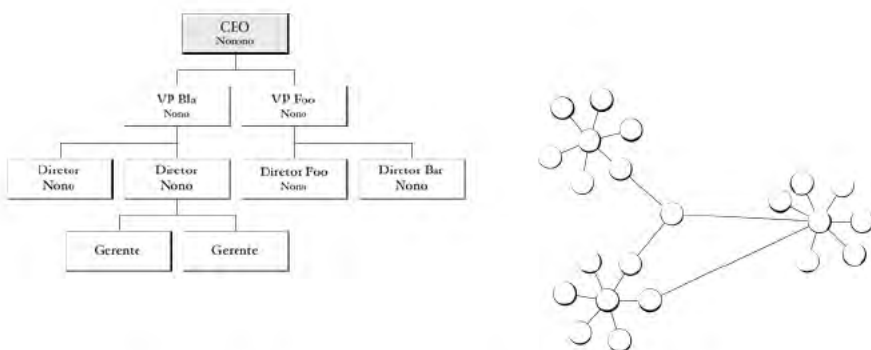


Fig. 3.1: As estruturas internas de uma organização

3.2 EMPATIA

No livro “Team of Rivals”, Doris Goodwyn traz uma biografia de A. Lincoln com uma exploração bastante profunda de suas manobras e habilidades políticas. O título refere-se à formação do gabinete presidencial abordando, em especial, quem eram os membros, sua percepção sobre Lincoln antes de ele assumir a presidência e como essa visão se modificou ao longo do mandato

do presidente. Goodwyn conta que, nas prévias eleitorais, Lincoln, um advogado desconhecido de Illinois, concorreu com políticos de grande prestígio e fama e acabou sendo eleito como o presidenciável pelo partido republicano. Uma vez eleito presidente dos Estados Unidos – e este é o principal fato que define o título do livro –, levou para compor o gabinete presidencial exatamente aquelas pessoas que ele havia derrotado nas prévias: grandes rivais, que tinham pouca ou nenhuma admiração por sua capacidade de liderança ou política. Durante o mandato, Lincoln se mostrou não só capaz de formar um time com esses rivais, combinando as melhores habilidades de cada um e permitindo que eles funcionassem em conjunto, como também ganhou o respeito e a amizade de quase todos os membros do gabinete.

Segundo a autora, uma das habilidades mais importantes do ex-presidente em sua carreira política foi sua grande empatia, que ela descreve como “[...] o dom ou maldição de se colocar no lugar de outro, sentir o que está sentindo e compreender suas razões e desejos.” Adam Smith também dá sua descrição: “Através da imaginação, nos colocamos na posição de outra pessoa [...] e até certo ponto, nos tornamos essa pessoa”.

Um dos soft-skills mais elementares em um projeto é compreender o que pensam e buscam as pessoas ao redor e antever como podem agir em determinada situação ou reagir a uma dada ação. Com isso, as ações dentro de um projeto se tornam mais efetivas e tendem a trazer resultados mais alinhados com o que se busca (e também com o que outros buscam). É importante aplicar a ideia de Smith de “se tornar um pouco a outra pessoa através de imaginação” para se ter uma visão mais ampla sobre como abordá-la, motivá-la ou mesmo influenciá-la. Para tanto, é necessário empatia: colocar-se no lugar de outra pessoa e, até certo ponto, ver o mundo como ela vê, entender seus objetivos e desejos e perceber os problemas e angústias que ela enfrenta. Isso é especialmente importante em um ambiente corporativo e, ainda mais, em tentativas de mudança. De fato, a falta desse tipo de compreensão leva, muitas vezes, ao desperdício de energia e de tempo, desgastando relacionamentos e trazendo frustração para todos.

3.3 MOTIVADORES

Motivadores são os mecanismos e influenciadores fundamentais subjacentes às ações. Compreender quais estão em jogo e o grau de influência que têm sobre cada pessoa é um elemento fundamental no exercício de empatia.

Os motivadores são classificados em dois tipos: motivadores extrínsecos e intrínsecos. Motivador extrínseco é qualquer coisa fora do sujeito que é necessário para aumentar a motivação. Alguns exemplos são: bônus, notas na escola, medalhas em competições esportivas etc. A motivação intrínseca é o oposto. É quando ganhar um bônus, uma medalha ou tirar uma nota alta não motiva tanto quanto a alegria do trabalho, da aprendizagem e de todos os fatores que refletem as paixões e a autoestima pessoal.

3.3.1 Motivadores extrínsecos

Kurt Lewin diz que o comportamento de uma pessoa é resultado de sua personalidade e do meio em que ela se encontra. Desse modo, é natural que se inicie a exploração pelo ambiente em que se está inserido: empresas e corporações e os motivadores que elas produzem. De modo geral, é bastante difundida a ideia de que uma empresa deve estabelecer metas, elaborar uma forma de medir seu cumprimento e monitorar a performance de sua execução.

Nessa perspectiva, nada mais natural que oferecer recompensas por resultados. Ao oferecer recompensas pelo cumprimento de um determinado objetivo, a empresa irá motivar as pessoas a trabalharem para sua obtenção, em uma simples relação de causa e efeito. Contudo, a literatura mostra que medir obtenção de objetivos não é tão simples quanto parece e que empresas e projetos são sistemas adaptativos complexos, que não possuem relações de causa e efeito claras e simples. Apesar dessas críticas à sua aplicação, é importante atentar para dois pontos: motivadores extrínsecos – na forma de recompensas por objetivos alcançados (ou punições) – e monitoramento de desempenho existem com frequência em empresas e eles causam disfunções, isto é, comportamentos desalinhados com os objetivos da organização ou do projeto (mas alinhados com as métricas coletadas).

Disfunções – na realidade, as métricas de desempenho que existem e as

recompensas e punições associadas – são o primeiro elemento a que se deve estar atento quando se pretende perceber o que está influenciando as pessoas ao redor. Trata-se de um elemento externo aos sujeitos e, portanto, mais objetivo de ser identificado. As disfunções normalmente possuem uma influência bastante poderosa na forma como as pessoas agem (de forma positiva ou inibidora). Tanto um bônus financeiro considerável quanto o receio de ser percebido como uma pessoa (ou um gerente, ou uma equipe) com baixo desempenho têm consequências bastante fortes para a carreira individual de qualquer um.

Por fim, apresenta-se um ponto sutil, mas fundamental: o que importa é o valor da métrica de desempenho ou da medida de objetivo alcançado, e não o desempenho real ou o atingimento de fato da missão da empresa (essa é a disfunção produzida). Muitas vezes, trata-se de buscar informações objetivas com relação à forma de avaliação da empresa, como por exemplo: “Quais métricas estão sendo coletadas?”, “Quais os critérios para pagamento de bônus deste ano?”, “Que metas estão em jogo?”, “Como a performance individual está sendo avaliada?”.

3.3.2 Motivadores intrínsecos

Muitas empresas têm a visão de que incentivos extrínsecos são suficientes para motivar as pessoas. Entretanto, vários trabalhos apontam para o sentido contrário e ressaltam a importância dos motivadores intrínsecos. Assim, apresentam-se algumas ferramentas que auxiliam a perceber os motivadores (ou inibidores) internos de uma pessoa. Nesse âmbito, tudo se torna mais sutil e subjetivo ainda (e a observação mais delicada). Tendo isso em vista, é importante lembrar que o que segue são apenas ferramentas inspiradoras e que o bom senso deve prevalecer sempre. É necessário separar fatos de suposições e tratar as conclusões como hipóteses que devem ser validadas com acontecimentos reais.

3.4 ALGUMAS FERRAMENTAS

No que segue, tenta-se desenvolver algumas ferramentas para criar o mapa de empatia de uma pessoa. Claro que seres humanos são muitos mais com-

plexos do que qualquer truque que se possa conceber e, dessa forma, não se tem a pretensão de desenvolver um modelo para o comportamento humano, mas sim fornecer ferramentas que auxiliem no exercício de imaginação recomendado por Smith. Todos esses instrumentos devem ser vistos e usados com uma bela pitada de bom senso.

3.4.1 A escala de necessidades de Maslow

Uma primeira ferramenta é a escala de necessidades de Maslow. Segundo essa teoria, as necessidades dos seres humanos podem ser largamente agrupadas e hierarquizadas em 5 categorias. Maslow diz que, enquanto as necessidades de determinada categoria não são satisfeitas, uma pessoa tem sérias dificuldades para dar atenção às necessidades do nível superior.

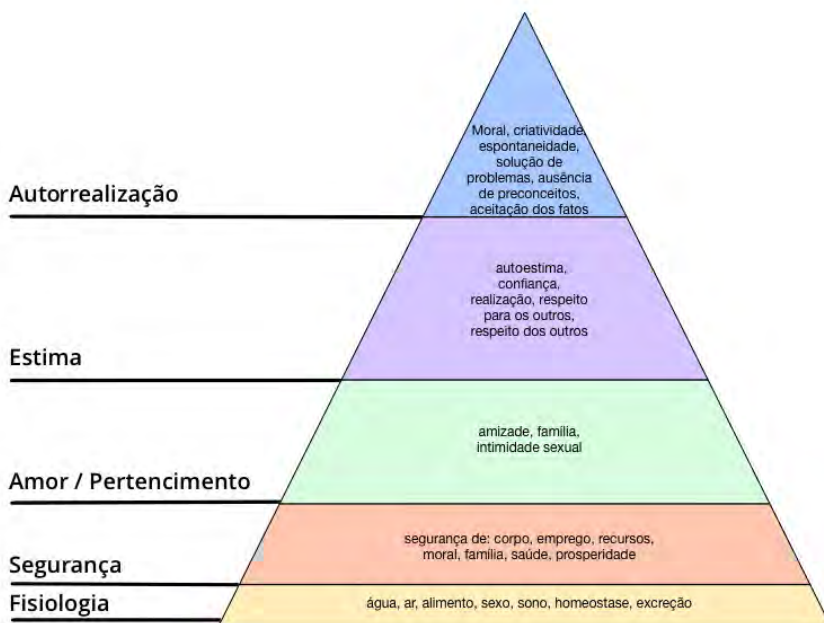


Fig. 3.2: A escala de necessidades de Maslow

Em um contexto de projeto, pode-se entender que um gerente terá sérias

dificuldades para se concentrar em fatores mais altos da pirâmide, como inovar nas práticas de formação de times (nível de realização pessoal), se estiver com preocupações grandes no nível de segurança (como perder o emprego, por exemplo). Desse modo, a teoria de Maslow é uma ferramenta que possibilita perceber o melhor âmbito de ação: tem-se pouco resultado tentando atuar no nível de realização pessoal se a pessoa estiver enfrentando insegurança. Nesse caso, é preciso atuar no nível mais baixo para se ter efetividade.

Em particular, é bastante válido inspecionar o nível de segurança das pessoas. Como mostra a pirâmide, segurança é o motivador mais básico em ambientes corporativos (assumindo que todos têm livre acesso aos recursos fisiológicos), sendo que uma pessoa com medo terá seu comportamento muito afetado por esse sentimento. Nesse caso, é fundamental entender a causa do medo e tratar dela antes de tomar qualquer outra medida. É muito comum medidas de performance serem vistas como ameaça e causarem esse efeito. Motivadores extrínsecos são um bom lugar para procurar por causadores de insegurança.

3.4.2 Os níveis lógicos de pensamento

Os níveis lógicos de pensamento, de Bateson e Dilts, é outra ferramenta interessante.



Fig. 3.3: Os níveis lógicos de pensamento

De acordo com esse modelo, as ações de uma pessoa têm origem em suas capacidades e estratégias, que são determinadas por suas crenças, e assim por diante. Essa escala fornece um guia análogo ao dado pela escala de necessidades de Maslow: não adianta tentar resolver o que uma pessoa faz se ela não possui as habilidades para tanto. Mais do que isso, não adianta esperar comportamento de liderança de um gerente se ele não se identifica como líder e não tem isso como uma missão pessoal. Esse modelo também é um instrumento que possibilita identificar em que nível atuar e onde está a raiz de determinado comportamento. Isso ajuda, por exemplo, a identificar passos para introduzir mudanças: “A mudança está alinhada com a visão e missão das pessoas?”, “Elas acreditam que trará benefícios?”, “Sabem como e o que deve ser feito? Têm as habilidades necessárias?”, e assim por diante.

3.4.3 Os 10 desejos intrínsecos

Jurgen Appelo traz uma lista de 10 desejos intrínsecos:

Aceitação	A necessidade de aprovação
Curiosidade	A necessidade de pensar
Poder	A necessidade de influenciar
Honra	A necessidade de ser fiel a um grupo
Contato social / Relacionamentos	A necessidade de amigos
Idealismo / Propósito	A necessidade de ideal / propósito
Status	A necessidade de reconhecimento social
Independência / Autonomia	A necessidade de ser um indivíduo
Ordem	A necessidade de ambientes estáveis
Competência / Maestria	A necessidade de se sentir capaz

Fig. 3.4: Os 10 desejos intrínsecos

Repassar essa lista com determinada pessoa em mente ajuda a entender que tipo de desejos ela tem. Tais desejos guiarão suas ações, seu pensamento e sua forma de perceber a realidade.

É natural que essas três ferramentas possuam intersecções e, algumas vezes, até alguma contradição na sua aplicação. É importante, porém, lembrar que não são instrumentos formais (nem mesmo esses modelos, por si só, o

são), mas guias que ajudam a compreender como outra pessoa vê e interage com o ambiente a sua volta.

3.4.4 Mapas de empatia

Mapas de empatia foram originalmente desenvolvidos para análise de segmentos de consumidores e são explorados por Osterwalder e Pigneur. Com pequenas modificações, podem ser muito úteis para diagramar informações e fornecer uma visualização resumida do que se coleta sobre determinada pessoa. A seguir, veja o desenho do mapa de empatia de uma pessoa (Beto), contendo os elementos que se pode preencher em cada seção.

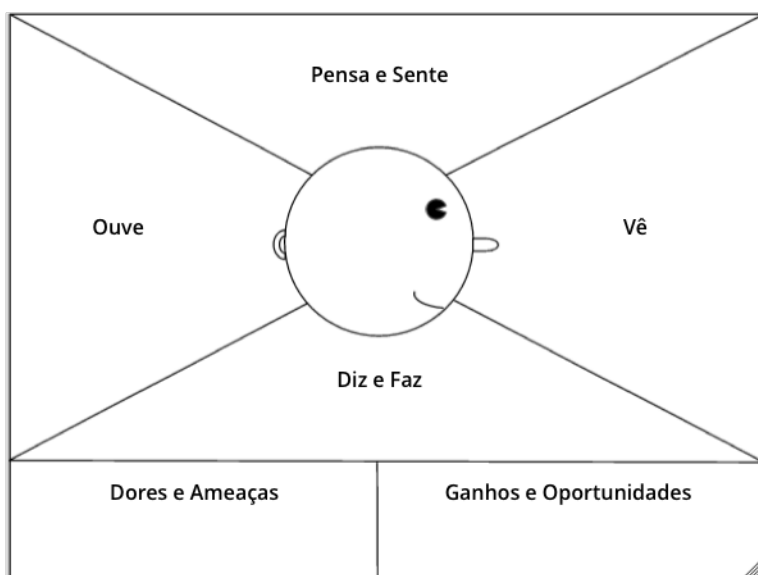


Fig. 3.5: Mapa de Empatia

Normalmente, é útil iniciar a análise de empatia de uma pessoa regis-

trando seus comportamentos objetivamente na seção *Diz e Faz* e, em seguida, derivar as outras seções.

Diz e Faz

- Quais são as ações de Beto, o que ele diz no dia a dia e como ele se comporta?

Ouve

- O que ou quem influencia Beto? Quem Beto escuta?

Vê

- O que Beto vê em relação ao ambiente e a outras pessoas?
- Como Beto nota-se percebido na empresa ou no ambiente?
- O que mais chama a atenção de Beto no ambiente?
- Quais são os motivadores extrínsecos que chamam a atenção de Beto?

Pensa e Sente

- O que Beto pensa? Como ele se sente?
- Quais necessidades (escala de Maslow) ele sente que não são atendidas?
- Quais são suas crenças (níveis lógicos de pensamento)?
- Quais são as habilidades que Beto percebe que tem (níveis lógicos de pensamento)? E quais sente que não tem?
- Quais são os desejos intrínsecos que motivam Beto (lista de 10 desejos intrínsecos)?

Dores e Ameaças

- Quais necessidades ele sente que não são atendidas (escala de Maslow)?
- Quais medidas externas de performance são vistas como risco ou ameaça?

Ganhos e Oportunidades

- Quais desejos intrínsecos satisfeitos trazem satisfação pessoal (lista de 10 desejos intrínsecos)?
- Quais motivadores extrínsecos e recompensas estão em jogo para Beto?

É bastante válido atentar para o fato de que as seções sofrem impactos entre si. Por exemplo, a forma como Beto pensa e sente influencia o que ele vê e a sua percepção da realidade: ele distorcerá o que vê e aplicará abstrações e analogias de acordo com o seu modo de pensar, as suas crenças e as suas habilidades. As pessoas que o influenciam causarão impactos na forma como ele pensa e sente, e assim por diante.

Por fim, vale comentar sobre como coletar essas informações. Essa ação é mais simples e menos fácil do que parece. Passa por conversas de bebedouro, almoços, happy hours, pela percepção de quem se relaciona e conversa com quem (“Quais são as pessoas que frequentemente almoçam juntas?”), pela percepção das expressões faciais durante conversas etc. É claro que usar fones de ouvido não ajuda em nada nesse processo. A boa notícia é que essa habilidade se desenvolve de forma bastante natural quando se está atento ao que acontece ao redor.

3.5 COMO USAR TUDO ISSO: SEU CÍRCULO DE INFLUÊNCIA

Uma visão clara do mapa de empatia de determinada pessoa traz melhores recursos para comunicação e relacionamento com esse sujeito, com vista a engajá-lo na solução de problemas, no projeto ou mesmo influenciá-lo. O

próximo conceito natural, nesse ponto, é dado por Stephen Covey: seu círculo de influência. Trata-se das pessoas e coisas que você pode influenciar. Parece um conceito simples, mas é extremamente poderoso. Tentar causar impactos fora do seu círculo pessoal é muito improdutivo. Em vez disso, procure usar o que está dentro do seu próprio círculo ou ampliá-lo.

Uma observação interessante é a de que se pode usar os círculos de influência de forma transitiva: influenciar uma pessoa que, então, influenciará a diretoria, por exemplo. A seção “Ouve” do mapa de empatia tenta capturar exatamente essas possibilidades. É importante combinar isso com as estruturas hierárquica e social da empresa. O uso de ambas é fundamental para a introdução de mudanças. Sobre esse tópico, Mary Linn Manns e Linda Rising trazem uma série de padrões que podem ser muito úteis.

Vale ainda comentar que seu círculo de influência muda com o tempo: ele pode crescer ou diminuir. Assim, é preciso esforço ativo tanto para trazer novas pessoas para ele quanto para manter as que já estão.

Lightstone cita o interessante conceito de *caches emocionais* (por vezes chamado de capital político): eles são uma abstração para brincar com a ideia de “quanto aquela pessoa está disposta a me ajudar, a fazer algo por mim”. Dividir sucessos, ajudar outras pessoas desinteressadamente, dar su-porte anonimamente e aumentar a autoestima são alguns exemplos de atitudes que podem aumentar seu cache emocional com outra pessoa. Cuidar bem dos caches ajuda a cultivar o círculo de influência. É sempre bom estar atento aos caches, pois chega a hora em que se pode precisar de auxílio.

Outros elementos que ajudam a aumentar o círculo e obter confiança de outras pessoas incluem: ser um “fazedor”, uma pessoa que conhece os aspectos técnicos envolvidos e consegue obter resultados; fazer bom uso de autoridade natural; gerar confiança (fazer aquilo que diz que vai fazer) e cuidar da comunicação. Sobre esse último ponto, vale ressaltar a questão de se ter consciência de comunicação pública e privada e redobrar a atenção para não expor pessoas em ambientes não seguros (assumindo sempre que reuniões não são ambientes seguros).

O conjunto de habilidades que se pode descrever como “soft-skills” ainda é muito mais amplo. Ele inclui bom gerenciamento de reuniões, de comunicação, introdução de mudanças, negociações etc. Entretanto, as noções de

empatia e influência aqui discutidas são, de certa forma, um primeiro passo na jornada (sem fim, claro) do desenvolvimento dessas habilidades.

CAPÍTULO 4

Psicologia cognitiva explicando Ágil

por Fabio Pereira

Psicologia?! Será que eu estou lendo o livro certo?

Se você se fez essa pergunta, vou tentar ajudá-lo a entender se esse assunto lhe é mesmo útil. Este capítulo é direcionado a pessoas curiosas, criativas, que pensam fora da caixa, que desejam aprender um pouco mais sobre como o cérebro humano funciona e, acima de tudo, que querem entender algumas razões por trás do imenso sucesso do Ágil, conceito que mudou a forma com que o mundo desenvolve software. Falarei sobre a relação entre pesquisas científicas que explicam o comportamento humano, discutirei sobre Psicologia cognitiva e sobre como alguns experimentos científicos realizados nessa área

podem ser correlacionados a diversos aspectos do mundo Ágil.

Não entrarei em detalhes no que diz respeito à Agilidade, portanto, algum conhecimento básico do Manifesto Ágil, como seus valores, princípios e práticas, é de extrema importância para o entendimento dos assuntos aqui apresentados. Se você pensou: “Eu sei o que é uma reunião de pé e que pessoas são mais importantes do que processos”, creio que já pode começar a ler e, se tiver alguma dúvida, converse, pergunte e discuta com seus colegas de trabalho e amigos. Esse é um dos objetivos que tenho ao escrever este artigo: gerar discussões para o compartilhamento de conhecimento.

Ao pensar em psicologia, muitos imaginam alguém sentado em um divã chorando e relembrando seus traumas de infância. Essa psicologia, muitas vezes estereotipada, é uma das subáreas da psicologia, a saber, a Psicanálise, que é diferente da Psicologia cognitiva. Como mostrarei em todos os experimentos aqui explicados, a Psicologia cognitiva utiliza o Método Científico [15] para chegar a resultados. Por este e outros motivos, as conclusões encontradas são uma prova científica de por que a revolução ágil mudou a forma como se desenvolve software no mundo.

4.1 NÃO SOMOS RACIONAIS ILUSÕES COGNITIVAS E UM CÉREBRO QUE PENSA DE FORMA RELATIVA

O meu interesse pelas áreas de cognição e tomada de decisões surgiu ao assistir a uma palestra que explicava o experimento da revista *The Economist* (<http://www.economist.com>), em que dois tipos de anúncio foram apresentados aos possíveis assinantes.

A imagem a seguir mostra os dois anúncios e os resultados das vendas:

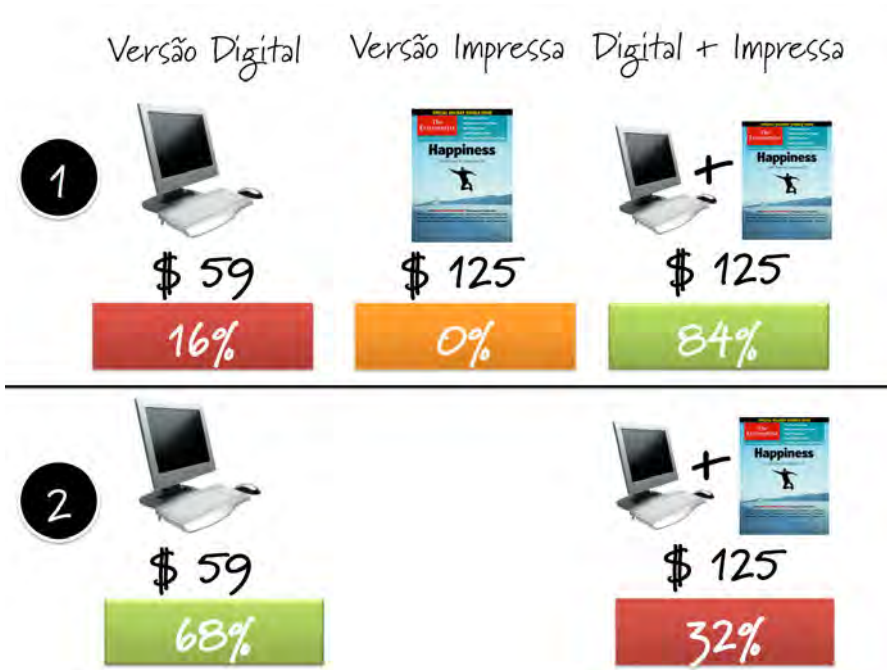


Fig. 4.1: Anúncios e resultados das vendas

Algumas observações, questionamentos e conclusões sobre o experimento e seus resultados são elencados a seguir:

- Para um grupo de pessoas, foi apresentado o anúncio 1, que possibilita 3 opções. Para outro grupo, foi apresentado o anúncio 2, que disponibiliza apenas 2 opções idênticas às do anúncio 1, com exceção da opção de assinatura da versão impressa, que foi retirada.
- A maioria (84%) dos assinantes que viu o anúncio 1 comprou o pacote digital e impresso. Perceba que essa alternativa custa o mesmo valor da versão apenas impressa, ou seja, comprando o pacote duplo pelo mesmo preço é como se o assinante estivesse ganhando um produto de graça.
- A maioria dos assinantes que viu o anúncio 2 (68%) comprou somente a assinatura digital da revista.

- A única diferença entre os dois anúncios é a opção de assinatura impressa, que custa o mesmo valor do pacote digital e impresso juntos. Essa opção não foi escolhida por nenhum dos assinantes.
- Por que adicionar ou remover uma opção que não é escolhida por ninguém modifica a maioria dos pacotes adquiridos? A explicação para essa pergunta é o “efeito decoy”, fenômeno que determina que os consumidores têm uma tendência maior a mudarem de opinião na escolha entre duas opções quando uma terceira alternativa é colocada na equação, contanto que essa terceira proposição contraste com uma das duas iniciais. Esse contraste significa que a terceira opção fará com que uma das alternativas iniciais pareça “melhor”, ficando, então, mais fácil comparar.

Por exemplo, supondo que as duas opções iniciais sejam A e B, ao introduzir uma opção -B, que faz B parecer melhor, existe uma probabilidade maior de a proposição B ser escolhida. No caso das assinaturas da revista *The Economist*, essa alternativa -B é a opção *decoy* ilustrada na figura a seguir.



Fig. 4.2: A opção decoy

Esse experimento, dentre outros, é apresentado no livro *Previsivelmente irracional* do autor Dan Ariely, que explica e comprova que, se agísse-mos de forma racional, não existiria efeito *decoy*, ou seja, as escolhas seriam as mesmas para os anúncios 1 e 2. O fato de uma opção que ninguém escolheu mudar o processo de tomada de decisões define o que ele chama de comportamento irracional. Segundo o autor, se agíssemos de forma racional, a opção *decoy* não deveria alterar em nada os resultados.

4.2 ILUSÕES DE ÓPTICA E ILUSÕES COGNITIVAS

As famosas e conhecidas ilusões de óptica são uma prova concreta de que o nosso cérebro e os nossos sentidos não são perfeitos. Veja a imagem a seguir:

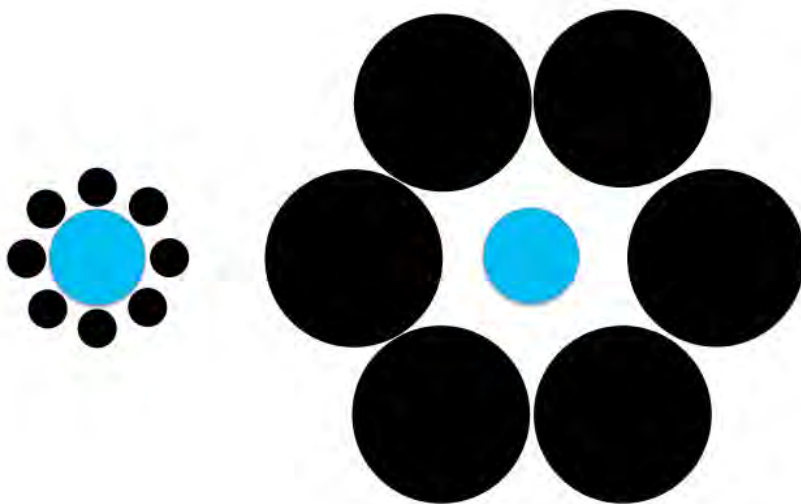


Fig. 4.3: Ilusões de óptica

A maioria das pessoas que olha para essa imagem tem a impressão de que o círculo interno da esquerda é levemente maior do que o da direita, quando, na verdade, os dois são do mesmo tamanho. Essa “ilusão” é causada pelos

círculos externos. Na esquerda, os círculos externos menores fazem pensar que o interno é maior. Da mesma forma, os círculos externos maiores, na direita, fazem pensar que o interno é menor. Esse é um exemplo ideal que mostra que o cérebro funciona de forma relativa, e não absoluta. Mesmo que se tente, não se consegue isolar e olhar somente para os círculos internos, ignorando os externos. Só se consegue perceber que os círculos internos são do mesmo tamanho ao remover os círculos externos.



Fig. 4.4: Círculos

Da mesma forma que a nossa percepção de imagens não é perfeita, também não são perfeitos os nossos processos cognitivos, a tomada de decisões e a percepção do mundo à nossa volta. É como se os círculos externos fizessem o papel de *decoy*. A retirada deles, assim como a exclusão da opção de assinatura que ninguém escolheu, faz com que o cérebro interprete o que está vendo de forma diferente.

O efeito *decoy*, assim como outros que mostrarei neste capítulo, é considerado uma ilusão cognitiva, podendo ser comparado com as ilusões de óptica. Ilusões cognitivas são o resultado de inferências que fazemos de forma inconsciente, mas que têm um impacto significativo no processo de tomada de decisões e no comportamento em geral.

4.3 O QUE AS ILUSÕES COGNITIVAS TÊM A VER COM DESENVOLVIMENTO ÁGIL?

Não só o efeito *decoy*, que mostra que tomamos decisões de forma relativa, mas outras ilusões cognitivas que discutirei mais adiante podem ser usadas como referência para entender o porquê das diversas práticas, princípios e valores ágeis. Além do mais, o primeiro valor do Manifesto Ágil sugere que se valorizem “indivíduos e interação entre eles mais que processos e ferramentas”. Entender o comportamento humano, questões sobre a memória, atenção, percepção, representação de conhecimento, raciocínio, criatividade, resolução de problemas e tomada de decisões, aspectos que são exatamente a base da Psicologia cognitiva, é extremamente compatível com esse foco nos indivíduos e em suas interações.

Uma vez contextualizado que ilusões cognitivas estão presentes no comportamento humano, abordarei algumas delas, juntamente com os experimentos que as comprovam. Durante esse processo, procurarei discutir similaridades dos experimentos e dos resultados com o desenvolvimento ágil.

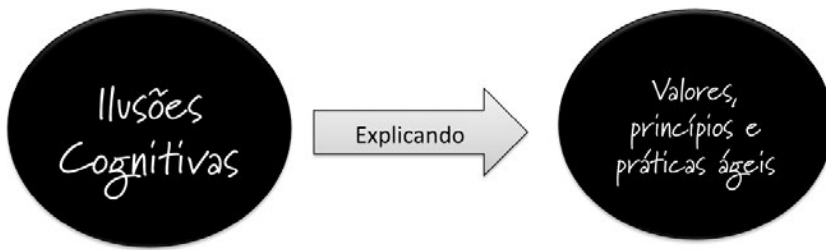


Fig. 4.5: Explicando ilusões cognitivas

4.4 SÍNDROME DO ESTUDANTE EXPLICANDO ITERAÇÕES

Em um dos seus experimentos, Dan Ariely, como professor, precisava que os alunos entregassem 3 trabalhos no final de um curso de 12 semanas. Ele queria estudar o impacto da definição de prazos e se os alunos os cumpririam. O pesquisador resolveu, como na maioria dos seus experimentos, utilizar o método científico, além de abordagens diferentes para algumas turmas.

- **Prazo único:** para uma turma, Ariely definiu que os 3 trabalhos deveriam ser entregues até o final do curso de 12 semanas, ou seja, só existia um prazo final para tudo o que deveria ser feito.
- **Prazos iterativos:** para outra turma foi estabelecido o prazo de entrega de um trabalho a cada 4 semanas.

Então, meu caro leitor, qual você acha que foi o resultado desse experimento? As conclusões a que Ariely chegou foram:

- A maioria dos alunos que tinha prazo único entregou todos os trabalhos no final, o que mostra que deixamos as coisas para a última hora. Tal fato não configura grande novidade. Ariely confirma que o ser humano é procrastinador por natureza, sofrendo da chamada **Síndrome do Estudante**. Quem já foi aluno sabe que a tendência é só estudar para a prova na véspera.
- Os alunos que tinham prazos iterativos, a cada 4 semanas, obtiveram as melhores notas. Isso mostra que uma ótima forma de diminuir a tendência à procrastinação e de melhorar a qualidade do que se faz é estabelecer prazos e comprometerimentos, o que resulta em melhores trabalhos.



Fig. 4.6: A síndrome do estudante

Prazos iterativos e comprometimentos lembram alguma coisa a você? Uma das grandes revoluções do desenvolvimento Ágil foi o modelo iterativo em contraste ao tradicional modelo cascata. Iterações de até 4 semanas, passando por todas as fases necessárias para produzir software que adicione valor de negócio, é uma das propostas de algumas metodologias ágeis.

Os resultados e conclusões desse experimento trazem clara contribuição para se conseguir explicar as vantagens do modelo iterativo em relação ao de cascata.

4.5 CEGUEIRA NÃO INTENCIONAL

Imagine que alguém pediu para você assistir a um vídeo de seis pessoas brincando com bolas de basquete. Dos seis sujeitos, metade está com camiseta branca e metade com camiseta preta. Seu objetivo é prestar muita atenção e contar quantas vezes as pessoas com camiseta branca passam a bola entre si.

Você assiste ao vídeo e, ao final, lhe perguntam se havia alguma coisa diferente no filme. Você responde que não e confirma acertadamente o número de passadas de bola. Perguntam, então, se havia um gorila no vídeo. Você se assusta e diz: “Um Gorila?! Claro que não!”

Esse experimento foi proposto pelos psicólogos Daniel Simons e Christopher Chabris. Metade dos voluntários não viu o gorila que apareceu no meio do vídeo, olhou para a câmera, bateu no peito e saiu. Esse resultado chocante revela que as pessoas, em geral, estão tão focadas em um evento que se tornam “cegas” a outros.

Esse e outros estudos comprovam o que é conhecido como *cegueira não intencional*. Tal situação é definida como a falha humana em detectar e perceber um estímulo ou evento inesperado quando o indivíduo está focado em outra tarefa que requer a sua atenção. Basicamente, quando se está muito focado em uma tarefa, não se percebe coisas importantes que acontecem ao redor.

Desenvolvimento de software exige muita atenção. Muitas vezes, os membros de uma equipe ficam tão focados em implementar funcionalidades, escrever código, testar e definir requisitos, que não prestam atenção a eventos extremamente importantes que acontecem à sua volta. É importante dizer que tais circunstâncias não têm um culpado, mas acontecem devido às “falhas” da cognição, como explicado anteriormente. Quem já não passou por uma situação clássica em que alguém pergunta:

“Você leu aquele e-mail?” E você pensa: “Que e-mail? Não me lembro de ter recebido”.

Comunicação é um dos pontos mais importantes em uma equipe ágil. No mundo ágil, existem metáforas que categorizam as formas com que a informação pode ser compartilhada em uma equipe, sendo elas a da geladeira ou a do radiador.

4.6 GELADEIRA OU RADIADOR?

Quando se precisa de algo que está dentro da geladeira, é necessário abri-la para buscar o que se quer. Deve-se, antes disso, saber se aquele item está dentro dela. O mesmo acontece com a informação “guardada” dentro de uma caixa de entrada de e-mail, por exemplo. É preciso abrir o e-mail e buscar para acessar a informação. Por isso, uma mensagem enviada quando um teste automatizado falha não é suficiente para chamar a atenção do desenvolvedor e, assim, corrigir o problema. Esse e-mail será, provavelmente, esquecido, não lido, ou seja, deixado dentro da geladeira.

Radiadores, como o nome diz, difundem informação o tempo todo, de forma visível e clara para todos os membros da equipe. Gráficos e informações grandes e visíveis são preferíveis quando comparados a informações guardadas em e-mails ou em alguma aplicação difícil de ser acessada. Estes são os chamados BVCs (*Big Visible Charts*). Equipes ágeis usam radiadores de informação para deixar claro para a equipe dados como:

- O estado atual do que foi planejado para ser entregue com um *Burn Up/Down Chart*;
- A qualidade e o feedback quando algo está quebrado, o que é rapidamente visível através de um *Build Light* ou um *Build Dashboard*;
- Quadros com tarefas e fotos de quem está trabalhando em qual funcionalidade;
- Diagrama com datas importantes de projeto;
- Quem está de férias e quando voltará;
- Diagrama de problemas causados pela dívida técnica comparados com o esforço necessário para resolver;
- Qualquer informação que seja importante para o projeto e que precise ser divulgada.



Fig. 4.7: Radiadores de informação

Outra prática, além dos radiadores de informação, que tem como objetivo trazer informações à equipe, é a famosa reunião de pé, talvez uma das técnicas mais conhecidas do desenvolvimento ágil. Um dos maiores benefícios da reunião de pé é difundir informação clara e sucinta para toda a equipe. Diversas vezes já presenciei aquele momento em que um membro da equipe ouve uma informação pela primeira vez e chega à conclusão de que só agora ficou claro o que estava acontecendo no dia anterior, quando então ele estava tão focado em outras tarefas que não percebeu. Ficou “cego de forma não intencional” ao que estava acontecendo ao seu redor.

Portanto, práticas como radiadores de informação e reuniões de pé possuem características que, claramente, previnem e mitigam os riscos causados pela cegueira não intencional.

4.7 UM DESMAIO NA RUA, A DIFUSÃO DE RESPONSABILIDADE E OS ITENS DA RETROSPECTIVA

Imagine que você está passando por uma calçada cheia de pessoas e uma delas está deitada no chão, desmaiada. O que você faz?

- Para e pergunta o que aconteceu? Chama uma ambulância?
- Pensa algo parecido com: “existem várias pessoas em volta, portanto alguém já deve ter tomado as providências necessárias”.

Diversos estudos psicossociais comprovam que, em uma situação em que alguém necessita de ajuda, a probabilidade de obter socorro é inversamente proporcional ao número de pessoas que estão em volta. Isso significa que, quanto mais gente houver, menor é a probabilidade de que alguém ajude. Intuitivamente se pensa que, quanto mais pessoas há, mais fácil é o socorro, quando a verdade é o oposto. Esse fenômeno é conhecido como *Difusão de responsabilidade*, o que faz com que a maioria das pessoas pense que, pelo fato de muitos estarem presentes, a responsabilidade é distribuída entre todos e que alguém já deve ter feito o que precisa ser feito.

Uma das estratégias para aumentar a probabilidade de ser ajudado é apontar um responsável. Se você estiver sendo assaltado na rua é mais eficiente apontar alguém específico para pedir ajuda, como por exemplo: “Você de camiseta verde! Preciso de ajuda!”. Tal atitude é mais eficaz do que gritar e fazer um pedido de ajuda genérico a qualquer um à sua volta.

O momento de retrospectiva de um projeto é extremamente importante, pois é nele que se consegue a melhoria contínua. Existem diversos modelos de retrospectiva, mas algo em comum à maioria deles é uma lista de itens a serem executados, os chamados *action items*. Um bom facilitador de retrospectivas sabe que esses itens precisam de “donos” ou responsáveis (*action owners*). Quando alguém se compromete a executar um determinado item e a ser o responsável por ele, é mais provável que o item seja finalizado. Também é de extrema importância revisar todos os itens da retrospectiva anterior e confirmar com os seus donos quais foram executados com sucesso ou não.

Quando os itens não possuem donos, é mais provável que todos fiquem esperando que alguém os faça e que ninguém acabe por executá-los. Portanto,

associar donos aos itens das retrospectivas é outra estratégia para evitar a difusão de responsabilidade.

Geralmente, times ágeis têm a sensação de que as responsabilidades são bem compartilhadas pelos membros da equipe. Entretanto, muitas vezes é difícil explicar o motivo disso. O uso das estratégias que evitam a difusão de responsabilidade, como a reunião de pé e a nomeação de donos para os itens das retrospectivas explica essa sensação de boa distribuição de trabalho e, conseqüentemente, de aumento de produtividade.

4.8 VALORIZO, POIS TAMBÉM SOU DONO

Outro experimento feito por pesquisadores consistiu em um sorteio de ingressos para um jogo muito disputado. Depois do sorteio, perguntou-se ao grupo de ganhadores por quanto eles estariam dispostos a vender e, ao grupo que não ganhou, quanto eles pagariam por esses ingressos. Os que ganharam venderiam, em média, por \$2400 (dois mil e quatrocentos dólares), enquanto que o outro grupo os compraria por \$170 (cento e setenta dólares), em média.

- “Vendo por \$2400”.
- “Compro por \$170”.

Tal resultado significa que a posse de um item, nesse caso o ingresso, resulta na alta valorização do objeto. Esse fenômeno é conhecido como *efeito endowment*.

Voltando ao mundo do desenvolvimento de software, o cuidado que os desenvolvedores têm com o código é diretamente ligado à qualidade do mesmo e, assim, ao número de problemas e bugs de um sistema. O que o efeito *endowment* sugere é que:

“Se os desenvolvedores se sentirem “donos” de um determinado código, existe uma probabilidade maior de ele ser valorizado e bem mantido.”

– efeito *endowment*

Código coletivo é uma das práticas do *Extreme Programming*, uma metodologia ágil também conhecida como XP. Um dos objetivos dessa prática é

fazer com que os desenvolvedores se sintam mais donos de todo o código e, assim, cuidem melhor dele, aumentando a qualidade total da base de código gerada e mantida.

A situação inversa e problemática cujo impacto o XP se propõe a minimizar é aquela em que cada parte do código tem um dono específico. Trata-se daquele momento no qual uma determinada classe precisa ser modificada e o desenvolvedor diz: “Não posso mudar este código porque ele é de fulano de tal e só ele modifica. Temos que esperar que ele volte de férias”. Portanto, o efeito *endowment* é uma das explicações para a prática de código coletivo, assim como a programação em par com rotação frequente, uma vez que evitam o surgimento da individualização de partes do código.

4.9 PRINCÍPIO DA PRIORIDADE RELATIVA

O Relatório do Caos foi uma das razões que trouxe à tona a necessidade de uma mudança de pensamento em relação à forma com que o mundo desenvolvia software. Um número publicado que chamou atenção foi a quantidade de funcionalidades entregues e nunca utilizadas pelos usuários, assim como o fato da não entrega do escopo total ser um dos grandes motivos para o fracasso de diversos projetos. Isso significa que projetos falhavam por não entregar o escopo total, enquanto grande parte do escopo entregue era desnecessária.

Uma possível solução para esses dois problemas é uma melhor priorização do escopo implementado, para que a parte mais importante e necessária do sistema seja implementada e disponibilizada para os usuários o quanto antes.

Priorizar o escopo é uma habilidade extremamente difícil e se encaixa perfeitamente no conceito de “tomada de decisão”, uma das áreas estudadas e conhecidas pelos pesquisadores de Psicologia cognitiva. Em um determinado momento de desenvolvimento de um software, membros da equipe precisam decidir a ordem em que as funcionalidades serão desenvolvidas, ou seja, a prioridade da lista, conhecida como *backlog*. Nesse momento, entra em ação a importante conceituação de dois tipos de prioridade: absoluta e relativa.

Antes de entrar em detalhes quanto aos modelos de priorização e de falar sobre as suas vantagens e desvantagens, passarei por mais um experimento

científico que isolou variáveis para entender como o cérebro humano funciona quando o indivíduo tem que tomar alguma decisão que envolve valores absolutos e relativos. O experimento sugere uma situação hipotética em que alguém precisa comprar uma caneta e sabe que em uma loja próxima o preço do objeto é de R\$6,00 (seis reais) e em uma loja um pouco mais longe, não tanto, a mesma caneta custa R\$2,00 (dois reais). Nessa situação, a pergunta é: você iria até a loja mais longe para pagar menos pela caneta? A maioria dos participantes diz que sim, que vale a pena ir até a loja mais distante para economizar. Quando a pergunta é um pouco diferente, a resposta é outra. Se, em vez de uma caneta, o item a ser comprado for um terno de R\$455,00 (quatrocentos e cinquenta e cinco reais) na loja mais próxima e R\$451,00 (quatrocentos e cinquenta e um reais) na loja mais distante, a maioria dos participantes diz que não iria até a segunda loja para fazer a economia.

		
Loja Perto	R\$ 6	R\$ 455
Loja Longe	R\$ 2	R\$ 451
Ganho Absoluto	R\$ 4	R\$ 4
Ganho Relativo	3 vezes	1,008 vezes

Fig. 4.8: Prioridade relativa

Como se pode observar, o ganho absoluto, nas duas situações, é o mesmo: R\$4,00 (quatro reais). Entretanto, o comportamento e a decisão tomada pelos participantes são opostos. Essa diferença é explicada pelo fato de que pensamos de forma relativa, e não absoluta. O ganho relativo da caneta é bem maior, três vezes mais barata na loja distante. O ganho relativo do terno é muito baixo, quatro reais a menos em um valor de R\$455,00 (1.008 vezes), o que parece não valer a pena. Chega-se, mais uma vez, à conclusão de que a relatividade é algo presente no nosso cérebro. Se Albert Einstein tivesse sido psicólogo, talvez essa fosse a sua Lei da Relatividade, e não a da Física. Voltando à priorização de itens de um backlog, serão definidos os dois tipos de priorização: absoluta e relativa.

- **Prioridade absoluta:** cada item é priorizado individualmente, sem levar em consideração o restante do backlog. A pergunta que caracteriza esse tipo de priorização é algo do tipo: “Este item é importante?”
- **Prioridade Relativa:** os itens são priorizados relativamente uns aos outros. As decisões de priorização são tomadas levando em consideração os itens em volta. Um item só obtém prioridade maior ou menor que outro, nunca individualmente.

Na priorização absoluta, em frente à pergunta “Este item é importante?”, fica difícil para o tomador de decisão imaginar as outras funcionalidades que são mais ou menos importantes do que a em questão, e ele acaba falando “sim”. Assim, o resultado é a alta importância de itens que deveriam ficar para o final ou até mesmo nunca ser implementados. Estes acabam sendo priorizados, implementados, entregues e nunca utilizados, como mostrou o relatório do caos. Dadas as características do nosso cérebro, a prioridade relativa se mostra mais eficaz para que itens menos importantes sejam deixados para o final e até mesmo não sejam implementados.

Geralmente, equipes ágeis mantêm um backlog pequeno, priorizado de forma relativa, revisado com frequência, utilizando o Princípio da prioridade relativa, que é definido como:

- Todas as decisões a respeito da prioridade de itens ou funcionalidades devem levar em consideração outros itens que podem ter prioridade

menor ou maior que o item em questão. Não existe prioridade absoluta, somente relativa a outros itens.

4.10 UM TACO, UMA BOLA, UM TESTE

Pense rápido e tente responder a pergunta a seguir:

- Você tem um taco e uma bola.
- A bola e o taco, juntos, custam R\$1,10 (um real e dez centavos).
- O taco custa um real a mais do que a bola.
- Quanto custa a bola?

Se a primeira resposta que veio a sua cabeça foi:

- A bola custa R\$ 0,10 (dez centavos)

Pense um pouco mais. Se a bola fosse mesmo R\$0,10, o taco, que custa um real mais caro, seria R\$1,10. A soma dos dois seria R\$1,20. Sabe-se, porém, que a soma dos dois é, na verdade, R\$1,10. Isso significa que a resposta certa para o valor da bola é R\$0,05, concorda?

Não se sinta mal por pensar assim, pois metade dos estudantes de algumas das melhores universidades do mundo (Harvard, Princeton e MIT) também cometeu esse erro. Na verdade, isso acontece quando pensamos rápido. Daniel Kahneman explica isso no livro *Rápido e devagar Duas formas de pensar* (*Thinking, Fast and Slow*), explicitando que temos dois agentes no nosso cérebro: um que toma decisões rápidas, porém não muito precisas, e outro mais lento, entretanto mais correto. Os dois lados são necessários. Imagine se você precisasse pensar lentamente todas as vezes que fizesse algo como andar, escovar os dentes, dirigir. Assim, precisa-se dos dois mundos do pensamento, o devagar e o rápido, para que o dia a dia seja mais eficiente. É importante saber escolher qual usar, dependendo da ocasião e do tipo de atividade que se pretende realizar.

Escrever testes automatizados é uma forma utilizada por desenvolvedores para validar se o código escrito, muitas vezes rapidamente, resolve mesmo o

problema proposto. O simples teste automatizado a seguir, escrito em uma pseudo-linguagem baseada em um framework de testes conhecido, resolveria o problema de se achar que a bola custa R\$0,10.

```
taco.preco.deve_ser == bola.preco + 1,00
(taco.preco + bola.preco).deve_ser == 1,10
```

No momento em que o desenvolvedor escrevesse a solução do problema como a bola custando R\$ 0,10, esse teste falharia, forçando-o a pensar um pouco mais. O custo do erro é imensamente menor se ele for encontrado mais cedo, no exemplo, no momento da implementação, em vez de ser diagnosticado em uma fase de teste tardia, por outra pessoa.

A existência desses dois agentes no cérebro, um rápido, que comete mais erros, e um devagar e mais correto, é um grande justificador para a escrita de testes automatizados no momento que se implementa uma determinada funcionalidade.

4.11 RESUMO DAS CORRELAÇÕES

A tabela a seguir é um resumo das correlações explicadas entre Psicologia cognitiva e alguns valores, princípios e práticas ágeis.

Psicologia Cognitiva	Ágil
Síndrome do estudante	Iterações e entrega contínua
Cegueira não intencional e difusão de responsabilidade	Radiadores de informação, reuniões de pé, retrospectiva (donos)
Efeito endonment	Código coletivo
Escolha relativa	Princípio da prioridade relativa
Rápido e devagar	Testes automatizados e TDD

Fig. 4.9: Resumo das correlações

É claro que essa lista não é finita e que a intenção, aqui, é que você, leitor, pense em outras relações e em outras explicações que possam ajudar no dia a dia.

4.12 A IMPORTÂNCIA DAS CORRELAÇÕES

A melhoria contínua e a adaptação às mudanças são características fundamentais para uma equipe ágil, portanto, entender as raízes e origens comportamentais é de extrema importância para a adaptação e a modificação dessas práticas de acordo com a sua realidade e contexto. Assim, entender as origens é essencial para:

- os membros de uma equipe que queiram melhorar e que estejam interessados em começar utilizando conceitos e práticas ágeis básicas;
- os membros de uma equipe que já vem utilizando o desenvolvimento ágil e que reconhecem a necessidade de adaptação e de melhoria;
- os consultores que ajudam outras equipes a se adaptarem a mudanças e a serem mais ágeis.

Como ficou claro nos experimentos aqui citados, a Psicologia cognitiva explica de uma forma mais científica a importância do desenvolvimento ágil e ajuda a entender o porquê do sucesso, antes praticamente inexplicável, desse movimento que revolucionou a forma como o mundo pensa durante o processo de criação, inovação e desenvolvimento de software.

4.13 PSICÓLOGO POR PAIXÃO

Tudo o que escrevi aqui é um resumo de anos de leitura e discussões com amigos e colegas. Na área de psicologia, eu ainda sou um amador. Esse ainda não é um trabalho “acadêmico”, são minhas próprias conclusões baseadas em outros trabalhos acadêmicos. Assim, meu conselho é: converse, discuta, aprenda e melhore... sempre!

CAPÍTULO 5

Queremos inovar!

por Francisco Trindade

A sentença do título parece ser o mantra mais repetido em empresas de tecnologia, atualmente. Por trás de muitas histórias de sucesso e um grande número de livros e materiais sobre o assunto, a inovação virou o assunto principal no mercado de TI em poucos anos. Vendo o exemplo de pequenas *startups* que estão irrompendo mercados, companhias já estabelecidas querem defender sua liderança, sendo elas mesmas inovadoras. Hoje em dia, aqueles que não inovam (ou nem tentam) estão ficando para trás.

Entretanto, como tudo na vida, criar uma empresa inovadora é mais fácil no discurso do que na prática. Com todas as notícias sobre pequenas equipes criando novos e incríveis produtos, é fácil esquecer todos os desafios a serem superados e acreditar que a inovação é fácil de ser atingida.

Se uma pequena empresa com poucos recursos pode resolver o quebra-cabeça da inovação, por que uma grande corporação, com um grande leque de clientes, pessoas e recursos, não poderia fazê-lo?

Este artigo irá examinar por que companhias já estabelecidas têm dificuldades ao tentar criar novos produtos, e comparar o que elas fazem com o que acontece em uma pequena startup. Também oferecerá exemplos de dificuldades que as empresas enfrentam quando lidam com a inovação e ideias de como superá-las.

5.1 UMA HISTÓRIA DA (FALTA DE) INOVAÇÃO

Este é um resumo do que vejo acontecendo no trabalho como consultor de empresas de tecnologia nos últimos anos. Apesar de os eventos descritos aqui não serem reais, tenho certeza de que, se você trabalha para uma empresa dessa área, vai se identificar.

Tudo começa com uma ideia. No caso de empresas maiores, ideias não apenas surgem do nada, mas são normalmente obtidas por meio de uma grande influência (alguém no alto escalão) ou por métodos mais formais, como pesquisa de mercado e pesquisa de feedback do cliente. Elas também não são raras, já que muitas organizações têm um excesso de ideias sobre produtos e serviços e falta de dinheiro e tempo para executá-las, então apenas algumas poucas selecionadas passam no filtro e vão para frente.

Acme é uma empresa imaginária para mera ilustração. E, nela, a história não foi diferente. Depois de um longo período de análises e discussões, uma proposta chegou ao final e foi ouvida e aprovada por todos que precisavam aceitá-la.

Como acontece em muitas empresas, a Acme estava preocupada com o sucesso de seus novos produtos. Eles geralmente levavam muito tempo e perdiam mercado, deixando as pessoas preocupadas com que isso acontecesse com a nova iniciativa. Contudo, com o sucesso do movimento de startups e o fato de que alguns concorrentes da Acme eram pequenas empresas, a inovação e os processos de *Lean Startup* foram todos estudados e conhecidos. A intenção era ter o novo produto operando como uma startup inovadora.

O primeiro passo para tirar essa iniciativa do papel era montar uma

equipe pequena, altamente capaz e multidisciplinar para trabalhar nela. O time era composto de poucos desenvolvedores, um analista de qualidade e um gerente de produto. Como a maioria das pessoas diria, pequenas equipes têm melhores desempenhos nestas situações, e a Acme queria tirar vantagem disso.

Um workshop foi organizado para começar. Todos os membros da equipe e *stakeholders* interessados passaram alguns dias juntos conversando sobre a iniciativa e definindo qual era o objetivo do projeto. Pesquisadores de mercado apresentaram seus *insights* sobre a mente dos consumidores e técnicos definiram como essa visão poderia ser obtida por meio de software. Com muito trabalho e negociação, o resultado obtido foi uma visão para um primeiro lançamento do produto, que poderia levar três meses para ser desenvolvido.

Nas primeiras semanas de trabalho, algumas questões surgiram com relação às necessidades do consumidor e ao modelo de negócio para o produto, mas, já que não havia clientes disponíveis ainda, foi difícil cortar elementos desnecessários do escopo.

Algumas semanas se passaram e, em um esforço para colocar o produto no mercado o mais rápido possível, o time começou a falar com stakeholders sobre a ideia de criar uma versão mínima do produto e colocá-lo no ar para verificar o engajamento do consumidor com a ideia do negócio. Depois de muita discussão, a Acme decidiu que era muito arriscado colocar um produto não aprovado internamente no mercado, então foram organizados testes de usuário com protótipos do mesmo.

Mesmo assim, a equipe estava trabalhando muito bem, entregando funcionalidade mais rápido do que acontecia normalmente dentro da empresa. Por causa disso, o projeto atraiu o interesse de outros gerentes que começaram a aparecer para demonstrações e fornecer ideias de quais outras funcionalidades o produto deveria ter. Com mais pessoas interessadas e mais exposição interna, ficava cada vez mais difícil para a equipe administrar o aumento do escopo.

O tempo passou e o prazo de três meses chegou, mas nenhum software estava perto de ser lançado. Com um escopo extra e mais pressão, as sessões de testes de usuário e o feedback do cliente estavam se tornando menos e me-

nos relevantes para o trabalho diário. O foco principal para todos era entregar o projeto conforme combinado.

Seis meses depois, a primeira versão do software estava finalmente no mercado. Seis meses para a entrega de um produto era ainda um grande feito para a Acme, de modo que o projeto foi considerado uma entrega bem-sucedida. Contudo, com uma quantidade excessiva de recursos e sem ser direcionado para um segmento específico de clientes, o produto fracassou.

5.2 MUDAR É NECESSÁRIO

Uma versão da história descrita anteriormente acontece todo dia em uma empresa de TI. Novas e incríveis ideias são propostas, recursos são investidos para entregar o produto da melhor maneira possível e, na maioria das vezes, ainda assim não dá certo.

Para entender por que isso continua acontecendo, precisamos voltar um pouco e ver por que a inovação é tão comum hoje em dia.

Já que imagens valem mais que mil palavras, não há maneira melhor de descrever a razão por trás da atual corrida pela inovação do que o quadro a seguir.

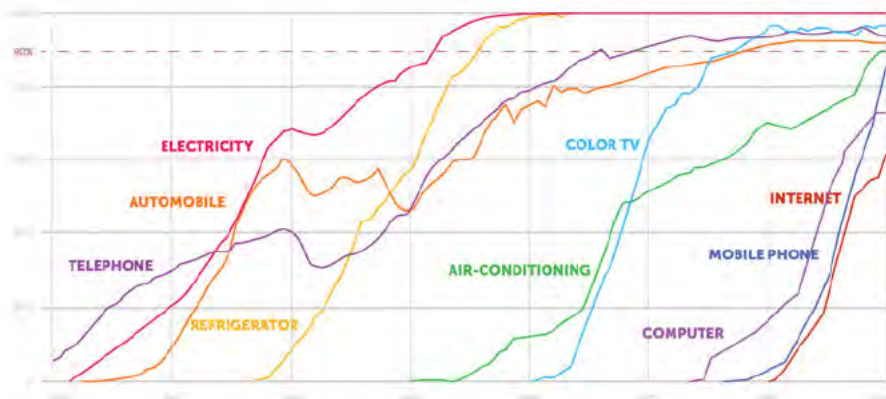


Fig. 5.1: Quadro Adoção de tecnologia

Ele mostra o período que leva para novas tecnologias serem adotadas ao

longo do tempo. É fácil notar que o intervalo entre a tecnologia ser criada até o uso em massa está diminuindo significativamente a cada nova onda, e que será reduzido ainda mais no futuro.

Não é só tecnologia. Com novas capacidades, comportamentos também estão mudando e novas necessidades e oportunidades estão surgindo. Como Gary Hamel afirma, “a mudança mudou” e tudo está sendo constantemente renovado em um ritmo muito mais rápido, portanto as empresas pre-cisam se adaptar.

Este fenômeno foi facilmente percebido e chamou a atenção do público com o sucesso do movimento startup. Empresas que começam pequenas estão tomando a frente, abalando antigos mercados e roubando grandes fatias de mercado de empresas existentes (por exemplo, Skype para chamadas telefônicas, Whatsapp para SMS), ou criando outros completamente novos (re-des sociais) nos quais é possível crescer rapidamente sem concorrência. À medida que se torna claro que a mudança veio para ficar, grandes empresas também querem participar e se manter à frente do mercado através de inovação.

Contudo, o aumento da velocidade da mudança não apenas cria novos mercados e oportunidades, mas também dificulta entender o que vai acontecer em seguida. Enquanto empresas consolidadas estão se mantendo tentando entregar uma versão melhor de seu produto existente, o mercado está mudando rapidamente, trazendo consumidores com novas necessidades que são atendidos por novos concorrentes. O desafio não é apenas entregar software mais rápido do que antes, mas sim conduzir uma abordagem completamente nova, necessária para se obter sucesso.

5.3 CONHEÇA A LEAN STARTUP

Se o objetivo é agir como empresas startup, que maneira melhor para replicá-las, senão observando-as? Isso é o processo de Lean Startup. Primeiramente descrito no livro homônimo, o processo combina ideias do sistema de produção enxuta com técnicas de desenvolvimento de clientes para criar um processo de inovação repetível. Em alto nível, o processo pode ser descrito como na figura a seguir.

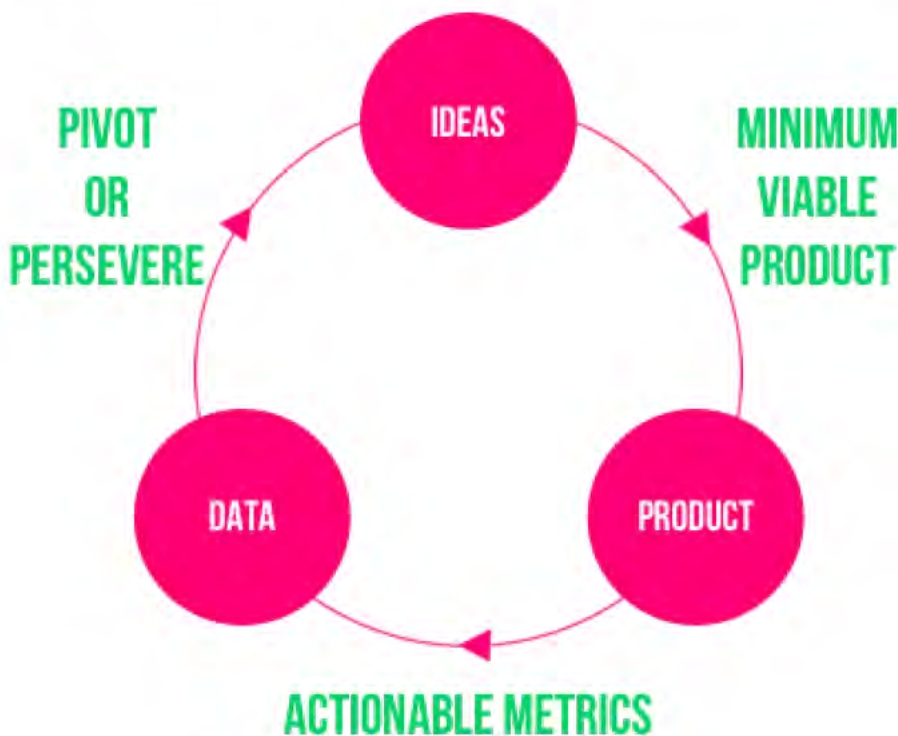


Fig. 5.2: Visão do processo de Lean Startup

Neste caso, o primeiro passo para criar um produto é refiná-lo até sua ideia central e construir um *produto minimamente viável*, que é uma versão-produto dessa ideia. Isso pode ser testado no mercado, juntando métricas que proporcionarão conhecimentos de como melhorá-lo. Com os dados coletados, uma decisão pode ser tomada no sentido de mudar o produto e/ou modelo de negócio (mudar) ou continuar a repetir a mesma ideia com versões mais refinadas do produto (preservar).

O ciclo é repetido várias vezes, usando feedback do mercado real para guiar as decisões relacionadas ao produto. Não é uma ideia nova lançar um produto e obter feedback sobre ele, mas é diferente quando o ciclo leva semanas e não meses, e quando o feedback já pode ser usado nos estágios iniciais de seu desenvolvimento.

5.4 ENTENDENDO AS DIFICULDADES

Enquanto ter um processo repetível que pode ser descrito e compartilhando por equipes torna as coisas muito mais fáceis, o desafio real não vem da compreensão da teoria. Como descrito na história da Acme no começo do artigo, a maioria das empresas já leram o livro e compreenderam como funciona, mas ainda lutam para executar o que está descrito nele. Por que isso acontece? Há uma necessidade de inovar e há uma vontade de fazê-lo. Toda empresa apoia a ideia de criar novos produtos mais rapidamente e obter feedback frequente de clientes. Na teoria, é a conjunção perfeita. Então, por que é tão difícil acertar ao inovar? Já que a maioria das empresas olha para o processo de Lean Startup e para as startups de tecnologia para se inspirar em como inovar, nós poderíamos simplificar o problema ao nos aprofundarmos e compararmos o que acontece em empresas consolidadas com o dia a dia de uma startup bem-sucedida. Se fizermos isso, veremos algumas diferenças:

5.4.1 Inovação não é só entregar software

Desenvolvimento de software é um trabalho duro. Entregar software de boa qualidade de forma confiável é um processo que requer muito investimento e muitas empresas têm uma maneira já consolidada de executar projetos para que o entreguem com sucesso.

Enquanto as práticas para criação de software de qualidade são essenciais para qualquer projeto, uma grande parte de como as empresas executam grandes projetos de software não é apropriada para empresas que estão começando, nas quais esse não é o principal desafio a ser superado. Como define Steve Blank, uma startup é uma organização em busca de um modelo de negócios escalável, e projetos de inovação deveriam ser tratados da mesma maneira. Não é apenas entregar software da mais alta qualidade, na maioria das vezes não é preciso nem entregar software! Contudo, devido aos times internos de inovação serem, em sua maioria, formados por desenvolvedores de software, é fácil cair na armadilha de apenas começar a escrever código sem considerar se ele vai ser realmente útil para qualquer cliente.

5.4.2 Sucesso não é apenas fazer o que você já faz bem

Não há dúvida de que empresas bem-sucedidas são boas em fazer, no mínimo, uma coisa bem. Elas trabalharam duro, ajudaram clientes e ganharam sua fatia no mercado ao serem excelentes no que fazem. Porém, tudo muda rapidamente e os líderes de mercado estão desaparecendo porque o que eles faziam não tem valor para novos clientes. Se as empresas querem ficar à frente, precisam deixar de apenas tomar o caminho mais fácil e resolver todos os problemas com as mesmas soluções de sempre.

Sucesso em inovação significa adotar um novo pensamento, ver o mundo com novos olhos e perceber novas oportunidades, mesmo competindo diretamente com produtos que a empresa já tem. O único caminho para não ter seu mercado capturado por novos competidores eventualmente é se assegurar de que sua empresa é a que está à frente da inovação.

5.4.3 Uma pessoa com uma ideia, a unidade de inovação

Com a chegada dos livros e processos sobre startups e como administrá-las, é fácil acreditar que começar um novo empreendimento é apenas seguir um processo rigoroso, e a maioria das empresas minimiza a importância das pessoas envolvidas em seu sucesso.

Startups trabalhando em novos produtos estão constantemente investigando e testando o mercado. Toda a informação reunida é essencial para entender como encontrar onde o produto se enquadra entre os consumidores. Da linguagem corporal mostrada por um cliente em uma entrevista aos dados do funil de conversão do produto, tudo será usado para tomar decisões que possam definir o sucesso da empresa. Por causa disso, ter um grupo central de pessoas que trabalha consistentemente na ideia e é capaz de conduzir essa informação é uma parte importante de ser capaz de encontrar um nicho que pode ser bem-sucedido.

Esse aspecto é frequentemente negligenciado por empresas já estabelecidas tentando inovar, já que estão acostumadas a definir processos e requerimentos que são repassados a diferentes grupos de pessoas. É comum, neste cenário, empresas separarem a pessoa que teve a ideia original da equipe que

vai executá-la, removendo a paixão pelo produto e adicionando uma grande quantidade de risco a todo o empreendimento.

Para criar condições melhores para a inovação, empresas têm que reconhecer o papel do empreendedor interno, o qual pode ter a ideia e ser a pessoa que a guia, sendo assistido por outros em seus diferentes estágios.

5.4.4 A regra é fracassar

A maioria das startups não dá certo. Às vezes, é difícil entender esse fato quando todas as notícias falam sobre histórias de sucesso, mesmo que as que foram bem-sucedidas sejam uma gota no oceano de empreendimentos que tentaram e falharam.

Isso é um grande contraste com a atual cultura empresarial, na qual todo projeto tem que ser bem-sucedido para ser considerado válido e para as pessoas envolvidas nele serem respeitadas. Se as empresas querem se tornar competentes em inovação, elas precisam entender que a maior parte das ideias não vai dar certo e que a única maneira de obter sucesso no longo prazo é aprender como testá-las com mínimo investimento.

Isso é de suma importância quando for avaliar projetos e também pessoas. O aprendizado é o resultado da maioria das atividades de inovação, por isso as empresas devem levar isso em consideração em suas avaliações, e não apenas resultados financeiros.

5.5 QUEBRANDO A UNIFORMIDADE

“E por muitos anos, ele não permitiu que nada fosse registrado sobre isso. Ele alegou que a melhoria é um processo interminável - e ao escrever sobre ela o processo ficaria cristalizado.”

– Ohno, T 1988 ‘Toyota Production System’ (prefácio)

Até agora, vimos os desafios em ser inovador em uma grande empresa e por que a iniciativa não funciona como esperado. Como pode ser feito, então? Não há respostas definitivas para essa pergunta, mas vamos focar em alguns passos que podem ser usados para atingir este objetivo.

Para criar uma cultura inovadora e se comportar como empresas de tecnologia inovadoras, organizações estabelecidas precisam sair da sua zona de conforto e entender melhor o que realmente são startups de sucesso. Isso não significa apenas a maneira como elas usam processos (Lean Startup e metodologias similares), mas a forma como pensam, tomam decisões e aproveitam oportunidades.

Inovação constante não acontecerá até que as empresas olhem fundo em seus valores e princípios e criem um ambiente próprio para experimentação, como acontece no mundo real. Para fazer isso, elas precisam aprender como compartilhar seu poder de decisão, se arriscar mais e adotar ideias diferentes. Empresas precisam inovar em sua estrutura.

5.6 AS FASES DA INOVAÇÃO

Enquanto a inovação não é um processo linear, para facilitar o entendimento desse artigo nós vamos analisar opções sobre como implementá-la em três diferentes fases: geração de insights, formando ideias e executando modelos de negócios.

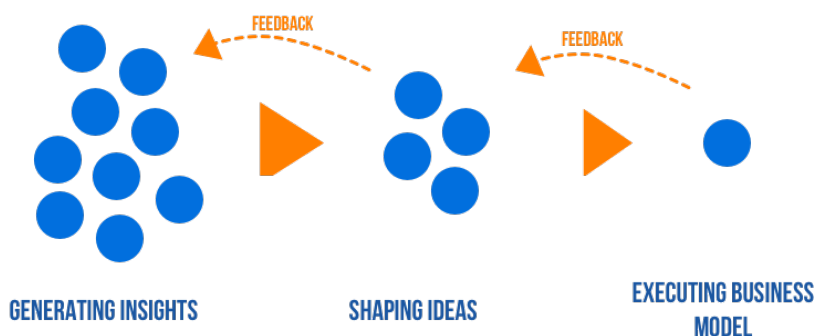


Fig. 5.3: Fases da inovação

Estas fases definem uma progressão de ideias para um negócio em ope-

ração, mas vale a pena apontar que se deve esperar que a maioria das ideias fracassem, e entender o feedback obtido nestes fracassos como o resultado mais valioso.

5.6.1 Geração de insights: como fazer as ideias fluírem?

Para criar uma oportunidade de negócio, alguém tem que ter uma ideia primeiro. A única opção para uma empresa estar constantemente à frente da tendência é obter todas as ideias disponíveis.

Isso parece mais difícil do que realmente é. A criatividade já existe e todas as empresas têm um ecossistema criado em torno delas, com empregados, clientes e outros *stakeholders*. Não há dúvida de que qualquer consumidor tem ideias e opiniões sobre como deixar o produto ou o serviço melhor e, da mesma maneira, todo empregado já pensou em novas maneiras de entregar valor para o cliente.

Além disso, há algumas coisas que podem ser feitas para manter as pessoas curiosas e as ideias fluindo. Todas surgem a partir da mesma premissa: as pessoas precisam de tempo para pensar e serem expostas a novas perspectivas para serem criativas.

Atividades como:

- workshops de *Design Thinking*, onde se imagina como solucionar um problema sob o ponto de vista do consumidor;
- passar tempo com consumidores, observando o que eles fazem e desafios que enfrentam no dia a dia;
- trocar de função com diferentes empregados para se ter uma perspectiva nova sobre os produtos da empresa;
- tornar disponíveis pesquisas de mercado realizadas pela empresa, para que todos tenham acesso à opinião do mercado sobre os produtos oferecidos.

Esses são todos exemplos de compartilhamento de diferentes perspectivas que levarão a novas ideias que podem ser de grande valor.

Contudo, o desafio não acaba aqui. Muitas pessoas, em qualquer empresa, já tiveram várias ideias mas normalmente não ficam confortáveis em divulgá-las. Na maioria das organizações, funcionários não teriam como transformar suas ideias em algo concreto. Além disso, a voz do consumidor (pesquisas de mercado, ligações para a central de suporte) normalmente para no *call center* ou no departamento de marketing sem ser aproveitada com eficiência. Reunir *insights* é mais do que apenas fazer uso da famosa caixa de sugestões.

Para utilizar esse capital intelectual, as empresas precisam tornar fácil para as pessoas explorarem o que têm em mente, de maneira segura. Além de sentirem que a organização está aberta para isso, os funcionários precisam de uma maneira para testar suas ideias e aprender com elas.

5.6.2 Modelando ideias: transformando pensamentos aleatórios em modelos de negócios

Quando ideias surgem, é fácil cair na armadilha de estar muito ocupado com o dia a dia dos negócios, em que ninguém tem tempo de explorar novas oportunidades porque todos estão ocupados fazendo o negócio andar. As empresas precisam de maneiras de tornar ideias incipientes em modelos de negócio viáveis, e pessoas com ideias precisam de maneiras de testá-las rapidamente sem tomar muito de seu tempo pessoal. Mas como fazer isso sem investir muito dinheiro em cada ideia?

Felizmente, isso não é um problema novo. Empreendedores e investidores o enfrentam todo dia e a indústria já estabeleceu alguns padrões para ajudar nessa situação que exploraremos a seguir.

- **Eventos de inovação:** uma maneira comum de auxiliar a dar forma às ideias é a criação de eventos de inovação, nos quais as pessoas podem dedicar um ou dois dias de trabalho a elas, obter feedback e transformá-las em algo viável. Há diferentes estilos de eventos que acontecem na indústria nessa área, como *Startup Weekends* e *Lean Startup Machine Workshops*. Este é um modelo que já foi testado na indústria em várias empresas, como na ShipIt, pela Atlassian, uma empresa que vende software para gerenciamento de projetos, e nos Hack Days, pelo REA Group, um grupo imobiliário da Austrália.

Enquanto essas iniciativas requerem um grande investimento quando feitas na empresa durante o horário de trabalho, há maneiras de começá-las em escala menor. Grupos focados com alguns funcionários interessados na ideia ou até mesmo eventos de fim de semana para testá-las são uma maneira fácil de experimentação.

- **Crowd-funding:** usar as massas para tomar decisões tornou-se algo comum nos últimos anos e tem sido bem documentado. Com a criação de sites de crowd-funding, como o Kickstarter, Pozible e muitos outros, isso também se tornou uma prática comum para conseguir fundos no mundo das startups.

Enquanto uma empresa de TI não tem milhões de pessoas para investir dinheiro em ideias, há normalmente um bom número de funcionários, clientes e interessados que podem votar e dar um feedback no que está sendo proposto. Iniciativas como esta estão começando a ser divulgadas por empresas interessadas em inovação e, enquanto o crowd-funding interno não funciona exatamente como o externo, é uma boa maneira de começar a ter feedback, especialmente se envolve consumidores.

- **Financiamento baseado em influência:** se você confia em seus funcionários para tomar decisões por sua empresa, por que não criar um sistema em que eles são donos de seu próprio tempo?

Do lado mais convencional, algumas empresas estão experimentando programas nos quais ideias são financiadas com tempo e dinheiro, e baseados em quantas pessoas as apoiam. Por exemplo, se eu sou um programador com uma ideia de negócio, eu tiraria um dia de trabalho para explorá-la, mas neste caso eu poderia fazer com que uma pessoa de vendas se juntasse a mim, e nós teríamos três dias para fazer o mesmo. Essa é uma forma barata de assegurar que seus funcionários tenham tempo de explorar novas oportunidades, enquanto usam o próprio ambiente interno da empresa para decidir o que pode ir para frente ou não.

No outro extremo, isso está se tornando um modo não tão raro de operação para empresas, com alguns exemplos de organizações, como a

Githube a Treehouse, que usam um modelo como esse como sua principal forma de trabalho, sem projeto ou estruturas formais e com funcionários capazes de decidir em que investirão seu tempo para alcançar os principais objetivos da empresa.

Esses modelos podem levar uma ideia adiante, mas é importante entender o que elas significam: dar uma chance para uma iniciativa que é desenvolvida dentro da empresa, sem os padrões e políticas de aprovação usuais. Tanto quanto possível, empresas devem evitar julgar as ideias até que seus idealizadores as coloquem em contato com clientes no mercado e obtenham feedback diretamente deles.

5.6.3 Executando modelos de negócios: transformando ideias em atividades lucrativas

Os formatos para ideias também são limitados na escala de investimentos que podem fornecer. Dias de inovação não duram muito e iniciativas de crowdfunding interno apenas mostrarão o que as pessoas veem como a melhor oportunidade. Elas ajudarão os funcionários a conseguir o feedback sobre o que estão pensando, ajudarão ideias fracas a sumirem rapidamente e ajudarão a empresa a selecionar quais são válidas para se investir. Mas ainda é importante entender o que fazer a seguir.

Com uma primeira seleção do que é válido investir, é hora de pegar as ideias e tentar implementá-las no mercado. Contudo, como em qualquer iniciativa em seu estágio inicial, a taxa de falha é ainda extremamente alta, então não seria aconselhável selecionar algumas delas e investir um orçamento de seis meses de operação em cada uma.

Ao olhar em volta para se inspirar, a oportunidade que um empreendedor com uma ideia válida teria é de participar de um programa de aceleração, que ajudaria a focar na execução e tentar provar um modelo de negócio viável. Há múltiplos exemplos de programas aceleradores bem-sucedidos que funcionam há anos na indústria de TI, como o YCombinator e o 500 Startups, e o prazo deles é normalmente de seis semanas, do começo ao fim.

O interessante sobre a fase de execução é que programas aceleradores internos podem, na teoria, ser mais bem-sucedidos que os externos. Enquanto

programas públicos têm iniciativas que abrangem diferentes indústrias, as iniciativas internas dentro de uma empresa normalmente ficarão em torno da atual indústria, na qual a companhia é bem-sucedida e isso pode trazer uma série de vantagens.

Contudo, há alguns pontos para se ter em mente quando executar um programa acelerador interno:

- **Ideias e empreendedores progridem juntos:** para evitar as interrupções comuns do negócio, empresas geralmente separam as ideias das pessoas que estão trabalhando com elas, alocando diferentes equipes para trabalhar no projeto ao longo do tempo. Enquanto isso é um mau hábito em qualquer fase de um produto de software, é particularmente prejudicial quando se tenta novas oportunidades de negócios. A maioria das informações neste estágio não é documentada em nenhum lugar, a não ser na memória das pessoas envolvidas. Além disso, a determinação de alguém com uma ideia não é facilmente substituída por nenhum tipo de processo. Se existe apenas uma mudança que as empresas podem fazer para melhorar a inovação é deixar as pessoas seguirem em frente com seus empreendimentos internos.
- **Acesso a clientes:** enquanto as startups em seu início lutarão muito para ter uma boa amostra de clientes para obter feedback deles, empresas de TI já estabelecidas têm um maior acesso a informações. Contudo, companhias usualmente evitam expor seus clientes a novas ideias, mesmo quando há muitos métodos já estabelecidos para fazer isso de maneira segura. Para fazer a inovação funcionar, o feedback do cliente é a coisa mais importante, então assegure-se de usar tudo o que tiver disponível.
- **Independência do negócio:** não importa quão grande é uma empresa, políticas internas sempre são fatores importantes na decisão de onde investir, e há sempre muito *status quo* a ser perdido quando produtos inovadores são bem-sucedidos. Enquanto o feedback é o resultado mais importante no processo de inovação, para dar a ideias inovadoras uma chance maior de sucesso, empresas têm que se proteger de críticas internas tanto quanto possível. Além disso, começar e testar um novo

negócio é muito diferente de desenvolver software. Enquanto as empresas de TI são normalmente bem equipadas em termos de sua capacidade de desenvolvimento, este estágio é aquele em que a organização interessada em criar uma cultura de inovação deve oferecer assistência em orientação e métodos que ajudem suas ideias a serem testadas no mercado.

5.7 REFLEXÕES

Inovar não é uma tarefa fácil. Milhares de startups vão à falência todo ano, e é fácil esquecer disso quando vemos apenas casos de sucesso no noticiário, o que nos leva a acreditar que cada ideia que uma empresa tem internamente deve funcionar.

Para criar uma verdadeira cultura inovadora, existe a necessidade de uma mudança de pensamento dentro da empresa. É uma tarefa grande, mas pode começar com pequenos experimentos que permitam à organização tentar diferentes alternativas para inovar sem corromper seu negócio principal.

Fundamentalmente, organizações precisam começar a deixar seus funcionários seguirem suas ideias e aspirações, criando um ambiente em que testes e execuções dessas ideias possam ser uma experiência gratificante para eles e viável para a empresa, mesmo quando não forem bem-sucedidas. Só assim as melhores ideias aparecerão continuamente, fazendo com que uma empresa fique sempre à frente no mercado.

CAPÍTULO 6

Mirebalais Melhorando a sociedade com tecnologia

por Alexandre Klaser, Glauber Ramos, Natalia Arsand e Mário Areias

Nossa missão é melhorar a humanidade por meio do software e ajudar na criação de um ecossistema socialmente responsável e economicamente justo. O impacto do software na sociedade pode (e deve!) ir muito além de redes sociais, e-commerce, ou aplicativos para celular.

Em todo o mundo, há diversos indivíduos e organizações humanitárias e não-governamentais comprometidas com esses mesmos ideais, com os quais devemos estabelecer parcerias para levar adiante nossa missão. Uma dessas organizações é a *Partners in Health* (PIH), uma ONG fundada em 1987 pelo

Dr. Paul Farmer, Jim Yong Kim e Ophelia Dahl, entre outros. O objetivo deles é fornecer um serviço preferencial de saúde aos menos privilegiados, como os residentes da região montanhosa do Platô Central, no Haiti.

Neste capítulo, contamos uma história real de como o software e o uso de técnicas e metodologias modernas ajudaram no grande trabalho de salvar vidas realizado pela PIH, servindo de exemplo e inspiração para o mundo da Tecnologia da Informação.



Fig. 6.1: Crianças do Haiti na vizinhança do hospital Mirebalais

6.1 CONTEXTO

Nossas primeiras interações com a PIH foram pequenas contribuições feitas no OpenMRS – um software livre de Prontuário Eletrônico de Paciente (PEP) –, quando escrevemos alguns módulos para o sistema e ajudamos com a documentação do produto.

A PIH, nos 25 anos desde sua fundação, expandiu sua atuação para outras regiões do Haiti e também lançou projetos adicionais ao redor do mundo. Uma destas expansões planejadas foi a de um hospital universitário em Mi-

rebalais, no Platô Central do Haiti. O grande terremoto que atingiu o país em 2010, comprometendo grande parte das instalações médicas na capital, Port-au-Prince, fez deste hospital não apenas um sonho ambicioso, mas também uma resposta urgente para preencher um enorme vazio.

Em março de 2013, o Hospital Universitário de Mirebalais (HUM) foi inaugurado, ocupando uma área de 23.000m² e disponibilizando 300 leitos. Construído pela PIH em cooperação com o Ministério da Saúde do Haiti, este hospital serve também como uma base de treinamento para os futuros médicos do Haiti e é um catalizador para o crescimento econômico da região.



Fig. 6.2: O hospital universitário Mirebalais

Nas palavras do *Chief Strategist* da Partners in Health, Dr. Paul Farmer, “este hospital é o ponto alto de um sonho que temos há um quarto de século e reitera o nosso compromisso com o país e o povo do Haiti, que é ainda mais forte do que antes do terremoto”.

A ThoughtWorks foi então contemplada com a oportunidade de levar sua colaboração com a PIH mais além, trabalhando com a equipe de Informática Médica da PIH em Boston para construir um sistema completo e integrado, baseado no OpenMRS para a operacionalização do hospital de Mirebalais.

6.2 O FOCO DIFERENCIADO

O desafio de implementar um sistema de prontuário eletrônico era imenso. Tanto para a ThoughtWorks – consultores especializados em software, agora inseridos no domínio hospitalar –, quanto para a PIH, que nunca antes havia trabalhado em uma instalação hospitalar tão grande e ambiciosa.

Um domínio novo para alguns membros da equipe é a realidade em grande parte dos projetos de uma empresa de consultoria. Se levarmos em conta que estávamos lidando com um domínio tão complexo quanto é o de saúde pública e também o fato de se tratar de um hospital no Haiti, um país onde a penetração das tecnologias de informação é bastante pequena, vemos que a informatização de um sistema destes se torna ainda mais desafiadora.

No entanto, o foco diferenciado da ThoughtWorks permitiu que esse obstáculo fosse mais facilmente superado: iniciando com a fase que chamamos de *Inception* (capítulo 1) para unir todos envolvidos no projeto em um workshop que, em uma semana focada, intensa e efetiva, permitiu que se construísse um conhecimento compartilhado da visão do produto e seus objetivos, abordando os possíveis desafios e discutindo soluções (seja do ponto de vista técnico ou do negócio). Ao final, foi montado um *roadmap* de implementação com um produto mínimo viável e uma projeção da direção futura a seguir para que o produto evoluísse de acordo com o cronograma de implementação das unidades dentro do hospital.

A Inception nos permitiu absorver os detalhes do domínio sendo trabalhado e as peculiaridades do contexto de um hospital em uma região do mundo tão carente de recursos, ao mesmo tempo em que demonstrávamos à equipe da PIH as vantagens de se trabalhar com uma visão de produto iterativa e incremental, focando nossa energia nos detalhes que mais importavam no momento, sem cometer o erro de excesso de planejamento e detalhamento exaustivo de todas as funcionalidades necessárias ao longo do ciclo de vida do sistema.

Um dos grandes desafios foi a experiência de usuário (leia os detalhes mais adiante), portanto nossa estratégia de agregar valor ao produto sempre partiu do ponto de vista daqueles que iriam efetivamente utilizar o sistema e tornar possível seu sucesso.

Nossa estratégia de um produto mínimo inicial e entregas contínuas para

agregar funcionalidades ao sistema também foi um dos grandes legados que deixamos para a equipe de Informática Médica da PIH e para a comunidade do OpenMRS em geral, provendo as técnicas e ferramentas necessárias que irão permitir que se continue seguindo uma abordagem enxuta para a evolução do OpenMRS.

6.2.1 Parte técnica

Antes do início do Mirebalais, o OpenMRS já era uma plataforma de sucesso com implantações em diversos países. No entanto, todos os *deployments* realizados eram manuais, dolorosos e lentos. Também era muito comum encontrar diversos erros em produção. Para mudar esse cenário, começamos a elaborar uma estratégia de testes.

Adotamos como estratégia a pirâmide de testes descrita no livro *Succeeding With Agile*. Apesar de usar tecnologias propícias para a criação de testes automatizados (como Java, Spring, Hibernate), o OpenMRS possuía poucos desses testes e, em sua maioria, eram testes de serviço, que, apesar de oferecerem uma boa cobertura, eram custosos e complexos de serem criados. Para resolver esse problema, começamos a usar TDD (*Test-Driven Development*) para criar testes unitários para código novo e para código legado que recebesse algum tipo de melhoria ou manutenção. Também usamos programação em pares para ajudar os desenvolvedores do OpenMRS a se familiarizarem com a utilização do TDD e com a criação dos testes.

Quando começamos a aplicar TDD, que veremos mais no capítulo 15, vários problemas no código começaram a aparecer. Pouca injeção de dependências, muitas chamadas de código estáticos, métodos grandes com muitas responsabilidades, entre outros. Isso fez com que começássemos a aplicar conceitos de Orientação a Objeto e Refactoring para deixar o código legado mais testável. Dessa forma, iniciamos a substituição de alguns antigos padrões de desenvolvimento do OpenMRS por padrões mais novos e ágeis.

Nossos códigos Javascript começaram a se tornar mais complexos, já que estávamos remodelando toda a interface do Mirebalais. Para testar código Javascript, nós implantamos o Jasmine e também criamos testes usando Selenium WebDriver para ter alguns testes de aceitação, principalmente para os caminhos críticos. Depois de alguns meses, a nova estratégia de testes

estava fazendo a diferença. O código estava com uma cobertura muito maior, os testes podiam ser rodados localmente pelo time e o tempo para rodar todos os testes era pequeno.

Como resultado desse trabalho, o número de bugs em produção diminuiu consideravelmente. Começamos a trabalhar em criar um *deploy* automatizado para o Mirebalais. A arquitetura do OpenMRS é modularizada e o Mirebalais nada mais é que apenas uma implementação desse sistema. Criamos alguns módulos novos, mas também usávamos alguns módulos que outros colaboradores criaram. Apesar de ser simples, a tarefa de fazer o *deploy* era manual e sujeita a erros, principalmente os relacionados ao versionamento de dependências.

Nós possuíamos diferentes ambientes para fazer o *deploy* da nossa aplicação. Um ambiente de testes, um ambiente para fazer demos, um ambiente para treinamento e um ambiente para produção. O ambiente de testes era um servidor em Boston que sempre estava atualizado com o último artefato gerado pelo CI. O ambiente de demo era outro servidor em Boston, a partir do qual se podia decidir que versão da aplicação seria implantada. O ambiente de treinamento e produção ficavam no hospital no Haiti. Todos os servidores eram provisionados via *Puppet* e possuíam o mesmo sistema operacional e software instalados.

Para realizarmos o *deploy*, a aplicação era colocada em um pacote *Debian*. Esse pacote era salvo no nosso repositório Debian e classificado como instável. O pacote instável era instalado nos ambientes de testes e demo. Quando recebíamos o OK para a produção, promovíamos o pacote para estável. Só então ele poderia ser instalado nos ambientes de treinamento e produção.

Isso fez com que instalar uma nova versão da nossa aplicação ou criar um novo servidor fosse uma tarefa simples, bem diferente do que era antes, quando o *deploy* era uma tarefa árdua, complexa e que demorava algumas horas para ser executada. Nós também começamos a usar essas práticas na própria comunidade OpenMRS, fazendo com que a plataforma toda se beneficiasse dessas novas práticas.

6.2.2 Produto Mínimo Viável

O Produto Mínimo Viável, ou MVP (do inglês, *Minimum Viable Product*), é a versão mais simples de um produto que pode ser disponibilizada para a validação de um pequeno conjunto de hipóteses sobre o negócio.

Basicamente, o que se quer evitar é o desperdício de tempo, dinheiro e esforço que ocorre quando se constrói um produto que não irá atender às expectativas. Para isso, é preciso identificar quais os objetivos do produto e suas possíveis hipóteses. Uma hipótese leva a um conjunto determinado de ações a serem executadas e que, por se tratar de uma hipótese, devem ser validadas (ou refutadas) o quanto antes e com o menor esforço possível. O MVP ajuda justamente com a validação e aprendizado da forma mais rápida possível.

Ao contrário de produtos criados da forma tradicional, que consiste em um longo período compreendido pela análise dos requisitos e construção do software para só depois ocorrer a validação por parte do usuário final, o MVP determina quais são as funcionalidades mais essenciais para que se tenha o mínimo de software funcional que possa agregar valor para o negócio (produto mínimo), e que possa ser efetivamente utilizado e validado pelo usuário final (produto viável). Esta versão é um produto bem menos elaborado que uma versão “final”, porém nos dá a garantia de só incluir as funcionalidades realmente necessárias para o processo de validação de hipóteses e aprendizagem sobre o negócio.

No caso do sistema sendo desenvolvido para o hospital de Mirebalais, as hipóteses que precisávamos validar estavam relacionadas a como o software seria utilizado e as melhores maneiras de permitir uma captura de dados fácil e intuitiva. Para isso, tínhamos de aprender como os usuários iriam interagir com o sistema. Nosso produto mínimo foi sendo determinado pelas funcionalidades que deveriam estar disponíveis assim que o hospital abrisse para o público. O cronograma de abertura do hospital era agressivo e sabia-se que nem todas as unidades seriam abertas ao mesmo tempo, o que permitiu que focássemos em um conjunto específico de objetivos a serem atingidos inicialmente.

Para determinar nosso MVP, as equipes da ThoughtWorks e PIH se encontraram no Haiti para uma semana de reuniões com usuários que já tinham

experiência com sistemas de PEP (que trabalhavam no Hospital Lacolline) e outros *stakeholders*, como o diretor do novo hospital, o diretor do treinamento da equipe de enfermagem e um médico.

Este período no Haiti foi dedicado a entender quais eram as limitações no PEP utilizado até então no Hospital Lacolline e aprender o máximo possível com os usuários locais, identificando prioridades e problemas. Como resultado, apontamos os objetivos do produto a ser construído, bem como as prioridades que formaram o MVP e nos deram uma projeção das entregas incrementais subsequentes, que guiaram a execução do projeto ao longo dos meses seguintes.

6.2.3 Experiência de usuário

Imagine um contexto em que há pobreza dominante, recursos naturais e tecnológicos escassos, quase nenhum investimento em educação e um sistema de saúde precário e parcialmente destruído por desastres naturais.

Imagine que, em um contexto desses, a porcentagem de pessoas iletradas é extrema, a necessidade de atendimento médico urgente e eficaz é gigantesca e tecnologia é algo considerado de outro mundo.

O nosso maior desafio como designers para o hospital de Mirebalais foi ter que responder a uma pergunta um tanto incomum:

O DESAFIO PARA OS DESIGNERS

Como criar um sistema de prontuário eletrônico que controle e agilize o atendimento de milhares de pessoas em situações de saúde grave, que poderá ser manipulado por pessoas iletradas em inglês e sem nenhum conhecimento em informática?

Quando os designers Glauber Ramos e Natalia Arsand entraram no projeto para trabalhar no design da experiência, não sabiam muito bem como encaixar as práticas de design de forma adequada no time. Apesar de várias referências de Agile e Lean UX, o time tinha uma forma bem peculiar de fazer Ágil e boa parte do sistema já estava pronto, dado que é desenvolvido em cima do OpenMRS.

A abordagem que decidimos seguir foi basicamente ciclos pequenos de pesquisa > refinamento > implementação > validação. E o início foi um período de bastante divergência, quando tentamos entender quem eram os usuários, quais eram as suas necessidades e em que situação estava o sistema para suportar isso tudo. Duas práticas foram usadas aqui para mapear o nosso entendimento: **Personas** e **Jornadas de usuários**.

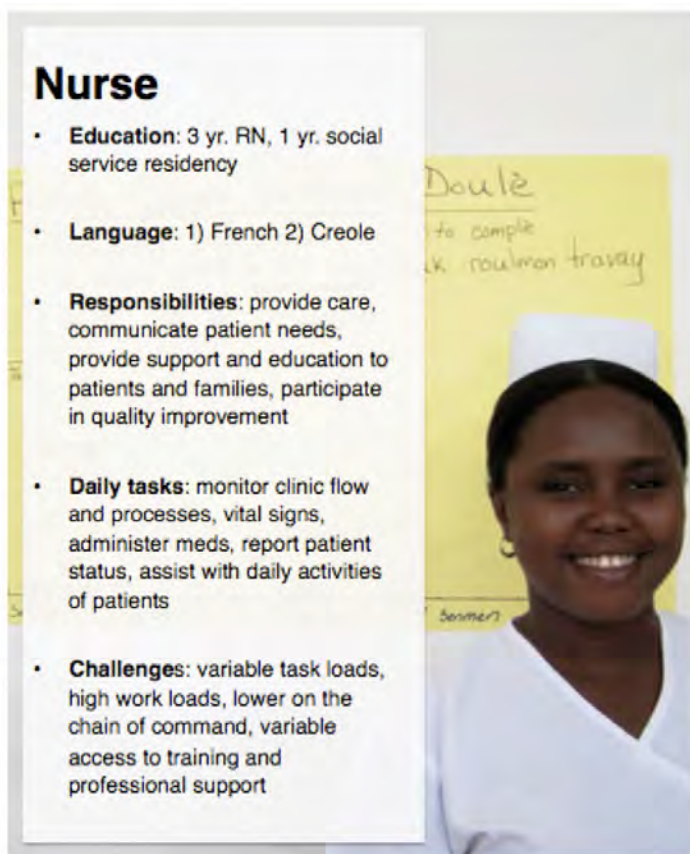


Fig. 6.3: Exemplo de persona

Personas são personagens fictícios que criamos para representar os diferentes grupos de usuários, considerando suas características demográficas, atitudes, comportamentos, objetivos e necessidades para com o sistema. Um

exemplo de persona do nosso sistema pode ser a enfermeira que cuida do processo de triagem. Mas como é uma enfermeira assim no Haiti? Quais são as responsabilidades dela exatamente? Com quem ela troca informação? Muitas outras perguntas nessa linha deveriam ser respondidas.

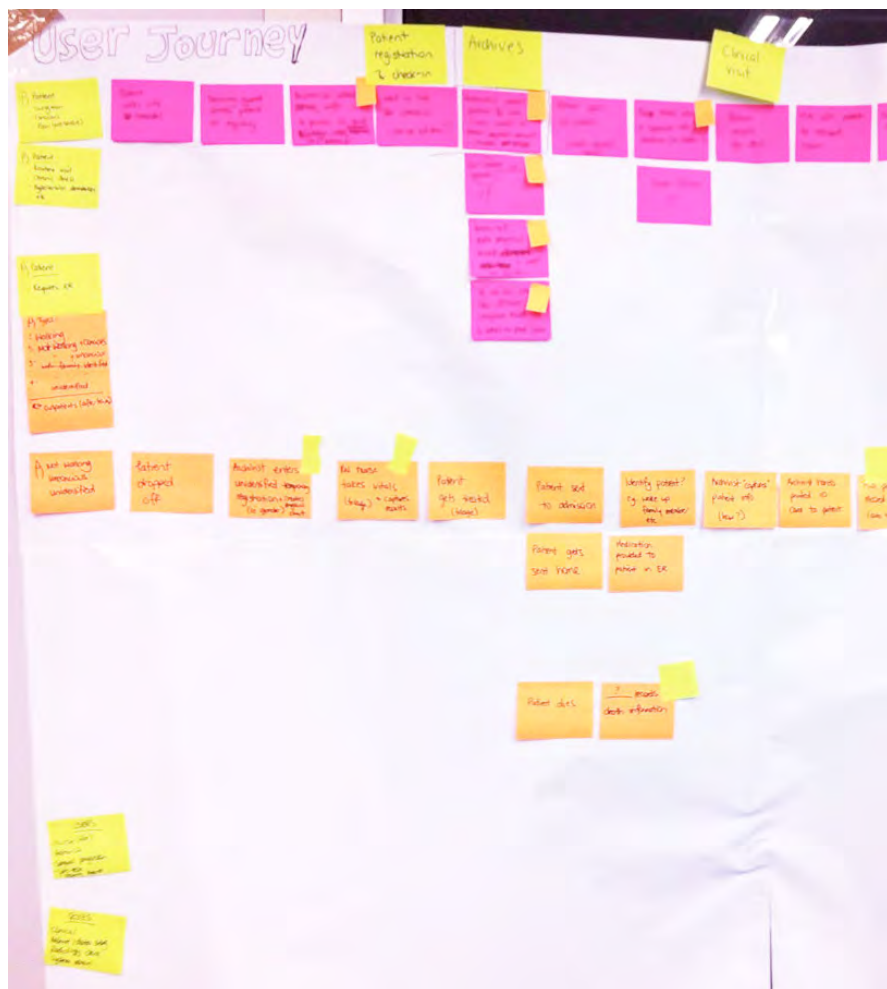


Fig. 6.4: Exemplo de jornada

A **Jornada do usuário** é uma prática dedicada a mapear o passo a passo de

cada persona dentro e fora do sistema, para podermos ter um entendimento amplo do que acontece e de onde se dá a interação com o sistema, e também levantar questionamentos e suposições para as ocorrências de cada etapa.

Usando a enfermeira novamente como exemplo, poderíamos descrever sua jornada da seguinte forma:

- 1) A enfermeira pega a lista de pacientes (De onde vem? Online? Offline? Tem que imprimir? Quem imprime?);
- 2) Caminha até a sala de espera (Qual a distância? O que tem no caminho? Quanto tempo leva?);
- 3) Chama o próximo paciente (Ele está lúcido? Está de cadeira de rodas? Tem acompanhante?);
- 4) Leva o paciente até a sala de triagem (Auxilia o paciente a caminhar? O acompanhante entra junto? Qual a distância? Tem obstáculos?);
- 5) Abre o sistema de triagem (Tem que logar? Tem que clicar? Computador móvel? Legível de onde ela está?);
- 6) Tira a pressão do paciente;
- 7) Pesa o paciente;
- 8) Anota no sistema os valores medidos (Anota tudo de uma vez só, ou a cada vez que mede uma coisa já anota? Anota primeiro no papel? Preenche planilha?).

Depois de termos um entendimento conciso a respeito dos usuários, suas jornadas e o sistema que estávamos desenvolvendo, estava na hora de validar as tantas suposições e hipóteses que surgiram, para isso planejamos uma viagem ao hospital com o objetivo de entender o projeto hospitalar e as condições onde cada parte do sistema seria usado, além de testar o nosso sistema ainda em desenvolvimento com representantes reais de nossas personas e jornadas.

Parte da nossa equipe partiu para o Haiti, montou um computador dentro do hospital ainda em construção, conversou e guiou testes de usabilidade com médicos, enfermeiros e outros funcionários que fariam parte do dia a dia do

hospital, e até mesmo pessoas que estavam por ali, trabalhando na construção ou simplesmente curiosos (no Haiti, qualquer computador pode virar uma atração).



Fig. 6.5: Aprendendo sobre o uso do sistema (no hospital)

Com os testes e um conhecimento maior sobre as dependências e fluxos do hospital, muitas das nossas suposições foram esclarecidas e nós conseguimos dar um direcionamento bem melhor para as nossas hipóteses de navegação, linguagem e usabilidade do sistema.

Uma das nossas descobertas mais valiosas foi que, para o sistema de registro de pacientes e triagem, teríamos que implementar uma navegação 100% pelo teclado e sem o uso da tecla `shift`, dado que os usuários desconheciam essa função e o uso do mouse tornaria o processo mais lento e difícil de aprender.

Todo o time estava envolvido nas dinâmicas e decisões de design, desenvolvedores, analistas de negócio, e não apenas os designers. Fazíamos sessões

rápidas de rabisco, nas quais testávamos várias alternativas para o mesmo problema, e trabalhávamos sempre em paralelo com o time de Desenvolvimento. Histórias em análise eram histórias que faziam tanto a análise do negócio quanto das necessidades do usuário, portanto sempre pensávamos no sistema como pequenas partes que deveriam se comunicar e encaixar perfeitamente no contexto e fluxo do restante das partes, mas considerando que cada parte poderia ter sua usabilidade única e peculiar para a situação na qual seria usada.

Utilizamos técnicas de prototipagem rápida em HTML/CSS que ajudaram a testar rapidamente diferentes conceitos de interface. A vantagem de utilizar HTML/CSS para prototipar é que se consegue ter uma ideia mais verossímil de como determinada funcionalidade vai ser na vida real e assim se consegue corrigir possíveis problemas e testá-la com maior facilidade.

Quando já tínhamos definido a maior parte dos componentes da interface e padrões de interações a serem seguidos no sistema, criamos um guia de estilo para auxiliar os desenvolvedores a reusarem e criarem novos elementos para a interface de forma rápida e concisa com o restante do sistema.

O maior valor que todo esse esforço teve foi chegar a um teste final e ouvir o feedback de que a interface é tão simples e de fácil utilização que não foi preciso mostrar aos usuários como fazer as tarefas, apenas mostrar o que estava disponível para se fazer.

6.3 INOVAÇÃO

Esta sequência de perguntas e respostas retrata a natureza inovadora deste projeto. As questões foram feitas para o Dr. David Walton, o principal responsável pela construção deste sistema.

1) Por que a PIH escolheu construir o Mirebalais EMR em vez de usar outro software?

Estamos comprometidos a usar um sistema de código aberto. A PIH teve uma ótima experiência com o OpenMRS, já que um dos fundadores do OpenMRS trabalhou na PIH à época, assim como o arquiteto-chefe do OpenMRS. Mas, apesar de tudo isso, sentimos que era o melhor caminho futuro em relação ao EMR baseado no que sabemos de versões anteriores de

OpenMRS.

2) O que exatamente o Mirebalais EMR fez diferente de outros EMRs?

Eu acho que a principal diferença entre o Mirebalais EMR é o upgrade para 2.0 e o novo UI/UX que melhorou enormemente a usabilidade para interações no ponto de atendimento entre médicos e pacientes.

3) Qual é a maior vantagem do Mirebalais EMR quando comparado com outros softwares?

O Mirebalais EMR foi criado e desenvolvido para locais com recursos limitados. O sistema não foi feito em qualquer departamento de TI. Mesmo não tendo o time 100% do tempo dentro do hospital, a sensação é de que ele foi construído dentro do hospital; não é um sistema de software, mas sim um sistema hospitalar.

4) Quais features no Mirebalais EMR teve um grande impacto no hospital?

Nossa, foram várias. Posso listar algumas que me vêm à cabeça:

- a integração com PACS;
- as novas interfaces com os usuários;
- os formulários de registro desenvolvidos visando usuários pouco letrados;
- o ponto de atendimento para os médicos;
- o monitoramento e o *dashboard* de avaliação para o hospital;
- o módulo dos pacientes, que nos traz informações essenciais para a melhoria do atendimento prestado. Alguns exemplos: acompanhar de onde estão vindo os pacientes; entender quem visita o hospital e mapear pelo país todo; acompanhar o número de visitas a pacientes, quais departamentos, entre outros, para entender os padrões de uso etc.

5) Você vê um modelo de negócio para o Mirebalais EMR?

Talvez, mas só como uma implementação. Eu acho que ele deve permanecer como software de código aberto gratuito para ser usado por qualquer

um que precise deste tipo de software para melhorar a entrega de assistência médica em clínicas ou hospitais. É muito mais que um modelo de negócio; é um modelo de melhoria social, da colaboração entre pessoas de diferentes países, competências e áreas de atuação.

6.4 CONCLUSÃO

O Hospital Universitário de Mirebalais causou um enorme impacto positivo no sistema de saúde do Haiti, melhorando a vida de milhares de pessoas em todo o país, fornecendo cuidados médicos para uma área de referência na qual 3,4 milhões de pessoas vivem, compreendendo a população de Mirebalais e duas outras regiões circundantes.

Nos primeiros seis meses desde a abertura do hospital, sua equipe registrou mais de 80.000 pacientes únicos, com mais de 200.000 visitas clínicas. Em outubro de 2013, a primeira turma de residentes iniciou o treinamento prático de pediatria, cirurgia geral e medicina interna. À medida que novas turmas de residentes iniciam a cada outono, o número de médicos treinados vai crescer cada vez mais.

Planos futuros de expansão do programa irão incluir outros profissionais de saúde, como anestesistas e outras enfermeiras especialistas, bem como mais especialidades médicas, como medicina de emergência – o primeiro programa deste tipo no país.

Este projeto representou uma grande jornada para todos que dela participaram, na qual um time de pessoas apaixonadas pelo que fazem empregaram as melhores práticas (como Lean UX, entrega contínua e desenvolvimento ágil) para entregar um sistema que está contribuindo para impactar positivamente a vida de milhares de pessoas no Haiti, transformando nossa missão – melhorar o mundo através da tecnologia – em realidade.

CAPÍTULO 7

Formando novos profissionais: um relato de parceria entre empresa e universidade

**por Alexandre Klaser, Émerson Hernandez, Guilherme Froes e
Luiza de Souza**

A dissonância entre a formação técnica de alunos na universidade e as necessidades do mercado é uma realidade amplamente discutida, porém poucas vezes resolvida com soluções integradas entre meio acadêmico e empresas de TI. Neste capítulo, compartilha-se uma alternativa para diminuir essa distância: um framework de capacitação de equipes. Essa abordagem tem se mostrado bem-sucedida e pode ser replicada por instituições que queiram imple-

mentar programas semelhantes com o objetivo de buscar aproximação entre o mundo acadêmico e o mercado de TI. Após algumas iterações aplicando este framework, conseguiu-se gerar evidência acadêmica que suportasse os métodos ágeis como processo a ser adotado por equipes de desenvolvimento. Ao mesmo tempo, foi possível capacitar alunos de graduação para o mercado de trabalho dentro das práticas mais modernas e eficazes de construção de software.



Fig. 7.1: Componentes da aceleradora

7.1 MOTIVAÇÃO

A cada ano que passa, mais profissionais estão se formando em cursos da área da Tecnologia da Informação (TI), como Ciências da Computação, Sistemas de Informação e Engenharia da Computação. Entretanto, o número não tem sido suficiente para atender à demanda do mercado. No levantamento mais recente feito pela Brasscom (Associação Brasileira de Empresas de Tecnolo-

gia da Informação e Comunicação), o mercado brasileiro de TI teria 78 mil novas vagas abertas em 2014, das quais apenas 33 mil seriam preenchidas por profissionais formados em cursos superiores.

Além da carência do ponto de vista numérico, existe também uma deficiência na formação dos novos profissionais. Em um primeiro momento, supõe-se que um egresso da universidade apresente diferencial competitivo em função de uma melhor formação teórica, porém, na realidade, as necessidades do mercado diferem muito daquilo para que os alunos foram capacitados. Muitos estudantes acabam não se envolvendo com atividades extracurriculares que poderiam aproximá-los de sua profissão durante o período em que frequentam a graduação.

Pensando nisso, diversas empresas oferecem programas de trainee que são projetados para aqueles que possuem pouca ou nenhuma experiência profissional. Alguns desses programas têm se mostrado eficazes. No entanto, a maioria deles está focada em necessidades específicas das organizações (buscando, acima de tudo, formar funcionários) e não em uma formação holística do profissional que o habilite a se colocar competitivamente no mercado. Dessa forma, os participantes de tais projetos acabam não tendo a oportunidade de aperfeiçoar amplamente suas habilidades técnicas e comportamentais. Isso significa que muitos programas resolvem problemas específicos, mas acabam não colaborando como poderiam para o desenvolvimento de novos profissionais.

Nesse contexto, um fator relevante identificado como ponto de melhoria de tais programas é o pré-requisito necessário para se fazer parte desse tipo de iniciativa. Uma das perguntas recorrentes quando se trata do assunto é: por que é necessário esperar que o profissional esteja formado ou nos últimos anos de graduação para oportunizar a ele a experiência profissional?

7.2 IDEALIZANDO O PROGRAMA

Refletindo sobre todas essas questões e sobre como a ThoughtWorks poderia contribuir para aperfeiçoar a formação de futuros integrantes do mercado de TI, foi proposta uma aproximação da referida empresa com a Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS) através da elaboração de

um novo programa de estágio. Como resultado, surgiu uma parceria entre o Centro de Inovação Microsoft – PUCRS, a Faculdade de Informática da PUCRS (Facin) e a ThoughtWorks. Através de um programa de TI preexistente, chamado Software Kaizen, financiado pelo Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), alunos da área de TI, matriculados em qualquer semestre de qualquer instituição de ensino superior, são expostos a um ambiente de simulação propício ao aprendizado, sem cobrança de vínculo empregatício.

A primeira edição do programa foi definida a partir de várias discussões entre a ThoughtWorks e a PUCRS, mantendo sempre a premissa de que era necessário que todos os envolvidos estivessem imersos em um ambiente de ensino, ou seja, que todos, sem exceções, estivessem aprendendo.

O programa em questão tem como foco principal o ensino de tecnologias e práticas para o gerenciamento de projeto, visando melhor organização, produtividade (capítulo 2) e qualidade. Espera-se que ele seja capaz de transformar um conjunto de alunos com pouca ou nenhuma experiência em desenvolvimento de software em uma equipe funcional, no período de 15 semanas. Ao término desse período, a equipe deve ser capaz de compreender os princípios dos métodos ágeis, além de práticas de desenvolvimento ágil como: desenvolvimento guiado por testes (*Test Driven Development*, capítulo 15), programação em par (*Pair Programming*, capítulo 8) e técnicas de governança. Além disso, é fundamental que cada participante seja capaz de identificar o valor e de problematizar cada prática apresentada em vez de possuir apenas o conhecimento teórico relativo a elas.

Visando alcançar esses objetivos, foi planejado um conjunto de ações que buscam simular ao máximo as situações de dia a dia encontradas em projetos de desenvolvimento de software. Com isso, a equipe participante tem a oportunidade de identificar as necessidades de um cliente real e de desenvolver um produto que agregue valor ao negócio. Esse produto pode ser um novo sistema ou parte dele, desenvolvido de forma incremental e iterativa, promovendo sempre o diálogo entre time e cliente.

Ao longo das iterações já realizadas do programa, observou-se que as equipes aprenderam a montar e manter um ambiente de desenvolvimento robusto, utilizando, para isso, algumas ferramentas e práticas com foco em

entrega contínua, como sistema de controle de versão, criação de testes automatizados e automatização de *build* e *release*.

7.3 O FRAMEWORK DE CAPACITAÇÃO

O programa tem duração de aproximadamente 15 semanas, exigindo dedicação de 30 horas semanais por parte dos alunos. Ele é dividido em iterações de 2 semanas cada, com o objetivo de converter as principais funcionalidades desejadas pelo cliente em software funcional. Entretanto, por se tratar de um programa focado no aprendizado de alunos com pouca ou nenhuma experiência profissional, foi definida uma iteração adicional, com objetivo diferenciado, chamada de Iteração 0.

Durante essa primeira iteração, os alunos ainda não trabalham diretamente no produto a ser desenvolvido. Esse período é destinado ao aprendizado de alguns conceitos, técnicas e ferramentas que darão suporte ao desenvolvimento. Entre os assuntos abordados e trabalhados, estão: fundamentos de métodos ágeis, ferramentas de controle de versão de código (Git), programação orientada a objetos e testes. Esse conteúdo é ministrado tanto pela mentoria técnica quanto por profissionais convidados, sob forma de palestras, workshops e *dojos*.



Fig. 7.2: O framework de aceleração

As iterações seguintes (de 1 a 6) são focadas na entrega. Nelas, os alunos trabalham em tarefas pequenas, mas diretamente relacionadas às histórias de usuário priorizadas por iteração. O time de mentoria auxilia os alunos a alcançarem um bom equilíbrio entre aprendizado e entrega de um software

funcionando.

7.3.1 Captação de participantes: divulgação e recrutamento

Uma divulgação adequada é fundamental para que mais pessoas possam conhecer o programa e se candidatar. As redes sociais são o principal meio de propagação, assegurando que um maior número de alunos, indiferentemente da instituição de ensino a que pertencem, seja informado a respeito da oportunidade. Além disso, elas também possibilitam que outras pessoas divulguem o programa através de compartilhamentos. Tanto a equipe de marketing da Thoughtworks quanto a PUCRS são os principais responsáveis pela divulgação do programa. Eles não apenas auxiliam a criar o material de divulgação, como também fazem propaganda em seus canais de comunicação. A figura a seguir mostra um exemplo de material de divulgação do programa.

Uma vez concluída a etapa de divulgação, inicia-se o processo de seleção dos candidatos. Nessa fase, a equipe de recrutamento da ThoughtWorks auxilia a organizar e executar o processo seletivo. O recrutamento é dividido em 3 fases: filtragem, avaliação técnica e avaliação comportamental.

**ThoughtWorks Inc.**
20 de março

Estamos procurando por VOCÊ que está querendo acelerar sua carreira!

Mais informações em:
<http://www.centrodeinovacao.org.br/Home/Noticias>

Interessados deviam enviar currículo para ci@pucrs.br

PROGRAMA GRATUITO DE CAPACITAÇÃO EM METODOLOGIAS ÁGEIS

Formação teórica e prática em 15 semanas

PARCERIA THOUGHTWORKS | FACIN | CENTRO DE INOVAÇÃO

“ Foram 15 semanas de aprendizado bastante intenso sobre técnicas de desenvolvimento de software e métodos ágeis. Depois foi só participar da seleção da ThoughtWorks e colocar todo conhecimento em prática.

Lucas Beter
Consultor Jr na TW Brasil

“ Me ajudou a descobrir um novo mundo de aventuras como TDD, code smells, entrega contínua e muito mais. É um treinamento completo e a experiência que eu precisava para seguir adiante na minha carreira!

Thais Hamilton
Consultora Jr na TW Brasil

ESTAMOS PROCURANDO POR VOCÊ QUE:

- É aluno de graduação ou dos últimos anos de cursos técnicos;
- Trará conhecimentos intermediários de programação web. Ainda pode trazer conhecimentos de Bancos de Dados e outras linguagens ou deseja aprender sobre o assunto;
- Tem perfil pro-ativo, com interesse em trabalho em equipe e muita vontade de aprender e se desenvolver;
- Tem disponibilidade das 8h30min às 15h30min (30 horas semanais).

Para mais informações acesse centrodeinovacao.org.br/Home/Noticias
Interessados devem enviar currículo para ci@pucrs.br



Curtir · Comentar · Compartilhar

 48  79 compartilhamentos

Fig. 7.3: Divulgação do programa nas redes sociais

A etapa de filtragem consiste em uma análise curricular do candidato. Para fazer parte do programa, é necessário que o interessado esteja matriculado em curso de graduação da área de TI, a fim de que seja possível que a ThoughtWorks forneça bolsa de estágio. Nessa fase, são removidos os currículos que não se adequam aos requisitos.

Depois da filtragem, a fase de escolha de candidatos é conduzida através de entrevistas técnicas e comportamentais. Em um primeiro momento, essas avaliações foram realizadas de forma individual. Entretanto, essa abordagem se mostrou muito custosa devido ao número de candidatos. A alternativa encontrada foi a realização de entrevistas coletivas e dinâmicas de grupo, processo que é utilizado atualmente.

Com foco na avaliação técnica, objetivando avaliar o conhecimento teórico do candidato, são usados exercícios em formato de dojo de programação (<http://codingdojo.org/>). Conforme representado na figura adiante, os candidatos devem resolver um problema de programação em conjunto, e a solução é analisada pelos mentores. Do ponto de vista de avaliação comportamental, o dojo também auxilia os selecionadores a identificar as relações de trabalho e a capacidade dos entrevistados de trabalhar em equipe.



Fig. 7.4: Exemplo de um dojo

As dinâmicas de grupo são realizadas com equipes de cinco ou seis candidatos. Cada grupo é acompanhado por dois facilitadores/observadores sendo, normalmente, uma pessoa do recrutamento da ThoughtWorks e um membro do time de mentores/monitores. A atividade é dividida em duas partes: inicialmente, todos os presentes se apresentam e os facilitadores respondem perguntas sobre o programa. Na segunda parte, é proposta uma tarefa que os candidatos devem realizar em grupo. Nessa fase, os inscritos devem escolher um dentre três temas propostos, realizar uma rápida pesquisa na internet sobre o assunto e apresentar os resultados aos facilitadores.

Uma vez finalizadas as etapas de recrutamento, os envolvidos se reúnem para definir quem serão os integrantes do programa. Essa escolha é realizada levando em consideração, principalmente, as características comportamentais do candidato. As atitudes deles devem estar alinhadas às três qualidades procuradas pela ThoughtWorks em seu recrutamento tradicional: aptidão,

atitude e integridade. Após a escolha dos integrantes, a equipe administrativa da PUCRS gerencia os contratos de estágio, realizando os procedimentos formais que possibilitam o ingresso dos candidatos no programa.

7.3.2 Estrutura: papéis e práticas

Para que o programa tenha o impacto esperado, é necessário que promova aos participantes um contato direto com profissionais mais experientes. Estes têm a função de acompanhar as atividades diárias do time e compreender não apenas as dificuldades técnicas como também identificar os aspectos comportamentais que devem ser desenvolvidos. A equipe de apoio é composta por membros da ThoughtWorks e da PUCRS.

Como o grupo envolvido é relativamente grande, é necessária uma divisão de papéis, de modo que as atividades possam ser exercidas por um ou mais integrantes. Cada integrante tem responsabilidades específicas, porém não excludentes. Os papéis definidos são os seguintes:

- **Monitor:** realiza o acompanhamento diário do time de alunos e auxilia na resolução de empecilhos não técnicos, como organizar reuniões e solucionar problemas com a documentação do programa. Além disso, é o responsável por coletar dados relacionados ao desempenho do time e repassar essas informações aos mentores;
- **Mentor ágil:** facilita o aprendizado de valores, princípios e práticas ágeis, enfatizando aspectos comportamentais, de negócio e de liderança. Além disso, é responsável por coletar dados para fins de análise e comparação entre as edições do programa;
- * **Mentor técnico:** auxilia no aprendizado de aspectos técnicos, tais como boas práticas, linguagens de programação e ferramentas. Cabe a ele monitorar o desenvolvimento do produto e garantir a qualidade do software através da realização de revisões de código;
- **Analista de negócio:** assessora o time no que diz respeito à visão de negócio, sendo o responsável por ajudar a equipe a chegar, coletivamente, no mínimo produto viável (MVP, que vimos no capítulo 6). Além disso,

cabe a ele auxiliar na criação das *user stories*, vistas no capítulo 1, e no planejamento das iterações, ou seja, dos ciclos de duas semanas;

- **Alunos:** são os responsáveis por desenvolver o produto, o que inclui toda a programação, criação da interface gráfica, definição da arquitetura do software e garantia de qualidade. São eles que realizam toda a comunicação com o cliente;
- **Representante do negócio:** detém a visão do produto que é desenvolvido pelos alunos e é o responsável por priorizar as funcionalidades que devem ser incluídas na lista de todas as funcionalidades desejadas para o MVP.

7.4 PRÁTICAS ÁGEIS ADOTADAS

Dentre as várias práticas ágeis que equipes maduras tradicionalmente utilizam em seus projetos, destacam-se as seguintes, que são adotadas e ensinadas no escopo do programa em questão:

1) Programação em par (Pair Programming)

Como o próprio nome sugere, na programação em par duas pessoas trabalham juntas no desenvolvimento de uma funcionalidade. Dessa forma, enquanto uma delas escreve o código, a outra faz uma revisão do programa que está sendo escrito. Este procedimento melhora não apenas a qualidade do software desenvolvido, como também o aprendizado e a comunicação entre os membros do time. Tal técnica é abordada já no primeiro dia do programa e a sua utilização é assegurada através da limitação do número de máquinas. Como há apenas uma máquina para cada dois alunos, eles obrigatoriamente têm de parear todos os dias e em todas as tarefas. A figura a seguir representa uma atividade desenvolvida por meio da programação em par.

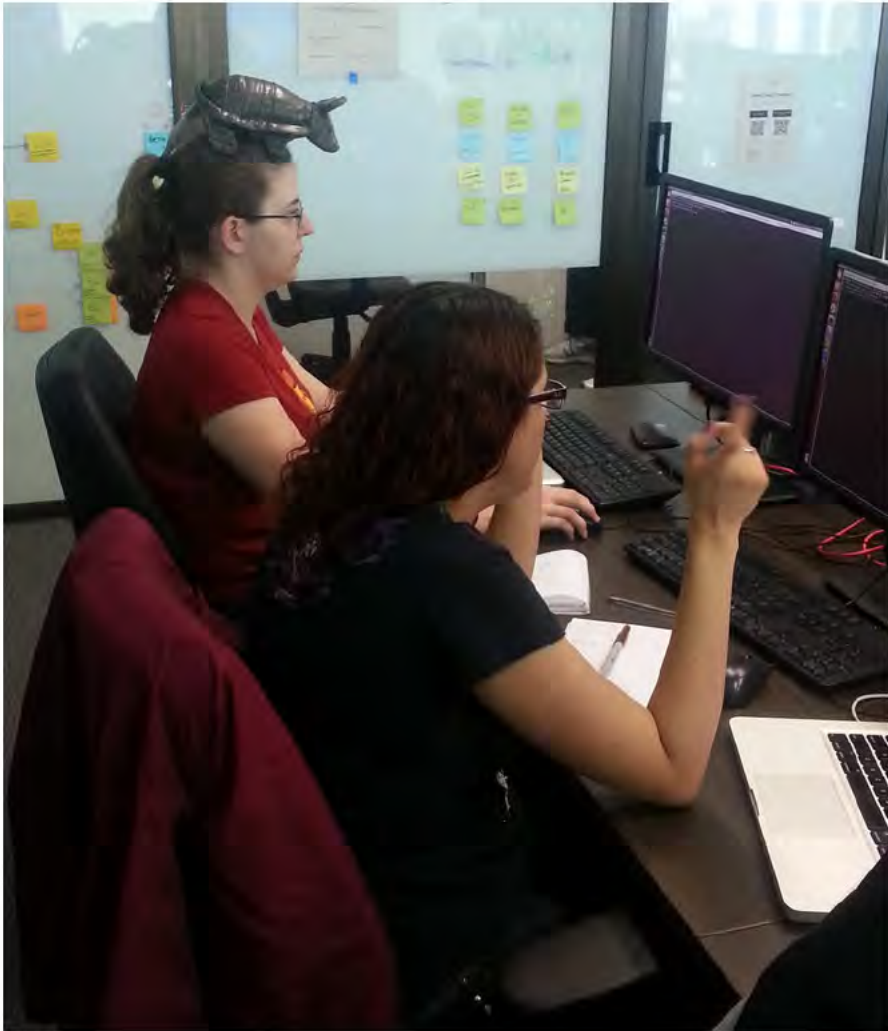


Fig. 7.5: Programação em par

Veremos mais sobre programação em par no capítulo 8.

2) Desenvolvimento guiado por testes (Test Driven Development)

Trata-se de um processo de desenvolvimento de software que se baseia em um curto ciclo de desenvolvimento. Inicialmente, é escrito um caso de

teste para uma nova funcionalidade, que deve falhar. Depois, implementa-se a menor solução para fazer com que esse caso de teste passe e, em seguida, tanto o código gerado quanto seus teste são refatorados. Isso beneficia o design e a simplicidade do software que está sendo desenvolvido e, como efeito colateral, assegura a cobertura de testes. Após algumas sessões explicativas, a utilização dessa técnica é cobrada dos alunos, possibilitando que eles a compreendam e percebam os seus benefícios. Veremos mais sobre TDD no capítulo 15.

3) Ideation

Consiste no processo de escolha do projeto que será desenvolvido pelos alunos. Essa decisão é tomada através de uma espécie de eleição, em que alguns projetos propostos são possíveis candidatos. Os critérios de avaliação das propostas são os seguintes:

- **Viés social:** o projeto deve ter um impacto real e positivo na sociedade;
- **Apoio ao aprendizado:** o projeto, sua estrutura e seus componentes técnicos devem favorecer o aprendizado de conceitos e técnicas, o que é o foco do programa;
- **Simplicidade técnica:** uma vez que os alunos são iniciantes, a complexidade técnica deve ser mínima, oferecendo poucas barreiras ao aprendizado;
- **Disponibilidade do mentor técnico:** o mentor técnico do projeto deve ter disponibilidade de tempo e conhecimento necessário para apoiar os alunos;
- **Disponibilidade do representante do projeto:** o representante do projeto deve ter disponibilidade para participar ativamente das atividades.

Uma vez entendidos os critérios, cada projeto candidato é apresentado pelo seu representante e todos os eleitores têm a oportunidade de esclarecer as suas dúvidas. Em seguida, tanto os alunos quanto os apoiadores votam no melhor projeto para cada um dos critérios, e o vencedor é aquele com o maior número de votos. A figura adiante apresenta os resultados de uma Ideation.

IDEIAS ↓ CRITÉRIOS	POA COMO VAMOS	DARIGEL MELHOR BRASIL!	CEP RUAS	INIMIGOS DA MESA
VIÉS SOCIAL	9 ☒ ☐	2 ☐	6 ☒ ☐	5 ☒
APÓIO AO APRENDIZADO	8 ☒ ☐	5 ☐ ☐		9 ☒ ☐
SIMPLES	5 ☐ ☐	2 ☐	8 ☒ ☐	7 ☒ ☐
COMPROMISSO DO MÉTODO TÉCNICO	1 ☒ ☐	4 ☐	6 ☒ ☐	4 ☐
DISPONIBILIDADE DO P.O.	8 ☒ ☐	2 ☐	2 ☐	9 ☒ ☐
TOTAL	39	15	22	34

Fig. 7.6: Resultado da Ideation

4) Inception

Trata-se de uma atividade ou de um conjunto de atividades realizadas no começo de um novo projeto para elucidar os requisitos. Sua função é criar um entendimento, em conjunto, daquilo que deve ser construído e do que define o sucesso do empreendimento. Por esse motivo, normalmente, a inception é executada com todos os envolvidos no projeto (cliente, desenvolvedores, usuários) para que eles cheguem, juntos, a uma definição do que deve ser o MVP. Esse processo é ilustrado na figura a seguir:

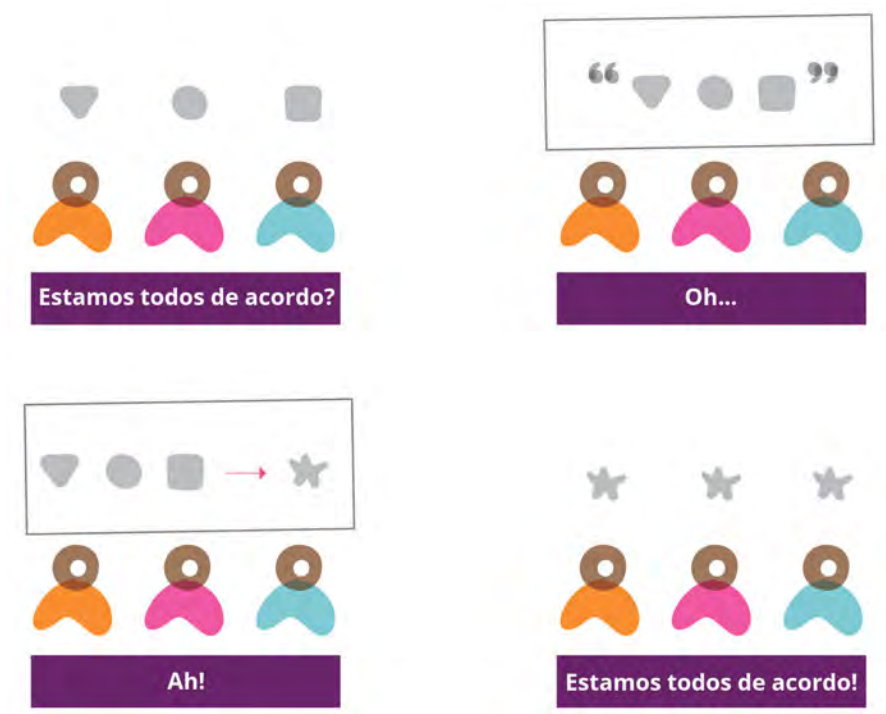


Fig. 7.7: Alinhamento do que será o produto através da inception

Vimos mais sobre inception no capítulo [1](#).

5) Entrega contínua

Trata-se uma disciplina de desenvolvimento de software que permite que o software seja colocado em produção a qualquer momento. Através de uma implantação automatizada que envolve o processo de compilar a aplicação, de implantá-la em um ambiente, de testá-la e de efetuar sua entrega final, é possível reduzir o custo, o tempo e o risco da entrega de mudanças incrementais aos usuários.

6) Gerenciamento de projeto

Do ponto de vista de gerenciamento de projetos ágeis, de forma iterativa e

incremental, as cerimônias realizadas são listadas a seguir. As reuniões acontecem para haver continuamente compartilhamento de conhecimento entre os membros da equipe e visibilidade necessária para interações de sucesso entre os membros do time.

- **Planejamento da iteração:** a reunião de planejamento marca o início de uma iteração. Nela são definidas as funcionalidades que serão entregues. Na realidade, cabe ao representante do projeto priorizar essas funcionalidades. O time deve informar a quantidade de trabalho que acredita ser capaz de desenvolver na iteração. A figura a seguir ilustra a priorização das funcionalidades realizada pelo representante do projeto.



Fig. 7.8: Priorização das tarefas que serão entregues ao término da iteração

- **Reunião diária:** é um encontro que costuma acontecer na parte da manhã, com duração de aproximadamente 15 minutos. Tem como obje-

tivo comunicar as informações importantes para todo o time. Normalmente, compartilha-se o que foi feito no dia anterior, o que será feito no dia atual e os problemas que estão atrapalhando o objetivo. Tal encontro é uma ótima forma de manter toda equipe alinhada, de identificar problemas que estejam atrapalhando o desempenho do time e de apontar soluções.

- **Apresentação de resultados da iteração (*Showcase*):** consiste em uma reunião que ocorre ao término de uma iteração, permitindo que o time mostre para o cliente o resultado do que foi desenvolvido. O representante do projeto analisa a funcionalidade que será entregue, aprovando-a ou solicitando mudanças.
- **Retrospectiva:** essa atividade costuma acontecer logo após o *Showcase* e permite ao time verificar o seu progresso e identificar as lições aprendidas. Nessa reunião, a equipe pode refletir sobre tudo o que aconteceu na iteração recém-terminada, visando assegurar melhorias contínuas, tanto do time quanto do processo. Normalmente, uma pessoa com mais experiência, mentor ou monitor, atua como facilitador dessa reunião para auxiliar a equipe. O resultado de uma retrospectiva é representado na figura a seguir.

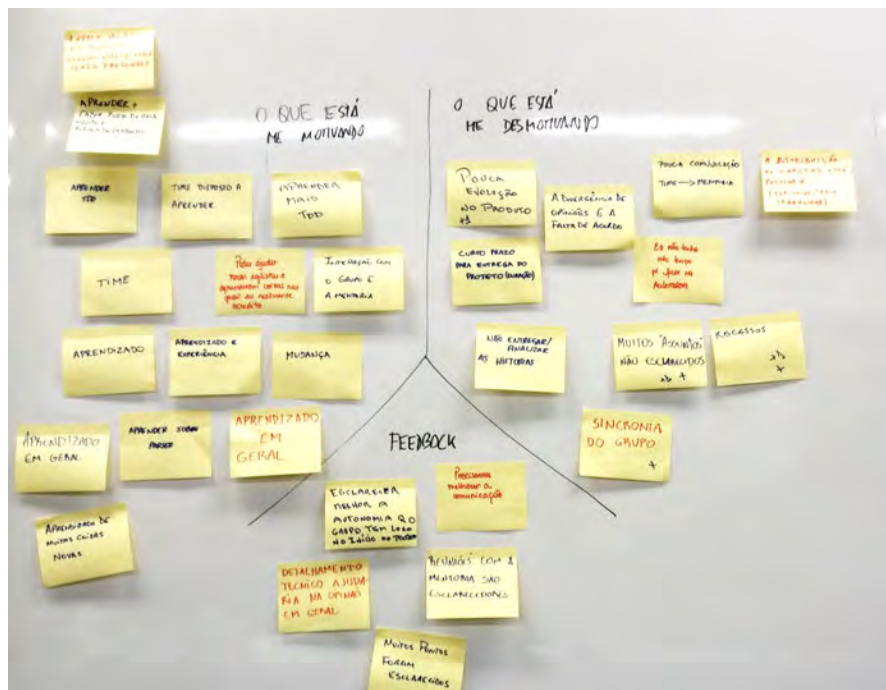


Fig. 7.9: Resultado de uma retrospectiva

- **Code review:** basicamente, ao terminar uma funcionalidade, os alunos submetem o seu código à equipe de mentoria. Esta, por sua vez, faz uma análise, certificando-se de que foram utilizadas boas práticas de programação. Caso qualquer problema seja identificado, os alunos deverão fazer as devidas alterações. Somente quando a equipe de mentoria aprova uma funcionalidade é que ela é considerada pronta para o *Showcase*.

7.5 CONCLUSÃO

A aproximação entre dois universos atualmente tão distintos como a formação técnica de alunos do ensino superior e as necessidades do mercado cabe não apenas às universidades. Deveria tratar-se de uma iniciativa conjunta entre instituições de ensino e empresas, sendo fundamental o papel destas úl-

timas. Promover esse relacionamento é benéfico para ambas as partes e ainda mais para a sociedade como um todo, pois possibilita que os novos profissionais que estão iniciando suas carreiras sejam preparados para encarar os desafios que terão pela frente de forma mais confiante, visando mais produtividade e melhores resultados.

A experiência com as três primeiras turmas foi muito relevante. Várias foram as lições aprendidas que levaram a resultados gratificantes. A parceria entre a PUCRS e a ThoughtWorks revelou, também, agradáveis surpresas, como o reconhecimento do programa, logo em seu primeiro ano, através do 4º Prêmio Inovação em Educação, promovido pelo Sindicato do Ensino Privado do Rio Grande do Sul (SINEPE-RS). Além disso, dois alunos da primeira turma participaram do processo seletivo da ThoughtWorks e foram contratados. Tal processo de admissão é historicamente conhecido como bastante rigoroso. Essa é uma prova inequívoca do sucesso do programa, que pode transformar alunos com pouca ou nenhuma experiência de mercado em profissionais capacitados.

7.6 AGRADECIMENTOS

Gostaríamos de agradecer a todos os Thoughtworkers envolvidos com as diversas turmas da Aceleradora. Eles tornaram possível o projeto, ajudando-nos com o recrutamento dos alunos, com a divulgação do programa, com a capacitação das turmas e com o apoio necessário.

Agradecemos também a Rafael Prikladnicki e Michael da Costa, ambos da PUCRS, por todo o esforço empenhado na criação, viabilização, divulgação e condução do programa. Eles representam muito bem a sinergia entre universidade e empresas, algo que acreditamos ser chave para a capacitação profissional dos alunos.

Queremos destacar o trabalho do Alejandro Olchik, pela sua enorme e incansável dedicação ao programa ao longo da ocorrência de todas as turmas, servindo como mentor ágil e tendo a nem sempre fácil tarefa de conciliar as diferentes visões, desejos e aspirações que se manifestam em grupos tão heterogêneos. Da mesma forma, agradecemos aos monitores Bernardo José, Pedro Guidoux e João Henrique pelo seu trabalho fundamental de acompa-

nhamento das turmas, atividade que se revelou cada vez mais importante ao longo das edições do programa. Eles representam um elo muito forte entre alunos e apoiadores, tendo de saber lidar não só com questões técnicas, mas também pessoais e comportamentais dos alunos.

Finalmente, queremos destacar o papel fundamental do Centro de Inovação Microsoft e de sua equipe, responsáveis por fornecer o espaço físico e a infraestrutura necessária para a execução do programa.

CAPÍTULO 8

Programação em par

por Roger Almeida

No dia a dia da ThoughtWorks, nós consideramos Programação em par algo essencial, que faz parte do cerne do nosso processo de desenvolvimento de software.

Mas fazer parte não quer dizer que trabalhamos em par 100% do tempo, há situações em que deliberadamente não trabalhamos em par. É importante entender os benefícios dessa prática para conseguir colher seus melhores frutos. Neste capítulo, vamos analisar situações em que a programação em par se encaixa melhor e técnicas para adotar essa prática.



Fig. 8.1: programação em par

8.1 DEFINIÇÃO

Programação em par é uma prática de desenvolvimento de software na qual duas pessoas trabalham em uma mesma tarefa, ao mesmo tempo. Uma das pessoas atua como piloto – ela é responsável por controlar o teclado e o mouse. A segunda pessoa é o navegador – ela é responsável por acompanhar e cuidar de tudo o que está sendo realizado, como decisões de design e a arquitetura durante o desenvolvimento. É importante que haja a troca de papéis entre as pessoas, para que ambas atuem tanto como piloto quanto como navegador. A origem da programação em par é creditada a Kent Beck e Ward Cunningham, os criadores do *eXtreme Programming* (XP), mas há relatos de que outras equipes de desenvolvimento já trabalhavam em par muito antes de Kent e Ward palestrarem sobre XP. Dijkstra diz que ele e J.A. Zonneveld já trabalhavam em par em 1969.

Laurie Williams e Alistair Cockburn mostraram que, trabalhando em par, dois desenvolvedores são apenas 15% mais lentos do que trabalhando sozi-

nhos, o que já mostra que trabalho em par não leva o dobro do tempo. Além disso, eles mostraram que o código resultante de programação em par contém 15% menos bugs. Como encontrar e corrigir bugs é mais custoso do que escrever o código, essa troca já se mostra muito vantajosa. Porém, existem outros benefícios que iremos tratar ao longo deste capítulo.

8.2 QUANDO PAREAR

Gostaria muito de ter uma resposta simples e direta para isso, mas infelizmente não tenho. Existem várias analogias sobre programação em par e outras profissões que utilizam trabalho em par, como cirurgiões e pilotos de avião. Mas uma que considero bem adequada é sobre corridas de Fórmula 1 e corridas de rally.

Essas duas modalidades de automobilismo têm algumas diferenças interessantes: em uma corrida de Fórmula 1, pelo fato de o circuito ser fechado, após a terceira ou quarta volta o piloto já está praticamente em modo automático, ou seja, ele consegue reagir às curvas sem muito esforço mental. Ele ainda tem de lidar com percalços, como fazer ultrapassagens ou parar nos boxes, mas, mesmo assim, na maior parte do tempo ele está trabalhando sobre algo conhecido: o percurso. Já o rally é uma corrida diferente. Nela, os participantes têm um ponto de partida e um ponto de chegada, com alguns pontos de passagem obrigatória pelo caminho. Cabe a eles a decisão do melhor caminho para chegar ao destino. Os participantes de rally estão o tempo todo em um estado de estresse mental elevado, pois o piloto precisa prestar muita atenção à “estrada”, enquanto o navegador precisa estar o tempo todo atento ao percurso.

Comparando as corridas ao desenvolvimento de software, existem momentos em que temos tarefas repetitivas, as quais espera-se que sejam executadas rapidamente e sem grande esforço mental (Fórmula 1), e há momentos em que a companhia de um navegador é a garantia de que estamos seguindo o caminho correto (rally).

Em geral, na maioria do tempo o desenvolvimento de software se assemelha muito mais ao rally do que a Fórmula 1, já que geralmente saímos de um ponto de partida e sabemos (às vezes) apenas qual nosso objetivo, sendo

o melhor percurso para desenvolver a(s) nova(s) funcionalidade(s) responsabilidade da equipe de desenvolvimento. Mas como desenvolvimento de software não é apenas escrever código, existem cenários em que a programação em par pode ser substituída por trabalho solo, sem maiores riscos.

Alguns cenários nos quais, geralmente, o trabalho em par pode **não** ser o modo mais efetivo são:

- Pesquisas online
- Leitura de documentação
- Preenchimento de formulários

Alguns cenários comuns para o uso de trabalho em par são:

- Projetando uma solução
- Escrevendo código
- Preparando os pontos de uma apresentação
- Bug-fix

O ponto fundamental é: a equipe de desenvolvimento decidir se o trabalho em par é o mais efetivo para as tarefas a serem realizadas naquele dia.

8.3 ILHAS DE CONHECIMENTO

A programação em par endereça vários riscos comuns a equipes de desenvolvimento. Um dos principais, e que as empresas nem sempre têm conhecimento, são as ilhas de conhecimento. Se sua equipe depende sempre de um membro específico para resolver determinadas tarefas, ela está criando uma ilha de conhecimento.

O que acontece se o principal desenvolvedor da sua equipe for atropelado por um ônibus? Parece uma piada de mau gosto, mas não é. O trabalho em par diminui o risco de que alguém da equipe domine sozinho algum conhecimento específico. Com isso, se um dos membros da sua equipe for atropelado

por um ônibus, o dano causado à equipe será menor, já que outros membros do time que tenham pareado com o finado desenvolvedor poderão, ainda que com dificuldade, continuar o trabalho dele. Weinberg tem uma máxima que diz: “Se um desenvolvedor é indispensável, livre-se dele o quanto antes”. Com programação em par, o conhecimento está sempre sendo armazenado com redundância. Ou seja, se um dos conteúdos de conhecimento faltar, sempre haverá uma redundância para resgatar esse conhecimento.

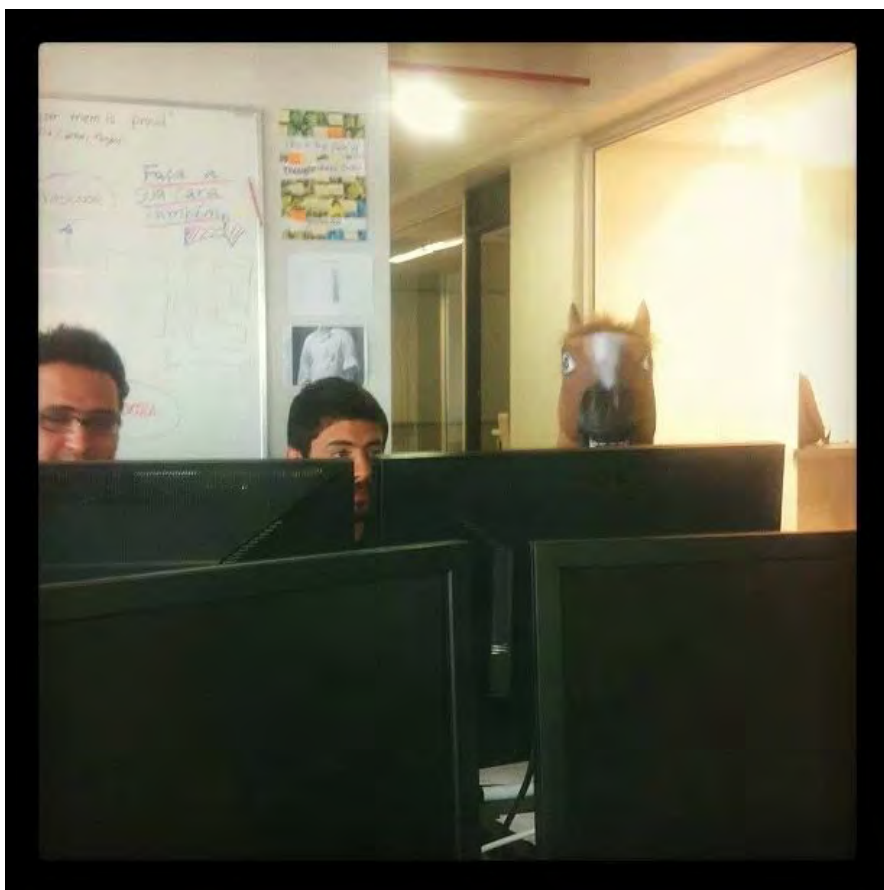


Fig. 8.2: Sabedoria compartilhada

8.4 FERRAMENTAL E PAREAMENTO REMOTO

A programação em par não exige um setup específico de máquina. Pode tranquilamente ser feita usando apenas uma estação de trabalho, quando o par está fisicamente junto. Mas, no dia a dia da ThoughtWorks, nós geralmente temos estações de pareamento que são máquinas com dois monitores, dois mouses e dois teclados. Isso não acontece em todos os projetos, mas é uma prática muito comum montarmos essas estações de pareamento. Isso proporciona o conforto ergonômico dos membros do par. Também usamos alguns softwares de compartilhamento de telas para permitir que acessemos o computador do par a partir da nossa estação de trabalho.

Atualmente, os softwares que mais usamos são o TeamViewer (<http://www.teamviewer.com>) , ScreenHero (<https://screenhero.com/>) e Synergy (<http://synergy-project.org/>) . Eles ajudam principalmente quando não temos estações de pareamento disponíveis ou quando estamos fazendo pareamento remoto. Inclusive usamos bastante o pareamento remoto quando pareamos com nossos clientes. Para tanto, geralmente adicionamos ao setup uma webcam e um microfone que ficam sempre ligados, de modo que podemos interagir com nosso par de forma visual e auditiva, quase como se estivéssemos fisicamente juntos.

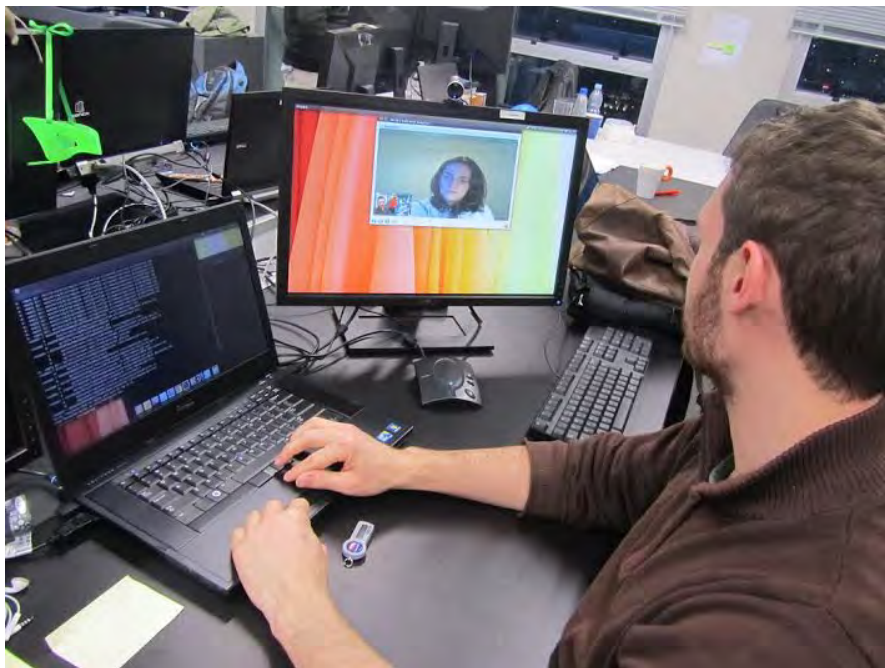


Fig. 8.3: Compartilhando a tela

8.5 TÉCNICAS

É normal ouvir pessoas que dizem “Eu tentei, mas não gostei. Não deu certo”. Como todas as outras práticas ágeis de desenvolvimento de software, não basta sentar e tentar fazer, é necessário entender a motivação por trás dela e procurar quais são as técnicas para aplicar essa prática. É muito comum, ao conversar com desenvolvedores ou equipes que tentaram implantar essa técnica, encontrar aqueles que se frustraram na implantação. Geralmente surgem justificativas como “Me senti desconfortável demais”, “Não sabíamos como fazer direito”, “Pra gente parece que não funciona” etc. A programação em par, assim como a maioria das práticas ágeis de desenvolvimento de software, é fácil de entender, mas difícil de dominar.

Ao longo dos anos na ThoughtWorks, algumas técnicas emergiram do dia a dia para ajudar aqueles que querem dar uma chance à programação em par.

Com essas técnicas, uma equipe que esteja começando ou deseja começar a utilizar programação em par pode evitar pequenas armadilhas que podem atrapalhar o sucesso nessa prática.

A intenção da lista a seguir é dar ciência às práticas mais comuns, para aplicá-las pode ser necessário fazer pequenas adaptações para que se adequem à realidade de cada equipe.

Nome da técnica	Dificuldade de implantação
Relógio de xadrez	Baixa
Pense alto	Baixa
Técnica do pomodoro	Baixa
Sua ideia primeiro	Baixa
Regra dos 10 segundos	Média
Ping-pong	Alta

Fig. 8.4: Lista de práticas comuns para programação em par

Relógio de xadrez

O par usa um relógio no estilo dos utilizados pelos jogadores de xadrez para marcar quanto tempo tem antes da próxima troca de posições.

Benefícios

Garante que haja a troca entre os papéis do par e que não haja um piloto dominante no par.

Dificuldade de implantação: Baixa

Basta arranjar um timer, que pode ser virtual ou online, e definir uma métrica de tempo que funcione para a dupla. Geralmente esses valores variam entre 10 minutos e 1 hora.

Pense alto

Quando um membro do par estiver pensando em como resolver um problema, é interessante que ele fale o que está pensando. Isso evita que tanto o piloto quanto o navegador fiquem parados com um olhar fixo no código, sem dizer nada e deixando seu parceiro sem a mínima pista do que está se passando.

Um dos princípios por trás da programação em par é a revisão e validação de ideias o mais cedo possível, ou seja, antes que ela seja adicionada ao código. Por isso, é importante que, quando uma ideia estiver surgindo, ela já seja compartilhada com o par, para que ele possa entender o mapa mental que está sendo montado na cabeça do parceiro.

Benefícios

Essa técnica ajuda a criar sinergia entre o par, para que as ideias geradas não sejam apenas as de um dos membros do par, mas sim o resultado de uma dupla validação de cada ideia. Além disso, evita que qualquer um dos dois gaste tempo e esforço mental desenvolvendo uma ideia que o par pode ajudar a validá-la ou descartá-la.

Dificuldade de implantação: Baixa

Uma dica para fazer essa técnica fazer parte do dia a dia é colar um *post-it* com o texto “Pense alto” próximo ao monitor para lembrar ambos que precisam pensar alto. Além disso, basta se sentir confortável para compartilhar ideias com o par. Vale lembrar que a regra dos 10 segundos, citada nos próximos tópicos, ajuda os dois a se sentirem mais confortáveis em compartilhar uma ideia. Uma outra dica é usar uma variação da regra dos 10 segundos, em que, se um dos membros ficar mais de 10 segundos parado sem falar nada, o par dele o lembra dizendo: “Pense alto”.

Técnica do pomodoro

A técnica do pomodoro (<http://pomodorotechnique.com>) não é uma regra originalmente da programação em par, mas tem se mostrado útil para garantir que o par se mantenha focado no trabalho e ainda haja tempo para ver um e-mail pessoal, pagar uma conta ou mesmo se atualizar com as novidades da sua rede social preferida, além de, é claro, gerar soluções para os problemas do dia a dia. Para aplicar a técnica, o par, após decidir qual será a próxima tarefa que irão fazer, separa um período de tempo para concluir essa tarefa. Após esse período de trabalho focado, vem um período de relaxamento mental, em que os membros da dupla podem ir fazer outras coisas, como ir ao banheiro, chat online, ou qualquer coisa. O tempo de trabalho focado e de pausa são determinados pela dupla, mas geralmente é usado o tempo-padrão da técnica do pomodoro: 25 minutos de trabalho focado e 5 minutos de des-

canso. Com isso, o par tem ciclos de 30 minutos. A cada quatro ciclos, a dupla faz uma pausa longa de aproximadamente 40 minutos, tempo suficiente para responder e-mails, atualizar a sua rede social ou fazer aquela transação no seu iBanking.

Um ponto interessante é tentar evitar que interrupções externas atrapalhem o par. Caso seja frequentemente interrompido por terceiros, essa técnica sugere tomar nota das constantes interrupções para que posteriormente ações sejam tomadas visando minimizá-las.

Benefícios

Fora os benefícios que a psicologia por trás da técnica do pomodoro já provou, também há a vantagem de essa técnica fazer com que a programação em par não pareça uma prisão para seus membros. Eles não precisam ficar pensando em como pedir um tempo para fazer algo pessoal, já que ambos sabem que estão sempre há no máximo 25 minutos de uma pausa. Essa técnica ajuda com coisas simples, como ir ao banheiro ou tomar um café. Além disso, a técnica ajuda a conseguir fazer aqueles dois membros mais antagônicos da equipe trabalharem juntos, pois o workaholic sabe que vai ter de parar de vez em quando e o procrastinador sabe que terá de se focar por um período.

Dificuldade de implantação: Baixa

O site da técnica do pomodoro tem bastante conteúdo que pode ajudar a entendê-la melhor. Para começar, é necessário um timer, que pode ser virtual. O par configura o timer para um período de 25 minutos para fazer algo e 5 minutos para descanso. Uma dica interessante é usar os ciclos do pomodoro com a técnica do relógio de xadrez.

Sua ideia primeiro

Quando tiver uma divergência entre qual caminho deve ser seguido pela dupla, um dos membros diz: “Vamos tentar sua ideia primeiro”. Com certeza haverá muitos momentos em que os membros do par não concordarão sobre algo em relação ao que estão implementando. Nesse momento, no entanto, não deve ser criada uma briga de egos, já que o objetivo de parear não é mostrar que um dos membros é melhor do que o outro, mas sim produzir o melhor dos dois. Ter momentos sem consenso não é necessariamente ruim. Na verdade, eles são benéficos, uma vez que exigem que as ideias sejam vali-

dadas, e acabam sendo um dos sinais de uma boa sessão de programação em par. Nesses momentos, geralmente o mais experiente do par puxa essa técnica do bolso e diz: “Vamos tentar sua ideia primeiro”.

Benefícios

Garante que haja a troca entre os papéis do par e que não haja um piloto dominante. Cria uma boa convivência entre os membros da dupla, e pode ajudar a evoluir a empatia (capacidade de ver algo a partir ponto de vista de outra pessoa) nos participantes. Também ajuda a expandir a capacidade mental para entender outras soluções, que foram geradas por terceiros.

Aplicar essa regra também faz o membro ganhar pontos de confiança com o seu par, ambos vão se sentir mais confortáveis para continuar compartilhando ideias. Se a primeira sugestão selecionada não agradar, o integrante da dupla que aplicou a regra pode dizer: “Legal, deixa eu te mostrar em que eu estava pensando”, e juntos poderão decidir qual ideia deve ser selecionada.

Dificuldade de implantação: Baixa

Não é necessário muito esforço para implantar essa técnica. Principalmente se os membros da dupla entenderem que ela beneficia muito quem a aplica, tanto mentalmente quanto socialmente.

Regra dos 10 segundos

Toda vez que o navegador perceber que o piloto está fazendo algo errado, ele conta mentalmente até 10 antes de interromper o piloto e dizer “Tá errado ali...”. Isso permite que o piloto desenvolva o raciocínio completamente e o trabalho fica mais fluido, sem tantas interrupções.

Benefícios

Essa técnica pode parecer simplória, mas pode ser a diferença entre conseguir implantar programação em par ou criar grandes inimizades com colegas de trabalho.

É muito chato ter alguém do lado falando a todo momento coisas do tipo “Esqueceu o ponto e vírgula”, “Faltou idantar”, “Tá certo isso?” etc. Até mesmo um simples “Hum...” pode ser algo que incomoda quando alguém está tentando resolver um problema. Por isso, é importante dar tempo para quem está no teclado rever o que acabou de digitar. Não precisa esperar até que o compilador ou o teste unitário pegue um erro que o navegador eficiente já

identificou. Mas dar uns 10 segundos para que o piloto revise seu próprio código e pensamento já alivia o clima entre o par.

Dificuldade de implantação: Média

Pode parecer fácil, mas não é. Muitas pessoas estão acostumadas a pensar e já falar ou mesmo falar sem pensar. Essa técnica exige que se pense em algo e decida quando falar, o que não é trivial. Alguém que começou com programação em par recentemente pode ter muita dificuldade em usar essa técnica e o seu domínio pode levar algum tempo.

Algo que ajuda é o piloto ter essa regra em mente enquanto pilota, assim ele pode gentilmente lembrar o seu navegador da regra dos 10 segundos quando as interrupções começarem a incomodar. Um ponto importante é: essa regra não deve ser usada para inibir o navegador de ajudar ou esclarecer dúvidas. Ele pode e deve fazer perguntas ou dar conselhos enquanto o piloto digita. O foco dessa regra deve ser apenas criar um ambiente em que o piloto se sinta confortável para errar e corrigir seus erros.

Ping-pong

O piloto escreve um teste sobre o próximo *baby step* no qual a dupla está trabalhando. Ele executa esse teste e o teste deve falhar (já que usando *Test Driven Development* (TDD), o código de teste foi escrito antes do código de produção). Nesse ponto é feita a troca de papéis e o objetivo do par, agora com um novo piloto, é fazer aquele teste que está quebrando passar. Nesse ponto, pode ser feito um *refactoring* sobre o código, fechando assim um ciclo completo de TDD. Ainda antes de fazer a troca de papéis, a próxima responsabilidade do par é escrever um novo teste que represente o próximo *baby step* que a dupla fará e, assim, o ciclo recomeça.

Benefícios

Ajuda a manter os dois membros do par com um bom conhecimento do caminho que deve ser seguido. Por envolver o TDD, essa técnica é ótima para transferir conhecimento sobre o TDD de um desenvolvedor mais experiente para um com menos experiência nessa técnica.

Dificuldade de implantação: Alta

É interessante que pelo menos um dos membros da dupla já tenha experiência com TDD, caso contrário a dupla terá de aprender duas coisas ao

mesmo tempo. Se pelo menos um dos dois membros já tem experiência com TDD, o foco pode ficar apenas no tamanho do próximo *baby step* e garantir que os dois estejam entendendo o objetivo do próximo teste. Além disso, um ponto que pode atrapalhar a implantação dessa técnica é a aderência do código ao TDD, já que não é todo código que irá facilitar o seu uso.

8.6 CONCLUSÃO

Nós, da ThoughtWorks, acreditamos em programação em par e que várias empresas podem se beneficiar dessa técnica. Há muito material com detalhes de pesquisas que mostram a efetividade da programação em par e hoje também existem vários cases documentados, mostrando como essa técnica pode trazer ganhos substanciais às equipes de desenvolvimento de software.

Quanto à adoção, as técnicas aqui citadas podem facilitar o processo. Vale a pena começar pelas práticas mais fáceis e ir evoluindo para as técnicas mais difíceis. Depois de um tempo, essas técnicas passarão a fazer parte do dia a dia e tendem a fluir sem muito esforço, o que torna as sessões de programação em par algo mais humano e prazeroso.

Algo que pode ajudar é tentar aplicar essas técnicas em sessões de *Coding Dojo*, em que a(s) dupla(s) pode(m) treinar o que vai ser feito em uma sessão de programação em par “real”.

CAPÍTULO 9

Vários projetos, objetivos diferentes, mas o código em comum

por Adriano Bonat e Carlos Lopes

Muitos projetos de desenvolvimento de software começam pequenos mas crescem rapidamente, seguindo a evolução das empresas que os implementam. Isso leva a um cenário de vários times trabalhando em projetos distintos, porém relacionados por uma mesma base de código.

Repositórios e sistemas de controle de versão ajudam a gerenciar várias pessoas e times que trabalham simultaneamente em uma mesma base de código. Na última década, muitos grupos se organizaram para trabalhar com

uma de suas funcionalidades mais exploradas: *branching*.

Neste capítulo, vamos ilustrar alguns problemas atualmente enfrentados pelas soluções de *branching*, bem como oferecer uma opção baseada em *trunk based development* e *feature toggles* para o desenvolvimento de diversos projetos concorrentes em um mesmo repositório, com o benefício de descobrir e solucionar problemas de integração muito mais rapidamente.

9.1 PROBLEMAS COM BRANCHING

“Feature Branching is a poor man’s modular architecture, instead of building systems with the ability to easy swap in and out features at runtime/deploytime they couple themselves to the source control providing this mechanism through manual merging.”

– Dan Bodart

Bodart diz, em tradução livre, que “o uso de *branches* é uma maneira pobre de estruturar um sistema, já que, em vez de desenvolver um sistema em que seja fácil adicionar e remover funcionalidades, isso é feito com *merges* manuais através do controle de versões”.

Para ilustrar uma situação com problema de *branching*, pode-se imaginar dois times completamente distintos trabalhando nos projetos B e C, por meses, sem nenhum tipo de comunicação. As duas funcionalidades são bastante importantes e cruciais para o lançamento da versão 2.0 do produto.

Entretanto, para que a versão possa ser efetivamente lançada, as duas funcionalidades precisam conviver em harmonia em uma única base de código que, nesse caso, é o *trunk* (representado pela linha escura central na Figura 9.1). Todavia, essas duas funcionalidades têm bastante código compartilhado, isto é, fazem uso de muitas classes em comum que, consequentemente, foram alteradas inúmeras vezes durante o desenvolvimento de ambas as funcionalidades, sendo que cada uma realizou modificações pertinentes aos seus objetivos. O que deve, pois, acontecer para que a versão 2.0 seja lançada?

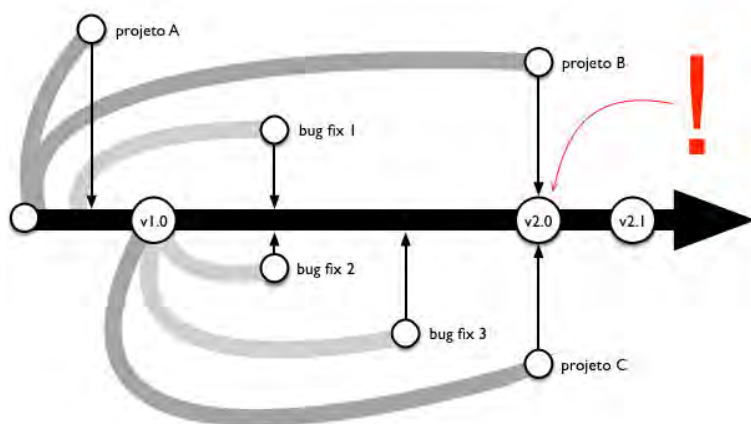


Fig. 9.1: Projetos sendo desenvolvidos em diferentes *branches*

9.1.1 Merge hell

Para o lançamento da versão 2.0, certamente seria preciso fazer com que finalmente as duas equipes trabalhassem em conjunto para realizar o *merge* de todo o código que foi desenvolvido de maneira isolada, ou seja, sem qualquer tipo de comunicação, durante meses. Entretanto, o cenário não é nada animador: cada projeto (B e C) não tem noção alguma sobre as mudanças realizadas pelo outro; o projeto B nunca teve conhecimento do código implementado pelo projeto A (pois o *branch* foi criado antes do *trunk* conter o código deste último), e os *bug fixes* realizados para a “suposta” versão 2.0 também podem ser causa de grandes mudanças.

Assim, a necessária integração entre os projetos poderá levar semanas ou meses, resultando em um processo extremamente doloroso, visto que grande parte do código envolvido é comum entre as duas funcionalidades. Mesmo que a integração seja finalizada, caso não se tenha uma cobertura de testes abrangente para todas as funcionalidades envolvidas, a chance de regressões e *bugs* é muito grande. Além disso, os *branches* (e inclusive o *trunk*) podem ficar bastante tempo em um estado “quebrado”, situação em que o código não pode ser implantado. A Figura 9.2 mostra um cenário com apenas 4 *branches*

e todas as dificuldades trazidas por eles.

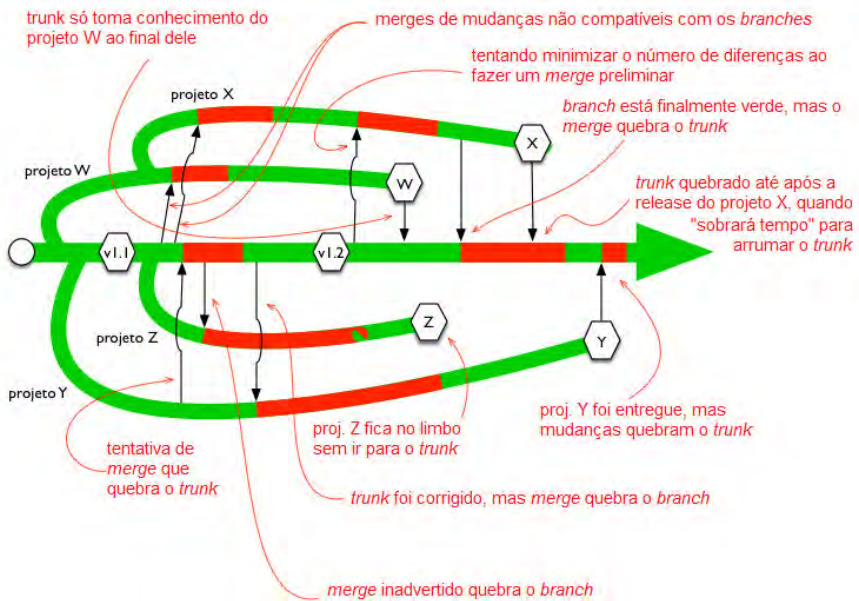


Fig. 9.2: Código fica muito tempo sem poder ser entregue

9.1.2 Merge monkey

Em cenários como esse, é comum se designar uma pessoa ou uma equipe (dependendo do tamanho dos projetos envolvidos) para atuar como *merge monkeys*. Mas o que esses “macacos” fazem?

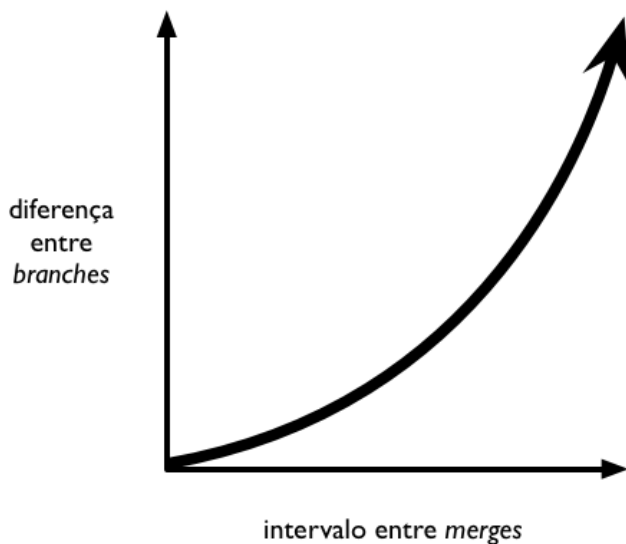


Fig. 9.3: O custo cresce exponencialmente quanto mais tempo fica sem realizar *merges*

Durante o tempo em que se fica sem realizar *merges*, o custo do projeto cresce exponencialmente.

Os *merge monkeys* têm o trabalho deveras ingrato de deixar a integração menos dolorosa, realizando *merges* frequentes entre todos os *branches* envolvidos.

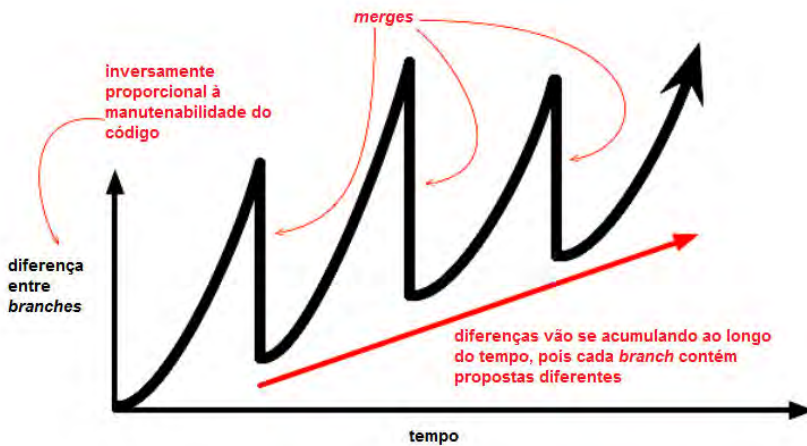


Fig. 9.4: O *merge monkey* reduz o risco, mas está longe de resolver todos os problemas

Essa prática certamente deixaria o lançamento da versão 2.0 um pouco menos doloroso, mas tornaria o processo de desenvolvimento um tanto quanto caótico, visto que, se há um time separado com a responsabilidade de realizar esses merges, existem grandes chances de conflitos semânticos, que necessitarão dos dois times envolvidos para serem resolvidos. Tal situação pode ser difícil, pois é comum os times estarem sob grande pressão para entregarem suas funcionalidades. Contudo, caso não haja o grupo de *merge monkeys*, o verdadeiro problema apenas estará sendo adiado.

9.1.3 Integração promíscua

Outra prática possível é a integração promíscua, em que diferentes *branches* realizam integrações entre si (conforme mostra a figura abaixo). Os problemas dessa prática estão fortemente ligados ao descaso em relação ao *trunk*, que deveria, em tese, ser o local onde o código da aplicação estivesse mais “saúdável”. No exemplo da figura, existem os projetos A e B sendo desenvolvidos em branches completamente distintos do *trunk*, que nunca é percebido pelos *branches*, já que estes somente se integram entre si próprios. Sendo assim, qualquer *bug fix* realizado no *trunk* passará despercebido pelos *branches*,

o que certamente causará complicações quando finalmente algum dos projetos resolver se integrar com o *trunk*.

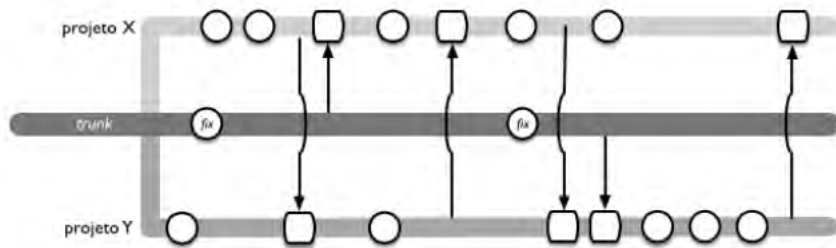


Fig. 9.5: Integração promíscua

9.2 TRUNK BASED DEVELOPMENT

Uma possível solução para evitar os problemas inerentes ao uso de *feature branches* é fazer com que todo o desenvolvimento de funcionalidades do time seja feito em um único *branch*, que geralmente é a linha principal ou o tronco (em inglês chama-se *trunk*). Por essa razão, esse tipo de prática é denominada *trunk based development (TBD)*. A utilização desse formato não descarta outra boa prática, a de fazer *commits* pequenos. Com desenvolvedores realizando mudanças pequenas muito mais frequentemente, os eventuais problemas de conflitos de integração (*merge*) de outras funcionalidades que estão sendo desenvolvidas serão minimizados. Isso acontece porque o delta de mudança é pequeno e, portanto, mais pontual, tornando fácil entender e resolver eventuais distorções.

Quando se está usando TBD, o processo de *deploy* (implantação) de uma nova versão do projeto em produção expõe tudo o que está no *trunk* naquela versão, enquanto com a utilização de *branches*, qualquer funcionalidade que ainda não está finalizada ou que ainda não foi homologada pelo time está fora do *deploy*, pois está fora do *trunk*.

9.2.1 Tipos de trunk

Existem diversos tipos de projetos e arquiteturas, e cada conceito apresentado ao longo deste artigo pode ou não aplicar-se à realidade de cada um. Dependendo do tipo de aplicação que está sendo desenvolvida, a abordagem utilizada para organizar o fluxo de trabalho no *trunk* pode mudar bastante e— o que é ainda mais comum — evoluir de um tipo para outro ao longo do tempo.

Trunk único, aplicação única



Fig. 9.6: Aplicação única, *trunk* único

O caso mais simples é aquele em que se tem uma aplicação única. A aplicação pode conter múltiplos módulos, mas estes são administrados como uma unidade. Os módulos não precisam necessariamente gerar um artefato único, mas, para fins de implantação, são tratados como um aglomerado.

É comum que projetos de software comecem adotando esse estilo, pois o processo de trabalho é bem simples (todo o código relacionado à aplicação

está em um só lugar) e atende muito bem os requisitos iniciais, o que geralmente significa lançar a versão inicial do produto.

Trunk único, várias aplicações

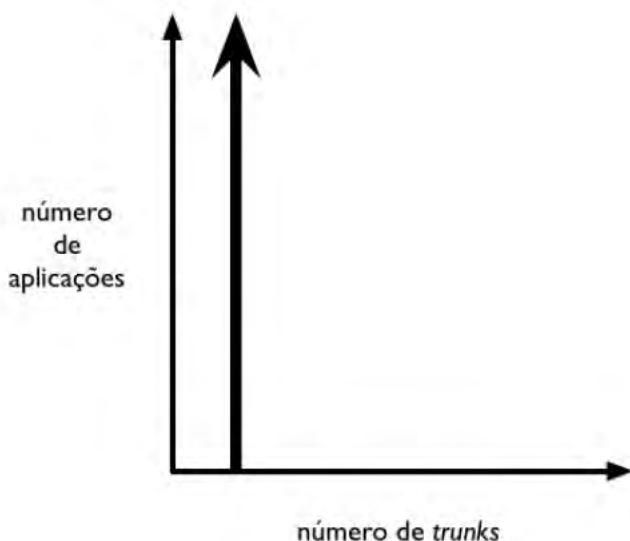


Fig. 9.7: Várias aplicações, *trunk* único

O caso em questão pode ser considerado uma evolução natural do anterior, quando, por exemplo, a aplicação tiver sido expandida obtendo diversas “ramificações”, mas ainda assim existir grande parte de lógica comum entre essas partes.

Um bom exemplo para retratar esse cenário é um sistema web de vendas para o varejo que seguia o modelo anterior (*trunk* único, aplicação única), que passa a incorporar uma aplicação para dispositivos móveis.

O maior benefício desse tipo de *trunk* é o fato de o código estar todo em um único lugar, fomentando a comunicação e a colaboração entre todos os times envolvidos e, conseqüentemente, minimizando os efeitos da *Lei de Conway*, que é o processo organizacional que inevitavelmente emerge

quando diferentes times são nominados “responsáveis” por certos módulos, fazendo com que a estrutura da aplicação fique muito similar à estrutura organizacional e passe a ditar como os grupos interagem. Por exemplo, se o time A for designado como o dono de certo módulo compartilhado, a tendência é que o módulo passe a ser tratado como um produto, tornando as mudanças necessárias pelos demais times mais lentas ou impossíveis, pois podem conflitar com as prioridades elencadas pelo time A.

Um ponto a ser observado é a estrutura de integração contínua. Como todo o código reside no mesmo local, é preciso ter diversos *pipelines* que garantam que mudanças — especialmente em código compartilhado — não quebrem nenhuma funcionalidade existente. Além disso, uma boa cobertura de testes é essencial nesse cenário (com especial atenção para testes de integração), pois se ela for baixa (ou simplesmente não existir) qualquer mudança se tornará problemática.

Outro detalhe muito importante dessa abordagem — e um dos desafios à medida que o número de módulos cresce — é que as aplicações devem, idealmente, ser implantadas ao mesmo tempo, podendo, inclusive, compartilhar *toggles*. Isso também torna o processo de integração contínua, e de implantação, muito mais simples, pois preocupações como compatibilidade entre versões, dependências entre diferentes módulos etc. não existem.

Vários *trunks*, várias aplicações

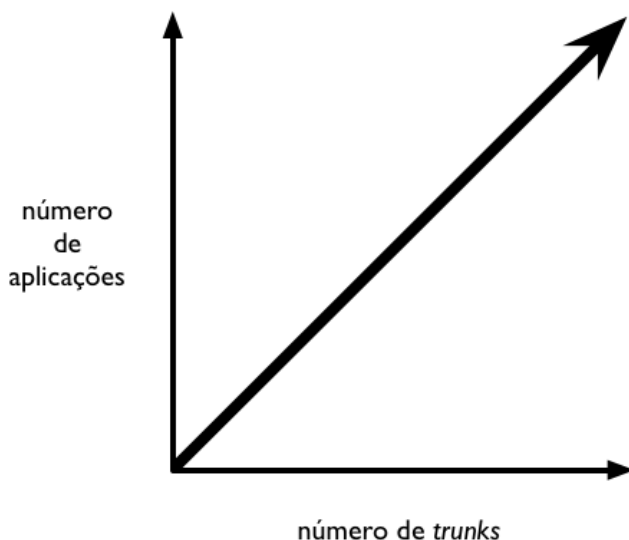


Fig. 9.8: Várias aplicações, vários *trunks*

Ainda utilizando o exemplo do sistema de vendas mencionado na seção anterior, pode-se imaginar que, com o tempo, ele cresceu, e agora conta com aplicativos para quatro diferentes plataformas de dispositivos móveis (além da versão web original) e também provê um canal de vendas por atacado, acessível tanto pela web quanto por todas as outras plataformas móveis. Com isso, tem 10 diferentes canais, que contam com uma grande base de código em comum, seja em forma de bibliotecas ou de serviços.

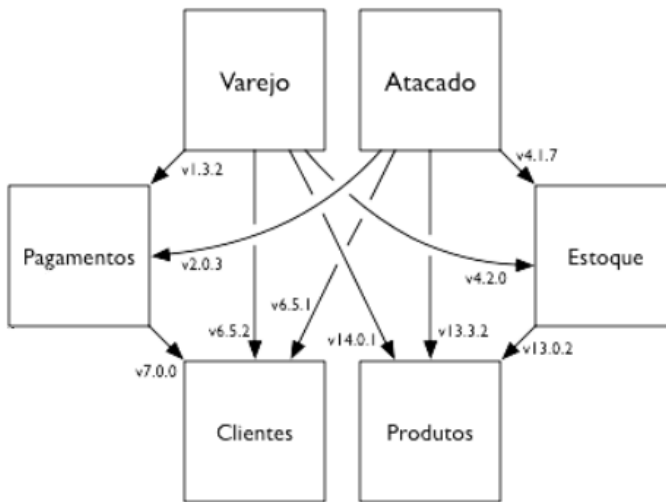


Fig. 9.9: Múltiplas dependências em diferentes versões de diferentes módulos

A estrutura desse sistema de vendas pode também contar com diversos times: várias equipes trabalhando nos serviços/bibliotecas de *backend*, bem como grupos distintos atuando nas aplicações para o varejo e para atacado. Com essa nova estrutura, várias mudanças podem ocorrer. Uma das mais inevitáveis é a segregação das equipes: como agora cada time tem módulos que pode chamar de seus, tende a trabalhar nos seus próprios “silos”, em um cenário onde a Lei de Conway é facilmente observada, o que pode, inclusive, significar que membros de diferentes times não possuem nem acesso de leitura ao código dos demais. É preciso tomar muito cuidado para que as equipes trabalhem juntas, focando em um mesmo objetivo, já que é muito comum cada time se fechar e criar obstáculos para que mudanças sejam incorporadas em seus módulos, tornando a colaboração entre os diferentes grupos muito rara. Desafios como acesso a versões de teste (seja de serviços ou arquivos binários) entram em questão: qual deve ser o processo para um time obter a versão de uma dependência produzida por outro time? De quem é a responsabilidade de compilar o código e de prover um servidor de testes ou um repositório de binários?

As implantações, nesse caso, tornam-se muito mais complicadas, principalmente devido à necessidade de gerenciamento de versões. O mesmo se aplica à integração contínua: como garantir que todas as combinações de diferentes versões funcionem em harmonia? São necessários vários *pipelines* que rodem diferentes combinações de versões para garantir a confiança de atualização de apenas um dos serviços desse ecossistema sem que nada pare de funcionar.

9.2.2 Feature toggles

Ao utilizar TBD, as funcionalidades incompletas e não aprovadas já estarão no *trunk*; como consequência, elas ficam acessíveis e visíveis para os usuários após o *deploy*. Isso não é algo desejável na maioria dos casos, sendo a solução mais comum o uso de *feature toggles*, que funcionam como interruptores para as funcionalidades. O uso dessa técnica seria desnecessário caso a implantação ocorresse apenas quando todo o trabalho do ciclo de desenvolvimento estivesse completo e aprovado, porém, como se vê mais à frente, a introdução de *feature toggles* tem alguns outros benefícios que podem ser úteis em determinados cenários, como detectar algum problema em uma das novas funcionalidades que foram colocadas em produção.



Fig. 9.10: A funcionalidade controlada pelo interruptor pode estar ligada ou desligada

Um *feature toggle* é, simplificando o conceito ao máximo, nada mais do que um condicional. Por exemplo:

```
if toggles.novaFuncionalidade?  
  novaFuncionalidade.executarCodigoParaNovaFuncionalidade  
end  
...
```

Esse código que controla as funcionalidades é uma complexidade adicional, sendo assim, quando houver uma decisão sobre a funcionalidade, deve-se remover o toggle e manter o seu código simples. Da mesma forma como o código funcional e o código de testes devem ser refatorados, o código relacionado aos toggles também deve ser modificado, melhorando seu entendimento e evitando a deterioração dos toggles.

O capítulo 10 aborda em mais detalhes o conceito de *feature toggles*.

9.3 BRANCHES SÃO DO MAL?

Muito pelo contrário. O problema do uso de *branches* emerge somente quando são utilizados de maneira equivocada ou em excesso ou, como por exemplo para “isolar” dois ou mais times por um longo período de tempo com a ilusão de que isso os fará desenvolver mais rápido — o que pode até ser verdade de início, mas ao chegar o momento para integrá-los, todo este tempo que foi economizado será cobrado a juros altos. Uma maneira mais inteligente de trabalhar com *branches* pode ser, por exemplo, através da combinação do uso de *branches* de curta duração e *pull requests*, de modo que, tão logo os desenvolvedores estejam contentes com a funcionalidade, podem enviar um conjunto de *commits* para o *branch* de onde a aplicação é efetivamente implantada. Isso possibilita que o trabalho seja realizado isoladamente por um curto período de tempo e traz alguns outros benefícios como revisões de código mais frequentes, já que se pode criar uma prática na qual outro par de desenvolvedores (não os que trabalham na funcionalidade) seja responsável por realizar o *merge* e, ao mesmo tempo, verificar que o código escrito siga os padrões definidos pelo time.

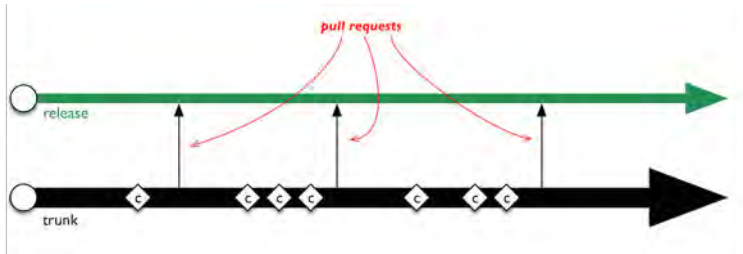


Fig. 9.11: *Pull requests* para um *branch* específico

9.4 CONCLUSÃO

A utilização ingênua do recurso de *branching* no desenvolvimento de múltiplas funcionalidades em paralelo encoraja os desenvolvedores a trabalharem de maneira isolada, causando problemas e desconfortos como *merge hell*, *merge monkey* e a integração promíscua. Para complicar ainda mais o cenário, muitos desses obstáculos serão notados somente pouco antes de o sistema ir para a produção.

Trunk based development e *feature toggles* são possíveis alternativas que possibilitam o desenvolvimento de diversos projetos concorrentes no mesmo repositório, com o benefício de se poder descobrir e solucionar problemas de integração muito mais rapidamente. Tais ferramentas também permitem uma maior flexibilidade quanto à entrega ou não de determinada funcionalidade, já que elas serão controladas por interruptores, podendo ser ligadas ou desligadas em produção.

CAPÍTULO 10

Dissecando *feature toggles*

por Adriano Bonat e Carlos Lopes

Supondo que você entregue novas funcionalidades com certa regularidade, é bastante comum que algumas delas demorem mais que um ciclo para serem finalizadas e, portanto, precisam ficar escondidas do usuário por algum tempo até que estejam totalmente prontas. Conforme descrito no capítulo 9, o uso de *branches* é uma possível saída para este problema, porém este mesmo capítulo também fala sobre como usá-los em excesso pode ser bastante prejudicial para a habilidade de entrega em times de desenvolvimento. Uma das técnicas para evitar o uso exagerado de *branches* é a utilização de *feature toggles*, que são, de maneira simplificada, nada mais que interruptores responsáveis por mostrar ou esconder certas funcionalidades do software em questão. A teoria por trás destes interruptores é simples. Entretanto, há variações im-

portantes sobre a sua aplicação e objetivos. Neste capítulo, veremos em mais detalhes o que são *feature toggles* e os benefícios e impactos de utilizá-los.

10.1 FEATURE TOGGLES NA PRÁTICA

Feature toggles são, na prática, interruptores responsáveis por esconder certos trechos de código de determinada aplicação. Tecnicamente falando, estruturas condicionais são a implementação mais simples de um *feature toggle*. Por exemplo:

```
if toggles.novaFuncionalidade?  
    novaFuncionalidade.executarCodigoParaNovaFuncionalidade  
end  
...
```



Fig. 10.1: *feature toggles* nada mais é do que interruptores responsáveis por ligar ou desligar certas partes da aplicação

10.2 PRECAUÇÕES

É importante que a estratégia para utilização de *feature toggles* seja definida o mais cedo possível, a fim de minimizar o impacto dos mesmos no código, pois, caso o uso não seja controlado, eles podem acabar se tornando um problema maior do que o que se propõem a resolver. Medidas simples, como a criação de métodos específicos no estilo `isFooEnabled` em vez de um gené-

rico `isEnabled("foo")` (tratando-se de linguagens estaticamente tipadas) podem facilitar e muito a manutenção dos *toggles* ao longo do código, ainda mais quando seu uso for extensivo, porque, com isso, pode-se contar com a ajuda da IDE para detectar referências no processo de *refactoring* e remoção de *toggles*.

Também é importante pensar em estratégias para centralizar o acesso aos dois “caminhos” que a aplicação pode tomar quando se utiliza um *feature toggle*. Por exemplo, imagine o cenário onde um novo layout será adicionado a uma aplicação já existente. Imagine também que o layout existente se baseia todo na cor azul, e a nova funcionalidade adicionará a opção de o usuário escolher entre as cores azul e vermelha. Ao esconder do usuário em produção apenas o seletor de cores e deixando como padrão o leiaute antigo, evita-se que o condicional deste *toggle* tenha que ser repetido em muitos lugares.

10.3 TIPOS DE *FEATURE TOGGLES*

É importante que quando um toggle for introduzido na aplicação o seu objetivo seja definido claramente, pois essa definição ajudará na conversa entre os diferentes participantes do projeto a entender melhor o propósito e características do toggle em questão. Por exemplo, em que situações ele vai estar ligado ou desligado? Quando, e se, ele vai ser removido do projeto? A definição do tipo de toggle ajuda a evitar o surgimento de alguns antipadrões (*anti-patterns*) na utilização dos mesmos, como toggles que ficam desnecessariamente muito tempo no código da aplicação, tornando a manutenção mais complexa. Por exemplo:

```
if toggles.programaDeFidelidade?
  # Toggle para o programa de fidelidade está habilitado...
  if toggles.novoSistemaDePagamento?
    # Temos que utilizar o novo sistema de pagamentos...
  else
    # O novo sistema de pagamentos ainda não está pronto...
else
  # O programa de fidelidade está desabilitado...
  if toggles.novoSistemaDePagamento?
    # Temos que utilizar o novo sistema de pagamentos...
```

```
else
    # O novo sistema de pagamentos ainda não está pronto...
```

Como cada toggle tem apenas dois estados (ligado ou desligado), o número de combinações lógicas possíveis será 2 elevado à enésima potência, em que n é o número de toggles envolvidos. No caso do nosso exemplo, temos 4 cenários que precisam ser tratados e testados. Eles também podem, possivelmente, ser refatorados para compartilhar código, evitando a proliferação da mesma lógica.

Na seção 10.4, serão abordadas algumas estratégias para a organização de uma aplicação de modo a evitar um aumento desnecessário na complexidade do código.

10.3.1 Toggle de desenvolvimento

Os desenvolvedores são os principais usuários desse tipo de toggle. Ele serve primariamente para esconder funcionalidades incompletas que ainda estão em desenvolvimento. Assim que a implementação estiver pronta e aprovada para ir para produção, ele poderá ser removido e a funcionalidade que foi desenvolvida estará integrada à aplicação.

Outra maneira de se liberar uma nova funcionalidade que está escondida por um toggle é simplesmente ligá-lo em produção, preservando-o por mais algum tempo. A vantagem desse tipo de abordagem é que, caso a nova funcionalidade apresente algum problema de implementação, segurança ou performance, ela poderá ser desativada através do toggle, dando tempo para o time de desenvolvimento efetuar as correções necessárias. Logo que elas estiverem em produção, o toggle poderá ser ligado novamente.

O time deve tomar cuidado para que o toggle não fique sem necessidade por um longo tempo na aplicação — ele deve ser removido assim que a funcionalidade estiver pronta e estável. Esse cuidado é muito importante, pois elimina a complexidade adicional que eles inevitavelmente agregam.

10.3.2 Toggle de operações

Nesse tipo de toggle, quem mais se beneficia é o time de operações. Algumas vezes a sua aplicação possui alguma funcionalidade que depende de algum

sistema externo, e esse sistema pode estar temporariamente indisponível. Seria desejável que a sua aplicação continuasse operando, porém sem a funcionalidade com essa dependência externa. Para isso, o time de operações pode simplesmente desligar o toggle que controla essa parte da aplicação até que o sistema do parceiro seja restabelecido. O desligamento pode até mesmo ser realizado automaticamente pela aplicação, caso ela seja capaz de monitorar quando o serviço externo se torna indisponível — padrão este que, por vezes, é conhecido como “disjuntor” (*circuit breaker*).

Esse tipo de *toggle* tem um tempo de vida maior que o de desenvolvimento, ele existirá enquanto a sua aplicação tiver essa dependência externa que pode apresentar alguma instabilidade temporária.

10.3.3 Toggle de negócio

Utilizados para alternar comportamentos da aplicação, o seu estado é diretamente manipulado pelo time de negócio, os chamados *stakeholders*. A aplicação pode oferecer um painel de controle para esses *toggles* na sua área de administração, assim o time de negócio vai poder ligar ou desligar funcionalidades, baseando-se em métricas de vendas e lucro, estratégias de oportunidade etc. O estado do *toggle* também pode variar sazonalmente, por exemplo, quando determinada funcionalidade só vai estar disponível em determinadas datas no ano.

Outra possibilidade é fazer a segmentação de usuários baseado no estado do toggle (prática também conhecida como teste A/B). Um grupo de usuários tem certa funcionalidade habilitada, enquanto outros não percebem diferença alguma na aplicação. Dessa maneira, o time de negócios pode efetuar comparações entre o comportamento desses dois grupos de usuários, e, por meio desse experimento, implementar melhorias na aplicação. É importante ressaltar que assim o toggle não afeta o sistema como um todo, mas somente a sessão de determinados usuários.

Também é importante distinguir este tipo dos demais, já que este não deve ser removido inadvertidamente e seu tempo de vida é uma decisão do time de negócio, pois pode ser parte importante do sistema e crítico para seu sucesso.

10.4 ARQUITETURA DA APLICAÇÃO

A explosão combinatorial deve ser encarada como um dos principais *code smells* ocasionados pelo uso de *feature toggles*. É importante que sua aplicação não se torne um imenso aglomerado de *feature toggles*. Geralmente, esse *smell* não é detectado cedo, pois no início o uso de toggles é visto como apenas um benefício, mas pode facilmente sair de controle. Exemplos clássicos são estruturas condicionais repetidas por diversas partes do código.

Podemos eliminar o uso de condicionais utilizando uma composição e o recurso de polimorfismo das linguagens orientadas a objeto. Refatorando o exemplo anterior:

```
class Pedido
  def initialize(sistemaDePagamento)
    @sistemaDePagamento = sistemaDePagamento
  end

  def processa
    @sistemaDePagamento.cobra!
  end
end
```

Dessa forma, `Pedido` simplesmente delega a responsabilidade de cobrança do pedido para o sistema de pagamento. A parte de sistema de pagamento teria duas implementações, a antiga e a nova, e ambas implementam a mesma interface:

```
class SistemaDePagamentoAntigo
  def cobra!
    # Cobra através do antigo sistema de pagamento
  end
end

class SistemaDePagamentoNovo
  def cobra!
    # Cobrança feita através do novo sistema de pagamento
  end
end
```

A decisão de qual sistema de pagamento será utilizado vai ser tomada pela biblioteca de injeção de dependência, ou por meio de uma fábrica de instâncias de `Pedido`, sendo que ambos consultarão o estado do *feature toggle* associado à funcionalidade do novo sistema de pagamentos.

É importante ressaltar que as implementações utilizadas como exemplos nesta seção são apenas sugestões, cujo foco é mostrar que *feature toggles* não é um conceito complicado de ser posto em prática. Em alguns casos, o mecanismo de controle de toggles pode ser um módulo por si só, como, por exemplo, em casos em que testes A/B sejam frequentes, pois o mecanismo de controle dos estados de cada toggle é essencial, portanto passa a ser tratado como um “cidadão de primeira categoria”, sendo muito apreciado pela área de negócios e não apenas uma técnica para ajudar o time de desenvolvimento.

10.4.1 Conteúdo estático

As possíveis implementações que vimos até então são pertinentes para o lado do servidor das aplicações (*backend*), mas não se aplicam tão naturalmente no lado cliente, para conteúdo estático como folhas de estilo CSS ou JavaScript.

A maneira mais simples de resolver este problema seria mover todo o conteúdo estático para ser gerado no lado do servidor, o que possibilitaria o uso do mesmo tipo de condicionais que o *backend* e as estratégias mostradas anteriormente. Esta abordagem pode ser a solução para aplicações internas ou com poucos acessos, mas é inviável para aplicações web com muitos acessos, pois este tipo de conteúdo é frequentemente *cacheado* ou hospedado em uma CDN.

CONTENT DELIVERY NETWORK (CDN)

Content Delivery Network (CDN) (Rede de Fornecimento de Conteúdo, em português) é um sistema interligado de servidores de cachê que cooperam de modo transparente para fornecer conteúdo a usuários finais na internet. Para tanto, sistemas CDN usam a proximidade geográfica de servidores de cachê como um dos critérios para a entrega de conteúdo na web, otimizando e acelerando sua distribuição.

Outra opção é gerar todos os artefatos estáticos (ou somente aqueles que façam uso de toggles) em tempo de *build* (ou *deploy*). O grande problema desta abordagem é que, para alterar o estado de um toggle, é preciso recompilar e/ou reimplantar a aplicação, o que em muitos ambientes pode ser inviável.

Uma possível estratégia para contornar os problemas citados é gerar duas versões de cada artefato: uma com a funcionalidade ativada e a outra com ela desativada. Dessa maneira, a versão adequada é apenas referenciada pela parte dinâmica da aplicação, permitindo que o conteúdo seja hospedado em uma CDN, ou então que o estado do toggle seja alterado em tempo de execução sem a necessidade de reimplantação.

No caso específico de JavaScript — que permite o uso de condicionais —, é possível também utilizar uma solução híbrida, na qual temos um artefato renderizado no lado do servidor que contém apenas o estado do toggle (possivelmente uma função no estilo `isFeatureOn()`) e os demais se utilizam desta função para decidir o código que deve ser executado. Essa estratégia torna funcionalidades em desenvolvimento (ou desligadas propositalmente) visíveis publicamente, para pessoas com algum conhecimento técnico, o que pode não ser aceitável em alguns casos. Nesse caso, a geração de múltiplos artefatos (um para cada estado do toggle) é uma solução plausível.

10.5 QUANDO REMOVER FEATURE TOGGLES

Essa é uma questão que não tem uma única resposta, mas, considerando que *feature toggles* são, afinal de contas, código extra, o ideal é que os mesmos sejam removidos o quanto antes, pois menos código significa menos complexidade e, assim, menor oportunidade para defeitos.

Logicamente, é importante levar o tipo de toggle em consideração. No caso de toggles de desenvolvimento, eles podem (e devem) ser removidos tão logo a funcionalidade começar a ser utilizada em produção. É comum times definirem um período de “validação” da funcionalidade antes de remover o toggle, deixando assim aberta a possibilidade de desativar a mesma caso ela gere algum problema no sistema. Uma prática comum para que ele não fique esquecido e nunca seja removido é adicionar à sua declaração um “tempo

de vida”, que pode ser uma estimativa bem rudimentar de quando se espera que a funcionalidade não necessite mais dele. Uma vez que se tem essa informação, pode-se ter algum processo que notifique o time periodicamente sobre toggles que, a princípio, não deveriam mais existir. Implementando algum processo simples como esse, pode-se garantir que não se gere uma lista imensa de toggles em um curto espaço de tempo, facilitando a vida de todo o time.

Já nos casos de toggles de operação ou de negócios, isso pode não ser tão simples, uma vez que eles consistem em importantes fatores da aplicação e muito provavelmente sejam desejados durante toda a vida dela. Nesses casos, talvez seja mais interessante distingui-los utilizando outro tipo de abstração, algo mais próximo a uma configuração.

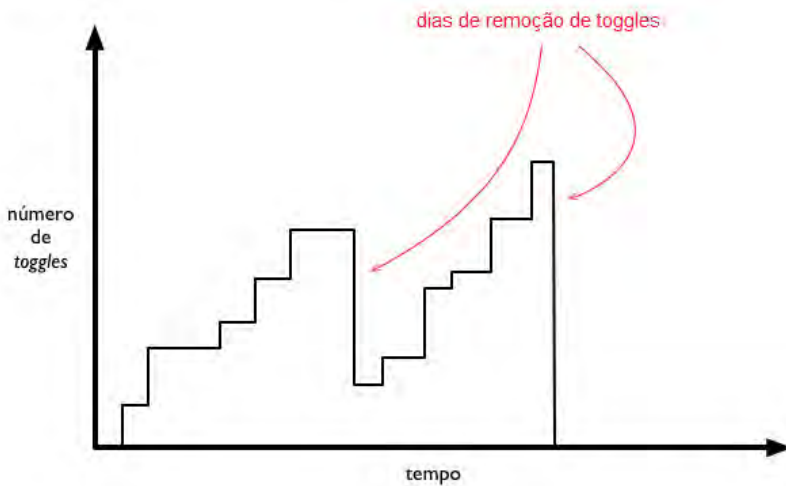


Fig. 10.2: Toggles sendo removidos esporadicamente e em grandes quantidades

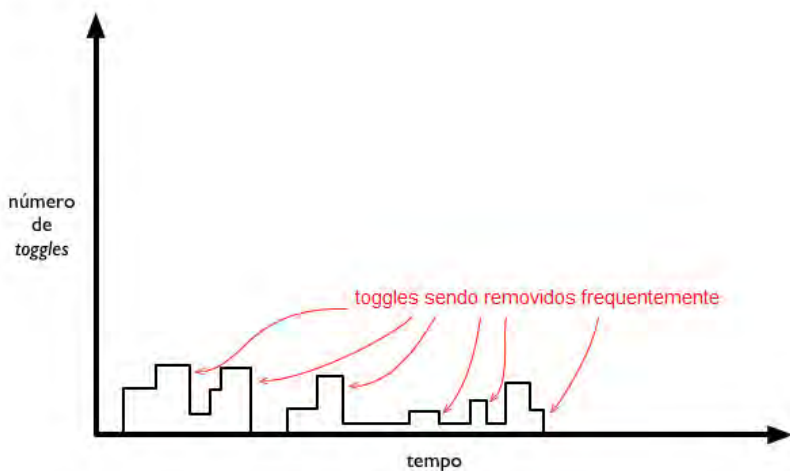


Fig. 10.3: Toggles sendo removidos constantemente, removendo complexidade desnecessária

É interessante notar que os toggles de operações e negócios tendem a ter uma vida mais estável que os de desenvolvimento, pois estes perdem seu valor assim que a funcionalidade relacionada está pronta.

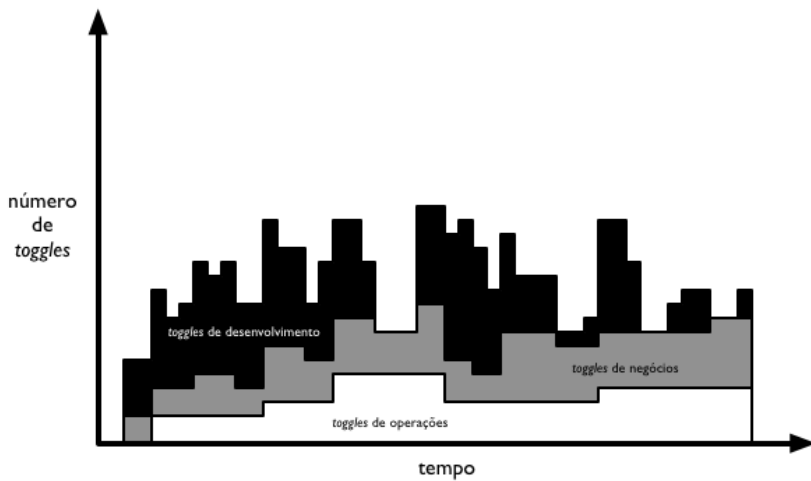


Fig. 10.4: Exemplo do número de toggles de diferentes tipos ao longo do tempo

10.6 CONCLUSÃO

Se você pretende melhorar o controle e eficiência com que desenvolve, opera e faz entregas, especialmente de forma contínua, você precisa ter o conceito de *feature toggles* como seu aliado. Como demonstrado ao longo do artigo, é uma técnica muito simples, porém extremamente efetiva, para ajudar times a entregar funcionalidades continuamente, mesmo em bases de código grandes, não só contribuindo no dia a dia dos times de desenvolvimento, mas também facilitando muito a vida de times de negócios e operações, por exemplo.

CAPÍTULO 11

Estudo de caso: automação como primeiro passo para entrega contínua

por Marcos Brizen

A história aqui compartilhada mostra como uma equipe melhorou o processo de implantação, passando de uma atuação manual, complexa, frágil e extremamente demorada, para uma sistemática em que uma única linha é executada e todas as etapas ocorrem de maneira simples, sendo os problemas facilmente identificados e recuperados e o tempo de execução praticamente constante e curto.

A entrega contínua está cada vez mais se consagrando como a prá-

tica que guiará times maduros para o futuro. Atualmente, a mesma equipe que adotou testes automatizados e configurou um servidor de integração contínua para aumentar a qualidade do produto final busca automatizar desde testes até a implantação do sistema, garantindo o máximo de qualidade e previsibilidade nas tarefas da equipe, principalmente nas mais críticas.

Então, como uma equipe madura, que utiliza testes automatizados, integração contínua, iterações curtas e várias outras práticas ágeis pode adotar a entrega contínua? Infelizmente, ainda não existe uma resposta simples e direta, mas muitas equipes estão em busca dessa dinâmica, cada uma à sua maneira. O objetivo deste texto é mostrar um caso em que a equipe busca realizar a entrega contínua, empregando a automação como primeiro passo.

11.1 EM BUSCA DA ENTREGA CONTÍNUA

Desde que as práticas de XP foram compartilhadas, desenvolvedores de software buscam melhorar a qualidade do seu trabalho através de técnicas como testes automatizados e desenvolvimento orientado a testes (TDD). Em um segundo momento, essas equipes caminham para a integração contínua, sendo que cada pedaço de código passa por todos os testes para garantir que problemas sejam detectados o mais rápido possível, fornecendo uma visão centralizada de qualidade e possibilitando que os problemas sejam expostos para toda a equipe.

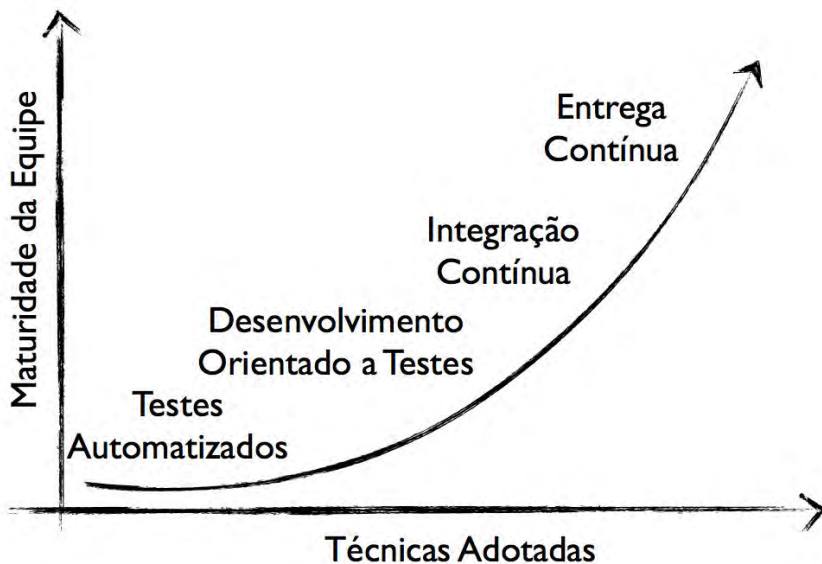


Fig. 11.1: Evolução das técnicas de desenvolvimento de software

Uma característica chave da entrega contínua é que ela precisa do envolvimento de toda a equipe. Adotar testes melhora a qualidade do produto final, o que é uma tarefa primordial dos desenvolvedores. Nesse sentido, a entrega contínua procura tirar as dificuldades técnicas do caminho e fazer com que a equipe de negócios (*stakeholders*, clientes etc.) possa decidir quando entregar. E quem melhor do que ela para tomar tal decisão?

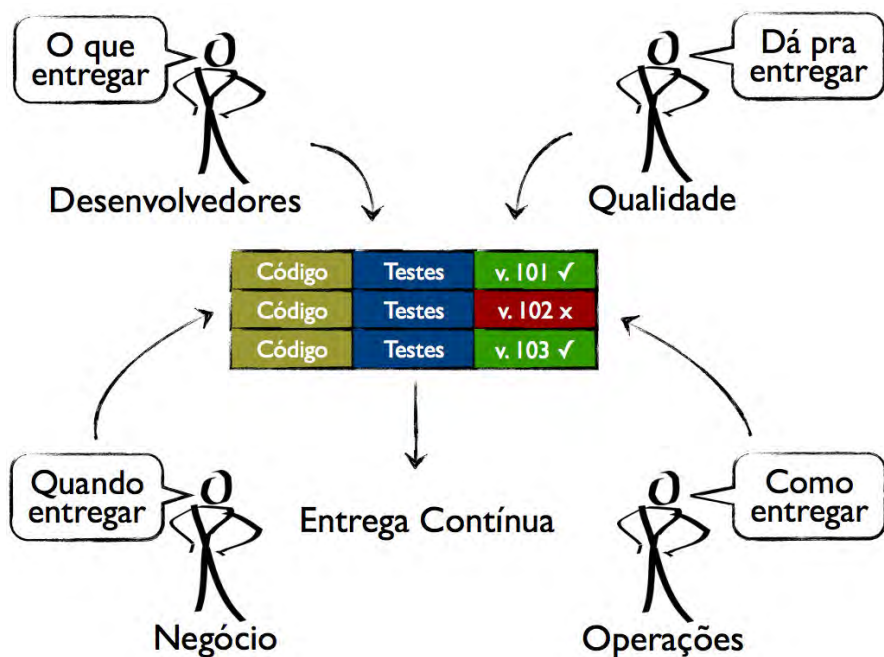


Fig. 11.2: Toda a equipe é envolvida para alcançar a entrega contínua

11.2 CONTEXTO

O projeto em questão existe há cerca de 6 anos e utiliza Ruby e Ruby on Rails, sempre com a mesma base de código. Atualmente, é relativamente simples dizer quais as melhores ferramentas para resolver problemas comuns, como testes automatizados, o que era diferente na época citada, quando a comunidade Rails estava começando a crescer e o framework dava seus primeiros passos. Muito do que hoje se tem como garantido, como tratamento de fuso horário, não era tão comum. Nessa perspectiva, apesar das tecnologias recentes, existia muita parte legada no projeto, sendo que apenas uma ou duas pessoas sabiam o que estava acontecendo.

Saindo um pouco da parte técnica, a equipe funciona em iterações com duração de 3 semanas e, ao final de cada uma delas, uma nova versão do sistema é implantada. Tal fato disciplina a equipe no processo de implantação e

permite que ela entregue valor constantemente.

Durante cada uma das iterações, são feitas as atividades de análise, de desenvolvimento e de testes em paralelo, com alguns dias focados para testes e resolução de problemas da nova versão. Antes de colocar a nova versão no ar, são realizados testes diários em ambientes de pré-produção e, por fim, de homologação, que é o mais semelhante possível ao ambiente de produção.

Antes de serem realizadas as melhorias de automação, o processo de implantação era basicamente manual. Assim, havia uma página com uma vasta documentação do processo, descrevendo os comandos que precisavam ser executados e o que poderia sair errado. É fácil imaginar o quão caótica era essa documentação, já que os comandos principais eram marcados em azul, os possíveis problemas em vermelho etc.

Apesar de haver um documento extensivamente descritivo, com vários exemplos de falha, realizar o processo de implantação não era o tipo de tarefa que poderia ser feita por qualquer membro do projeto. Um pequeno grupo de pessoas com muito conhecimento e inserido no contexto se revezava na implantação, enquanto novos membros participavam para aprender as rotinas.

Dessa forma, o processo em questão era caótico, exigia um alto nível de conhecimento e tinha duração totalmente imprevisível, podendo variar de 4 até 10 horas. Para agravar ainda mais a situação, as implantações aconteciam durante períodos de baixa utilização da aplicação. Isso significa que várias pessoas da equipe precisavam passar a noite acordadas realizando um processo manual, extremamente frágil e altamente crítico.

11.3 A INICIATIVA DE MELHORIA DE IMPLANTAÇÕES

É fácil perceber que colocar uma pessoa acordada, de madrugada, para realizar um processo manual, frágil e crítico não é uma boa ideia. Foi assim que o time aqui descrito decidiu investir esforços para melhorar o processo de implantação.

Entretanto, tal mudança geralmente não ocorre motivada apenas por um desejo da equipe. Um processo crítico para a empresa dificilmente é modificado sem que haja uma cooperação entre todos os níveis, desde a equipe de

desenvolvedores, que sofre na pele as dificuldades, até a de estrategistas, que percebem a importância e o risco das mudanças.

Na organização em que a equipe em questão estava inserida, era bastante comum a formação de “iniciativas”. Iniciativas são como subprojetos, em que uma pequena parte da equipe é alocada especificamente para atividades relacionadas a essa perspectiva. Nesse sentido, o primeiro passo para melhorar o processo de entrega foi inserir a proposta em uma iniciativa, seguindo o padrão de trabalho da empresa.

A iniciativa foi nomeada como “Iniciativa de melhoria de implantações” e obteve apoio multifuncional da empresa: a equipe de desenvolvimento dava suporte à gerência, mostrando quais seriam as melhorias e como elas seriam realizadas, e a gerência lutava por investimentos para a iniciativa, evidenciando os resultados esperados e alcançados.

Medir e mostrar o real problema foram as primeiras ações da equipe. Existiam indicadores óbvios da necessidade de melhoria como a quantidade de horas e de pessoas de que o processo precisava. A empresa pretendia fazer com que o processo fosse mais rápido e eficiente. Assim, vendeu-se a ideia de automatizá-lo, como uma primeira etapa da iniciativa.

11.3.1 Fase 1 – Automação

Automatizar o processo de implantação foi o objetivo da primeira parte das mudanças. No livro *Entrega contínua*, o autor Jez Humble aponta a implantação manual como o primeiro antipadrão da entrega de software. Quando o processo é manual, existe um fator humano muito forte. Assim, as decisões que precisam ser tomadas durante a execução do processo muitas vezes mudam o resultado final. Em alguns cenários faz sentido deixar uma pessoa tomar decisões, mas não quando o principal produto de uma grande empresa está indo ao ar.

Documentos executáveis

Alguns sintomas da entrega manual apontados estavam fortemente presentes na equipe em análise, como produção de uma documentação extensiva para descrever o processo de implantação, confiança em testes manuais para

verificar que a aplicação está sendo executada, correções frequentes ao software implantado e trabalho exaustivo, até a madrugada, para tentar descobrir por que algo não está funcionando.

A partir desses pressupostos, fica bem mais fácil entender que a automação é essencial para alcançar entrega contínua. A equipe mapeou todas as etapas do processo de implantação e foi atacando pequenas partes. Por exemplo, em vez de ter uma pessoa para atualizar o código, fazer um *checkout* da versão que deveria ser implantada e iniciar o *script*, colocou a máquina para fazer isso. Aos poucos, as melhorias começaram a ser notadas, pois a documentação do processo diminuiu, bem como as exigências de tomadas de decisão.

Feedback e visibilidade

Um dos pontos importantes que precisou de pouco investimento da equipe e deu retorno muito grande foi a monitoração da saúde da aplicação. Neste caso, existiam vários processos e serviços que deveriam ser executados o tempo todo nos servidores. Como já eram utilizadas ferramentas de monitoração, foi necessário apenas adicionar uma verificação a mais para procurar por erros, em vez de deixar uma pessoa olhando para a página de monitoramento procurando por pontos vermelhos.

Assim, conforme as tarefas foram sendo automatizadas, tudo passou a ser uma simples questão de organização. Como a implantação agora possui vários pequenos passos, bastou unir todos eles em um único *script* para que todo o processo de implantação fosse automatizado. O documento que descrevia as etapas e comandos para executar a implantação transformou-se em baixar o código na máquina onde o processo seria iniciado e executar uma linha, a partir de alguns parâmetros.

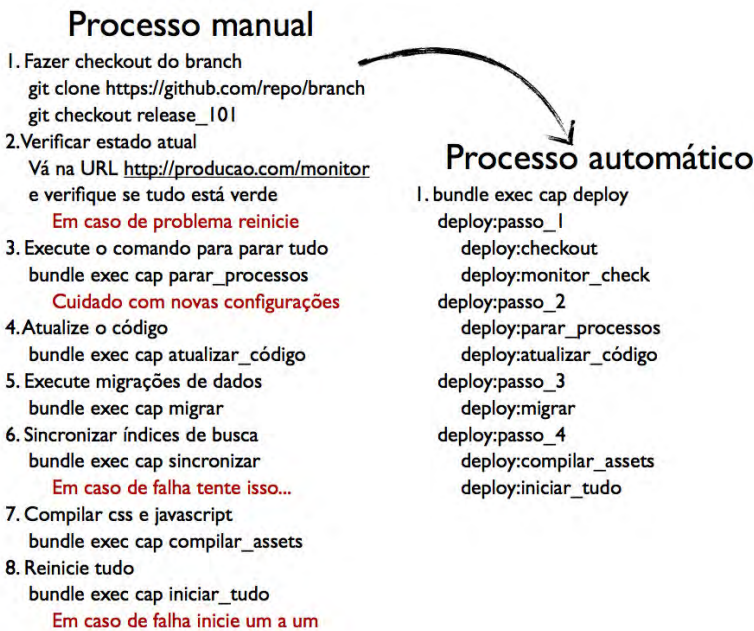


Fig. 11.3: Migrando processos manuais para automatizados

Organizando scripts externos

No entanto, um problema ainda chateava bastante o time: a quantidade de passos externos ao script principal. Como no início o processo era manual, era comum que passos extras fossem adicionados ao documento (limpar algo no banco de dados, por exemplo). Mesmo com um único script automatizado, ainda era necessário um pequeno conjunto de decisões.

A solução:

- 1) fazer uma espécie de versionamento desses *scripts* externos, para garantir que eles sejam executados apenas uma vez;
- 2) tornar o *script* de implantação flexível para que novas tarefas pudessem ser executadas junto com as principais.

Para a fase 1, bastou utilizar uma tabela no banco de dados para verificar

quais *scripts* ainda não haviam sido executados e quais precisariam ser executados durante a implantação. Já a parte 2 envolveu um esforço um pouco maior, mas graças ao *Capistrano*: – ferramenta para automação de servidores remotos – foi possível adicionar alguns ganchos no *script* principal e adicionar a chamada aos *scripts* externos antes ou depois de uma determinada tarefa. Desde então, a documentação do processo de implantação não sofreu nenhuma alteração significativa.

11.3.2 Fase 2 Melhorando a confiança nos scripts

A automação das tarefas permitiu que a equipe enxergasse mais claramente os problemas. Onde antes existiam milhares de pontos de falha, que nem sempre eram relacionados ao processo em si, agora havia apenas um. Com isso, notou-se que, mesmo com uma única linha sendo executada, vários problemas ainda ocorriam.

Para melhorar o processo e caminhar em direção ao objetivo de executar uma implantação mais rapidamente, decidiu-se aumentar a confiabilidade em vez de investir em performance. Para isso, criou-se uma segunda etapa de melhorias, com a finalidade específica de aperfeiçoar a confiança da equipe no *script* de implantação, o que também faria com que o tempo de duração do processo fosse menor.

Aproximando desenvolvimento e operações – DevOps

Com a maior visibilidade e autoria de *scripts* de implantação, os desenvolvedores começaram a lançar um olhar bem mais crítico aos problemas que aconteciam durante o processo. Alguns deles estavam fora do controle da equipe, como a parte de provisionamento de máquinas, que era controlada por outra equipe de operações e geralmente causava surpresas durante a implantação.

Mesmo com equipes de operações e desenvolvimento separadas por causa da organização da empresa, aos poucos os desenvolvedores foram se inteirando melhor sobre como o provisionamento, utilizando Puppet, funcionava, e passaram a se envolver mais com as mudanças que aconteciam. Foi apenas questão de tempo para que os benefícios de tais ações comessem a apare-

cer. Agora a equipe entendia o que acontecia e, em caso de algum problema, podia entrar em ação ao invés de simplesmente esperar uma resposta.

Correção e automação

Outras falhas eram causadas devido à ingenuidade do *script* de implantação, que considerava apenas o caminho positivo sem se preocupar muito com falhas. Isso exigiu um esforço bem maior na melhoria do *script* em si. A equipe decidiu que parte do empenho da iteração seria dedicada a investigar os problemas ocorridos e a garantir que as mesmas situações não acontecessem duas vezes. Após cada entrega, os problemas eram analisados para que não se repetissem nas implantações seguintes. Histórias eram adicionadas na iteração para que a investigação apropriada ocorresse, dando visibilidade dessas atividades à gerência do projeto e ao cliente.

Alguns exemplos de mudanças feitas são: em vez de ler/escrever informações de um arquivo diretamente, primeiro verifica-se se o arquivo está lá e, caso não esteja, cria-se a estrutura de diretórios para que ele possa ser gerado antes de ser processado; adicionam-se tratamentos de exceções para tentar mais de uma vez caso algum problema de rede aconteça; melhora-se a forma como é verificado se algum processo está sendo executado ou não, para que falhas reais sejam vistas mais cedo.

Scripts mais inteligentes

Já que as falhas eram um cenário comum, era normal que o processo precisasse ser reiniciado ou completado manualmente. Uma grande mudança feita para melhorar a confiança foi tornar o *script* idempotente, ou seja, atuar no sentido de executá-lo diversas vezes e obter sempre o mesmo resultado.

Para alcançar isso, fez-se com que o *script* entendesse quais tarefas já haviam sido executadas a fim de não as executar novamente. Caso as migrações de dados fossem executadas com sucesso, mas a tarefa seguinte desse erro, não seria necessário esperar pelo banco de dados para prosseguir e descobrir que o problema ainda não tinha sido corrigido. Foi realizado, então, um mapeamento das tarefas e um agrupamento em passos. Um deles seria um conjunto de atividades que, uma vez executadas, não precisariam ser refeitas. Cada passo possuía apenas uma tarefa de alto risco, que seria executada logo

no início, garantindo que os problemas surgiriam o mais cedo possível. Além disso, cada passo teria conhecimento sobre o passo seguinte, de modo que o *script* poderia começar em qualquer passo e o processo seria igualmente terminado por completo.



Fig. 11.4: Passo 1 não é executado duas vezes em caso de falha

11.4 AVALIANDO O RESULTADO

Um processo automatizado, estável e que continua a ser executado em caso de falhas foi a grande vantagem obtida ao final das etapas de melhoria. Pelo menos, essa é a percepção do ponto de vista dos desenvolvedores, já que eram eles os executores e garantidores de que tudo estivesse em ordem. No entanto, também é importante o envolvimento da equipe de desenvolvimento com a gerência e o cliente, para garantir que não somente os desenvolvedores vejam os benefícios das alterações no processo.

Para dar maior visibilidade aos resultados obtidos, nas reuniões de planejamento da iteração, era exibido o desempenho de cada tarefa de melhoria, mostrando o antes e o depois de sua implantação. Dessa forma, todas as pessoas da equipe, a saber, desenvolvedores, analistas de negócio e de qualidade, gerentes e proprietários da aplicação, podiam ver o progresso.

Antes das melhorias, o processo demorava em média de 4 a 6 horas, chegando ao ponto máximo de 10 horas! Em razão disso, a gerência da empresa decidiu colocar uma meta agressiva para a iniciativa: executar uma entrega

em menos de duas horas, mesmo que isso não parecesse factível. No final das duas etapas, conseguiu-se fazer com que a execução do *script* de implantação levasse de 1 hora a 1 hora e meia, adicionando a isso mais meia hora para a equipe de qualidade realizar verificações manuais antes de liberar a aplicação. Com isso, pôde-se manter todo o processo em uma média de 2 horas!

A equipe, diretamente afetada, ficou extremamente satisfeita com o resultado final alcançado depois de oito iterações (6 meses). Apesar de parecer um tempo longo, em grandes empresas as mudanças tendem a ser mais demoradas, mais ainda quando se trata da principal aplicação. O mesmo sentimento foi compartilhado pelo setor de liderança da empresa cliente, pois a meta que inicialmente parecia distante foi concretizada. Por ser a principal aplicação da organização, até mesmo a alta gerência e os vice-presidentes agradeceram pelas melhorias.

11.5 CONCLUSÃO

Transformar um processo manual, lento e arriscado em uma tarefa mais simples e automatizada é um grande benefício. Entretanto, entrega contínua vai bem além disso. O seu código precisa ser mais flexível e tolerante, o gerenciamento de ambientes deve ser simples, as verificações necessitam ocorrer mais rapidamente, entre vários outros fatores.

Após dar o primeiro passo, a equipe estuda maneiras de desacoplar mudanças de código, de dados e de infraestrutura, garantindo maior agilidade no processo de entrega. É preciso avaliar bem o contexto da equipe e do projeto para saber qual a melhor maneira de resolver o problema, diminuindo as dificuldades e aumentando os benefícios.

CAPÍTULO 12

Explorando padrões de implementação em Ruby

por Hugo Corbucci

O fim da década de 90 e o começo dos anos 2000 foram amplamente dominados por linguagens orientadas a objetos de tipagem forte e com sintaxe próxima de C. Conforme avançamos nos anos 2000, com a evolução de métodos ágeis e a ênfase em testes automatizados, linguagens dinamicamente tipadas ganharam forças novamente. Ruby, em especial, ganhou muitos adeptos graças ao framework web Ruby on Rails, carinhosamente apelidado de Rails.

Conforme a comunidade Ruby crescia e as bibliotecas ao seu redor amadureciam, as práticas recomendadas foram mudando e a forma de implementar alguns padrões mudou. Este capítulo apresenta quatro padrões de projeto

conhecidos e algumas implementações diferentes em Ruby, acompanhadas de uma análise crítica sobre suas vantagens e desvantagens.

Os padrões escolhidos *State*, *Active Record*, *Construtores* e *Map Reduce* são amplamente usados na comunidade Ruby e evoluíram muito nos últimos anos. Quando estiver lendo os códigos de exemplo, tente identificar as limitações que cada implementação apresenta.

12.1 PADRÃO STATE

O padrão *State* foi originalmente apresentado no famoso livro de padrões de projeto *Design Patterns*, também conhecido como GoF (de *Gang of Four*, por conta dos seus quatro autores). O padrão *State* é uma das formas de se atacar o problema de variar o comportamento de determinado objeto ou algoritmo de acordo com algum fator como, por exemplo, o estado interno da memória. Como linguagens orientadas a objetos têm por princípio a ideia de encapsulamento de dados e comportamentos, faz muito sentido um objeto mudar seu comportamento de acordo com seus dados. No entanto, existem muitas formas de atingir esse objetivo.

A mais simples é, obviamente, o uso de condicionais como no código a seguir:

```
class Produto
  def preco
    if disponivel?
      valor_unitario
    else
      nil
    end
  end
end
```

Nesse caso, o método `preco` tem dois comportamentos de acordo com o estado interno do objeto. No caso de o produto estar disponível, o preço é o valor unitário, caso contrário é *nil*. Nessa implementação super-simples, o padrão *State* se apresenta com condicional.

No entanto, é fácil perceber como essa solução simples tem problemas quando precisamos implementar mais métodos, como, por exemplo, o `foto_destaque`:

```
class Produto
  def foto_destaque
    if disponivel?
      imagens.first
    else
      IMAGEM_INDISPONIVEL
    end
  end
end
```

Com o condicional, acabamos tendo que repetir o mesmo código de verificação várias vezes. Em Java ou C#, a solução mais comum seria criar uma enumeração com a lista de estados possíveis para o produto e manter nele a referência para o estado atual. Em Ruby, não há enumeração e, em geral, módulos são a forma mais comum de substituir enumerações. A ideia é que cada instância da enumeração é um módulo e o produto apenas inclui o módulo desejado. Nesse caso, o código ficaria:

```
module Disponivel
  def preco
    valor_unitario
  end
  def foto_destaque
    imagens.first
  end
end

module Indisponivel
  def preco
    nil
  end
  def foto_destaque
    IMAGEM_INDISPONIVEL
  end
end
```

```
class Produto
  if disponivel?
    include Disponivel
  else
    include Indisponivel
  end
end
```

Um dos problemas dessa solução vem do fato de ela não ser muito dinâmica, já que módulos em Ruby não podem ser removidos, apenas adicionados. Por conta disso, se o produto tiver sido criado como `Indisponível`, não podemos apenas fazer `include Disponivel` quando ele se tornar disponível. Como fazer para lidar com o uso de diferentes métodos? De acordo com boas práticas de orientação a objetos, podemos pensar em manter as implementações separadas e apenas delegar a implementação para a classe e o método desejado. Algo como:

```
class Produto
  def preco
    if disponivel?
      preco_disponivel
    else
      preco_indisponivel
    end
  end
  private
  def preco_disponivel
    valor_unitario
  end
  def preco_indisponivel
    nil
  end
end
```

Isso é muito semelhante à solução que propusemos originalmente. A vantagem de implementar esse método é que, agora, com um pouco de consistência nos nomes dos métodos, podemos aproveitar outro aspecto dinâmico de Ruby para centralizar o condicional:

```
class Produto
  def preco
    send("preco_#{estado}")
  end
  private
  def estado
    disponivel? ? 'disponivel' : 'indisponivel'
  end
  def preco_disponivel
    valor_unitario
  end
  def preco_indisponivel
    nil
  end
end
```

Neste caso, o método `send` decide dinamicamente, montando o nome do método que deseja invocar, qual implementação é utilizada. A vantagem fica na centralização do condicional no método `estado`, que sabe o nome do estado atual. A principal desvantagem é que, conforme o `Produto` depender mais desse estado, a classe vai dobrando de tamanho.

SOBRE USO DE `SEND` COM NOMES COMPOSTOS DINAMICAMENTE

Outro problema um pouco menos evidente do uso de `send` é que ele torna mais complicada a avaliação para determinar quais métodos precisam existir e quais não são necessários. No exemplo apresentado, a situação é bem simples, pois só há dois estados e a lógica é bem simples para determinar todas as combinações possíveis (`preco_disponivel` e `preco_indisponivel`).

Agora, imagine que o produto tenha quatro estados diferentes e, em alguns casos, há estados compostos. Para saber quais métodos são usados e necessários, precisamos entender todas as regras de negócio sobre como o estado pode ser representado. Essa é uma complicação comum com uso de metaprogramação, uma técnica da qual `send` é uma das ferramentas.

Mas, neste caso, por que não separar os objetos como fizemos com módulos? A ideia fica a mesma, porém, em vez de decidir qual método é chamado dinamicamente, decidimos para qual objeto a chamada deve ser direcionada.

```
class Produto
  def preco
    estado.preco
  end
  def foto_destaque
    estado.foto_destaque
  end
  private
  def estado
    if disponivel?
      Produto::Disponivel.new(self)
    else
      Produto::Indisponivel.new(self)
    end
  end
end
```

Dessa forma, a decisão fica centralizada, o código separado e é fácil alterar e testar cada estado possível. Também fica muito fácil alterar os possíveis estados e modificar o comportamento no futuro. A grande vantagem dessa solução é sua simplicidade. Como as únicas coisas necessárias para essa implementação são objetos simples e delegação, fica bem claro quais métodos estão sendo chamados e como eles podem ser alterados. Ruby ainda oferece o módulo `Forwardable`, que disponibiliza alguns métodos que facilitam a escrita de classes que delegam para objetos internos. Usando esse módulo, o código anterior fica assim:

```
require 'forwardable'
class Produto
  extend Forwardable
  def_delegators :estado, :preco, :foto_destaque
  private
  def estado
    if disponivel?
```

```
        Produto::Disponivel.new(self)
      else
        Produto::Indisponivel.new(self)
      end
    end
  end
end

class Produto::Disponivel
  def initialize(produto)
    @produto = produto
  end
  def preco
    @produto.valor_unitario
  end
  def foto_destaque
    @produto.imagens.first
  end
end

class Produto::Indisponivel
  CAMINHO_DA_IMAGEM = 'indisponivel.png'
  def initialize(produto)
  end
  def preco
    nil
  end
  def foto_destaque
    CAMINHO_DA_IMAGEM
  end
end
```

No fim das contas, apesar de Ruby oferecer muitas formas de implementar o padrão *State*, a última solução apresenta os maiores benefícios. As outras soluções começam mais simples, mas rapidamente tendem a exigir mais conhecimento, paciência ou código para serem entendidas ou estendidas.

12.2 PADRÃO *ACTIVE RECORD*

Rails popularizou o padrão *Active Record* por meio da biblioteca *ActiveRecord* para lidar com persistência de dados. O problema que este padrão procura resolver é o de associar um objeto em memória com sua representação no banco de dados. Outro padrão que resolve um problema semelhante é o *Data Access Object* ou DAO, que é bastante comum no mundo Java. A diferença fica pela forma com que a aplicação interage com um objeto que implementa o padrão *Active Record*. Esses objetos oferecem métodos para inserção, atualização e remoção de suas representações na camada de persistência.

A biblioteca usada no Ruby é implementada com uso pesado de metaprogramação e gera, dinamicamente, métodos no objeto que refletem sua constituição no banco de dados. Portanto, se tivermos um objeto *ActiveRecord* da seguinte forma:

```
class Produto < ActiveRecord::Base
end
```

E uma tabela no banco de dados:

```
mysql> describe produtos;
+-----+-----+-----+-----+-----+
| Field          | Type          | Null | Key | Default |
+-----+-----+-----+-----+-----+
| id             | int(11)       | NO   | PRI | NULL     |
| nome          | char(50)      | NO   |     | NULL     |
| preco         | double        | NO   |     | NULL     |
| quantidade_em_estoque | int(11)       | NO   |     | NULL     |
| descricao     | text          | YES  |     | NULL     |
+-----+-----+-----+-----+-----+
5 rows in set (0.00 sec)
```

Ao executarmos:

```
> metodos_dinamicos = (Produto.new.methods -
>   ActiveRecord::Base.instance_methods)
=> ["id_change", "quantidade_em_estoque_will_change!",
    "quantidade_em_estoque_was", "reset_nome!", "preco",
```

```

"preco=", "reset_quantidade_em_estoque!",
"preco_before_type_cast", "id_will_change!", "descricao",
"descricao=", "id_was", "descricao_before_type_cast", "preco?",
"preco_changed?", "descricao?", "reset_id!",
"descricao_changed?", "descricao_change", "preco_will_change!",
"preco_was", "nome", "descricao_will_change!", "descricao_was",
"nome=", "reset_preco!", "nome_before_type_cast",
"quantidade_em_estoque", "quantidade_em_estoque=",
"reset_descricao!", "id_before_type_cast", "nome?",
"nome_changed?", "preco_change",
"quantidade_em_estoque_before_type_cast",
"quantidade_em_estoque?",

"quantidade_em_estoque_changed?",
"quantidade_em_estoque_change",
"nome_will_change!", "nome_was", "id_changed?"]

```

São 43 métodos criados por conta daquelas cinco colunas disponíveis no banco de dados. Se você notar, são nove métodos por coluna, a não ser pelo ID, que é tratado de forma diferente por ser a chave primária do banco de dados e por todo objeto em Ruby (versões 1.8 ou anteriores) ter o método `id` definido. Veja os exemplos:

```

> metodos_dinamicos.grep /nome/
=> ["reset_nome!", "nome", "nome=", "nome_before_type_cast",
    "nome?", "nome_changed?", "nome_change", "nome_will_change!",
    "nome_was"]
> _.size
=> 9
> metodos_dinamicos.grep /(?:_id)|(?:id_)|(?:^id$)/
=> ["reset_id!", "id_before_type_cast", "id_changed?",
    "id_change", "id_will_change!", "id_was"]
> _.size
=> 6

```

O ID não tem métodos para atribuição (`id=`) ou verificação de existência (`id?`). Note também que, como excluimos os métodos de `ActiveRecord::Base`, ao gerar os métodos dinâmicos (`Produto.new.methods -`

`ActiveRecord::Base.instance_methods`), na listagem faltou o método de consulta (`id`), porque ele está definido na superclasse `Object` (até a versão 1.8 do Ruby) da qual `ActiveRecord::Base` herda.

`ActiveRecord::Base` também oferece muitos outros métodos (206 no total), dos quais por volta de 15 são os mais usados:

```
=> [attributes, attributes=, changed?, changes, new_record?,  
    reload, valid?, save, save!, update, update!, update_attributes,  
    update_attributes!, destroy, destroy!]
```

Esses são os métodos com os quais a maior parte das interações com objetos `ActiveRecord` se dá. Eles permitem salvar um objeto, atualizá-lo, apagá-lo, assim como verificar algumas coisas dentro dele. Com esses métodos aliados aos que foram gerados de acordo com o banco de dados, é possível criar objetos e atualizar facilmente como se estivéssemos lidando apenas com atribuições em memória.

Apesar de o `ActiveRecord` ser uma *gem* independente, a maioria dos projetos Ruby que a usam são projetos Rails. Rails força um padrão *Model-View-Controller* [77], ou MVC, e os exemplos mais simples do uso do framework utilizam objetos que herdam de `ActiveRecord::Base` como seus modelos, isto é, a camada do MVC que mantém a lógica de negócios.

Existem muitas consequências na concentração dessas duas responsabilidades (persistência e lógica de negócio) em um único objeto. Uma das mais evidentes é a ligação direta entre a modelagem de objetos e a do banco de dados. Para aqueles que estudaram um pouco de modelagem de banco de dados, vocês devem lembrar dos princípios de normalização de dados. A ideia é simples: evitar duplicação de dados por meio da extração dos dados duplicados em um único local e criar uma referência aos dados extraídos.

Ao atrelar o objeto responsável pela persistência com o seu modelo lógico, essas regras de normalização passam a afetar a sua modelagem do problema na sua camada de negócio. Em vários casos, isso tem um impacto positivo. Fica muito mais fácil identificar novos objetos de negócio relevantes ao perceber que certos dados vão juntos uns com os outros. No entanto, conforme seus objetos crescem, também fica mais difícil separar objetos de negócio de acordo com seu uso no sistema, já que o foco fica nos dados. Vejamos um exemplo para entender melhor esse argumentos.

Considere a modelagem de um sistema de leilão online, como o eBay [57]. Parece relativamente simples imaginar que precisamos de usuários com endereços, formas de pagamento e itens a serem leiloados. Ao criarmos a primeira modelagem desse sistema, provavelmente criaremos as seguintes classes: `Usuario`, `Endereço`, `FormaDePagamento` e `Item`. Parece uma modelagem relativamente robusta. Usuários têm um ou mais endereços para os quais os itens que comprarem serão enviados, eles podem comprar com qualquer forma de pagamento previamente cadastrada e podem oferecer um ou vários itens para leilão. Não vou facilmente cometer o erro de concentrar o endereço e a forma de pagamento na tabela do usuário, já que fica claro que posso ter mais de um endereço ou forma de pagamento e posso decidir mudá-los sem alterar alguma informação do meu usuário.

Conforme evoluímos o sistema, rapidamente descobrimos que o `Item` precisa receber lances e criamos logo uma classe `Lance`, que associa um usuário a um item e um valor. O item passa a ter um ciclo de vida em que está sendo preparado para um leilão, um momento em que recebe lances e um momento em que foi vendido ou não, mas não aceita mais lances. Em termos de dados, isso é muito simples e basta adicionar uma coluna na tabela de itens para modelar essa mudança do modelo de negócios. Não há duplicação de dados e o objeto `Item` simplesmente ganhou 9 métodos.

A biblioteca `ActiveRecord` deixou tão fácil agregar esses métodos e dados que não paramos para pensar no sistema de leilão que é o principal do negócio. Um item que está em leilão é muito mais importante para meu negócio do que um item que já foi vendido ou que não está pronto para ser leilado. Ao agregar os dados, eu dei a mesma importância a todos eles, compliquei minha lógica de negócio tendo que filtrar o status do item para a maioria das minhas ações e, no caso de conseguir ser tão popular quanto o eBay, agora terei que lidar com bilhões de itens na minha tabela de itens em vez dos meros 10 milhões que tenho em geral em leilão.

Para evitar esse tipo de problema, a comunidade Rails está lentamente evoluindo para uma estrutura em que os objetos que lidam com as regras de negócio não são os objetos que lidam com a persistência. A ideia é que os objetos que herdaram de `ActiveRecord::Base` contenham apenas a lógica necessária para realizar a persistência corretamente. Outros objetos são res-

ponsáveis por manter a lógica de negócio e garantir a consistência entre os objetos persistidos. A prática atual é que esses últimos objetos são chamados de *Active Models* e aproveitam alguns tantos métodos para facilitar a validação do objeto.

12.3 PADRÕES CONSTRUTORES

Ruby é uma linguagem orientada a objetos. Por consequência, boa parte do trabalho de sistemas orientados a objeto é a construção de objetos. O mundo Java possui bibliotecas que cuidam de facilitar a construção de objetos graças a uma técnica chamada **injeção de dependência** (*Dependency Injection* ou DI), que é uma implementação do padrão **inversão de controle** (*Inversion of Control* ou IoC).

Não existem muitas bibliotecas populares em Ruby para facilitar o uso dessas técnicas. Um dos motivos para a ausência dessas bibliotecas é a dinamicidade da linguagem que dificulta identificar o tipo do elemento que precisa ser passado. No entanto, esse mesmo aspecto dinâmico foi um dos principais motivadores para o fortalecimento da cultura de testes automatizados na comunidade Ruby.

A consequência é que muitas das decisões relacionadas a como objetos são construídos precisam ser tomadas pelo desenvolvedor em cada ocasião. O padrão da linguagem é que objetos são construídos por meio do método de classe `new` que instancia um objeto novo e chama o método `initialize` nessa instância com os parâmetros passados ao `new`. Para a maioria dos casos em que objetos são simples e só podem ser construídos de uma forma, essa solução é perfeitamente adequada e resolve todos os problemas. No entanto, assim que o objeto passa a ser mais complexo e possui mais de uma forma de ser construído, é preciso encontrar novos padrões.

Uma das soluções em projetos Ruby é o uso de um mapa de atributos:

```
Transaction.new(:credit => credit_account,  
               :debit => debit_account,  
               :total => 1000)
```

A implementação dessa solução usa metaprogramação mas não é tão complicada:

```
class Transaction
  def initialize(attributes)
    attributes.each do |key, value|
      send("#{key}=", value)
    end
  end
  private
  def credit=(value)
    @credit = value
  end
  def debit=(value)
    @debit = value
  end
  def total=(value)
    @total = value
  end
end
```

Essa é a implementação escolhida pela comunidade Rails para construir seus objetos que herdam de `ActiveRecord::Base`. Ela permite extensibilidade, já que a implementação de cada um dos métodos `credit=` ou `debit=` pode ser alterada de acordo com a necessidade. Novos atributos ou atributos que foram alterados podem ser introduzidos desde que os métodos de atribuição continuem tendo o efeito desejado. Por outro lado, não podemos garantir que o objeto criado seja válido, isto é, internamente consistente. Se as regras para criar o objeto começam a ficar complexas, é muito fácil criar objetos com valores inesperados de forma que os métodos do objeto não se comportam conforme desejado.

Em alguns casos, podemos simplesmente adicionar uma validação ao final do `initialize` para lançar exceções, caso o objeto seja inválido após todas as atribuições. Isso garante que objetos inválidos não possam ser criados. No entanto, é mais difícil para os usuários do objeto saberem o que é o mínimo e máximo permitido para criação daquele objeto.

Para lidar com esse problema, podemos criar métodos de classe com nomes descritivos para a criação dos objetos, como:

```
Transaction.with_credit_and_debit_and_total(
```

```
    credit_account, debit_account, total
)
```

Dessa forma, fica claro quais são objetos esperados para criação da transação. Infelizmente, também é óbvio, com esse exemplo, que essa solução rapidamente se torna inviável caso o objeto precise de mais argumentos. Não somente o nome fica longo, mas, quando for preciso adicionar um atributo obrigatório ao construtor, precisaremos achar todos os locais do código que usam esse construtor e atualizá-lo. Em Java, essa tarefa é bem simples, já que a tipagem estática faz com que a maioria das ferramentas de análise de código seja capaz de determinar corretamente todos os usos de determinado método. Em Ruby, a dinamicidade da linguagem torna essa tarefa muito mais difícil e a maioria das ferramentas disponíveis não consegue ajudar a fazer esse tipo de mudanças com muita segurança.

Cada uma das soluções anteriores lida bem com a construção de objetos menores e oferece mais ou menos rigidez e informação aos seus usuários com relação ao que é permitido para aquele objeto. A última solução é a mais complexa, mais flexível e pode ser mais estrita. Caso seu objeto seja complexo de construir e precise de muitas regras que os usuários precisam seguir, o comum é criar uma outra classe ou conjunto de classes, chamadas de *Builders*, cuja responsabilidade é descrever como criar o objeto complexo. Essas classes oferecem métodos rígidos, e possivelmente, encadeamentos que restringem os usuários. Por exemplo:

```
TransactionBuilder.from(credit_account).
    to(debit_account).of_value(amount)
```

```
class TransactionBuilder
  def self.from(credit)
    TransactionFromBuilder.new(credit)
  end
end

class TransactionFromBuilder
  def initialize(credit)
    @credit = credit
  end
  def to(debit)
```

```
        TransactionBetweenBuilder.new(@credit, debit)
    end
end
class TransactionBetweenBuilder
    def initialize(credit, debit)
        @credit = credit
        @debit = debit
    end
    def of_value(amount)
        Transaction.new(:credit => @credit,
            :debit => @debit,
            :amount => amount)
    end
end
```

Note que esse exemplo possui três classes muito simples apenas para facilitar a criação da transação. Nesse caso simples, essas três classes são claramente uma modelagem excessivamente complicada. Mas nada impede que algumas delas sejam juntadas, se fizer sentido.

Apesar de essa solução ter mais código, ela permite tanto uma flexibilidade em termos de evolução que as outras não permitem, como uma usabilidade mais evidente. Como todos os padrões apresentados neste capítulo, existem situações em que cada solução encontra seu limite. Pense nas vantagens e desvantagens de cada uma delas e evolua seu sistema conforme necessário.

12.4 PADRÃO MAP/REDUCE

Linguagens orientadas a objetos e linguagens funcionais são frequentemente consideradas mutualmente exclusivas. No entanto, existem muitas características funcionais em várias linguagens orientadas a objetos inclusive Ruby.

Existe uma funcionalidade e um módulo da linguagem de permitem tirar proveito de muitas ideias de linguagens funcionais. A primeira é a habilidade de armazenar funções em variáveis, uma funcionalidade também conhecida como funções de primeira ordem. A segunda é o módulo *Enumerable*, que é incluído ou implementado em todas as coleções disponíveis na linguagem, seja uma lista, um conjunto ou um dicionário.

Este módulo oferece uma série de métodos que permite realizar várias operações na coleção desejada. O método mais comum é `each`. Praticamente todo programador Ruby já viu o uso do método `each` para iterar na coleção, como no exemplo a seguir:

```
[1, 2, 3].each {|n| puts n }  
1  
2  
3  
> [1, 2, 3]
```

Esse exemplo bem simples reproduz apenas o comportamento de laços imperativos como `for` ou `while`. Apesar de interessante, ele não traz nada de muito funcional, já que o resultado do uso do método é a própria lista inicial e, portanto, só tem efeitos colaterais.

Outros métodos são mais interessantes, como, por exemplo, `select`:

```
[1, 2, 3].select {|n| n % 2 == 1}  
> [1, 3]
```

Ou `reject`:

```
[1, 2, 3].reject {|n| n % 2 == 0}  
> [1, 3]
```

Nesse caso, apesar de podermos ter efeitos colaterais, o mais útil é o resultado da chamada do método. O próximo método que vai nessa linha é o `map`:

```
[1, 2, 3].map {|n| n ** 2}  
> [1, 4, 9]
```

Com o `map` podemos criar uma nova coleção a partir de uma coleção existente. Com isso, já podemos pegar qualquer coleção, filtrá-la para apenas os elementos que quisermos e mapeá-los para outros valores desejados. Pense em um carrinho de compra:

```
carrinho.itens.map {|item| item.total }
```

O que era uma lista com objetos complexos agora já acumula apenas os preços dos itens que desejamos comprar. Para determinar o total, falta um método: `reduce`.

```
[1, 2, 3].reduce {|acumulador, n| acumulador + n}  
> 6
```

No caso do carrinho de compras:

```
total = carrinho.itens.  
  map {|item| item.total }.  
  reduce {|acumulador, valor| acumulador + valor}
```

Como essas funções são bastante usadas nesse tipo de conta, o Ruby também disponibiliza uma forma de criar uma função anônima de forma mais simples:

```
total = carrinho.itens.map(&:total).reduce(&:+)
```

O operador unário `&` (não confunda com o operador binário `&` que faz um *e* lógico bit a bit) usado com um símbolo faz com que se chame o método `to_proc` no símbolo. O resultado é uma função anônima correspondente à chamada do método cujo nome o símbolo representa. No caso do `+`, o primeiro objeto (acumulador) recebe a chamada do `+` com o segundo argumento (valor) como parâmetro.

Imagine que, em vez do valor dos itens somados, gostaríamos de exibir a quantidade de itens. A mudança é muito simples:

```
quantidade_de_itens = carrinho.itens.map(&:quantidade)  
  .reduce(&:+)
```

Essas três funções todas têm sinônimos que são conhecidos em uma ou outra linguagem como `filter`, `collect` e `inject`. Também existem algumas que existem para parâmetros muito comuns como o `compact`, que é um `reject` de objetos que sejam *nil*, ou o `join`, que é um `reduce` que concatena *strings*.

O nome *MapReduce* ficou muito conhecido alguns anos atrás com o crescimento do Google e seu algoritmo de ranqueamento de páginas que “era

baseado em *MapReduce*”. A ideia é a mesma e a grande vantagem é a habilidade de paralelizar os passos do mapeamento e, muitas vezes, da redução também (depende um pouco de usar operações associativas um detalhe fora do escopo deste capítulo).

Para programadores em Ruby, além da possibilidade de isso acontecer dependendo da máquina virtual que utilizar, o poder do *MapReduce* é de facilitar o entendimento das transformações que acontecem na coleção e como elas servem para gerar o resultado final. Compare o exemplo anterior à versão procedural:

```
tamanho = carrinho.itens.size
i = 0
quantidade_de_itens = 0
while i < tamanho do
  item = carrinho.itens[i]
  quantidade_de_itens += item.quantidade
  i += 1
end
quantidade_de_itens
```

O mapeamento e a redução estão misturados ao processo iterativo que navega pela coleção. Além de ter mais código, o resultado é mais difícil de entender porque não há uma boa abstração. Em resumo, `select`, `map` e `reduce` são operações que abstraem comportamentos que desejamos frequentemente quando trabalhamos com coleções. Usar essas abstrações nos ajuda a simplificar nosso código, aumentar a coesão e facilitar mudanças.

12.5 CONCLUSÃO

Ruby é uma linguagem orientada a objetos com origens bem claras em *Small-talk*. Muitas das ideias, padrões e soluções encontradas e apresentadas por essa comunidade são fáceis de aplicar em Ruby. Os padrões de implementação demonstrados aqui são uma pequena amostra desse conhecimento.

Eles reforçam o que a comunidade de orientação a objetos tem apresentado como boas práticas há muito tempo. Reduzir o acoplamento, aumentar a coesão e encapsular comportamentos são formas de facilitar a manutenção do código que esses padrões tentam realçar.

Existem várias outras opções de implementações e cada uma tem seu mérito. Procure sempre pensar nas consequências de optar por uma ou outra implementação. Há grandes chances de que uma implementação um pouco mais robusta, mas que permita isolar as responsabilidades, seja mais vantajosa a médio e longo prazo do que uma que pareça mais simples de imediato.

CAPÍTULO 13

Testando JavaScript com o Protractor: um exemplo de solução integradora

por Daniel Amorim

Vamos conversar sobre o framework **Protractor** e sobre o seu papel como ferramenta de teste e como integrador de soluções, combinando ferramentas e tecnologias atualmente poderosas: AngularJS, NodeJS, Selenium, WebDriver, Jasmine, Cucumber e Mocha. Este capítulo trata especificamente do framework em questão nos últimos dois anos, ou seja, 2013 e 2014. Embora o Protractor seja recente, a ideia por trás da solução apresentada por ele se repete no mundo Open Source de soluções integra-

doras entre famílias de frameworks, tecnologias específicas, bem-sucedidas e complementares.

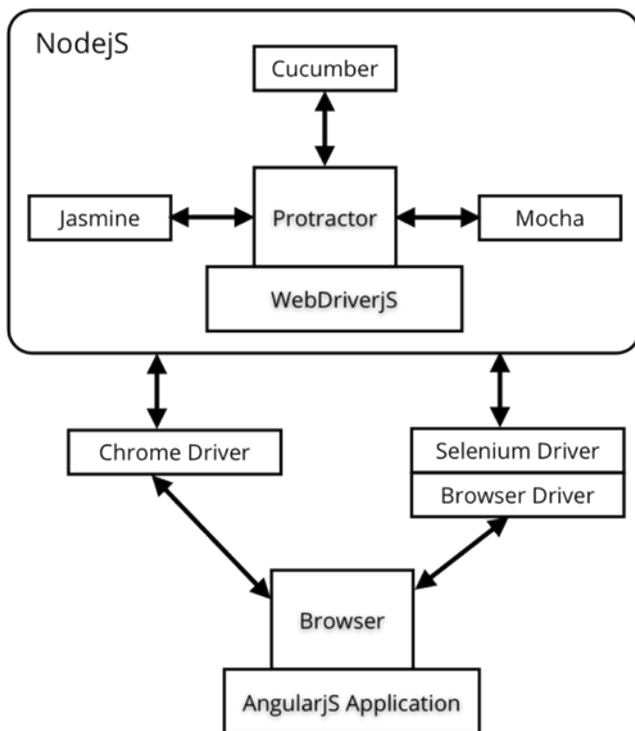


Fig. 13.1: Arquitetura do Protractor

13.1 PRESS START

A primeira versão do Protractor foi lançada em julho de 2013, configurando-se basicamente como um protótipo de um framework de testes. A Google, porém, com o apoio da comunidades de testes, está desenvolvendo-o para que ele acompanhe a evolução do AngularJS e satisfaça as necessidades da comunidade que o está utilizando.

O projeto do Protractor é público no Github e é possível acompanhar as *issues* do projeto, adicionar issues de interesse, comentar as issues abertas pelos outros, fazer *pull requests* para colaborar com o projeto etc. Qualquer um que

esteja interessado em colaborar com o crescimento do projeto é bem-vindo!

Link do projeto: <https://github.com/angular/protractor>

13.2 ENTENDENDO O PROTRACTOR MAIS A FUNDO

O Protractor é um framework de automação de testes funcionais, cujo intuito não é ser a única forma de testar a aplicação AngularJS, mas sim cobrir os critérios de aceitação exigidos pelo usuário. Mesmo existindo testes que executem em nível de UI através do Protractor, ainda assim é necessária a criação de testes unitários e de integração.

O Protractor roda em cima do Selenium, e, dessa forma, já contém todos os benefícios e facilidades desse último, além das funcionalidades para testar aplicações AngularJS, especificamente. Como solução baseada no Selenium, é possível a utilização dos drivers que implementam o WebDriver, tais como ChromeDriver e GhostDriver. No caso do ChromeDriver, é viável rodar os testes em cima dele sem a necessidade do Selenium Server. Já para utilizar o GhostDriver, é necessário usar o PhantomJS, o qual se utiliza do GhostDriver e, nesse caso, permite rodar os testes em *headless mode*.

O framework permite que seja empregado Jasmine para organizar e criar testes e resultados esperados. O Jasmine é compatível com o Protractor porque todos os recursos que são extraídos do browser para fazer os testes são `promises`, e o comando `expect` do Jasmine trata internamente tais `promises`, tornando transparentes as validações dos testes.

No início, como o Protractor ainda era recente e estava em fase de maturação, a documentação para usá-lo era restrita e ficava desatualizada rapidamente, devido à constante evolução da ferramenta. Porém, nos últimos meses, a colaboração da comunidade tem crescido bastante e a documentação tem se mantido mais atualizada, em razão da facilidade com que as pessoas questionam e colaboram através do projeto público no Github. A única burocracia para participar é a necessidade de assinar o contrato digital de colaborador da Google.

Passo 1 Instalando

Um prerequisite para o uso do Protractor é o Node.js, pois o framework

de testes é um pacote node que deve ser instalado através do NPM (*Node Package Manager*).

Com apenas um simples comando `npm install protractor -g`, o Protractor está instalado na máquina e pronto para ser executado através do comando `protractor`. Após a execução desse comando, é emitida uma mensagem exigindo um parâmetro de execução do Protractor. Tal parâmetro é o caminho do arquivo de configuração, e se trata de elemento necessário para executar os testes, pois guia a ferramenta da maneira como ela deve ser executada.

Passo 2 Configurando

Existem vários parâmetros que permitem a configuração do Protractor. A seguir, são descritos alguns, que se pensa serem os mais importantes:

- `SeleniumAddress`: permite informar uma URL do Selenium Server que o Protractor usará para executar os testes. Neste caso, o Selenium Server deve estar previamente iniciado antes de serem rodados os testes no Protractor.
- `SeleniumServerJar`: permite informar o caminho do arquivo `.jar` do Selenium Server. Caso este parâmetro seja utilizado, o Protractor irá controlar a inicialização e a finalização do Selenium Server, não sendo necessário iniciá-lo antes de executar o Protractor.
- `SauceUser` e `SauceKey`: caso informados tais parâmetros, o Protractor irá ignorar o `SeleniumServerJar` e executar os testes contra um Selenium Server no SauceLabs. Um array de arquivos de testes, os quais o Protractor irá executar, pode ser passado através do parâmetro `specs`. O caminho dos arquivos de teste deve ser relativo ao arquivo de configuração.
- `seleniumArgs`: permite passar parâmetros para o Selenium caso o Protractor inicialize através do `SeleniumServerJar`.

Parâmetros também podem ser passados para o WebDriver através do `capabilities`, onde é informado o browser em que o Protractor vai executar os testes contra.

Uma URL de acesso padrão pode ser passada para o Protractor através do parâmetro `baseUrl`. Com isso, toda a chamada feita para um browser será para essa url especificada.

O framework de testes e de *assertions* que será utilizado pelo Protractor pode ser determinado pelo parâmetro `framework`. Para isso, existem as seguintes opções: Jasmine, Cucumber e Mocha.

Para configurar um *timeout* para cada teste executado, o Protractor provê o parâmetro `allScriptsTimeout`, o qual deve receber um valor em milissegundos.

Todos esses parâmetros são encapsulados em um objeto `node` com nome de `config`, para que o Protractor possa identificá-los. No código a seguir, tem-se um exemplo de arquivo de configuração do Protractor, o qual foi salvo como `config.js`.

```
exports.config = {  
  
  seleniumServerJar:  
    './node_modules/protractor/selenium/  
      selenium-server-standalone-2.39.0.jar',  
  
  specs: [  
    'tests/hello_world.js'  
  ],  
  
  seleniumArgs: ['-browserTimeout=60'],  
  
  capabilities: {  
    'browserName': 'chrome'  
  },  
  
  baseUrl: 'http://localhost:8000',  
  
  allScriptsTimeout: 30000  
};
```

Nesse exemplo, está configurado o parâmetro `seleniumServerJar` para iniciar o Selenium Server através do Protractor. O arquivo de testes que

será executado nesse caso é o `hello_world.js`, que está dentro da pasta `tests`. Tais testes serão executados contra o navegador Chrome, devido ao parâmetro `capabilities` estar configurado com o atributo `browserName` como `chrome`. O timeout estabelecido para a execução de cada teste é de 30 segundos.

Após o arquivo de configuração estar pronto, basta ir até a pasta do arquivo e executar o comando `protractor config.js` e o Protractor será executado seguindo as instruções passadas no arquivo. Porém, a mensagem a seguir será exibida, devido à inexistência do arquivo de testes chamado `hello_world.js` ainda.

```
/usr/local/lib/node_modules/protractor/lib/cli.js:91
    throw new Error('Test file ' + specs[i] + ' did ...');
    ^
Error: Test file tests/hello_world.js did not match any files.
    at run (/usr/local/lib/
      node_modules/protractor/lib/cli.js:91:13)
    at Object.<anonymous>
      (/usr/local/lib/node_modules/protractor/lib/cli.js:265:1)
    at Module._compile (module.js:456:26)
    at Object.Module._extensions..js (module.js:474:10)
    at Module.load (module.js:356:32)
    at Function.Module._load (module.js:312:12)
    at Function.Module.runMain (module.js:497:10)
    at startup (node.js:119:16)
    at node.js:901:3
```

Na sequência, são criados os testes para que o Protractor possa executá-los.

Passo 3 Criando testes

Como já citado na introdução, o Protractor roda em cima do Selenium e, por isso, utiliza todas as vantagens que o Selenium tem embutido em si. Porém, o que faz do Protractor o melhor framework de automação de testes para aplicações AngularJS são justamente as facilidades criadas nele especificamente para testar esse tipo de aplicação.

Os comandos customizados pelo Protractor visam capturar os elementos da interface da aplicação através das diretivas do AngularJS. Isso torna o framework interessante pelo fato de o profissional que aprender a utilizá-lo, automaticamente, também apreende o comportamento do AngularJS em relação à renderização de elementos na interface. O inverso também acontece: a partir do momento em que se conhece o AngularJS, facilmente se aprende a utilizar o Protractor.

O AngularJS utiliza algumas técnicas específicas para manipular o DOM, inserindo ou extraindo informações do HTML. Exemplos disso são a utilização do `ng-model` para a entrada de dados, do *binding* de atributos para exibição de dados no DOM e do `ng-repeat` para apresentação de informações no HTML que estejam contidas em uma lista no javascript. Para capturar um elemento na interface que contenha um `ng-model`, por exemplo, o Protractor provê o comando `by.model("nome")`. Através dele, o framework retorna o `WebElement`, o qual contém a diretiva `ng-model` com o valor `nome`.

```
<div>
  <label>Nome</label>
  <input type="text" ng-model="nome">
  <label>Endereço</label>
  <input type="text" ng-model="endereco">
</div>
```

Dado esse exemplo, quando executado o comando `element(by.model("nome"))`, o Protractor retorna o `WebElement` a seguir:

```
<input type="text" ng-model="nome">
```

Para capturar elementos aos quais o AngularJS faz *binding* de alguma informação, o comando é um pouco diferente, mas a lógica é a mesma do `model`. O exemplo a seguir demonstra como utilizar:

```
<div>
<label>{{name}}</label>
<label>{{endereco}}</label>
```

```
<label>{{CEP}}</label>
</div>
```

Aqui, tem-se um código em que foram feitos alguns *bindings* de informações através do AngularJS. Ao executar o comando `element(by.binding("endereco"))` pelo Protractor, é retornado o `WebElement` adiante:

```
<label>{{endereco}}</label>
```

Na utilização do `ng-repeat`, o Protractor fornece o comando `by.repeater("aluno in alunos")`, o qual retorna um array de elementos. Esse comando permite encadear as chamadas funções `row` e `column` para tornar a busca mais específica.

Encadeado com o comando `row`, retorna o elemento em que o `ng-repeat` está, porém com as informações do objeto na posição passadas por parâmetro ao comando `row`. Por exemplo: `by.repeater("aluno in alunos").row(0)` retorna o primeiro elemento da lista de nomes, pois o javascript inicia a contagem em 0. Caso o objetivo seja buscar um elemento específico dentro da estrutura do *repeater*, é possível utilizar, ainda, o comando `column` encadeado com os outros dois anteriores. Nesse comando, passa-se o nome do atributo ao qual está sendo feito *binding* no HTML e, com isso, o Protractor retornará o `WebElement` ao qual o *binding* está sendo feito.

O exemplo a seguir demonstra como funcionam esses comandos.

```
<div ng-repeat="aluno in alunos">
<span>{{aluno.nome}}</span>
<span>Nota {{aluno.nota}}</span>
<input type="text" ng-model="aluno.nota"/>
</div>
```

O código é uma estrutura HTML criada para rodar uma aplicação AngularJS. Dado que uma lista de alunos seja um array com os nomes André, Fernando e José, o HTML gerado pelo AngularJS ficaria da seguinte forma:

```
<div ng-repeat="aluno in alunos">
  <span>André</span>
  <span>Nota 8</span>
```

```
<input type="text" ng-model="aluno.nota"/>
</div>
<div ng-repeat="aluno in alunos">
  <span>Fernando</span>
  <span>Nota 9</span>
  <input type="text" ng-model="aluno.nota"/>
</div>
<div ng-repeat="aluno in alunos">
  <span>José</span>
  <span>Nota 7</span>
  <input type="text" ng-model="aluno.nota"/>
</div>
```

A partir desse exemplo, pode-se entender como funciona o comando `by.repeater` do Protractor.

Em caso de execução do comando `element(by.repeater("aluno in alunos"))`, o Protractor irá retornar toda a estrutura HTML apresentada.

Executando o comando `element(by.repeater("aluno in alunos").row(1))`, é retornada apenas a estrutura de HTML a seguir:

```
<div ng-repeat="aluno in alunos">
<span>Fernando</span>
<span>Nota 9</span>
<input type="text" ng-model="aluno.nota"/>
</div>
```

E, no caso mais específico, para buscar um elemento final, executando o comando `element(by.repeater("aluno in alunos").row(1).column("nota"))` o Protractor retorna apenas a tag `span` adiante:

```
<span>Nota 9</span>
```

Após conhecidos alguns dos comandos do Protractor, são criados testes para demonstrar, na prática, o uso deles. Assim, utiliza-se o exemplo do site do AngularJS (<http://angularjs.org/>) apresentado na sequência:

```

<div ng-repeat="aluno in alunos">
  <span>André</span>
  <span>Nota 8</span>
  <input type="text" ng-model="aluno.nota"/>
</div>
<div ng-repeat="aluno in alunos">
  <span>Fernando</span>
  <span>Nota 9</span>
  <input type="text" ng-model="aluno.nota"/>
</div>
<div ng-repeat="aluno in alunos">
  <span>José</span>
  <span>Nota 7</span><html ng-app>
<head>
</head>
<body>
  <div>
    <label>Name:</label>
    <input type="text" ng-model="yourName"
      placeholder="Enter a name here">
    <hr>
    <h1>Hello {{yourName}}!</h1>
  </div>
</body>
</html>
  <input type="text" ng-model="aluno.nota"/>
</div>

```

Para essa aplicação, pode-se criar 2 testes como exemplo. No primeiro, verifica-se se o texto inicial da tag `h1` é "Hello !", pois não há ainda nenhum valor no atributo `yourName`.

Em um segundo teste, digita-se algo no campo texto que corresponde à entrada do atributo `yourName` e depois, verifica-se novamente o texto dentro da tag `h1` com o novo valor exibido, que deve conter, além de "Hello !", também o texto digitado no campo de entrada.

```

describe('Hello World form', function() {

  it('should display Hello !', function() {

```

```
browser.get('');
expect(element(by.binding('yourName'))
        .getText()).toEqual("Hello !");
});

it('should be able to insert text',function(){
  browser.get('');
  element(by.model('yourName')).sendKeys("Protractor");
  expect(element(by.binding('yourName'))
        .getText()).toEqual("Hello Protractor!");
});
});
```

Verifica-se que no segundo teste há um comando `sendKeys("Protractor")` que não foi mencionado anteriormente. Esse é um método do `WebElement` para digitar valores em um campo texto, conforme utilizado no teste. Colocando esses testes dentro do arquivo `hello_world.js`, conforme configurado no capítulo anterior, eles irão fazer parte da atual suíte de testes.

Agora, é necessário rodar novamente o Protractor, sendo que não mais aparecerá uma mensagem de erro, e sim o status de cada um dos testes executados.

```
Finished in 2.663 seconds
2 tests, 2 assertions, 0 failures
```

A partir desse momento, os testes estão rodando e se pode incrementar a suíte o quanto for preciso ou, ainda melhor, incluí-la em uma ferramenta de integração contínua para manter os testes rodando constantemente.

13.3 POR QUE USAR PROTRACTOR?

Ao desenvolver uma aplicação AngularJS, pode-se usar Protractor para testá-la. O Protractor foi criado exclusivamente para isso, e traz em seu framework uma gama de customizações do Selenium para facilitar a interação dos testes com a aplicação. Não é necessário encher os testes de `sleeps` ou `Waits`, pois

o Protractor já controla tudo isso. A ferramenta é orientada aos conceitos do AngularJS, o que facilita o aprendizado, tanto de quem desenvolve AngularJS e inicia a utilizar Protractor, como o contrário. Ela possibilita uma estruturação baseada no Jasmine, o que facilita a utilização da mesma estrutura tanto em testes funcionais quanto em unitários. O Protractor permite rodar em browsers ou em modo *headless*. O que mais um framework de automação de testes precisa ter?

CAPÍTULO 14

Melhore seus testes

por Marcos Brizen

Neste capítulo, vou compartilhar como melhorar seus testes unitários e evitar dores de cabeça com a manutenção da suíte de testes, descrevendo um conjunto pequeno de práticas. Mesmo não sendo uma lista completa, ela vai prover um caminho inicial e vários exemplos de onde começar.

14.1 TRATE CÓDIGO DE TESTE COMO CÓDIGO DE PRODUÇÃO

Muitas pessoas falam sobre a importância de ter código limpo, várias regras são criadas, como convenções para nomear variáveis, entre outras. Como resultado, temos vários padrões que são seguidos pela maioria dos desenvol-

vedores, como, por exemplo, escrever métodos pequenos.

14.1.1 Repetir ou não se repetir?

O mesmo cuidado com código de produção também precisa ser aplicado no código de testes, mesmo não sendo muito comum ouvir falar especificamente deles. O foco principal é dar a mesma atenção à qualidade do código. No entanto, ela tem um significado diferente no contexto de testes.

Observe o seguinte caso de teste:

```
```ruby
context 'confirm' do
 setup do
 @user = Factory.create_user
 @confirm_action = Proc.new do |item, session|
 login_as @user do
 post :confirm, item, session
 end
 end
 end

 should 'render JSON as response with sale attributes' do
 item = Factory.create_item
 session = {last_login: 1.day.ago }
 @confirm_action[item, session]
 json_response = JSON.parse(@response.body)
 assert json_response.include?('sale')
 end

 should 'render JSON as response without sale attributes
 if item is already sold' do
 sold_item = Factory.create_sold_item
 session = {last_login: 1.day.ago }
 @confirm_action[sold_item, session]
 json_response = JSON.parse(@response.body)
 assert_false json_response.include?('sale')
 end
end
```
```

Este caso de teste possui pouca repetição, pois, como a chamada da ação do *controller* (`confirm`) está embutida dentro do *setup*, cada teste não precisa se preocupar em chamar a ação. Basta chamar `@confirm_action` e passar o item e a sessão como parâmetros.

14.1.2 Legibilidade vs. repetição

Mesmo com pouca repetição, é muito difícil entender o código anterior. A sintaxe utilizada para executar o `Proc` não é muito comum, e até a utilização de `Procs` pode dificultar a leitura do código por um iniciante em Ruby. Vamos fazer uma pequena modificação e avaliar como fica:

```
```ruby
context 'confirm' do
 setup do
 @user = Factory.create_user
 end

 should 'render JSON as response with sale attributes' do
 item = Factory.create_item
 session = {last_login: 1.day.ago }
 login_as @user do
 post :confirm, item, session
 end
 json_response = JSON.parse(@response.body)
 assert json_response.include?('sale')
 end

 should 'render JSON as response without sale attributes
 if item is already sold' do
 sold_item = Factory.create_sold_item
 session = {last_login: 1.day.ago }
 login_as @user do
 post :confirm, sold_item, session
 end
 json_response = JSON.parse(@response.body)
 assert_false json_response.include?('sale')
 end
end
```

```
end
...

```

Utilizar o bloco de código diretamente no teste deixa bem mais claro o que está acontecendo. Não precisamos ficar pulando entre o teste e o setup para ver o que acontece. No entanto, sacrificamos um pouco a simplicidade adicionando repetição, já que cada um dos testes vai ter o bloco que faz o login e chama o *controller* da mesma forma.

### 14.1.3 Qualidade do código de testes

O exemplo citado veio da simplificação de um código real, extraído de um projeto, e mostra bem a dualidade entre legibilidade e simplicidade. Na maioria dos casos, escrever testes que alcançam tanto boa legibilidade quanto simplicidade não é muito difícil, principalmente com a evolução das ferramentas de testes.

Testes oferecem tranquilidade para modificar código de produção, mas o código de produção não dá tranquilidade para modificar testes. O foco principal nos testes deve ser a legibilidade, enquanto que código de produção foca em evitar repetição. É claro que existem várias outras boas características, e legibilidade não necessariamente exclui simplicidade. Mas o foco principal da qualidade é diferente.

Robert Martin (Uncle Bob), autor do famoso livro *Clean Code* [81], dedicou um capítulo inteiro para falar sobre a importância de se ter cuidado com a qualidade dos códigos de teste. Nesse capítulo, ele fala sobre uma equipe que decidiu explicitamente não dedicar tempo para cuidar da qualidade de testes, já que eles não vão para produção, mas acabou perdendo a confiança na suíte de testes.

Uma suíte de testes difícil de ler não ajuda a perceber o que está errado com o código. Testes são uma das mais poderosas técnicas para alcançar a flexibilidade, fazer uma mudança, além de que saber que tudo está bem em poucos minutos traz muita confiança para o time. Portanto, seus testes devem ser robustos e limpos. Mudanças acontecem todos os dias e precisamos estar preparados.

## 14.2 UTILIZE PADRÕES DE TESTES

Padrões de projeto são conhecidos pela flexibilidade e legibilidade que agregam ao código. Para pessoas que conhecem padrões, fica mais fácil entender o código quando reconhecem a utilização de um determinado padrão. Como o foco principal dos testes deve ser legibilidade, vale a pena aprender sobre padrões de testes.

Vou apresentar agora um pequeno subconjunto de padrões bem conhecidos e bastante úteis, ilustrando os variados tipos existentes. O livro *XUnit Test Patterns*, de Gerard Meszaros, descreve um grande conjunto de padrões, técnicas e *code smells* voltados especificamente para testes, que são facilmente reconhecidos em frameworks modernos.

### 14.2.1 Padrões de organização de código

Primeiro, vamos apresentar padrões de testes que focam na organização, evitando repetições e acomodando mudanças facilmente. Um exemplo é o *Arrange Act Assert* ou apenas 3-As. Veja o exemplo de código a seguir:

```
```ruby
should 'render JSON as response with sale attributes' do
  item = Factory.create_item
  assert_false item.sold?
  session = {last_login: 1.day.ago }
  login_as @user do
    post :confirm, item, session
    json_response = JSON.parse(@response.body)
    assert json_response.include?('sale')
    assert_response :success
  end
  assert item.reload.sold?
end
```
```

À primeira vista, não fica claro o que o teste verifica. Temos várias asserções espalhadas pelo código, fica difícil perceber o que realmente importa, mesmo que a descrição do teste mencione apenas que a resposta deve ter os atributos de uma venda.

O padrão 3-As sugere uma divisão e organização do código de testes em:

- **Arrange (Organizar):** configure toda a informação necessária para o teste;
- **Act (Ação):** faça a chamada para o que você irá testar;
- **Assert (Verifique):** veja se o que era esperado realmente aconteceu – ou não.

Poderíamos refatorar o código, apenas mudando linhas de lugar e melhorar a legibilidade:

```
```ruby
should 'render JSON as response with sale attributes' do
  #Arrange
  item = Factory.create_item
  session = {last_login: 1.day.ago }
  #Act
  login_as @user do
    post :confirm, item, session
  end
  #Assert
  json_response = JSON.parse(@response.body)
  assert json_response.include?('sale')
  assert_response :success
  assert item.reload.sold?
end
```
```

Organizar e separar o seu código facilita o entendimento do que esta sendo testado. Outras pessoas que também são familiares com o padrão vão rapidamente entender o que cada parágrafo de código é e que tipo de informação encontrarão lá.

Outro padrão semelhante é o *Given When Then*, que alcançou grande audiência com a popularidade do *Behaviour Driven Development* (BDD). A divisão e organização de cada sessão também são bem parecidas:

- **Given (Dado):** pré-condições para que o teste seja executado;

- **When (Quando):** comportamento que está sendo descrito;
- **Then (Então):** mudanças esperadas depois do comportamento descrito.

A diferença entre os dois padrões está no foco que eles dão. O *3-As* foca em organizar a estrutura interna do código, já o *Given When Then* visa facilitar escrever testes que podem ser lidos por pessoas que não programam. *3-As* e *Given When Then* são apenas os mais conhecidos, existem vários outros padrões de organização de código, basta encontrar o que melhor se aplica às suas necessidades.

#### 14.2.2 Padrões de banco de dados

Um bom exemplo de padrão de teste voltado para banco de dados é a *Caixa de Areia*, que ficou popular com o framework Rails. A ideia é criar um banco de dados separado para que testes não sejam influenciados por mudanças externas enquanto executam.

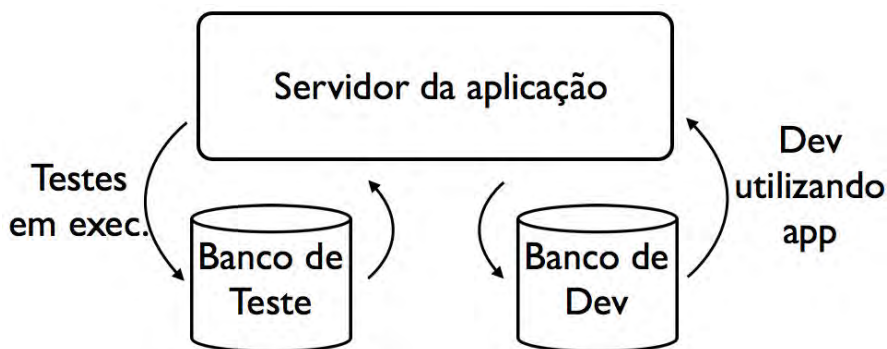


Fig. 14.1: Testes executando em um banco de dados Caixa de Areia.

Outra utilização do mesmo padrão permite a execução de testes em paralelo sem que um lote interfira com outro. Ferramentas de paralelização geralmente criam um banco de dados diferente, mas com mesmo esquema, para cada agente de execução:

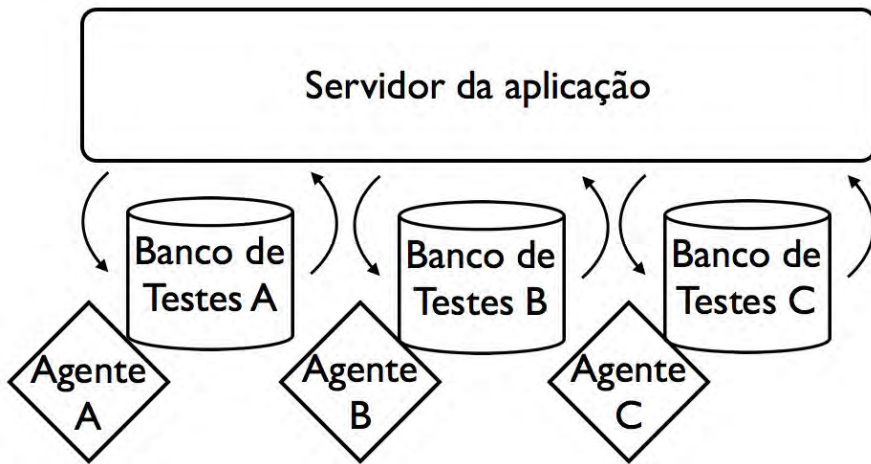


Fig. 14.2: Testes simultâneos utilizando banco de dados Caixa de Areia diferentes

### 14.2.3 Padrões de verificação de resultado

Um padrão com foco nas asserções é a *Asserção Não Terminada*. É comum encontrar marcações de `TO-DO` no código; geralmente, elas indicam algo que precisa ser revisitado depois. A ideia do padrão é fazer com que o teste falhe propositalmente quando não estiver completo, garantindo que o código não seja esquecido na cabeça de alguém.

A implementação pode ser bem simples, como:

```
```ruby
should 'render JSON as response with sale attributes' do
  item = Factory.create_item
  session = {last_login: 1.day.ago }
  login_as @user do
    post :confirm, item, session
  end
  json_response = JSON.parse(@response.body)
  assert json_response.include?('sale')

  assert false,

```

```
        'Need to check what sales attributes are important'  
    end  
    ...
```

Assim, ao executarmos a suíte de testes, essa asserção vai falhar e a mensagem de erro indicará o que precisa ser revisto. Muitos frameworks possuem uma forma de ignorar o teste quando ele é marcado como pendente, assim o código pode ser integrado sem o risco de quebrar o *build*, mas vai exibir um aviso enquanto a pendência não for resolvida:

```
```ruby  
should 'render JSON as response with sale attributes' do
 item = Factory.create_item
 session = {last_login: 1.day.ago }
 login_as @user do
 post :confirm, item, session
 end
 json_response = JSON.parse(@response.body)
 assert json_response.include?('sale')

 pending 'Need to check what sales attributes are important'
end
```
```

14.2.4 Vários outros tipos de padrões

Você já deve estar utilizando muitos dos padrões mostrados aqui e vários outros, mesmo sem saber. No entanto, é interessante ter consciência do que está acontecendo para tomar melhores decisões. Caso note que um padrão pode lhe ajudar, vale a pena investigar sua aplicação e avaliar os benefícios de utilizá-lo.

14.3 EVITE TESTES INSTÁVEIS

“Os testes falharam de novo, mais um build quebrado.”

– um desenvolvedor

“Basta rodar de novo, esse teste tá passando na minha máquina.”

– outro desenvolvedor

Se você já esteve em uma situação em que ouviu “é só rodar de novo”, ou até mesmo se já falou isso para alguém, então você tem um problema. Testes que não trazem confiança é como não ter teste nenhum. Toda a confiança de esperar alguns minutos e saber que tudo está funcionando como deveria é jogada fora.

14.3.1 Testes não determinísticos

Martin Fowler descreve este tipo de teste como *Não Determinístico*: testes que passam, mas falham algumas vezes, sem nenhuma causa aparente. Este tipo de problema geralmente afeta testes que possuem um grande escopo, como os de aceitação que fazem uso de um navegador para acessar o sistema como um usuário.

Quando se tem uma suíte estável, ver que ela falhou lhe garante um enorme benefício: saber o que introduziu o problema. Se os testes quebraram depois que você enviou seu código para o servidor de integração, você já está a meio caminho de encontrar o código com falha. Se a suíte falha aleatoriamente, é necessário gastar um bom tempo tentando resolver a causa do problema para só então descobrir se a sua mudança causou algum mal. Muitas vezes esse processo é bem demorado e pode tirar a sua vantagem de saber imediatamente quando o problema foi introduzido.

Outro problema de uma suíte de testes instável é saber se os testes estão instáveis ou se o sistema está com um comportamento instável. Nesses casos, é necessária uma grande investigação para encontrar possíveis causas, que muitas vezes não se mostra muito produtiva.

14.3.2 Possíveis causas de instabilidade

O livro *XUnit Test Patterns* fala bastante sobre testes frágeis e suas possíveis causas e remédios. Os exemplos de causas mais comuns que já vi são sensibilidade de dados e sensibilidade de contexto.

Sensibilidade de dados

A sensibilidade de dados fica clara quando alterações nos dados ocorrem fora do teste, fazendo com o que ele falhe ou passe sem relação direta com o código que está sendo testado. Por exemplo, a estratégia de limpeza do banco de dados pode influenciar o teste, caso a tabela seja truncada ou limpa, apenas removendo os dados.

EFEITOS COLATERAIS

Em um projeto anterior, utilizávamos uma *gem* para limpar o banco de dados entre a execução de testes e ela oferecia diferentes estratégias de limpeza de banco. Em uma determinada distribuição de Linux que estava sendo utilizada, notamos que a limpeza do banco tomava muito mais tempo do que em outras distribuições. Descobrimos que esse era um problema conhecido com a versão do banco de dados e a formatação do disco rígido utilizada pela distribuição Linux.

Modificamos a estratégia de limpeza para fugir do problema, porém, ao fazer isso, vários testes começaram a quebrar! A maneira como o banco de dados era limpadado influenciava os testes, fazendo-os falhar, eventualmente. A solução nesse caso foi rever os testes instáveis e entender a dependência com o banco de dados, o que não foi simples de resolver, mas valeu a pena para termos de volta uma suíte confiável.

Sensibilidade de contexto

Já a sensibilidade de contexto ocorre quando o teste depende de um conjunto bem específico de condições para ser executado, mas o próprio teste não tem controle. Um teste que dependa do tempo tem alta probabilidade de se tornar instável caso o framework não permita boas maneiras de manipulação temporal.

Dependência com serviços externos são outro bom exemplo de sensibilidade de contexto. Suponha que uma página precise chamar um serviço externo para exibir informação ao usuário. Caso esse serviço demore mais do que o normal para responder ou esteja completamente inaccessível, os seus tes-

tes vão falhar independente de mudanças. Até mesmo serviços internos que fazem processamento assíncrono, como *Jobs* e *Workers*, podem causar instabilidade.

CONTEXTO PARALELOS

Um dos projetos em que trabalhei possuía uma suíte de testes de aceitação que utilizava Selenium e tinha uma alta quantidade de testes através da interface. A quantidade era tão grande que rodar todos eles em uma única máquina sem nenhuma paralelização demoraria cerca de oito horas para terminar. A solução adotada foi executar a suíte em um servidor de integração distribuindo os testes entre oito máquinas diferentes. Assim, conseguimos reduzir o tempo total de execução para cerca de uma hora. O problema que surgiu foi que muitas vezes eles falhavam, pois o browser não conseguia ser inicializado em algumas das máquinas pois ainda estava executando outro grupo de testes. Inicialmente, executá-los novamente pareceu uma boa solução, mas, quando se passaram semanas sem que o *build* ficasse verde, decidimos investigar melhor o problema. No fim, investimos em atualizar a infraestrutura de testes para suportar novas versões. Após alguns dias de esforço dedicado, conseguimos voltar a ter uma suíte menos instável.

14.3.3 Melhorando a instabilidade

Uma vez que você se encontra com uma suíte de testes instável em suas mãos, existem algumas técnicas que podem ser utilizadas para se ter de volta uma suíte saudável.

Para descobrir se determinado teste é instável ou não, uma boa técnica é a da Quarentena. A ideia é analisar os últimos *logs* de sua suíte, verificar quais falham comumente e separá-los do resto dos seus testes. Mesmo diminuindo a cobertura, você vai ter uma suíte confiável e vai poder se focar nos testes em quarentena.

O livro *XUnit Test Patterns* apresenta um diagrama com uma série de perguntas e possíveis causas de instabilidade. Mas não existe mágica, muitas ve-

zes é necessário olhar *logs*, colocar pontos de parada e adicionar mais e mais *logs*.

Mesmo com muita investigação, nem sempre conseguimos encontrar as causas. Uma saída é reescrever os testes, seja de uma maneira diferente ou em um nível diferente. Claro que um teste unitário não tem a mesma cobertura que um de aceitação, mas essa pode ser uma maneira de obter uma suíte estável de volta.

A solução ideal para o problema é não ter o problema! Evitar que testes instáveis apareçam é o melhor que pode ser feito. Ignorar um *build* vermelho e enviar o código mesmo assim é o primeiro passo para criar testes não determinísticos. Reforçar a disciplina de não enviar código quando a suíte estiver vermelha, a não ser que seja para consertá-la, é a melhor solução.

14.4 TESTE NO NÍVEL APROPRIADO

Além de tomar todos os cuidados descritos até agora, é preciso também escrever a quantidade certa de testes. A princípio, pode parecer que, quanto mais testes existirem na sua suíte, melhor, mas, com o passar do tempo, tê-los demais pode causar um grande problema: lentidão.

14.4.1 Custos e níveis de testes

Imagine que você tem uma suíte de teste que lhe dá uma cobertura de 100%, mas que demora uma hora e meia para executar por completo. Dificilmente todos os desenvolvedores vão esperar todo esse tempo para cada mudança de código, é mais provável que eles acumulem mudanças para serem testadas de uma só vez. E aquela vantagem de saber exatamente o que introduziu o problema vai pela janela.

Todo teste que você escreve tem um custo e é importante entendê-lo. Pessoas que não gostam de testes automatizados reclamam logo que eles aumentam o tempo de desenvolvimento, pois precisamos escrever o código de produção e o teste. No entanto, executá-los também tem um custo, que muitas vezes é esquecido.

Os testes podem ser divididos de maneira bem simples em três tipos: Aceitação, Serviço e Unitário. Cada tipo possui um escopo diferente, bem

como custos para escrevê-los e mantê-los.

Testes unitários são os mais conhecidos e mal interpretados. Eles focam em um único comportamento de cada vez e são executados muito rapidamente, pois as ações que eles executam são mínimas. No entanto, este tipo de teste não oferece uma cobertura muito grande, já que, como as ações são mínimas, a integração entre serviços fica de lado.

Testes de serviço, ou de integração, têm um escopo maior que os unitários, portanto dão maior cobertura. Eles geralmente não vão fazer muita utilização de *Dublês de testes* (ou *Mocks*, como são mais conhecidos) e testam a comunicação entre serviços diferentes, até mesmo em aplicações diferentes. A desvantagem é que eles podem demorar um pouco mais que os unitários e possuem altos riscos de se tornarem instáveis.

Testes de aceitação, ou de interface, vão testar o sistema do ponto de vista do usuário, tendo o maior escopo entre os três, permitindo testar praticamente qualquer funcionalidade. Um teste que abre o navegador e interage com o sistema é um bom exemplo de teste de aceitação. O problema com eles é que, assim como os de serviço, eles demoram muito para serem executados e são extremamente sensíveis ao contexto.

14.4.2 Pirâmide de testes

Ter uma suíte de testes rápida, mas de baixa cobertura, não é melhor. Problemas que passam despercebidos são piores do que aqueles que demoram para ser encontrados. A chave está em encontrar uma balança entre cobertura e velocidade. Uma das soluções mais conhecidas é a **Pirâmide de testes**, de Mike Cohn.

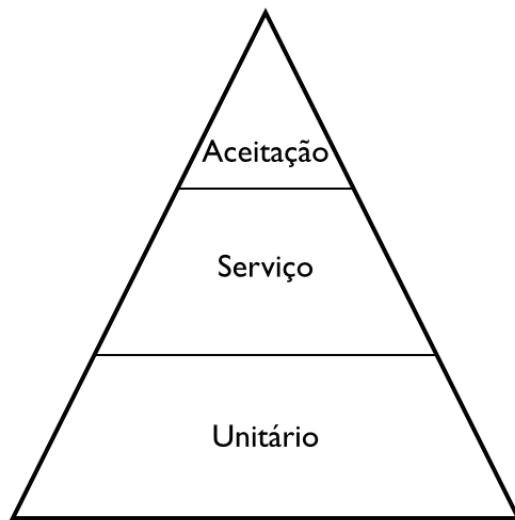


Fig. 14.3: Balanceamento de quantidade com Pirâmide de testes

A ideia da pirâmide é que você tenha uma quantidade maior de testes unitários, pois executam rapidamente e possuem uma cobertura aceitável, menos testes de serviço, utilizando-os apenas para validar pontos de integração, e a menor quantidade possível de testes de aceitação.

Para definir qual o melhor nível para escrever o teste, é necessário primeiro identificar o que vai ser testado. Uma sequência de diferentes entradas para uma ação de um controlador pode ser testada tanto em um teste de aceitação que abre o browser, modifica os valores na interface e submete clicando em um botão, quanto em um nível unitário chamando a ação do controlador e passando os valores diretamente.

O objeto do teste vai ser testado em ambos os casos. No entanto, o foco do teste unitário fica exatamente na ação do controlador e não nos *ids* dos elementos da página. Quanto menor for o foco, mais rápido e fácil é manter o teste.

14.4.3 Antipadrões de balanceamento

Um antipadrão à Pirâmide de testes é a **Casquinha de sorvete** criada por Alister Scott. Neste cenário, existe uma grande quantidade de testes manuais e de aceitação e poucos testes de serviço ou unitários. A imagem da pirâmide ficaria invertida como em uma casquinha de sorvete.

Mesmo assim, ainda é possível ter uma boa cobertura, mas confiar muito em testes mais lentos diminui o ciclo de feedback de desenvolvimento e aumenta a dificuldade de investigar problemas no código, já que rodá-lo não é tão simples e rápido.

Outro antipadrão à Pirâmide de testes é o **Cupcake de testes** descrito por Fabio Pereira. Nele, a equipe tenta alcançar o máximo possível de cobertura nos testes em todos os níveis.

Além do problema da lentidão, devido à grande quantidade de testes manuais e de aceitação, existe muita duplicação de esforços, que também é considerada uma forma de desperdício no desenvolvimento de software.

14.4.4 Como diminuir o problema

Fazer com que os testes se mantenham balanceados é a melhor solução. Fique de olho na quantidade de testes escritos e no tempo que eles demoram para serem executados. Existem ferramentas que apontam os mais lentos: talvez valha a pena investigá-los ou até mesmo reescrevê-los em níveis mais baixos.

Rebalancear os testes não é a única solução para uma suíte lenta. Paralelização, local ou entre computadores em uma mesma rede, requer um esforço inicial para configurar a infraestrutura, mas gera bons resultados sem precisar modificar a suíte de testes.

Outra solução é dividir a suíte e executá-la em partes. Por exemplo, uma suíte de testes de aceitação que demora muito pode não fazer parte dos testes que são executados antes de enviar código, mas é executada pelo servidor de integração em paralelo. No entanto, isso exigiria uma grande disciplina para monitorar o *build* e encontrar problemas o quanto antes.

14.5 CONCLUSÃO

Como dito antes, mesmo não sendo uma lista completa, as práticas aqui descritas devem prover informações suficientes para que você possa aprender mais sobre testes. Cada passo pode e deve ser estudado com mais profundidade, principalmente pelo fato de que o contexto do projeto vai impactar diretamente nas suas decisões.

Tente voltar para este texto de tempos em tempos e olhe com mais atenção cada uma das práticas e procure mais informações, leia as referências citadas neste capítulo e melhore cada vez mais sua habilidade de escrever testes!

CAPÍTULO 15

Entendendo e utilizando dublês de teste

por Marcos Brizen

Dublês de teste ou *Mocks*, como são geralmente conhecidos, ajudam a testar e a projetar as interações dos testes com os componentes do código, deixando-o mais coeso e com baixo acoplamento. Existem diversos tipos de dublês, e o primeiro passo para utilizá-los bem é entendê-los.

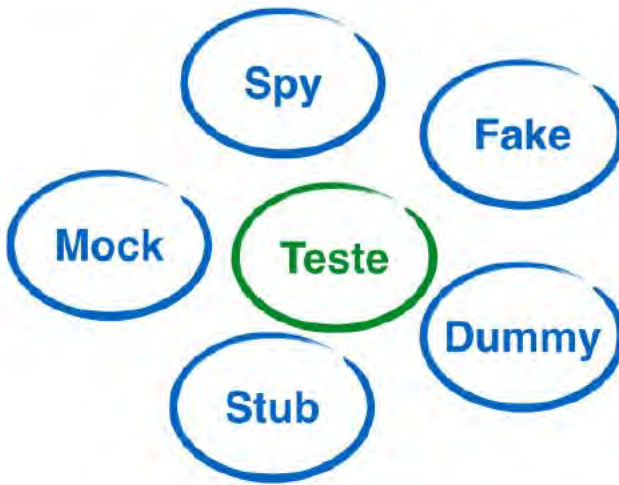


Fig. 15.1: Qual tipo de dublê usar?

Vamos avaliar 5 dublês: *Stubs*, *Spy*, *Mock*, *Fake* e *Dummy*. Para cada um deles, será exemplificado o uso e no final serão discutidos problemas com a má utilização.

15.1 STUB

O tipo mais conhecido de dublê talvez seja o *Stub*. Um *Stub* é utilizado para substituir um comportamento, geralmente um método que deixa de ser executado. Ele também pode retornar um valor absoluto quando é chamado, caso o teste dependa do resultado mas não da execução do comportamento.

Exemplo de uso

Vamos supor que temos um método que busca, em uma lista de produtos, quais estão com preço superior a 20% da média de preços. A média de preços vem de um serviço externo, então precisamos chamar esse serviço para cada item da lista e verificar se o valor do item + 20% é superior ao preço médio. Uma implementação poderia ser a seguinte:

```
def find_overpriced_items_in(items_list)
    items = Item.find(items_list)
```

```
items.select do |item|
  average_price =
    AveragePrice::Service.get_average_price(item.global_id)
  item.price > (average_price * 1.2)
end
end
...
```

Para testar esse método, poderíamos utilizar `Stubs` para que a lógica de chamada ao serviço não seja executada e o teste não dependa da chamada externa:

```
should "return all overpriced items in the list" do
  overpriced_item = Factory.create_item :price => 125
  AveragePrice::Service.stubs(:get_average_price).returns(100)
  assert_equal [overpriced_item],
    find_overpriced_items_in([overpriced_item])
end
...
```

Neste exemplo, qualquer chamada ao serviço sempre vai retornar 100, independente do parâmetro. A maioria dos frameworks de testes atuais permite uma maior flexibilidade aos `Stubs`, como retornar valores diferentes dependendo do parâmetro passado. Poderíamos deixar o teste mais rico passando dois itens, um que está acima da média, e outro que não está:

```
should "return all overpriced items in the list" do
  overpriced_item = Factory.create_item :price => 125
  item = Factory.create_item :price => 220
  AveragePrice::Service.stubs(:get_average_price).
    with(overpriced_item.global_id).returns(100)
  AveragePrice::Service.stubs(:get_average_price).
    with(item.global_id).returns(200)
  assert_equal [overpriced_item],
    find_overpriced_items_in([overpriced_item, item])
end
...
```

Outra boa utilização de `Stubs` é quando precisamos lidar com tempo. É muito difícil manipular o tempo diretamente, e utilizar *sleeps* no meio dos

testes não deve ser uma opção. Para exemplificar, pode-se criar um método que liste todos os itens promocionais não vendidos nas últimas 2 horas:

```
def promo_items_not_sold
  PromoItems.all(
    :conditions => ["created_on <= ? AND sold != ?",
                   DateTime.now - 2.hours, true])
end
...
```

O teste deste método poderia criar um item e utilizar um `Stub` em `DateTime.now` para que, ao executar o método, o tempo atual seja igual a duas horas no futuro e o item promocional não vendido seja retornado:

```
should "return list of promotional items that were
      not sold in the last 2 hours" do
  promo_item = Factory.create_promo_item
  DateTime.stubs(:now).returns(item.created_on + 2.hours)
  assert_equal [promo_item], promo_items_not_sold
end
...
```

`Stubs` em alto nível podem acabar por criar **Testes Tautológicos**, que verificam o valor que é diretamente retornado pelo `Stub`. É recomendável utilizar `Stubs` no menor nível possível que faça sentido para o teste. Se o objetivo é evitar acesso ao banco de dados, é melhor utilizar `Stubs` apenas nos métodos do framework que fazem a conexão com o banco de dados, em vez de métodos do modelo. Dessa forma, a maior parte do comportamento da aplicação continuará sendo utilizada e testada.

15.2 FAKE

Um `dublê Fake` é bem parecido com o `Stub`, ele também substitui um comportamento, garantindo que uma dependência externa não seja utilizada dentro do caso de teste sem necessidade. Geralmente, o `Fake` é um objeto completo e não apenas um método.

Exemplo de uso

No exemplo do Stub, fizemos com que o método `get_average_price` de `AveragePrice::Service` sempre retorne 100. Vamos supor, agora, que vários testes executem a chamada ao serviço. Seria interessante usar `Fake`, que retorna sempre o mesmo valor, e empregar `Stubs` apenas para os testes que precisam de um resultado específico da chamada ao serviço.

```
module AveragePrice
  class Service
    def get_average_price(global_id)
      150
    end
  end
end
...
```

Antes de rodar os testes, poderíamos carregar esta classe para que ela fosse sobrescrita e qualquer chamada ao serviço retornaria 150. Também seria possível estabelecer um conjunto específico de valores para `global_id` e fazer o `Fake` retornar valores diferentes:

```
module AveragePrice
  class Service
    ITEM_PRICE_100 = 1
    ITEM_PRICE_200 = 2
    def get_average_price(global_id)
      return 100 if(global_id == ITEM_PRICE_100)
      return 200 if(global_id == ITEM_PRICE_200)
      150
    end
  end
end
...
```

O teste poderia criar um item utilizando um `global_id` conhecido e não precisaria utilizar `Stubs`:

```

should "return all overpriced items in the list" do
  overpriced_item = Factory.create_item :price => 125,
    :global_id => AveragePrice::Service::ITEM_PRICE_100
  item = Factory.create_item :price => 220,
    :global_id => AveragePrice::Service::ITEM_PRICE_200
  assert_equal [overpriced_item],
    find_overpriced_items_in([overpriced_item, item])
end
...

```

A diferença principal do `Fake` é que ele possui valor de negócio. Apesar do exemplo simples, `Fakes` podem carregar consigo lógicas complexas que talvez até precisem ter testes próprios.

15.3 DUMMY

O duplê `Dummy` pode ser considerado uma simplificação do `Fake`. Um objeto `Dummy` não carrega nenhum valor e, ao ser utilizado, provavelmente vai retornar apenas nulo. Ele é usado para facilitar o entendimento do teste. Em vez de apenas passar um valor nulo, podemos passar um objeto cujo nome diz o que seria utilizado em um caso real.

Exemplo de uso

O exemplo a seguir supõe que para instanciar um item promocional é necessário passar o item original. No entanto, o método `overpriced?`, que diz se um item está acima do preço, não utiliza o item original:

```

class PromoItem
  def initialize(item)
    @item = item
  end

  def overpriced?
    price >
      (AveragePrice::Service.get_average_price(@item.global_id)*1.2)
  end
end
...

```

Para o caso de teste do método `overpriced?` não é necessário o `global_id` do `@item`, pois nele será utilizado um `Stub`. Pode-se usar um `Dummy` que retornará nulo, mas deixando claro que é esperado um objeto `Item`:

```
class DummyItem
  def method_missing(method, *args, &block)
    nil
  end
end
...
```

Em Ruby, podemos utilizar o método `method_missing`, que é chamado quando um método não é encontrado, para retornar nulo em qualquer chamada no objeto. Dessa forma, ao chamar `global_id` no objeto de `DummyItem`, será retornado nulo. O caso de teste poderia ser:

```
should "return false if item price is
      less or equal than 20% plus average price" do
  dummy_item = DummyItem.new
  promo_item = PromoItem.new(dummy_item, :price => 100)
  AveragePrice::Service.stubs(:get_average_price).returns(120)
  assert_false promo_item.overpriced?
end
...
```

Objetos `Dummy` também podem ter sua própria implementação, por exemplo, um método que nunca deve ser chamado poderia lançar uma exceção alertando o fato de que, para aquele exemplo, é necessário um objeto real.

Dublês do tipo `Fake` e `Dummy` geralmente são utilizados globalmente para evitar a repetição dentro dos testes. No entanto, podem ser empregados em um teste mesmo que a pessoa que os escreveu não tenha a intenção de usá-los, por isso é importante deixar claro para a equipe que esses dublês existem e onde podem ser encontrados.

15.4 MOCKS

Mocks são *Dublês* que não se importam muito com valores específicos sendo retornados. A ideia de utilizá-los é saber se determinado método foi chamado ou não, com quais parâmetros e quantas vezes. *Mocks* possuem asserções próprias para verificar se as condições especificadas ocorreram ou não.

Exemplo de uso

Supondo que queremos testar o mesmo código do exemplo anterior, mas que, ao executar o método `overpriced?` de `PromoItem`, o serviço `AveragePrice::Service` deve ser chamado uma única vez e com o parâmetro `global_id` do item. Utilizando um `Mock`, poderíamos fazer o seguinte caso de teste:

```
should "use AveragePrice service to get the average price" do
  item = Factory.create_item(:global_id => 99)
  promo_item = PromoItem.new(item, :price => 100)
  AveragePrice::Service.expects(:get_average_price).
    with(item.global_id).once
  promo_item.overpriced?
end
...
```

Mocks são bem parecidos com *Stubs*. A diferença é que, caso o método `get_average_price` não seja chamado conforme especificado, o teste vai falhar. O mesmo acontece caso o teste termine e o método `get_average_price` nunca seja chamado.

Um exemplo mais interessante de *Mocks* pode ser visualizado em um teste no qual queremos garantir que exista uma estratégia de guardar o resultado da chamada do serviço `AveragePrice` em um cache. Queremos que, mesmo que o método `overpriced?` seja chamado mais de uma vez, o serviço só será chamado uma vez:

```
should "use AveragePrice service to get the average price" do
  dummy_item = Object.new
  promo_item = PromoItem.new(dummy_item, :price => 100)
  AveragePrice::Service.expects(:get_average_price).
```

```
    with(promo_item.global_id).once
      promo_item.overpriced?
      promo_item.overpriced?
  end
  ...
```

15.5 SPY

O `spy` é um duplê que também verifica informações sobre chamadas de métodos, porém guarda informações sobre a execução, permitindo que asserções sejam feitas após a ação do teste.

Exemplo de uso

Um método que recebe uma lista de itens e exclui os que estão acima do preço poderia ser como o seguinte:

```
def remove_overpriced_items_in(items)
  items.reject { |item| item.overpriced? }
end
...
```

Podemos utilizar um `spy` para ter certeza de que o método `overpriced?` foi chamado no objeto `item` passado:

```
should "use AveragePrice service to get the average price" do
  item_spy = mock()
  remove_overpriced_items_in([item_spy])
  assert_received(item_spy, :overpriced) { |expect| expect.once }
end
...
```

A principal diferença entre um `spy` e um `mock` é que as expectativas de um `mock` são configuradas antes da ação do teste, enquanto as do `spy` são feitas depois.

Utilizar o `spy` é especialmente útil para visualizar resultados indiretos que não podem ser facilmente verificados com `asserts`. Mas, no geral, o `spy` pode facilmente ser convertido por um `mock`.

Utilizar `Mock` e `Spy` aumenta muito o acoplamento do teste com a implementação, pois eles especificam um determinado método que precisa ser chamado com um conjunto de parâmetros definido. Qualquer refatoração que seja feita pode acabar por quebrar testes. Recomenda-se utilizá-los apenas quando não houver outra maneira de escrever a asserção.

15.6 DUBLÊS DE TESTE E TDD

A técnica de escrever testes antes para guiar o desenvolvimento (TDD, *Test-Driven Development*) é uma abordagem segundo a qual se escreve o código de um teste falhando, em seguida, o código que corrige a falha e, por fim, refatorando o que for possível.

Um dos benefícios de utilizar TDD é o design evolutivo, no qual se pensa primeiro em como um componente será utilizado para só então implementá-lo. O isolamento correto do componente é vital para o desenvolvimento guiado por testes, pois testes fracamente acoplados facilitam mudanças. Dublês são um ferramental essencial para a prática de TDD.

Analisando o código a seguir, podemos ver facilmente que um `PromoItem` interage com um `Item` e com o `AveragePrice::Service`, no entanto, não é necessário ter conhecimento mais aprofundado sobre como criar um item ou sobre qual é a configuração de serviços externos.

```
should "use AveragePrice service to get the average price" do
  dummy_item = DummyItem.new
  promo_item = PromoItem.new(dummy_item, :price => 100)
  AveragePrice::Service.expects(:get_average_price).
    with(promo_item.global_id).once
  promo_item.overpriced?
end
...
```

Em alguns casos, você vai escrever testes que não utilizam dublês. Por exemplo, se fosse escrever um teste que expõe um defeito no código, talvez seja melhor usar objetos reais e deixar explícitas as condições que causam o problema. Entretanto, em caso de execução de TDD e de definição do design do sistema pelos testes, os dublês facilitarão o design evolutivo.

15.7 UTILIZAR DUBLÊS, SIM

Dublês ajudam a testar de acordo com a especificidade da situação e a evoluir o seu código. Sem eles, fica muito difícil remover dependências dos casos de teste e evoluir o design do código. Reconheça a necessidade e decida pelo uso do dublê adequado.

Use, mas não abuse! Lembre-se que o foco não são os dublês, mas sim os testes. Cuide para não exagerar e acabar por não testar o que você deveria estar testando.

CAPÍTULO 16

Desmistificando o BDD

por Juraci de Lima Vieira Neto e Nicholas Pufal

O BDD (do inglês, *Behavior Driven Development* ou, em português, “desenvolvimento orientado por comportamento”) promete resolver diversos problemas de comunicação no processo de um time ágil. No entanto, é fácil ouvir reclamações sobre as aplicações da metodologia em questão. Percebe-se, através de um *case* e do exame das principais reclamações, que boa parte dos times não faz uso correto dessa prática.

Neste capítulo, faz-se uma breve introdução dos fundamentos do BDD. Em seguida, mostram-se as principais reclamações e comenta-se a sua pertinência ou não quanto ao que dispõe o BDD. Para finalizar, apresenta-se um *case* de sucesso, pelo qual, graças ao uso de técnicas inspiradas nos princípios de BDD, tais como uma sessão de “3 amigos”, o time responsável pôde

compreender melhor um domínio complexo de negócio.

16.1 BDD É GERALMENTE MAL COMPREENDIDO

Atenção, este capítulo não é sobre testes, tampouco está relacionado a uma técnica ou prática da fase de testes. Leia-o e entenda por quê.

O BDD se tornou um termo da moda entre desenvolvedores, analistas de qualidade e analistas de negócios. Apesar de suas fortes ideias, é geralmente mal compreendido. Seguidamente, escutam-se times afirmando estar utilizando o BDD, mas, olhando mais de perto, percebe-se que a equipe usa uma ferramenta de BDD para automação de testes, e não aplica os conceitos em si.

No final das contas, escutam-se pessoas reclamando da ferramenta em questão e, ao mesmo tempo, não contextualizando as ideias que inspiraram a criação dela. O resultado disso é uma série de queixas expostas em diversos blogs da internet por pessoas que rejeitam toda a ideia por trás do BDD porque tentaram usar uma ferramenta sem antes mudar de atitude com relação à forma com que desenvolvem software. Na medida em que se começa a compreender quais são, de fato, as ideias por trás do BDD, vê-se que tais reclamações são, por vezes, mal colocadas.

16.2 OS FUNDAMENTOS DE BDD

Apesar de ser um termo da moda e parecer algo completamente novo, o BDD, concebido em 2003 por Dan North, trata-se de uma evolução do TDD (*Test Driven Development*, ou “desenvolvimento guiado por testes”). Na verdade, o próprio Dan North afirma que a sua ideia foi clarificar qual seria a forma correta de interpretar o TDD, tendo em vista que muitos desenvolvedores ainda ficavam em dúvida quanto ao que testar, quando testar e como testar (*Behavior Driven Development*, 2014). O BDD incrementa o TDD de forma a resolver essas questões em aberto, e para tanto, sugere:

- Um fluxo de trabalho mais bem definido, através de uma abordagem *outside-in* (“de fora para dentro”, em uma tradução livre) a qual será descrita no decorrer deste capítulo.

- Uma clara compreensão das funcionalidades por ambas as partes, isto é, pela equipe técnica e pelos *stakeholders*. Para tanto, usa-se linguagem natural e procura-se ser o mais descritivo possível no momento da escrita de cenários de teste e de seus respectivos exemplos, uma vez que estes são, na realidade, especificações de um comportamento desejado, sendo que todos devem ser capazes de contribuir para o seu aprimoramento.
- Funcionalidades que possuam um valor claro e verificável para o negócio, de modo a priorizar o mais importante e evitar o que pode ser desnecessário.
- Reforço da importância do uso de exemplos concretos para descrever uma funcionalidade, seja ela uma especificação de baixo ou de alto nível.
- Um mínimo planejamento futuro, escrevendo especificações para o menor e mais prioritário subconjunto de funcionalidades possível, desenvolvendo-o antes de adicionar mais funcionalidades.

Em suma, o BDD tem como principal objetivo tornar um time mais eficaz através do aprimoramento da comunicação. Isso significa que, independente da função de um membro no time, seja ele um analista de negócios, analista de qualidade ou desenvolvedor, conseguirá fazer uso da sua bagagem cognitiva para auxiliar na clareza com que foi descrita uma funcionalidade da aplicação.

16.3 O BDD APRIMORANDO O TDD

“Eu decidi que deve ser possível apresentar TDD (Test Driven Development, ou desenvolvimento baseado em testes) de uma forma que vá direto às suas coisas boas e evite todas as suas armadilhas.”

– Dan North

Quando Dan North disse isso, referiu-se ao fato de os desenvolvedores estarem realmente perdidos com toda a ideia por trás de TDD. O T (Teste) do TDD levou a uma série de confusões. Ainda nos dias de hoje é muito

comum ouvir um desenvolvedor proferindo comentários como: “Eu não vou escrever todos aqueles testes”, “É um código muito simples, não precisa ser testado” e “Testes são uma perda de tempo”.

Por conta disso, é essencial não se utilizar a palavra teste. Teste transmite a ideia de verificação, o que significa que o sistema já deve implementar todos os requisitos funcionais. Em outras palavras, comunica o sentido de que é uma etapa que deve ocorrer após a implementação do sistema – uma ideia similar a que se tem com modelos clássicos, como o cascata. Quando se fala em BDD, trata-se de comportamento. Por comportamento quer-se dizer que se utiliza boa parte do tempo pensando no que se deseja alcançar.

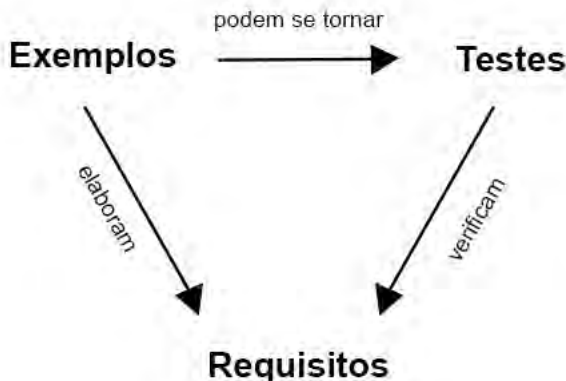


Fig. 16.1: Relação entre exemplos, requisitos e testes.

É por isso que todos os frameworks de teste que se dizem próprios para BDD fazem uso de expressões como *should* (“devo”, em português) e *expect* (“espero que”, em português). É uma maneira de facilitar a troca de paradigma por parte do desenvolvedor, o qual deve focar mais, por exemplo, em como a comunicação entre o objeto A e o objeto B deve ocorrer. Por essa razão é que, em BDD, tem-se o chamado ciclo de desenvolvimento *outside-in* (de fora para dentro), ilustrado pela figura a seguir.



Fig. 16.2: Fluxo de trabalho outside-in.

Esse ciclo é basicamente o mesmo **vermelho-verde-refatorar** que se tem em TDD, apenas se adiciona uma nova camada, a qual introduz as especificações em nível de funcionalidade que, por sua vez, possui exemplos concretos de como uma dada funcionalidade deve se comportar sob a perspectiva do usuário. No ciclo de desenvolvimento, na primeira execução, os cenários vão falhar, visto que não se tem nenhum código implementado. Então, desce-se uma camada e se chega ao nível das especificações de código. Elas também irão falhar. Implementa-se o básico para fazê-las passar, e tem-se a etapa verde. Refatora-se o código, e se mantém as especificações de nível de código passando. A seguir, quando se volta ao nível das funcionalidades, o primeiro cenário deve estar finalmente passando.

16.4 OS TRÊS PRINCÍPIOS DO BDD

Os três princípios a seguir são uma tradução livre dos originais extraídos do capítulo 11 do livro *The RSpec Book*.

- 1) **Nada além do suficiente:** pensar e planejar o design da aplicação a longo prazo não é benéfico no que se refere a valor para o negócio. Não se deve fazer menos do que o necessário para começar, mas qualquer coisa além disso é desgaste desnecessário.
- 2) **Entregar valor aos *stakeholders*:** se você estiver trabalhando em algo que não entrega valor e nem auxilia na habilidade de entregar valor, pare de fazê-lo agora mesmo.
- 3) **Tudo se baseia em comportamento:** tanto em nível de código como de especificações da aplicação, pode-se usar o mesmo pensamento e a mesma linguística para descrever comportamento, independente do nível de granularidade.

16.5 PRINCIPAIS RECLAMAÇÕES SOBRE BDD

A seguir, são descritas as três principais reclamações que foram compiladas após pesquisas em blogs e conversas com pessoas que atuam ativamente no desenvolvimento de software.

16.5.1 O cliente não se importa com testes, logo, não lhe interessa se o time aplica BDD ou não

Essa é a principal reclamação. Tal afirmação faz todo o sentido, visto que, para o cliente, o que realmente importa é um software que atenda às suas necessidades e que funcione. Se você começar uma discussão sobre testes, é muito provável que as pessoas envolvidas com o negócio sintam-se desinteressadas no assunto. Além disso, a palavra teste, infelizmente, carrega consigo uma conotação negativa na comunidade de desenvolvimento de software. Os profissionais da área de testes geralmente são vistos como cidadãos de segunda classe em uma equipe de desenvolvimento.

Mas espere um pouco, aqui se fala de BDD, que é desenvolvimento orientado por comportamento, e isso nada tem a ver com testes. Testar é algo que você não pode fazer enquanto o software não existir. Testar significa verificar e, em BDD, trata-se, antes de mais nada, de especificar. Classificar BDD como uma técnica de testes ou como pertencente à fase de testes é um indício claro de que não se sabe ou não se procurou informação sobre do que essa metodologia realmente trata.

BDD é uma atividade de design, em que você constrói partes da funcionalidade de maneira incremental, guiado pelo comportamento esperado. Em BDD, saímos da perspectiva orientada a testes e entramos na perspectiva orientada a especificações, o que significa que essa reclamação nasceu mal colocada.

16.5.2 O cliente não quer escrever as especificações na ferramenta destinada a isso

Essa é a segunda reclamação mais usada. Falaremos sobre ela em duas partes.

O cliente deve escrever as especificações por conta própria

Quem afirma isso está sugerindo que o cliente proponha a solução para o seu próprio problema, situação que deveria ser resolvida pelo software. Se o próprio cliente escrever as especificações, não irá se beneficiar de algo chamado diversidade cognitiva, aspecto que só aparece em grupos heterogêneos de pessoas trabalhando juntas em prol de um objetivo comum.

O cliente precisa do conselho de desenvolvedores que compreendem os aspectos técnicos do problema que ele está tentando resolver. Também precisa do paradigma de um analista de qualidade, o qual vai auxiliar na criação de cenários que ninguém pensou antes. Caso contrário, a solução encontrada pode ser muito mais complexa do que precisa ser.

Compreende-se que é injusto que o time de desenvolvimento reclame sobre algo que é de sua responsabilidade, ou seja, que ele mesmo deveria resolver para os seus clientes.

O cliente precisa interagir diretamente com a ferramenta de BDD

Essa não é a ideia. O que o cliente realmente precisa fazer é prover o time com informações sobre o problema que ele quer resolver, e juntos, podem pensar nos exemplos concretos que vão nortear o processo de desenvolvimento.

[subsection Consegue-se alcançar o mesmo resultado sem uma linguagem específica de domínio]

Esse argumento alcançar o mesmo resultado sem uma linguagem específica de domínio (DSL) é comumente encontrado entre desenvolvedores. A maior parte deles argumenta que não existe um benefício real em acrescentar mais essa camada, que descreve comportamento em linguagem natural, visto que ela apenas adiciona complexidade e faz com que o conjunto de testes fique lento.

Quando se olha tal reclamação focando uma *tech stack* em Ruby, isso geralmente significa que em vez de usar o Cucumber (Listagem 2), você pode usar o Capybara + RSpec (Listagem 1) para obter o mesmo benefício, e ainda por cima ter uma melhor performance ao rodar seus testes.

Listagem 1. Exemplo de uma funcionalidade descrita utilizando Capybara + RSpec:

```
feature 'Sacar dinheiro de um caixa eletrônico' do
  background do
    user = User.make(name: 'Fulano')
    card = Card.make(user: user, number:
                                     , password: 'foobar')
    Account.make(user: user, card: card, amount: 100.00)
  end

  scenario 'Tentativa de sacar dinheiro usando um cartão
           inválido para minha conta' do
    visit '/atm'

    within("#atm") do
      fill_in 'Cartão', with: '0000-0000-0000-0000'
      fill_in 'Quantia', with: 50.00
      fill_in 'Senha', :with => 'foobar'
```

```
end

click_button 'Sacar'
expect(page).to have_content
'Por questão de segurança, o cartão não será ejetado'
expect(page).to have_content
'Favor entrar em contato com o seu gerente'
end
end
```

Listagem 2. Exemplo de uma funcionalidade descrita com o Cucumber:
Funcionalidade: sacar dinheiro de um caixa eletrônico.

Cenário: tentativa de sacar dinheiro usando um cartão inválido para a conta.

Dado que eu tenho R\$ 100,00 na conta
Mas meu cartão é inválido
Quando eu solicitar R\$ 50,00
Então meu cartão deve ficar preso no caixa eletrônico
E eu devo receber um aviso solicitando para contatar o banco

A verdade é que essa comparação não faz sentido. É como relacionar maçãs e laranjas: ambas são frutas, entretanto bem distintas.

O benefício de se utilizar uma linguagem específica de domínio que pessoas do negócio podem ler – como as especificações escritas no Cucumber, neste caso – vai além do que a perspectiva de um desenvolvedor é capaz de compreender. Não se trata de código e, sim, de aprimorar a comunicação entre todos os membros do time.

Um bom exemplo disso é a diferente forma com que os cenários são descritos em cada uma dessas ferramentas. Conforme se pode ver, em uma ferramenta que particulariza cenários em linguagem natural (Listagem 2), consegue-se alcançar um nível de detalhamento maior, transmitindo uma informação mais clara do que com uma ferramenta de menor nível (Listagem 1).

Isso significa que o benefício real é ter os analistas de negócios dialogando com os desenvolvedores e analistas de qualidade, todos eles aprimorando aquele único arquivo que é uma maneira não abstrata de demonstrar

ideias – o arquivo das especificações, o qual descreve todos os cenários da funcionalidade. Além disso, esses diferentes profissionais usarão cada qual a sua capacidade cognitiva para, juntos, pensarem no melhor caminho para transformar uma especificação na concretização das necessidades do negócio.

16.6 COMUNICAÇÃO E O FLUXO DE TRABALHO

Por mais que equipes ágeis estejam cada vez mais disseminadas e presentes no universo de TI, ainda é muito comum haver certa divisão interna dos times. Nessa separação, comumente se encontra o analista de negócios como o “oráculo”, no que diz respeito ao conhecimento do domínio do negócio. É ele quem acaba interagindo mais com o cliente e, naturalmente, adquirindo um conhecimento mais aprofundado sobre o domínio, enquanto desenvolvedores e analistas de qualidade ficam sem muita ideia do que está acontecendo. O ponto de contato acaba sendo alguma documentação estática, a qual, teoricamente, contém todos os critérios de aceitação, o escopo e os requisitos de que o time necessita para implementar e posteriormente validar procurar por bugs e/ou requisitos mal compreendidos.

Nesse modelo, as informações que estão estáticas na documentação rapidamente se tornam obsoletas. Muitas vezes, existe uma grande diferença entre o que está descrito no documento, o que foi implementado no software e o que o cliente de fato desejava. Além disso, por vezes os requisitos estão documentados de forma imperativa e ambígua, logo, sujeitos a diferentes interpretações por parte de quem os lê. Essa situação ilustra bem o efeito um tanto catastrófico de tais barreiras internas em um time que se considera ágil (figura a seguir).



Fig. 16.3: Barreiras na comunicação do time.

Com isso, chega-se à configuração do fluxo de trabalho ilustrado na próxima figura.



Fig. 16.4: Problemas encontrados apenas mais adiante no processo.

Em um primeiro momento, tem-se as histórias sendo criadas. Tais histórias vão parar na mão de um desenvolvedor, o qual deverá transformar algo vago e ambíguo em algo concreto (software funcional), geralmente baseado em critérios de aceitação e escopo superficiais. Esse desenvolvedor, por sua vez, eventualmente encontrará inconsistências e falhas na descrição da funcionalidade, empurrando-a de volta à fase de análise, para correção. Após tal fase de aperfeiçoamento, um analista de qualidade finalmente irá trabalhar com a referida funcionalidade. Agora, esse analista descobrirá que o desenvolvedor não compreendeu corretamente alguns requisitos, deixando escapar alguns detalhes que são importantes ao negócio. E assim, o ciclo continua.

Não é difícil encontrar situação parecida em times que se dizem ágeis. Tal situação é um processo muito doloroso de vai e vem, que se baseia na descoberta de problemas e de suas respectivas correções.

16.7 SESSÃO DE 3 AMIGOS

Com o intuito de auxiliar na quebra das barreiras citadas no tópico anterior, uma prática muito interessante, que deriva do pensamento colaborativo existente em BDD, é a chamada **sessão de 3 amigos**.

Ao se averiguar a função de analistas de qualidade e de desenvolvedores, chega-se rapidamente à conclusão de que eles são *stakeholders* indiretos, o que significa que devem fazer parte do processo de escrita das funcionalidades. Dessa forma, em vez de haver diversas surpresas ao longo do processo, todos podem debater sobre os diferentes paradigmas juntos, em um estágio inicial.

É exatamente nessa perspectiva que se baseia a sessão de 3 amigos. No momento de escrever ou aprimorar o arquivo texto que detalha uma funcionalidade em linguagem natural (em inglês, conhecido como *feature file*), todos os integrantes do time participam. O nome “3 amigos” vem da interação entre desenvolvedores, analistas de qualidade e analistas de negócios. Sendo assim, antes de qualquer etapa de desenvolvimento, o time deve trabalhar em conjunto para deixar claro qual é a melhor forma de descrever os cenários e quais são os melhores exemplos que tratam de uma dada funcionalidade.

Tal prática traz uma série de benefícios para a comunicação do time. O fato de a equipe usar exemplos concretos – e não apenas requisitos imperativos, que não têm valor algum para aqueles que desconhecem um determinado domínio de negócio – facilita muito que um analista de negócios passe o seu conhecimento para os analistas de qualidade e para os desenvolvedores.

Caso exista um excesso de informações nesse arquivo – tais como um cenário que poderia ser facilmente coberto através de especificações em nível de código, com execução mais rápida – o desenvolvedor sugere ao analista de negócios que reavalie quais cenários são realmente vitais e quais poderiam ser movidos para outro nível de especificação. Da mesma forma, é muito importante a visão de um analista de qualidade, visto que ele pode sugerir alterações que aprimorem o nível de cobertura de um determinado cenário ou até mesmo contribuir sugerindo a criação de um novo cenário.



Fig. 16.5: Membros do time durante uma sessão de 3 amigos.

O processo em questão é totalmente colaborativo, uma vez que todos se beneficiam de visões complementares. A sessão de 3 amigos torna possível que pessoas de diferentes áreas estejam no mesmo patamar de informação quando dado tópico for discutido. Além disso, geralmente o resultado de uma sessão dessas é que:

- o analista de negócios passa a ter uma melhor noção dos desafios técnicos envolvidos;
- o desenvolvedor passa a ter uma ideia mais clara das necessidades de negócio e, portanto, do escopo do que realmente deve ser desenvolvido;
- o analista de qualidade passa a ter um melhor conhecimento sobre quais áreas deverá explorar quando trabalhar na história.



Fig. 16.6: Problemas de comunicação agora resolvidos em um momento inicial.

Através da Sessão de 3 amigos, as falhas de comunicação que antes causariam problemas no decorrer do processo agora podem ser comunicadas e desfeitas em um estágio inicial.

16.8 UM CASO DE SUCESSO USANDO BDD

Quão complicado seria para você explicar para uma criança de 3 anos de idade como funciona uma transação bancária? O mesmo desafio se aplica ao desenvolvimento de software, visto que o domínio do cliente pode, por diversas vezes, ser bastante vago e nebuloso para o time de desenvolvimento.

O ser humano precisa de exemplos para compreender um tópico. Exemplos reais são excelentes formas de se comunicar e, no dia a dia, são usados sem se perceber. Ao trabalhar com exemplos reais, as pessoas se comunicam melhor, pois são capazes de se relacionar com as situações mais facilmente.

Isso tudo é muito mais perceptível quando o domínio do negócio é complexo. Um bom exemplo disso é um projeto de um banco de investimentos com o qual trabalhou a equipe que desenvolveu este artigo. Como é de se esperar em um domínio desses, as terminologias eram muito complicadas, tornando um tanto quanto difícil a vida dos desenvolvedores na hora de manter

um diálogo com os analistas de negócios do banco. Com o intuito de melhorar a comunicação, parte do processo era, antes de começar uma história, ter uma rápida reunião entre o desenvolvedor, o analista de qualidade e o analista de negócios responsáveis por ela.

O analista de negócios compartilhava com os outros o arquivo com as especificações – também conhecido como *feature file* –, o qual continha todos os cenários pensados por ele. O analista de qualidade focava em aprimorar os cenários já criados, sugerindo a elaboração de cenários que ainda não haviam sido explorados, enquanto o desenvolvedor se concentrava nos desafios técnicos da implementação como, por exemplo, sugerir mover um cenário para uma especificação em nível de código – que oferece um feedback mais rápido no conjunto de testes. Assim, todos sugeriam mudanças nos passos de um cenário, caso isso fizesse mais sentido à compreensão global.

Ao utilizar esse processo, os analistas de negócios expandiam o seu conhecimento ao compreender melhor os desafios técnicos de cada cenário e os desenvolvedores conseguiam ter uma ideia mais clara das necessidades do negócio, facilitando a compreensão do que realmente precisava ser desenvolvido. Além disso, sempre que mudanças fossem necessárias, apenas aquele único pedaço de informação seria alterado, o que significa que todo o time estaria sempre atualizado.

16.9 CONSIDERAÇÕES FINAIS

O BDD é uma metodologia ágil e completa. O BDD começou como um simples aperfeiçoamento de TDD, entretanto evoluiu para uma metodologia ágil completa. Por conta disso, o BDD é conhecido como uma metodologia ágil de segunda geração, construída em cima de um pensamento ágil clássico, mas solucionando problemas de uma maneira pragmática. Suas práticas são aplicáveis desde a etapa mais inicial, como a coleta de requisitos, até as etapas envolvidas no ciclo de entrega.

Neste capítulo, procurou-se esclarecer os benefícios do BDD, desmistificando que as suas ferramentas são apenas complementares à metodologia ágil completa. A principal ideia por trás do BDD é a prevenção de falhas de comunicação entre os envolvidos através de um contato frequente, qualitativo

e baseados em exemplos reais, isto é, não somente em abstrações e requisitos imperativos.