DESENVOLVIMENTO ÁGIL LIMPO

Para todos os programadores que já enfrentaram o desafio do método cascata.

1 Introdução à Metodologia Ágil

Em fevereiro de 2001, um grupo de dezessete especialistas em software se reuniu em Snowbird, Utah, para conversar sobre o estado lamentável do desenvolvimento de software. Naquela época, grande parte dos softwares era arquitetada usando processos ineficazes, pesados e cabalísticos, como o Método Cascata e instâncias sobrecarregadas do Processo Unificado da Rational (RUP). O objetivo desses dezessete especialistas era criar um manifesto que introduzisse uma abordagem mais efetiva e mais leve.

Um trabalho nada fácil. Os dezessete especialistas eram pessoas com experiências diversificadas e opiniões fortes e divergentes. Esperar que esse grupo chegasse a um consenso era uma hipótese remota. E, no entanto, contra todas as probabilidades, eles chegaram ao consenso: o Manifesto Ágil foi escrito e nasceu um dos movimentos mais influentes e duradouros no campo de software.

Os movimentos na área de software seguem um caminho previsível. A princípio, existe uma minoria de apoiadores, outra de críticos entusiasmados e uma grande maioria que simplesmente não se importa. Parte dos movimentos morrem nessa fase ou nem saem dela. Pense em Programação Orientada a Aspectos (POA), Programação Lógica ou cartões CRC. Alguns, no entanto, atravessam o abismo e se tornam extraordinariamente populares e controversos. Outros até conseguem superar a controvérsia e basicamente se tornam parte da tendência dominante do pensamento. Um exemplo é a Programação Orientada a Objetos (OOP), e também a metodologia ágil.

Infelizmente, uma vez que um movimento se torna popular, seu nome é corrompido por mal-entendidos e usurpações. Os produtos e métodos que não têm nada a ver com o movimento em si se aproveitam de seus nomes a fim de lucrar com a popularidade e relevância deles. E assim tem sido com a agilidade.

A finalidade deste livro, escrito quase duas décadas após a reunião de Snowbird, é esclarecer as coisas. É uma tentativa de ser o mais pragmático possível, descrevendo o ágil sem baboseira e sem terminologia duvidosa.

Aqui, são apresentados os fundamentos da agilidade. Muitos enriquecem e contribuem com essas ideias — e não há nada de errado nisso. No entanto,

esses enriquecimentos e contribuições são tudo, menos agilidade. Eles são ágeis e mais alguma coisa.

Nesta obra, você lerá o que é a metodologia ágil, o que era e o que inevitavelmente sempre será.

A História da Agilidade

Quando o ágil começou? Provavelmente, há mais de 50 mil anos, no momento em que os humanos decidiram colaborar em prol de um objetivo comum. A ideia de escolher pequenos objetivos intermediários e calcular o progresso após cada um deles é muito intuitiva e humana demais para ser considerada qualquer tipo de revolução.

Quando o ágil começou na indústria moderna? Difícil saber. Imagino que o primeiro motor a vapor, o primeiro moinho, o primeiro motor de combustão interna e o primeiro avião foram produzidos por meio de técnicas que agora chamaríamos de ágeis. A razão disso é que avançar a pequenos passos calculados é bastante natural e humano para que as coisas tenham ocorrido de outra forma.

Mas então, quando o ágil começou no software? Eu gostaria de ter sido uma mosca quando Alan Turing estava redigindo seu artigo de 1936.[1] Acredito que muitos dos "programas" que ele escreveu naquele livro foram desenvolvidos em pequenas etapas com muita verificação. Imagino também que o primeiro código que ele escreveu para o Automatic Computing Engine, em 1946, foi desenvolvido em pequenas etapas, com bastante verificação e até mesmo alguns testes reais.

Os primórdios do software estão repletos de exemplos de comportamento que agora descreveríamos como ágeis. Por exemplo, os programadores que escreveram o software de controle para a cápsula espacial Mercury trabalharam em etapas de meio-dia intercaladas por testes unitários.

1. Turing, A. M. 1936. Em números computáveis, com um aplicativo para o Entscheidungsproblem [proof]. *Proceedings of the London Mathematical Society, 2* (publicado em 1937), 42 (1): 230–65. O melhor jeito de entender este artigo é ler a obra-prima de Charles Petzold: Petzold, C. 2008. *A Guided Tour through Alan Turing's Historic Paper on Computability and the Turing Machine.* Indianapolis, IN: Wiley.

Existe muita coisa documentada em outros lugares sobre esse período. Craig Larman e Vic Basili escreveram uma história resumida do primeiro wiki de Ward Cunningham,[2] e também no livro de Lerman *Agile & Iterative Development: A manager's guide [Desenvolvimento Ágil e Iterativo: Um guia para gerentes].*[3]

Mas o ágil não era a única saída. Na realidade, existia uma metodologia competitiva que obtivera um sucesso considerável na manufatura e na indústria em geral: a Administração Científica, ou Taylorismo.

A Administração Científica é uma abordagem top-down, de comando e controle. Os gerentes usam técnicas científicas a fim de determinar os melhores procedimentos para alcançar uma meta e, em seguida, instruem todos os subordinados a seguirem seu planejamento à risca. Dito de outro modo, um grande planejamento é feito antecipadamente, seguido por uma implementação cuidadosa e detalhada.

Possivelmente, a Administração Científica é tão antiga quanto as pirâmides, Stonehenge ou qualquer outra grande obra dos tempos remotos, pois é impossível acreditar que essas obras poderiam ter sido construídas sem ela. Mais uma vez, a ideia de repetir um processo bem-sucedido é intuitiva e humana demais para ser considerada algum tipo de revolução.

A Administração Científica, ou Taylorismo, recebeu esse nome graças ao trabalho de Frederick Winslow Taylor na década de 1880. Taylor oficializou e comercializou a abordagem e fez sua fortuna como consultor de gerenciamento. A técnica foi um grande sucesso e resultou em ganhos de eficiência e produtividade gigantescos durante as décadas que se seguiram.

E foi assim que, em 1970, o mundo do software estava no cruzamento dessas duas técnicas divergentes. Por um lado, o pré-ágil (a metodologia ágil antes de ser chamada de "ágil") seguiu passos breves e reativos que foram calculados e refinados rumo a um escalonamento, em uma caminhada aleatória direcionada, visando um bom resultado. Do

2. O wiki de Ward, c2.com, é o wiki original — o primeiro a aparecer na internet. Foi de grande ajuda durante um bom tempo.
3. Larman, C. 2004. *Agile & Iterative Development: A manager's guide.* Boston, MA: Addison-Wesley.

outro lado, a Administração Científica postergava a ação até que uma análise completa e um planejamento detalhado resultante tivessem sido elaborados. O pré-ágil funcionava bem em projetos com baixo custo de mudança e resolvia os problemas parcialmente definidos com objetivos especificados de modo informal. A Administração Científica funcionava melhor em projetos com alto custo de mudança e solucionava problemas bem definidos com objetivos extremamente específicos.

A questão era: que tipo de projeto eram os projetos de software? Eles tinham altos custos de mudança e eram bem definidos com objetivos específicos ou tinham baixos custos de mudança e eram parcialmente definidos com objetivos informais?

Não leia muito esse parágrafo anterior. Ninguém, que eu saiba, fez essa pergunta. Ironicamente, o caminho que escolhemos na década de 1970 parece ter sido mais obra do acaso do que a intenção.

Em 1970, Winston Royce produziu um artigo[4] que descreveu suas ideias para gerenciar projetos de software em larga escala. O artigo tinha um diagrama (Figura 1.1) que retratava seu plano. Royce não criou esse diagrama, nem o defendia como um plano. Na realidade, o diagrama servia como falácia do espantalho para que ele desmantelasse nas páginas seguintes de seu artigo.

4. Royce, W. W. 1970. Managing the development of large software systems. *Proceedings, IEEE WESCON,* agosto: 1–9. Acesse em: http://www-scf.usc.edu/~csci201/lectures/Lecture11/royce1970.pdf. [conteúdo em inglês]

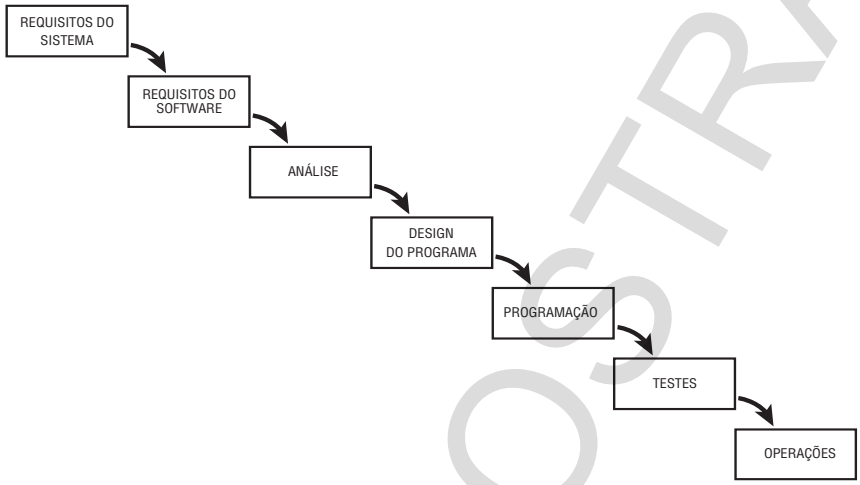

Figura 1.1 Diagrama de Winston Royce que inspirou o Desenvolvimento em Cascata

No entanto, a colocação proeminente do diagrama, e a tendência das pessoas deduzirem o conteúdo de um artigo a partir do diagrama na primeira ou na segunda página, resultaram em uma mudança drástica na indústria de software.

O diagrama inicial de Royce parecia tanto com a água fluindo e descendo sobre algumas pedras que a técnica ficou conhecida como "Cascata".

O Método Cascata descendia, logicamente, da Administração Científica. Tudo era uma questão de fazer uma análise completa, elaborar um planejamento detalhado e, em seguida, executá-lo até concluí-lo.

Ainda que Royce não o recomendasse, era o conceito que as pessoas extraíram do trabalho dele e que dominou as próximas três décadas.[5]

É aqui que entro na história. Em 1970, eu tinha 18 anos e trabalhava como programador em uma empresa chamada ASC Tabulating em Lake Bluff, Illinois. A empresa tinha um IBM 360/30 com 16K de núcleo, um IBM 360/40 com 64K de núcleo e um minicomputador Varian 620/f com 64K

5. Repare que minha interpretação dessa linha do tempo foi contestada no Capítulo 7 de Bossavit, L. 2012. *The Leprechauns of Software Engineering: How folklore turns into fact and what to do about it*. Leanpub.

de núcleo. Programei os 360s em COBOL, PL/1, Fortran e Assembler. No 620/f, eu programava apenas em Assembler.

É importante relembrar como era ser um programador naquela época. Escrevíamos a lápis nosso código em formulários de programação e solicitávamos aos perfuradores de cartões que perfurassem os cartões para nós. Submetíamos nossos cartões cuidadosamente verificados aos operadores de computadores que, por sua vez, executavam nossas compilações e testes durante o terceiro turno, porque os computadores estavam ocupados demais durante o dia fazendo trabalho de verdade. Não raro, demorava-se dias para ir do código inicial à primeira compilação, e cada recuperação subsequente levava em torno de um dia.

Já com o 620/f, era um pouco diferente. Essa máquina era dedicada à nossa equipe, por isso, tínhamos acesso a ela 24 horas por dia, todos os dias. Poderíamos fazer dois, três, talvez até quatro recuperações e testes por dia. A equipe da qual eu fazia parte também era composta de pessoas que, diferentemente da maioria dos programadores do turno diário, sabiam digitar. Desse modo, perfurávamos nossos próprios baralhos de cartões em vez de entregá-los aos caprichos dos perfuradores.

Qual era o processo que usávamos naquela época? Com certeza não era o Método Cascata. Não tínhamos a noção de seguir um planejamento detalhado. Entrávamos no sistema todos os dias, executávamos compilações, testávamos nosso código e corrigíamos os bugs. Era um ciclo infinito que não tinha estrutura. Também não era ágil, e nem pré-ágil. Não existia disciplina no modo como trabalhávamos. Não existia um conjunto de testes nem intervalos de tempo calculados. Era somente código e correção, código e correção, dia após dia, mês após mês.

A primeira vez que li a respeito do Método Cascata foi em uma revista especializada, por volta de 1972. Parecia uma dádiva de Deus para mim. Seria realmente possível analisar o problema de forma antecipada, projetar uma solução para esse problema e depois implementá-la? Poderíamos de fato elaborar um cronograma baseado nessas três fases? Quando concluíssemos a análise, teríamos realmente um terço do projeto? Eu sentia o poder do conceito. Eu queria acreditar porque, se funcionasse, era praticamente um sonho se tornando realidade.

Aparentemente, eu não era o único a me sentir assim, porque muitos outros programadores e escritórios de programação também tinham sido contaminados por esse sentimento. E, como disse antes, o Método Cascata começou a dominar o modo como pensávamos.

Dominou, mas não funcionava. Pelos próximos trinta anos, eu, meus colegas e meus irmãos e irmãs programadores mundo afora, tentamos e tentamos fazer essa análise e design darem certo. Mas sempre que pensávamos que havíamos dominado o Método Cascata, ele escapava por nossos dedos durante a fase de implementação. Todos os nossos meses de planejamento cuidadoso se tornavam irrelevantes devido à inevitável pressa desenfreada, materializada bem diante dos olhos flagrantes de gerentes e de clientes, e criavam terríveis atrasos com os prazos.

Apesar do fluxo quase interminável de falhas, persistíamos com o mindset do Método Cascata. Afinal de contas, como isso poderia dar errado? Como a análise minuciosa do problema, o design cuidadoso de uma solução e a implementação desse design repetidamente não conseguiam dar conta do recado? Era inconcebível[6] que o problema estivesse com a nossa estratégia. O problema deveria ser a gente. De alguma forma, estávamos fazendo algo de errado.

O nível em que o mindset do Método Cascata nos dominava pode ser exemplificado com as linguagens de programação atuais. Quando Dijkstra criou a Programação Estruturada, em 1968, a Análise Estruturada[7] e o Design Estruturado[8] não estavam muito atrás. Quando a Programação Orientada a Objetos (OOP) começou a se popularizar em 1988, a Análise Orientada a Objetos[9] e o Design Orientado a Objetos[10] (OOD) também não ficaram muito atrás. Estávamos à mercê desse trio de conceitos e desse triunvirato de fases. Simplesmente não conseguíamos conceber uma maneira diferente de trabalhar.

6. Assista *A Princesa Prometida* (1987) em versão legendada para ouvir a entonação correta da palava *inconceivable* (inconcebível).
7. DeMarco, T. 1979. *Structured Analysis and System Specification.* Upper Saddle River, NJ: Yourdon Press.
8. Page-Jones, M. 1980. *The Practical Guide to Structured Systems Design.* Englewood Cliffs, NJ: Yourdon Press.
9. Coad, P. e E. Yourdon. 1990. *Object-Oriented Analysis.* Englewood Cliffs, NJ: Yourdon Press.
10. Booch, G. 1991. *Object Oriented Design with Applications.* Redwood City, CA: Benjamin-Cummings Publishing Co.

E então, de repente, conseguimos.

O início da reformulação da agilidade começou no final dos anos de 1980 ou início dos anos 1990. Na década de 1980, a comunidade da linguagem de programação Smalltalk já começava a demonstrar os sinais da agilidade. Também havia indícios dela no livro de Booch de 1991 sobre OOD.[10] Mais resolução apareceu em Crystal Methods, de Cockburn, em 1991. A comunidade de Design Patterns começou a discuti-la em 1994, estimulada por um artigo escrito por James Coplien.[11]

Em 1995, Beedle,[12] Devos, Sharon, Schwaber e Sutherland haviam escrito o famoso artigo sobre o Scrum.[13] E as comportas foram abertas. O bastião do Método Cascata havia sido violado e não tinha mais volta.

E assim, mais uma vez, eu entro na história. O que relato a seguir é proveniente das minhas memórias; não tentei verificar nada com os envolvidos. Logo, você deve estar ciente de que minhas recordações têm muitas omissões e coisas inacreditáveis, ou, no mínimo, um tanto imprecisas. Mas não se assuste, pelo menos tentei ser um pouco divertido.

Vi Kent Beck pela primeira vez na conferência PLOP[14] de 1994, na qual o artigo dele foi apresentado. Era uma reunião casual, e nada de interessante aconteceu. Eu o conheci em fevereiro de 1999 na conferência da OOP em Munique. Mas, dessa vez, eu sabia muito mais a seu respeito.

Na época, eu era consultor em C++ e OOD, voando para tudo quanto é canto e ajudando as pessoas a projetar e implementar aplicativos em C++ empregando técnicas de OOD. Meus clientes começaram a me perguntar sobre o processo. Eles ouviram dizer que o Método Cascata não

11. Coplien, J. O. 1995. A generative development-process pattern language. *Pattern Languages of Program Design.* Reading, MA: Addison-Wesley, p. 183.
12. Mike Beedle foi assassinado em 23 de março de 2018, em Chicago, por um sem-teto com transtornos mentais que havia sido preso e libertado trocentas vezes antes. Ele deveria ter sido internado em uma instituição psiquiátrica. Mike Beedle era meu amigo.
13. Beedle, M., M. Devos, Y. Sharon, K. Schwaber e J. Sutherland. SCRUM: An extension pattern language for hyperproductive software development. Disponível em: http://jeffsutherland.org/scrum/scrum_ plop.pdf.
14. Pattern Languages of Programming foi uma conferência realizada nos anos de 1990 próximo à Universidade de Illinois.

se integrava com a OO e queriam o meu conselho. Concordei[15] sobre a integração da OO com o Método Cascata, e pensei muito no assunto. Até cogitei desenvolver meu próprio processo OO. Felizmente, abri mão dessa ideia logo de início, porque havia me deparado com os escritos de Kent Beck sobre a Extreme Programming (XP).

Quanto mais eu lia sobre XP, mais aficionado ficava. As ideias eram revolucionárias (ou assim eu pensava na época). Elas faziam sentido, sobretudo em um contexto OO (de novo, assim eu pensava na época). Logo, eu estava ávido para aprender mais.

Para minha surpresa, naquela conferência OOP em Munique, eu me vi palestrando do lado oposto a Kent Beck no corredor. Esbarrei com ele durante um intervalo e disse que deveríamos nos encontrar para almoçar a fim de discutir XP. Esse almoço preparou o terreno para uma parceria significativa. Minhas discussões com ele me levaram a viajar para sua casa em Medford, Oregon, com o objetivo de trabalharmos juntos no projeto de um curso sobre XP. Durante essa visita, tive o meu primeiro contato com o Desenvolvimento Orientado a Testes (TDD) e fiquei obcecado.

Naquela época, eu cuidava de uma empresa chamada Object Mentor. Fizemos uma parceria com Kent para oferecer um curso de treinamento de cinco dias em XP, que chamamos de *XP Immersion.* Do final de 1999 até 11 de setembro de 2001,[16] o curso foi um grande sucesso! Treinamos centenas de pessoas.

No verão de 2000, Kent convidou uma determinada quantidade de pessoas da comunidade XP e Patterns para uma reunião perto de sua casa. Ele chamou a reunião de "XP Leadership". Fizemos um passeio de barco e subimos as margens do rio Rogue. E nos reunimos para decidir exatamente o que queríamos fazer com a XP.

Uma das ideias era criar uma organização de XP sem fins lucrativos. Eu era a favor, só que muitos eram contra. Aparentemente, eles tiveram uma experiência negativa com um grupo semelhante de Design Patterns. Deixei

15. Essa é uma daquelas estranhas coincidências que ocorrem de tempos em tempos. Não há nada de especial na OO que a torne menos possível de se integrar com o Método Cascata, e mesmo assim, naqueles dias, essa ideia era dominante.
16. O significado dessa data não deve ser esquecido.

a sessão frustrado, porém Martin Fowler me seguiu e sugeriu que nos encontrássemos mais tarde em Chicago para conversar. Concordei.

No outono de 2000, Martin e eu nos encontramos em uma cafeteria próxima ao escritório da ThoughtWorks, onde ele trabalhava. Contei-lhe sobre a minha ideia de reunir todos os apoiadores de processos simplificados concorrentes com o objetivo de lançar um manifesto unificado. Martin fez diversas recomendações para uma lista de convites e colaboramos em sua elaboração. Naquele dia, mais tarde, enviei todas as cartas-convite. O assunto era *Light Weight Process Summit* ["Reunião sobre Processos Leves", em tradução livre].

Um dos convidados era Alistair Cockburn. Ele me ligou para dizer que estava prestes a realizar um encontro parecido, mas que gostava mais da nossa lista de convidados do que da própria. Ele se ofereceu para combinar sua lista com a nossa e fazer o trabalho de campo de agendar a reunião, se concordássemos em nos reunir na estação de esqui de Snowbird, perto de Salt Lake City.

E assim, a reunião em Snowbird foi marcada.

SNOWBIRD

Fiquei bem surpreso que tantas pessoas concordaram em ir. Tipo, quem é que quer participar de uma reunião cujo nome era "Reunião sobre Processos Leves"? Mas, lá estávamos, na sala Aspen, no Lodge, em Snowbird.

Éramos dezessete. Desde então, fomos criticados por sermos dezessete homens brancos de meia-idade. As críticas são justas até certo ponto. No entanto, pelo menos uma mulher, Agneta Jacobson, foi convidada, mas não pôde comparecer. E, afinal de contas, a grande maioria dos programadores seniores do mundo, na época, eram homens brancos de meia-idade — as razões disso já são uma história para outro livro.

Nós, os dezessete, representávamos alguns pontos de vista diferentes, incluindo cinco processos leves distintos. A turma maior era a equipe XP: Kent Beck, eu, James Grenning, Ward Cunningham e Ron Jeffries.

Em seguida, vinha a equipe do Scrum: Ken Schwaber, Mike Beedle e Jeff Sutherland. Jon Kern representava o Desenvolvimento Orientado à Funcionalidade (FDD) e Arie van Bennekum representava a Metodologia de Desenvolvimento de Sistemas Dinâmicos (DSDM). E, por fim, Alistair Cockburn representava sua Família Crystal de Metodologia.

O restante do pessoal não era muito ligado a nenhum processo. Andy Hunt e Dave Thomas eram os programadores pragmáticos. Brian Marick era consultor de testes. Jim Highsmith era consultor de gerenciamento de software. Steve Mellor estava lá para impedir que desconsiderássemos as regras, porque estava representando a filosofia orientada a modelos, da qual muitos de nós desconfiavam. E, por fim, tínhamos Martin Fowler, que, embora tivesse relações pessoais íntimas com a equipe XP, desconfiava de qualquer tipo de processo de marca e simpatizava com todos.

Não me recordo muito dos dois dias em que nos reunimos. Outros que estiveram lá têm uma lembrança diferente da minha.[17] Então, vou lhe contar o que me lembro, mas são memórias de quase duas décadas de um homem de 65 anos. Não vou lembrar de alguns detalhes, mas a essência provavelmente não se perderá.

Foi acordado, de alguma forma, que eu daria início à reunião. Agradeci a todos por terem vindo e sugeri que nossa missão fosse criar um manifesto que descrevesse o que acreditávamos ser comum em relação a todos esses processos leves e no que dizia respeito ao desenvolvimento de software em geral. Depois, me sentei. Acredito que essa foi a minha única contribuição para a reunião.

Fizemos o tipo de coisa padrão em que escrevemos os problemas em cartões e, em seguida, os colocamos no chão e os classificamos em grupos de afinidade. Nem sei se isso ajudou em alguma coisa; apenas lembro de fazê-lo.

17. Recentemente, publicaram uma história do evento no *The Atlantic:* Mimbs Nyce, C. 2017. The winter getaway that turned the software world upside down. *The Atlantic.* 8 de dezembro. Acesse em: https://www.theatlantic.com/technology/archive/2017/12/agile-manifesto-a-history/547715/ [conteúdo em inglês]. No momento em que eu redigia este texto, não li esse artigo porque não queria que ele contaminasse as lembranças que estou compartilhando aqui.

Não me recordo se a mágica ocorreu no primeiro ou no segundo dia. Parece-me que foi no final do primeiro dia. Talvez os grupos de afinidade tenham determinado os quatro valores: Indivíduos e Interações, Validação do Software, Colaboração com o Cliente e Resposta à Mudança. Alguém os escreveu no quadro branco na frente da sala e teve a brilhante ideia de dizer que aqueles eram preferenciais, mas não substituíam os valores complementares de processos, ferramentas, documentação, contratos e planos.

Essa é a essência do Manifesto Ágil, e ninguém consegue se lembrar muito bem de quem a colocou primeiro no quadro. Lembro-me de ter sido Ward Cunningham. Mas Ward acredita que foi Martin Fowler.

Veja a foto na página agilemanifesto.org. Ward diz que tirou essa foto para registrar aquele momento. Ela mostra claramente Martin no quadro, com muitos de nós ao seu redor.[18] Isso dá credibilidade à versão de Ward de que foi Martin quem teve a ideia.

Por outro lado, talvez seja melhor que nunca saibamos realmente.

Uma vez que a mágica se concretizou, todo o grupo se reuniu em torno dela. Havia algumas melhorias de texto, ajustes e refinamentos a serem feitos. Pelo que me lembro, foi Ward quem escreveu o preâmbulo: "Estamos identificando formas melhores de desenvolver software fazendo isso e ajudando os outros a fazê-lo." Outros de nós realizaram alterações e sugestões ínfimas, mas ficou claro que estava terminado. Pairava um sentimento de trabalho concluído na sala. Nada de desacordo. Sem discussão. Nem ao menos uma discussão real de alternativas. Eram somente estas quatro linhas:

- **Pessoas e interações**, em detrimento de processos e ferramentas.
- **Validação do software,** em vez de uma documentação exaustiva e longa.

18. Da esquerda para a direita, em um semicírculo em volta de Martin, essa foto mostra Dave Thomas, Andy Hunt (ou talvez Jon Kern), eu (você pode ver pelo jeans e pelo Leatherman no meu cinto), Jim Highsmith, alguém, Ron Jeffries e James Grenning. Existe alguém sentado atrás de Ron, e no chão, ao lado do sapato dele, talvez seja um dos cartões que usamos nos grupos de afinidade.

- **Colaboração com o cliente,** em detrimento da negociação de contrato.
- **Resposta à mudança,** em vez de seguir um plano cegamente.

Eu disse que terminamos? Parecia. Naturalmente, havia muitos detalhes para resolver. Por um lado, como chamaríamos aquilo que identificamos?

O nome "Ágil" não era um tiro certeiro. Tínhamos muitos candidatos diferentes. Acontece que eu gostava de "Peso leve", mas ninguém mais gostava. Eles achavam que sugeria "inconsequência". Outros gostaram da palavra "Adaptável". A palavra "Ágil" foi mencionada e uma pessoa comentou que, naquele momento, era uma palavra de ordem popular nas Forças Armadas. No final, embora ninguém realmente tenha amado de paixão a palavra "Ágil", ela era a melhor dentre as várias alternativas ruins.

No final do segundo dia, Ward se ofereceu para publicar o site agilemanifesto.org. Acredito que foi ideia dele que as pessoas o assinassem.

Pós-Snowbird

As duas semanas seguintes não foram tão românticas ou agitadas quanto aqueles dois dias em Snowbird. Elas foram predominantemente ocupadas pelo trabalho duro de elaborar os documentos de princípios que Ward por fim colocou no site.

A ideia de redigir esse documento foi algo que todos concordamos ser necessário com o intuito de explicar e direcionar os quatro valores. Afinal, os quatro valores são os tipos de declarações com as quais todos podem concordar sem ter que mudar nada sobre o modo como trabalham. Os princípios deixam claro que esses quatro valores têm consequências além da conotação de "serem coisas em que todos acreditam".

Não me lembro muito desse período, além do fato de termos enviado por e-mail o documento que relaciona os princípios entre si e que fazíamos muitos ajustes de texto. Trabalhamos arduamente, mas acho que todos sentimos que valeu a pena o esforço. Feito tudo isso, todos voltamos aos nossos trabalhos, atividades e vidas normais. Presumo que a maioria de nós pensou que a história terminaria por aí mesmo.

Nenhum de nós esperava o gigantesco movimento de apoio que se seguiu. Nenhum de nós previu a dimensão das consequências daqueles dois dias. Mas, para que eu não corra o risco de exagerar as coisas por ter participado de tudo, lembro-me sempre de que Alistair estava prestes a realizar uma reunião semelhante. E não deixo de pensar quantos outros estavam prestes a fazer a mesma coisa. Por isso, me contentei com a ideia de que o tempo era propício e que, se nós, dezessete, não tivéssemos nos encontrado naquela montanha em Utah, algum outro grupo teria se encontrado em outro lugar e chegado a uma conclusão parecida.

VISÃO GERAL DO ÁGIL

Como se gerencia um projeto de software? Existiram muitas abordagens ao longo dos anos — a maioria delas, péssimas. A esperança e a oração são populares entre os gerentes que acreditam que existem deuses que governam o destino dos projetos de software. Aqueles que não compartilham dessa crença geralmente recorrem a técnicas motivacionais, como cumprir o cronograma apelando para o chicote, para as correntes e para as fotos de pessoas escalando montanhas e gaivotas voando sobre o mar.

Essas abordagens, quase de forma universal, resultam em manifestações características da má administração de software: equipes de desenvolvimento que estão sempre atrasadas, apesar de trabalharem demais. Equipes que desenvolvem produtos claramente de baixa qualidade e que nem chegam perto de atender às necessidades do cliente.

RESTRIÇÃO TRIPLA

O motivo pelo qual essas técnicas simplesmente não funcionam é que os gerentes que as empregam não compreendem a lógica fundamental dos projetos de software. Essa lógica restringe todos os projetos a obedecer a uma regra irrefutável chamada de *Restrição Tripla* de gerenciamento de projetos. Bom, rápido, barato, concluído: escolha somente três. Você não pode ter os quatro. Pode ter um projeto que seja bom, rápido e barato, mas não será concluído. Pode ter um projeto concluído, barato e rápido, mas não será nem um pouco bom.

A realidade é que um bom gerente de projetos entende que esses quatro atributos têm coeficientes. Um gerente competente coordena um projeto para ser bom, rápido e barato o suficiente e concluído quando for necessário. Um bom gerente administra os coeficientes desses atributos, em vez de exigir que todos esses coeficientes sejam 100%. A metodologia ágil se esforça para atingir esse tipo de gerenciamento.

A partir desse momento, quero ter certeza de que você entende que o ágil é um framework que *ajuda* os desenvolvedores e os gerentes que desempenham esse tipo de gerenciamento pragmático de projetos. No entanto, esse gerenciamento não ocorre automaticamente, e não existe a garantia de que os gerentes tomem as decisões adequadas. Na verdade, é perfeitamente possível trabalhar com o framework ágil e ainda gerenciar mal o projeto, jogando tudo por água abaixo.

Gráficos na Parede

Então, como a agilidade pode contribuir com esse tipo de gerenciamento? *A agilidade fornece dados*. Uma equipe de desenvolvimento ágil produz somente os tipos de dados que os gerentes precisam para tomar boas decisões.

Veja a Figura 1.2. Imagine que ela está pregada na parede da sala do projeto. Isso não seria um espetáculo?

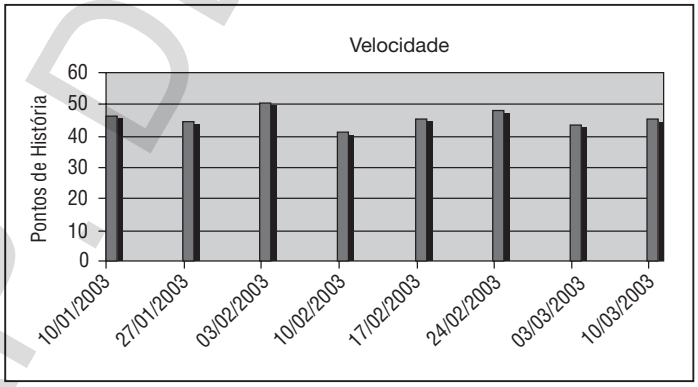

Figura 1.2 A velocidade da equipe

Esse gráfico mostra a quantidade de trabalhado que a equipe de desenvolvimento faz toda semana. A unidade de medida é "pontos de história". Mais adiante, falaremos sobre esses pontos. Por ora, basta analisar esse gráfico. Qualquer um pode bater o olho nele e ver com que rapidez a equipe está avançando. Demora menos de dez segundos para ver que a velocidade média é de aproximadamente 45 pontos por semana.

Qualquer um, até mesmo um gerente, pode prever que na próxima semana a equipe terá cerca de 45 pontos. Nas próximas dez semanas, eles devem apresentar 450 pontos. Isso é poder! É sobretudo poderoso se os gerentes e a equipe tiverem uma boa noção do número de pontos no projeto. De fato, equipes ágeis competentes têm noção dessas informações por meio de mais um gráfico na parede.

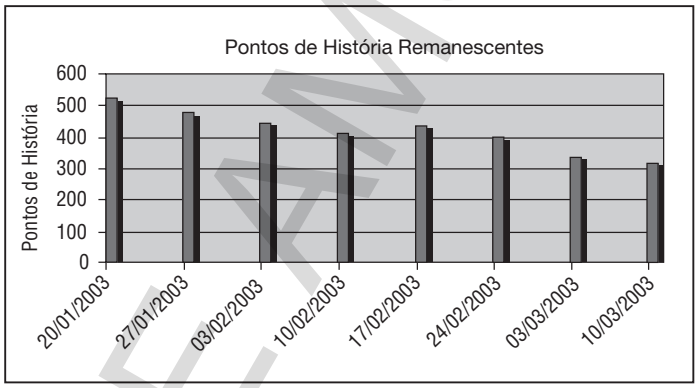

Figura 1.3 Gráfico de burn-down

A Figura 1.3 se chama *gráfico de burn-down.* Ele mostra quantos pontos restam até o próximo grande marco. Observe como diminui a cada semana. Repare que ele diminui menos que o número de pontos no gráfico de velocidade. Isso ocorre porque há um histórico constante de novos requisitos e problemas sendo descobertos durante o desenvolvimento.

Observe que o gráfico de burn-down tem uma curva que prevê quando o marco provavelmente será atingido. Praticamente qualquer pessoa pode entrar na sala, olhar para esses dois gráficos e chegar à conclusão de que o marco será alcançado em junho a uma velocidade de 45 pontos por semana.

Repare que existe uma anomalia no gráfico de burn-down. A semana de 17 de fevereiro, de alguma forma, perdeu terreno. Isso pode ter ocorrido devido ao acréscimo de uma funcionalidade nova ou a alguma outra mudança importante nos requisitos. Talvez seja o resultado de os desenvolvedores reestimarem o trabalho que falta. Em ambos os casos, queremos saber o impacto no cronograma a fim de que o projeto possa ser gerenciado de forma adequada.

É de suma importância para a metodologia ágil que esses dois gráficos estejam na parede. Uma das motivações principais para o desenvolvimento de software ágil é fornecer os dados que os gerentes precisam a fim de decidir como definir os coeficientes na Restrição Tripla e conduzir o projeto rumo ao melhor resultado possível.

Muitas pessoas não concordariam com esse último parágrafo. Afinal, os gráficos não são mencionados no Manifesto Ágil, e nem todas as equipes ágeis usam esses gráficos. E, francamente, não são os gráficos que importam. O importante são os dados.

Antes de mais nada, o desenvolvimento ágil é uma abordagem orientada a feedback. Cada semana, cada dia, cada hora e até cada minuto são orientados pela análise dos resultados da semana, dia, hora e minuto anteriores e, em seguida, pela realização dos ajustes apropriados. Isso vale para os próprios programadores e também para o gerenciamento de toda a equipe. Sem os dados, o projeto não pode ser gerenciado.[19]

Portanto, ainda que você não coloque esses dois gráficos na parede, faça questão de disponibilizar esses dados na cara dos gerentes. Verifique se eles sabem com que rapidez a equipe está avançando e quanto falta para a equipe concluir o trabalho. E apresente essas informações de maneira transparente, pública e clara — por exemplo, colocando os dois gráficos na parede.

19. Isso está estritamente relacionado ao ciclo de OODA de John Boyd, resumido aqui: https://en.wikipedia.org/wiki/OODA_loop. Boyd, J. R. 1987. *A Discourse on Winning and Losing.* Maxwell Air Force Base, AL: Air University Library, Document No. M-U 43947 [conteúdo em inglês].

Mas por que isso é tão importante? É possível gerenciar efetivamente um projeto sem esses dados? Durante trinta anos, até tentamos. E foi assim que aconteceu…

A Primeira Coisa que Você Sabe

Qual é a primeira coisa que você sabe a respeito de um projeto? Antes de saber o nome dele ou qualquer um dos requisitos, há um dado que precede todos os demais. *O Prazo,* é claro. E uma vez que *ele* é escolhido, é congelado. Nem adianta tentar negociar *O Prazo* porque ele foi escolhido por boas razões comerciais. Caso *ele* seja em setembro, é porque existe alguma feira ou reunião de acionistas em setembro ou simplesmente o financiamento acaba nesse mês. Seja lá qual for o motivo, é por razões *comerciais*, e isso não mudará apenas porque alguns desenvolvedores acham que talvez não consigam cumprir o prazo.

Ao mesmo tempo, os requisitos estão em constante movimento e nunca podem ser congelados. Isso ocorre porque os clientes não sabem realmente o que querem. Eles meio que sabem qual problema querem *resolver,* mas traduzir isso em forma de requisitos de um sistema nunca é uma coisa simples. Desse modo, os requisitos estão sendo constantemente reavaliados e repensados. Adiciona-se novas funcionalidades. Remove-se as funcionalidades antigas. A interface do usuário altera o formulário semanalmente, se não todos os dias.

Este é o mundo de uma equipe de desenvolvimento de software. É um mundo em que os prazos são congelados e os requisitos constantemente mudam. E, de alguma forma, nesse contexto, a equipe de desenvolvimento deve fazer com que o projeto tenha um bom resultado.

A Reunião

O Modelo Cascata prometia nos oferecer uma saída para lidar com esse problema. Com o intuito de compreender o quanto isso era tentador e ineficaz, mostrarei como era *A Reunião.*

É 1º de maio. O chefão chama todos nós para uma sala de reunião.

"Temos um projeto novo", diz o poderoso chefão. "Ele deve estar concluído no dia 1º de novembro. Ainda não temos requisitos, mas teremos nas próximas semanas."

"Agora, quanto tempo vocês levam para fazer a análise?"

Todos se olham de rabo de olho. Ninguém está disposto a falar. Como você responde a uma pergunta dessa? Um de nós murmura: "Mas ainda não temos requisitos."

"Finjam que têm os requisitos!", fala o chefão, aos gritos. "Vocês sabem como isso funciona. São todos profissionais. Eu não preciso de uma data exata. Só preciso de alguma coisa para colocar no cronograma. Lembrem-se de que, se demorar mais de dois meses, é melhor nem pegarmos esse projeto."

As palavras "dois meses?" escapa da boca de alguém, mas o poderoso chefão toma isso como uma afirmação. "Bom! — Foi o que pensei. Agora, quanto tempo vocês levarão para fazer o design?"

Mais uma vez, paira um silêncio mortal na sala. Você faz as contas. Primeiro, se dá conta de que faltam seis meses para novembro. A conclusão é óbvia. "Dois meses?", você pergunta.

"Exato!", sorri o poderoso chefão. "Exatamente o que pensei. E isso nos deixa dois meses para a implementação. Obrigado por virem à minha reunião."

Muitos leitores já estiveram em uma reunião assim. Considere-se sortudo se nunca esteve.

A Fase de Análise

Em seguida, todos nós deixamos a sala de reuniões e retornamos às nossas mesas. O que estamos fazendo? Este é o início da Fase de Análise, logo, deveríamos estar analisando. Mas, o que seria exatamente uma *análise*?

Se você ler livros sobre análise de software, descobrirá que existem tantas definições de análise quanto autores. Não existe um consenso real acerca

do que é análise. Ela pode ser efetuada a partir de uma estrutura analítica dos requisitos. Pode ser a identificação e elaboração dos requisitos. Talvez a criação de um modelo de dados subjacente, ou modelo de objeto, ou… Eis a melhor definição de análise: é o que os analistas fazem.

Naturalmente, algumas coisas são óbvias. Devemos fazer os cálculos do projeto e realizar as projeções básicas de viabilidade e recursos humanos. Devemos garantir que o cronograma seja cumprido. Isso é o mínimo que nossos chefes esperariam de nós. Seja lá o que for esse negócio de análise, é o que faremos nos próximos dois meses.

Esta é a fase de lua de mel do projeto. Todo mundo feliz e contente navegando na internet, negociando um pouco e se reunindo com clientes e usuários, fazendo rascunhos de diagramas legais e, em geral, se divertindo.

Então, em 1º de julho, um milagre acontece. Terminamos a análise.

Por que terminamos a análise? Porque é 1º de julho. O cronograma dizia que deveríamos concluí-la em 1º de julho, então concluímos em 1º de julho. Por que se atrasar?

Então, comemoramos com uma festinha, com balões e discursos, nossa passagem pelos portões das fases e nossa entrada na Fase de Design.

A Fase do Design

Mas, o que estamos fazendo agora? Estamos projetando, claro. Mas o que é *design*?

Sabemos um pouco mais a respeito da definição do design de software. Design de software é quando dividimos o projeto em módulos e projetamos as interfaces entre eles. É também quando determinamos de quantas equipes precisamos e quais devem ser as conexões entre essas equipes. Via de regra, espera-se que isso refine o cronograma para a elaboração de um planejamento de implementação que seja alcançável de forma realística.

Obviamente, as coisas mudam de forma inesperada durante essa fase. Acrescentam-se funcionalidades novas. Removem-se ou alteram-se as

funcionalidades antigas. E adoraríamos voltar e analisar novamente essas mudanças; mas o tempo é curto. Então, meio que jogamos essas mudanças no design.

E então, outro milagre acontece. É 1º de setembro e concluímos o design. Por que já terminamos? Porque é dia 1º de setembro. O cronograma diz que devemos terminar, então por que nos atrasar?

Logo, temos outra festa. Balões e discursos. E nós atravessamos o portão para a Fase de Implementação.

Se ao menos pudéssemos fazer isso mais uma vez. Se pudéssemos apenas *dizer* que terminamos a implementação. Mas não podemos, porque a questão da implementação é que, na verdade, ela precisa *estar* pronta. Análise e design não são *entregas binárias*. Essas fases não têm critérios de conclusão inequívocos. Não existe uma maneira real de saber que você as concluiu. Portanto, é melhor terminarmos as coisas no prazo.

A Fase de Implementação

Por outro lado, a implementação tem critérios de conclusão definidos. Não há como fingir que você implementou alguma coisa.[20]

É totalmente inequívoco o que estamos fazendo durante a Fase de Implementação. Estamos desenvolvendo e programando. E é melhor também programarmos como loucos, como se não houvesse amanhã, porque já estouramos quatro meses de prazo desse projeto.

Nesse ínterim, os requisitos ainda estão mudando. Acrescentam-se funcionalidades novas. Removem-se ou alteram-se as funcionalidades antigas Gostaríamos muito de dar um passo para trás, analisar e reprojetar essas mudanças, mas temos apenas semanas. E, assim, lidamos, gerenciamos e encaramos de frente essas mudanças no código.

Ao analisar o código e compará-lo com o design, percebemos que devíamos estar fumando alguma coisa bem forte quando criamos esse design, pois claramente o código não está nada parecido com aqueles

20. Embora os desenvolvedores do healthcare.gov tenham tentado.

diagramas belos e elegantes que desenhamos. No entanto, não temos tempo para nos preocupar com isso, porque estamos correndo contra o relógio e as horas extras só aumentam.

Assim, por volta do dia 15 de outubro, alguém diz: "Ei, qual é prazo? Quando ele termina?" Nesse momento, percebemos que restam somente duas semanas e nunca concluiremos o projeto até o dia 1º de novembro. Também é a primeira vez que as partes interessadas são informadas de que pode ocorrer um pequeno contratempo com o projeto.

Imagine a angústia das partes interessadas. "Vocês não poderiam ter nos informado isso na fase de análise? Não era quando deveriam dimensionar o projeto e comprovar a viabilidade do cronograma? Vocês não poderiam ter nos informado isso durante a Fase de Design? Não era quando deveriam dividir o design em módulos, atribuí-los a equipes e fazer as projeções dos recursos humanos? Por que estão nos informando isso apenas duas semanas antes do prazo?"

E eles têm razão, não têm?

Fase: Marchando para a Morte

Agora, entramos na Fase Marchando para a Morte do projeto. Os clientes estão furiosos. As partes interessadas estão possessas. A pressão aumenta. As horas extras disparam. As pessoas desistem. Só desgraça.

Em março, entregamos meio nas coxas alguma coisa que faz mais ou menos o que os clientes querem. Todo mundo está chateado e desmotivado. E prometemos a nós mesmos que *nunca* mais faremos outro projeto como este. Da próxima vez, vamos fazer a coisa certa! Da próxima vez faremos *mais* análise e *mais* design.

Chamo isso de *Inflação Desenfreada do Processo* (*Runaway Process Inflation*). Vamos fazer coisas que não funcionam, e faremos *muitas* delas.

Exagero?

Obviamente, essa é uma história exagerada. Engloba em um só lugar praticamente todas as coisas ruins que já aconteceram em um projeto de

software. Grande parte dos projetos em Cascata não deu tão errado assim. Na verdade, alguns, por pura sorte, conseguiram ser concluídos com um bocado de sucesso. Por outro lado, participei de reuniões desse tipo em mais de uma ocasião e já trabalhei em projetos semelhantes, e não sou o único. A história pode até ser exagerada, mas ainda é real.

Se você me perguntasse quantos projetos em Cascata foram um completo desastre como o projeto descrito anteriormente, eu diria que poucos — por outro lado, ainda assim deram errado e estão longe de ser muitos. Além do mais, a grande maioria sofreu problemas semelhantes em um grau menor (ou, às vezes, maior).

O Método Cascata não foi um desastre absoluto. Não reduziu a pó cada projeto. Mas foi e continua sendo uma forma desastrosa de gerenciar um projeto de software.

Uma Ideia Melhor

O problema da ideia do Método Cascata é que ela faz todo o sentido. Primeiro, analisamos o problema, então projetamos a solução e depois implementamos o design.

Simples. Direto. Óbvio. E errado.

A abordagem ágil de um projeto é totalmente diferente do que você acabou de ler, mas também faz todo o sentido. Na realidade, ao ler isso, acho que você verá que faz mais sentido do que as três fases do Método Cascata.

Um projeto ágil começa com a análise, no entanto, é uma análise que nunca termina. No diagrama da Figura 1.4, vemos o projeto inteiro. À direita está a data final, 1º de novembro. Lembre-se: a primeira coisa que você sabe é a data. Subdividimos esse tempo em incrementos regulares chamados *iterações* ou *sprints*.[21]

21. *Sprint* é um termo usado no Scrum. Não gosto dele porque implica correr o mais rápido possível. Um projeto de software é uma maratona, e acredite, você não quer disparar em uma maratona.

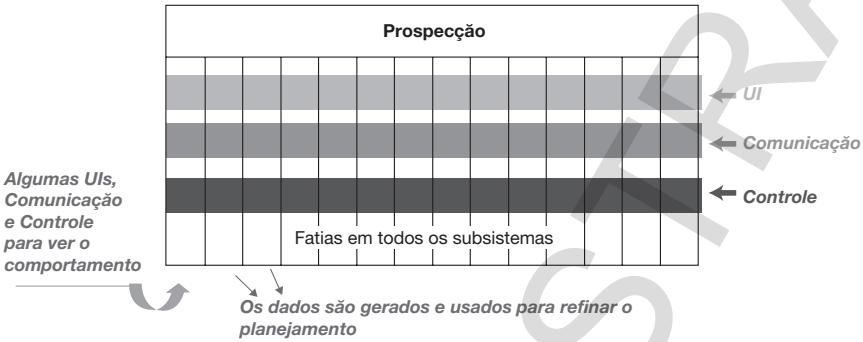

Figura 1.4 Visão macro do projeto

A extensão de uma iteração geralmente é de uma ou duas semanas. Prefiro uma semana porque muita coisa pode dar errado em duas semanas. Outras pessoas preferem duas semanas porque receiam não conseguir fazer o suficiente em uma semana.

ITERAÇÃO ZERO

A primeira iteração, às vezes conhecida como *Iteração Zero,* é usada para gerar uma pequena lista de funcionalidades, chamadas *histórias.* Falaremos mais a respeito nos próximos capítulos. Por ora, pense nelas como funcionalidades que precisam ser desenvolvidas. A Iteração Zero também é utilizada para definir o ambiente de desenvolvimento, estimar as histórias e traçar o planejamento inicial. Esse planejamento é simplesmente uma alocação provisória das histórias para as primeiras iterações. Por fim, a Iteração Zero é empregada pelos desenvolvedores e arquitetos com o intuito de traçar o panorama inicial do design para o sistema com base na lista provisória de histórias.

Esse processo de escrever histórias, estimá-las, planejá-las e projetá-las *nunca para.* Por isso, existe uma barra horizontal em todo o projeto chamada *Prospecção.* Toda iteração no projeto, do começo ao fim, terá um pouco de análise, design e implementação. Em um projeto ágil, estamos *sempre* analisando e projetando.

Algumas pessoas acham que a metodologia ágil é apenas uma série de mini-Métodos Cascatas. *Não* é o caso. As iterações não são subdivididas

em três seções. Não se realiza a análise somente no início da iteração, nem se realiza a implementação apenas no final da iteração. Ao contrário, as atividades de análise, arquitetura, design e implementação de requisitos são constantes durante toda a iteração.

Caso ache tudo muito confuso, não se preocupe. Vamos estudar os detalhes nos próximos capítulos. Lembre-se apenas de que as iterações não são o menor nível em um projeto ágil. Existem muitos outros níveis. E a análise, o design e a implementação ocorrem em cada um desses níveis. São dependências sem fim de um sistema.

A Agilidade Gera Dados

A iteração 1 começa com uma estimativa de quantas histórias serão concluídas. A equipe trabalha durante o período da iteração na elaboração dessas histórias. Mais adiante, falaremos sobre o que acontece dentro da iteração. Por enquanto, quais são as possibilidades de a equipe concluir todas as histórias que planejou terminar?

Praticamente zero, é claro. Isso ocorre porque o software não é um processo de estimativa confiável. Nós, programadores, simplesmente não sabemos quanto tempo as coisas levarão. Isso não significa que somos incompetentes ou preguiçosos; significa que não temos como saber exatamente o quanto uma tarefa será complicada até que ela esteja em andamento e seja concluída. Mas, conforme veremos, nem tudo está perdido.

No final da iteração, uma parte das histórias que planejamos terminar será concluída. Isso nos fornece nossa primeira avaliação de quanto pode ser concluído em uma iteração. Ou seja, são *dados reais*. Se assumirmos que toda iteração será semelhante, podemos utilizar esses dados para ajustar nosso planejamento original e calcular uma nova data final para o projeto (Figura 1.5).