Arquitetura Limpa

Arquitetura Limpa

O Guia do Artesão para Estrutura e Design de Software

Robert C. Martin

ALTA BOOKS
EDITORA
Rio de Janeiro, 2019

Este livro é dedicado à minha amada esposa, meus quatro filhos espetaculares e suas famílias, incluindo meu kit completo com cinco netos — a grande dádiva da minha vida.

Sumário

Prefácio

Do que estamos falando quando falamos de arquitetura?

Como qualquer metáfora, descrever software por meio das lentes da arquitetura pode esconder tanto quanto pode revelar. Pode prometer mais do que entregar e entregar mais que o prometido.

O apelo óbvio da arquitetura é a estrutura, que domina os paradigmas e discussões sobre o desenvolvimento de software — componentes, classes, funções, módulos, camadas e serviços, micro ou macro. No entanto, muitas vezes, é difícil confirmar ou compreender a estrutura bruta de vários sistemas de software — esquemas corporativos ao estilo soviético em vias de se tornarem legado, improváveis torres de equilíbrio se estendendo em direção à nuvem, camadas arqueológicas enterradas em um slide que parece uma imensa bola de lama. Pelo jeito, a estrutura de um software não parece tão intuitiva quanto a estrutura de um prédio.

Prédios têm uma estrutura física óbvia, em pedra ou concreto, com arcos altos ou largos, grande ou pequena, magnífica ou mundana. Essas estruturas têm pouca escolha além de respeitar os limites impostos pela gravidade e pelos seus materiais. Por outro lado — exceto no sentido de seriedade — o software tem pouco tempo para a gravidade. E do que o software é feito? Diferente dos prédios, que podem ser feitos de tijolos, concreto, madeira, aço e vidro, o software é feito de software.

Grandes construções de software são compostas de componentes de software menores, que, por sua vez, são formados por componentes ainda menores de software, e assim por diante. É um código dentro do outro, do início ao fim.

Na arquitetura de software, por natureza, o software é recursivo, fractal e esboçado e desenvolvido em código. Os detalhes são essenciais. Também ocorre o entrelaçamento de níveis de detalhes na arquitetura de prédios, mas não faz sentido falar de escala física em software. O software tem estrutura — muitas estruturas e muitos tipos delas — e essa variedade supera o conjunto de estruturas físicas encontradas nos prédios. Você pode até argumentar de forma muito convincente que, na arquitetura de software, há mais atividade e foco no design do que na construção civil — neste sentido, não é insensato considerar a arquitetura de software mais arquitetural do que a arquitetura de prédios!

Mas a escala física é algo que os humanos entendem e buscam no mundo. Embora atraentes e visualmente óbvias, caixas em um diagrama de PowerPoint não são arquitetura de sistemas de software. Sem dúvida, elas representam uma visão particular de uma arquitetura, mas confundir essas caixas com *a* imagem maior — com *a* arquitetura — é perder a imagem maior e a arquitetura: a arquitetura de software não se parece com nada. Uma visualização específica é uma escolha, não uma determinação. É uma escolha baseada em um conjunto maior de escolhas: o que incluir; o que excluir; o que enfatizar utilizando forma ou cor; o que retirar do destaque por meio da uniformização ou omissão. Não há nada natural ou intrínseco que diferencie uma visão da outra.

Talvez não seja muito útil falar sobre física e escala física em arquitetura de software, mas certas restrições físicas merecem a nossa consideração e atenção. A velocidade do processador e a largura de banda da rede podem fornecer um veredito duro sobre a performance de um sistema. A memória e o armazenamento podem limitar as ambições de qualquer base de código. O software pode ser feito do mesmo material que os sonhos, mas funciona no mundo físico.

> *Nisto é que consiste a monstruosidade no amor, senhora, em ser infinita a vontade e restrita a execução; em serem limitados os desejos e o ato escravo do limite.*
>
> *— William Shakespeare*

Nós e nossas empresas e economias vivemos no mundo físico. Isso nos dá uma outra perspectiva pela qual podemos entender a arquitetura do software e falar e raciocinar em termos de forças menos físicas e quantidades diferentes.

> *A arquitetura representa as decisões significativas de design que moldam um sistema, onde a significância é medida pelo custo da mudança.*
>
> *— Grady Booch*

Tempo, dinheiro e esforço nos dão um sentido de escala para classificar o grande e o pequeno e distinguir as coisas arquiteturais do resto. Essa medida também nos diz como podemos determinar se uma arquitetura é boa ou não: uma arquitetura não deve apenas atender às demandas dos usuários, desenvolvedores e proprietários em um determinado momento, mas também corresponder a essas expectativas ao longo do tempo.

> *Se você acha que uma arquitetura boa é cara, experimente uma arquitetura ruim.*
>
> *— Brian Foote e Joseph Yoder*

Os tipos de mudanças que ocorrem normalmente durante o desenvolvimento de um sistema não precisam ser caros, complexos ou abordados em projetos específicos gerenciados, fora do fluxo de trabalho diário e semanal.

Esse ponto está ligado a um problema de importância considerável para a física: a viagem no tempo. Como podemos saber as consequências dessas mudanças típicas para que possamos adaptar as principais decisões que tomamos a respeito delas? Como podemos reduzir o esforço e o custo de desenvolvimento no futuro sem recorrer a bolas de cristal ou máquinas do tempo?

> *A arquitetura é o conjunto de decisões que você queria ter tomado logo no início de um projeto, mas, como todo mundo, não teve a imaginação necessária.*
>
> *— Ralph Johnson*

Se entender o passado é muito difícil e nossa compreensão do presente, no máximo, incerta, prever o futuro não é nada trivial.

É aqui que a estrada se bifurca em vários caminhos.

Na estrada mais escura, surge a ideia de que uma arquitetura forte e estável se faz com autoridade e rigidez. Se for cara, a mudança é eliminada e suas causas são ignoradas ou atiradas em um fosso burocrático. A atuação do arquiteto é irrestrita e totalitária, e a arquitetura se transforma em uma distopia para os desenvolvedores e uma fonte de frustração constante para todos.

Um cheiro forte de generalidade especulativa vem de outro caminho. Essa é uma rota cheia de adivinhação codificada, incontáveis parâmetros, tumbas de código morto e mais complexidade acidental do que você poderia resolver com um orçamento de manutenção.

O caminho que nos interessa mais é o mais limpo. Nele, reconhecemos a suavidade do software e queremos preservá-lo como a principal propriedade do sistema. Admitimos que operamos com conhecimento incompleto, mas também sabemos que, como seres humanos, operar com conhecimento incompleto é o que fazemos de melhor. Nesse caminho, priorizamos os nossos pontos fortes em vez das deficiências. Criamos e descobrimos coisas. Fazemos perguntas e conduzimos experimentos. Uma arquitetura boa vem de compreendê-la mais como uma jornada do que como um destino, mais como um processo contínuo de investigação do que como um artefato congelado.

> *A arquitetura é uma hipótese que precisa ser comprovada por implementação e medição.*
>
> *— Tom Gilb*

Andar por este caminho requer cuidado e atenção, consideração e observação, prática e princípio. Em um primeiro momento, isso pode parecer lento, mas tudo depende da forma de andar.

> *A única maneira de ir rápido, é ir bem.*
>
> *— Robert C. Martin*

Aproveite a jornada.

— Kevlin Henney
Maio de 2017

Apresentação

O título deste livro é *Arquitetura Limpa*. Trata-se de um nome audacioso. Alguns até diriam arrogante. Então, por que decidi escrever este livro e escolher esse título?

Escrevi minha primeira linha de código em 1964, aos 12 anos. Há mais de meio século mexo com programação. Nesse meio-tempo, aprendi algumas coisas sobre como estruturar sistemas de software — informações que, na minha opinião, outras pessoas acharão valiosas.

Aprendi tudo isso construindo muitos sistemas, grandes e pequenos. Construí pequenos sistemas embarcados e grandes sistemas de processamento de lotes. Sistemas em tempo real e sistemas web. Aplicações de console, aplicações GUI, aplicações de controle de processo, jogos, sistemas de contabilidade, de telecomunicação, ferramentas de design, aplicações de desenho e muitos, muitos outros.

Desenvolvi aplicações single-threaded, aplicações multithreaded, aplicações com poucos processos pesados, aplicações com muitos processos leves, aplicações multiprocessadoras, aplicações de base de

dados, aplicações matemáticas, aplicações de geometria computacional e muitos, muitos outros.

Criei muitas aplicações. Construí muitos sistemas. E a dedicação com que trabalhei em todos esses projetos me ensinou algo surpreendente.

As regras da arquitetura são sempre as mesmas!

Isso é surpreendente porque os sistemas que criei eram radicalmente diferentes entre si. Então, por que sistemas tão diferentes compartilham regras similares de arquitetura? Cheguei à conclusão de que *as regras da arquitetura de software são independentes de todas as outras variáveis*.

Isso se torna ainda mais surpreendente quando você considera as mudanças ocorridas no hardware ao longo desse meio século. Comecei a programar em máquinas do tamanho de geladeiras que tinham tempos de ciclo de meio megahertz, 4K de memória central, 32K de memória de disco e uma interface de teletipo de 10 caracteres por segundo. Agora, estou fazendo uma excursão de ônibus pela África do Sul enquanto escrevo este prefácio, em um MacBook com um processador Intel i7 de quatro núcleos, rodando a 2.8 gigahertz cada. O MacBook tem 16 gigabytes de RAM, um terabyte de SSD e um display de retina, com resolução máxima de 2880X1800 pixels, que exibe vídeos em definição extremamente alta. A diferença em poder computacional é inacreditável. Qualquer análise razoável mostrará que este MacBook é pelo menos 10^{22} vezes mais potente do que aqueles computadores antigos em que comecei a programar, meio século atrás.

Vinte e duas ordens de magnitude é um número muito grande. É o número de angströms da Terra até Alpha Centauri. É o número de elétron das moedas que estão no seu bolso ou bolsa. Além disso tudo, esse número — *por alto* — expressa o aumento do poder computacional registrado ao longo da minha vida até agora.

E qual foi o efeito desse grande salto no poder computacional sobre os softwares que eu escrevo? Com certeza, eles ficaram maiores. Eu costumava pensar que tinha um programa grande quando escrevia 2.000 linhas. Afinal, era uma caixa cheia de cartões que pesava 4,5kg. Hoje em dia, um programa não é realmente grande a menos que ultrapasse 100.000 linhas.

O software também ficou muito mais eficiente. Podemos fazer coisas com que mal sonhávamos nos anos 1960. Os filmes *Colossus 1980*, *Revolta na Lua* e *2001: Uma Odisseia no Espaço* tentaram imaginar como seria o futuro, mas passaram bem longe ao preverem máquinas enormes que adquiriam senciência. Em vez disso, o que temos hoje são máquinas infinitamente pequenas, mas que ainda são… só máquinas.

Outra característica do software atual em comparação com o anterior: *ele continua sendo feito com os mesmos elementos*. É formado por declarações `if`, declarações de atribuição e laços `while`.

Ah, você pode discordar e dizer que agora temos linguagens muito melhores e paradigmas superiores. Afinal de contas, programamos em Java, C# e Ruby e usamos o design orientado a objetos. Isso é verdade, mas, ainda assim, o código continua a ser apenas uma reunião de sequências, seleções e iterações, como nos anos 1950 e 1960.

Quando observa bem de perto a prática de programar computadores, você percebe que muito pouco mudou em 50 anos. As linguagens ficaram um pouco melhores e as ferramentas, fantasticamente melhores. Mas os blocos de construção básicos de um programa de computador não mudaram.

Se eu levasse uma programadora[1] de 1966 para 2016, colocasse ela em frente ao meu MacBook executando o IntelliJ e a apresentasse ao Java, talvez ela precisasse de 24 horas para se recuperar do choque. Mas, logo em seguida, ela poderia escrever o código. Java não é tão diferente de C ou mesmo de Fortran.

Por outro lado, se eu levasse você de volta para 1966 e ensinasse a escrever e editar código PDP-8 perfurando uma fita de papel em um teletipo de 10 caracteres por segundo, talvez você precisasse de 24 horas para se recuperar da decepção. Mas, logo em seguida, poderia escrever o código, que não mudou tanto assim.

1. *E ela provavelmente seria uma mulher já que, naquela época, as mulheres eram uma grande fração dos programadores.*

Eis o segredo: essa imutabilidade do código é a razão pela qual as regras da arquitetura de software são tão consistentes entre os diversos tipos de sistemas. As regras da arquitetura de software são princípios para ordenar e montar os blocos de construção de programas. E já que esses blocos de construção são universais e não mudaram, as regras para ordená-los são, também, universais e imutáveis.

Os programadores mais novos podem pensar que isso é loucura. Podem insistir que, atualmente, tudo é novo e diferente e que as regras do passado estão no passado. Se pensam assim, estão redondamente enganados. As regras não mudaram. Apesar das novas linguagens, frameworks e paradigmas, as regras continuam as mesmas de quando Alan Turing escreveu seu primeiro código de máquina em 1946.

Mas uma coisa mudou: como não conhecíamos as regras naquela época, acabávamos infringindo essas normas várias vezes. Agora, com meio século de experiência nas costas, compreendemos essas regras.

E são dessas regras — atemporais e imutáveis — que este livro trata.

Agradecimentos

Gostaria de agradecer as pessoas abaixo pelo papel que tiveram na criação deste livro — sem nenhuma ordem específica:

Chris Guzikowski
Chris Zahn
Matt Heuser
Jeff Overbey
Micah Martin
Justin Martin
Carl Hickman
James Grenning
Simon Brown
Kevlin Henney
Jason Gorman
Doug Bradbury
Colin Jones
Grady Booch
Kent Beck

Martin Fowler
Alistair Cockburn
James O. Coplien
Tim Conrad
Richard Lloyd
Ken Finder
Kris Iyer (CK)
Mike Carew
Jerry Fitzpatrick
Jim Newkirk
Ed Thelen
Joe Mabel
Bill Degnan

E muitos outros, numerosos demais para citar.

Na revisão final deste livro, eu lia o capítulo Arquitetura Gritante quando o sorriso de olhos brilhantes e a risada melódica de Jim Weirich ecoaram em minha mente. Boa sorte, Jim!

Sobre o Autor

Robert C. Martin (Uncle Bob) atua como programador desde 1970. Cofundador da cleancoders.com, que oferece treinamento online em vídeo para desenvolvedores de software, também fundou a Uncle Bob Consulting LLC, que presta serviços de consultoria de software, treinamento e desenvolvimento de capacidades para as principais corporações do mundo. Ocupou o cargo de artesão-mestre na 8th Light, Inc., uma firma de consultoria de software sediada em Chicago. Já publicou dúzias de artigos em vários periódicos do setor e faz palestras regularmente em conferências internacionais e feiras. Também atuou três anos como editor-chefe da *C++ Report* e foi o primeiro presidente da Agile Alliance.

Martin escreveu e editou muitos livros, como *O Codificador Limpo*, *Código Limpo*, *UML for Java Programmers*, *Agile Software Development*, *Extreme Programming in Practice*, *More C++ Gems*, *Pattern Languages of Program Design 3* e *Designing Object Oriented C++ Applications Using the Booch Method* (os dois primeiros traduzidos e publicados pela Alta Books).

1 Introdução

Ninguém precisa de muitas habilidades e conhecimentos para fazer um programa funcionar. Alunos do ensino médio fazem isso o tempo todo. Jovens universitários iniciam negócios bilionários com esboços de algumas linhas em PHP ou Ruby. Entrincheirados em cubículos no mundo inteiro, hordas de programadores juniores enfrentam uma torrente de documentos de requerimentos, armazenados em imensos sistemas de acompanhamento de defeitos, para fazer os seus sistemas "funcionarem" utilizando somente a pura e bruta força de *vontade*. O código que eles produzem pode não ser bonito, mas funciona. Funciona porque fazer algo funcionar — uma vez — não é tão difícil.

Fazer direito é outra questão. Criar software de maneira correta é *difícil*. Requer conhecimentos e habilidades que a maioria dos jovens programadores ainda não adquiriu. Requer um grau de raciocínio e insight que a maioria dos programadores não se empenha em desenvolver. Requer um nível de disciplina e dedicação que a maioria dos programadores nunca sonhou que precisaria. Principalmente, requer paixão pela programação e o desejo de se tornar um profissional.

Quando você acerta o software, algo mágico acontece: você não precisa de hordas de programadores para mantê-lo funcionando. Você não precisa de uma torrente de documentos de requerimentos e imensos sistemas de

acompanhamento de defeitos. Você não precisa de fazendas globais de cubículos nem de programação 24 horas por dia, sete dias por semana.

Quando o software é feito da maneira certa, ele exige só uma fração dos recursos humanos para ser criado e mantido. As mudanças são simples e rápidas. Os poucos defeitos surgem distantes uns dos outros. O esforço é minimizado enquanto a funcionalidade e a flexibilidade são maximizadas.

Sim, essa visão parece um pouco utópica. Mas eu estive lá e vi acontecer. Já trabalhei em projetos onde o design e a arquitetura do sistema facilitavam a sua escrita e manutenção. Já participei de projetos que exigiram uma fração dos recursos humanos previstos. Já atuei em sistemas que apresentaram taxas de defeitos extremamente baixas. Já observei o efeito extraordinário que uma boa arquitetura de software pode ter sobre um sistema, um projeto e uma equipe. Já estive na terra prometida.

Mas não acredite em mim. Examine a sua própria experiência. Você já vivenciou o oposto disso tudo? Já trabalhou em sistemas tão interconectados e intrincadamente acoplados que qualquer mudança, por mais trivial que seja, levava semanas e envolvia grandes riscos? Já experimentou a impedância de um código ruim e um péssimo design? O design de algum dos sistemas em que você trabalhou já provocou um efeito negativo arrasador sobre a moral da equipe, a confiança dos clientes e a paciência dos gerentes? Você já viu equipes, departamentos e até empresas serem prejudicados pela péssima estrutura de um software? Já esteve no inferno da programação?

Eu já estive — e até certo ponto, a maioria de nós também. É muito mais comum enfrentar incríveis dificuldades com o design dos softwares do que curtir a oportunidade de trabalhar com um bom design.

1 O que são Design e Arquitetura?

Tem havido muita confusão entre os termos design e arquitetura ao longo dos anos. O que é design? O que é arquitetura? Quais são as diferenças entre os dois?

Um dos objetivos deste livro é acabar com essa confusão e definir, de uma vez por todas, o que são design e arquitetura. Para começar, afirmo que não há diferença entre os dois termos. *Nenhuma diferença*.

Em geral, a palavra "arquitetura" é usada no contexto de algo em um nível mais alto e que independe dos detalhes dos níveis mais baixos, enquanto "design" parece muitas vezes sugerir as estruturas e decisões de nível mais baixo. Porém, esses usos perdem o sentido quando observamos arquitetos de verdade em ação.

Considere o arquiteto que projetou minha casa nova. A casa tem uma arquitetura? É claro que tem. E o que é essa arquitetura? Bem, é a forma da casa, a aparência externa, as elevações e o layout dos espaços e salas. Mas, quando analiso as plantas produzidas pelo arquiteto, noto um número imenso de detalhes de baixo nível. Vejo onde cada tomada, interruptor de luz e lâmpada será colocado. Vejo quais interruptores controlam quais lâmpadas. Vejo onde o forno será instalado e o tamanho e o local do aquecedor de água e da bomba. Vejo descrições detalhadas de como as paredes, tetos e fundações serão construídas.

Resumindo, vejo os pequenos detalhes que servem de base para as decisões de alto nível. Também vejo como esses detalhes de baixo nível e as decisões de alto nível fazem parte do design da casa como um todo.

Ocorre o mesmo com o design de software. Os detalhes de baixo nível e a estrutura de alto nível são partes do mesmo todo. Juntos, formam um tecido contínuo que define e molda o sistema. Você não pode ter um sem o outro e, de fato, nenhuma linha os separa claramente. Há simplesmente uma linha constante de decisões que se estende dos níveis mais altos para os mais baixos.

O Objetivo?

Qual é o objetivo dessas decisões? O objetivo do bom design de software? Esse objetivo não é nada menos do que a minha descrição utópica:

> *O objetivo da arquitetura de software é minimizar os recursos humanos necessários para construir e manter um determinado sistema.*

A medida da qualidade do design corresponde à medida do esforço necessário para satisfazer as demandas do cliente. Se o esforço for baixo e se mantiver assim ao longo da vida do sistema, o design é bom. Se o esforço aumentar a cada novo release ou nova versão, o design é ruim. Simples assim.

Estudo de Caso

Como exemplo, considere o seguinte estudo de caso. Ele inclui dados reais de uma empresa real que deseja se manter anônima.

Primeiro, vamos observar o crescimento da equipe de engenharia. Estou certo que você concordará que essa tendência é muito encorajadora. O crescimento, como o apresentado na Figura 1.1, deve ser um indicativo de sucesso expressivo!

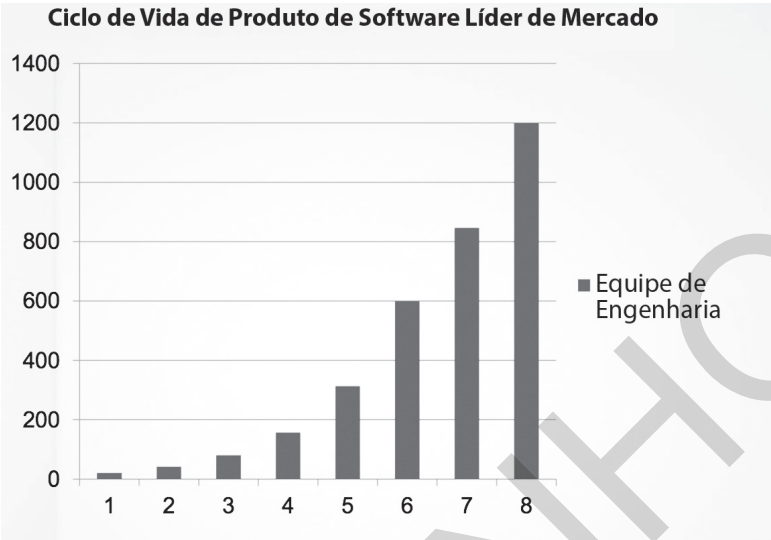

Figura 1.1 Crescimento da equipe de engenharia

Reprodução autorizada de uma apresentação de slides de Jason Gorman

Agora, vamos analisar a produtividade da empresa ao longo do mesmo período, medida simplesmente por linhas de código (Figura 1.2).

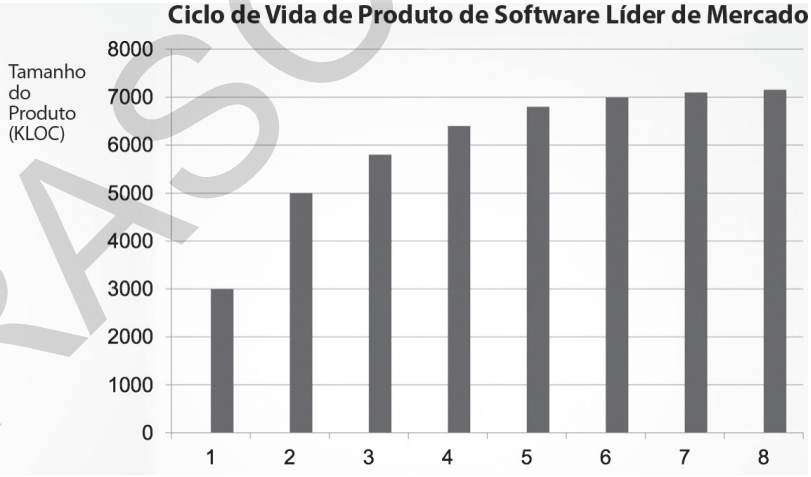

Figura 1.2 Produtividade ao longo do mesmo período

Claramente há algo errado aqui. Embora cada release seja operado por um número cada vez maior de desenvolvedores, o crescimento do código parece estar se aproximando de uma assíntota.

Mas aqui está o gráfico realmente assustador: a Figura 1.3 mostra como o custo por linha de código mudou com o tempo.

Essas tendências não são sustentáveis. Seja qual for a lucratividade registrada pela empresa no momento, essas curvas drenarão catastroficamente o lucro do modelo de negócios e causarão a estagnação da empresa, que poderá até sofrer um colapso total.

O que causou essa mudança notável na produtividade? Por que o custo de produção do código aumentou 40 vezes entre o release 1 e o release 8?

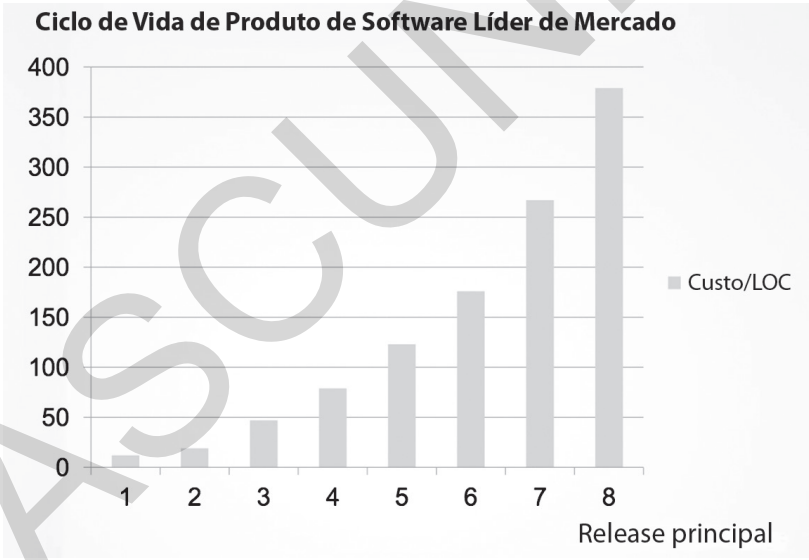

Figura 1.3 Custo por linha de código em função do tempo

A Marca Registrada de uma Bagunça

Você está olhando para a marca registrada de uma bagunça. Quando sistemas são desenvolvidos às pressas, quando o número total de programadores se torna o único gerador de resultados, e quando houver pouca ou nenhuma preocupação com a limpeza do código e a estrutura

do design, você pode ter certeza que irá seguir esta curva até o seu terrível final.

A Figura 1.4 mostra como esta curva é vista pelos desenvolvedores. Eles começaram com quase 100% de produtividade, mas, a cada release, sua produtividade caiu. No quarto release, ficou claro que a produtividade estava em queda livre, tendendo a se estabilizar em uma assíntota próxima a zero.

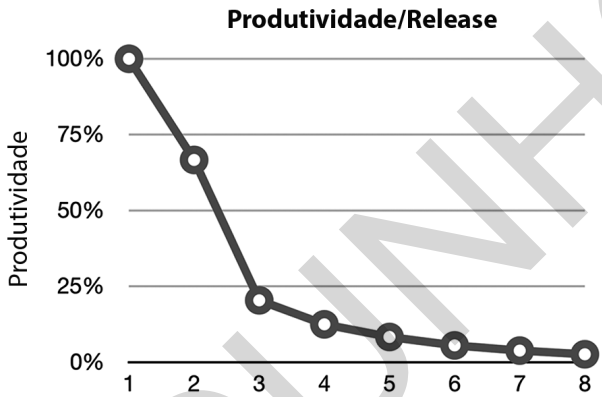

Figura 1.4 Produtividade por release

Isso é tremendamente frustrante para os desenvolvedores, porque todos estão trabalhando *duro*. Ninguém diminuiu o ritmo.

E ainda assim, apesar de todos os atos de heroísmo, horas extras e dedicação, eles simplesmente não conseguem fazer quase nada. Todos os esforços foram desviados da criação de recursos e agora são consumidos pela gestão da bagunça. Nesse ponto, o trabalho mudou para transferir a bagunça de um lugar para outro, para o próximo e para o lugar seguinte até que possam adicionar mais um recursozinho insignificante.

A Visão Executiva

Mas, se você acha que *isso* é ruim, imagine como essa imagem chega aos executivos! Considere a Figura 1.5, que descreve a folha de pagamentos mensal do desenvolvimento para o mesmo período.

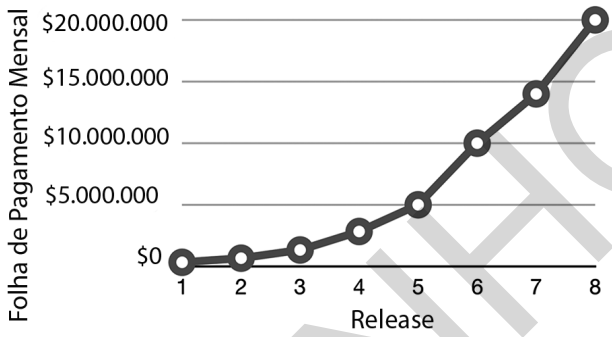

Figura 1.5 Folha de pagamento mensal do desenvolvimento por release

Quando o release 1 foi entregue, a folha de pagamento mensal estava orçada em algumas centenas de milhares de dólares. O segundo release custou algumas centenas de milhares de dólares a mais. No oitavo release, a folha de pagamento mensal era de US$20 milhões e tendia a aumentar.

Esse gráfico por si só já é assustador. Claramente algo alarmante está acontecendo. Espera-se que a receita esteja excedendo os custos e, portanto, justificando a despesa. Mas não importa como você olhe para esta curva, ela é motivo para preocupações.

Agora, compare a curva da Figura 1.5 com as linhas de código escritas por release da Figura 1.2. Aquelas centenas de milhares de dólares mensais no início trouxeram muita funcionalidade — mas os US$20 milhões finais não geraram quase nada! Qualquer CFO olharia para esses dois gráficos e saberia que é necessário tomar uma ação imediata para evitar um desastre.

Mas que ação pode ser tomada? O que deu errado? O que causou esse declínio incrível na produtividade? O que os executivos podem fazer além de bater os pés e gritar com os desenvolvedores?

O que Deu Errado?

Há quase 2600 anos, Esopo contou a história da Tartaruga e da Lebre. A moral dessa história foi explicada muitas vezes e de diversas formas diferentes:

- "Devagar e sempre, e você vence a corrida."
- "A corrida não é para os velozes, nem a batalha para os fortes."
- "Quanto mais pressa, menor a velocidade."

A própria história ilustra a tolice de ter confiança demais. A Lebre, tão confiante em sua velocidade intrínseca, não leva a corrida a sério e tira um cochilo, enquanto a Tartaruga cruza a linha de chegada.

Os desenvolvedores modernos estão em uma corrida similar e demonstram um excesso de confiança parecido. Ah, eles não dormem — longe disso. A maioria dos desenvolvedores modernos trabalha demais. Mas uma parte de seu cérebro dorme *sim* — a parte que sabe que um código bom, limpo e bem projetado *é importante*.

Esses desenvolvedores acreditam em uma mentira bem conhecida: "Podemos limpar tudo depois, mas, primeiro, temos que colocá-lo no mercado!" Evidentemente, as coisas nunca são limpas mais tarde porque a pressão do mercado nunca diminui. A preocupação de lançar o produto no mercado o quanto antes significa que você agora tem uma horda de concorrentes nos seus calcanhares e precisa ficar à frente deles, correndo o mais rápido que puder.

E assim os desenvolvedores nunca alternam entre modos. Não podem voltar e limpar tudo porque precisam terminar o próximo recurso, e o próximo, e o próximo, e o próximo. Nesse ritmo, a bagunça vai se acumulando e a produtividade continua a se aproximar assintoticamente do zero.

Como a Lebre que depositava uma confiança excessiva na própria velocidade, os desenvolvedores confiam excessivamente na sua habilidade de permanecerem produtivos. Mas a crescente bagunça no código, que mina a sua produtividade, nunca dorme e nunca cede. Se tiver espaço, ela reduzirá a produtividade a zero em uma questão de meses.

A maior mentira em que os desenvolvedores acreditam é a noção de que escrever um código bagunçado permite fazer tudo mais rápido a curto prazo e só diminui o ritmo a longo prazo. Os desenvolvedores que caem nessa cilada demonstram o mesmo excesso de confiança da lebre ao acreditarem na sua habilidade de alternar entre os modos de fazer bagunça e o de limpar bagunça em algum ponto no futuro, mas acabam cometendo um simples erro de fato. Esquecem que *fazer bagunça é sempre mais lento do que manter tudo limpo*, seja qual for a escala de tempo utilizada.

Observe na Figura 1.6 os resultados de um experimento notável, conduzido por Jason Gorman ao longo de seis dias. A cada dia ele concluía um programa simples que convertia números inteiros em numerais romanos. Ele sabia que o trabalho estava completo quando o conjunto predefinido de testes de aceitação era aprovado. A tarefa demorava um pouco menos de 30 minutos por dia. Jason usou uma disciplina bem conhecida de limpeza chamada desenvolvimento guiado por testes (TDD — do inglês test-driven development) no primeiro, terceiro e quinto dias. Nos outros três dias, ele escreveu o código sem essa disciplina.

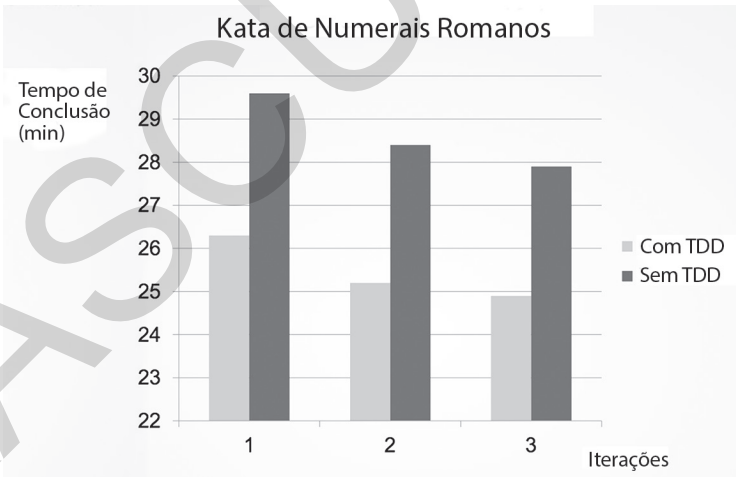

Figura 1.6 Tempo de conclusão por iterações e uso/não uso de TDD

Primeiro, observe a curva de aprendizagem aparente na Figura 1.6. O trabalho nos últimos dias foi concluído mais rapidamente do que nos primeiros. Note também que o trabalho nos dias de TDD evoluíram a uma velocidade aproximadamente 10% maior do que nos dias sem TDD. De fato, mesmo o dia mais lento com TDD foi mais rápido do que o dia mais rápido sem TDD.

Algumas pessoas podem achar esse resultado extraordinário. Mas para aqueles que não foram iludidos pelo excesso de confiança da Lebre, o resultado já era esperado. Isso porque eles conhecem uma verdade simples do desenvolvimento de software:

A única maneira de seguir rápido é seguir bem.

Essa é a resposta para o dilema dos executivos. A única maneira de reverter o declínio na produtividade e o aumento no custo é fazer com que os desenvolvedores parem de pensar como uma Lebre superconfiante e comecem a assumir a responsabilidade pela bagunça que fizeram.

Os desenvolvedores podem achar que a resposta consiste em começar de novo do zero e reprojetar o sistema inteiro — mas essa é só a Lebre falando outra vez. O mesmo excesso de confiança que causou a bagunça está dizendo que eles conseguirão melhorar tudo se recomeçarem a corrida. Mas a realidade não é tão bonita assim:

Devido ao seu excesso de confiança, eles reformularão o projeto com a mesma bagunça do original.

Conclusão

Em qualquer caso, a melhor opção para a organização de desenvolvimento é reconhecer e evitar o seu próprio excesso de confiança e começar a levar a sério a qualidade da arquitetura do software.

Para levar a sério a arquitetura do software, você precisa saber como é uma boa arquitetura de software. Para construir um sistema com um design e uma arquitetura que minimizem o esforço e maximizem a produtividade, você precisa saber quais atributos da arquitetura do sistema podem concretizar esses objetivos.

É isso que abordo neste livro. Irei descrever arquiteturas e designs limpos e eficientes para que os desenvolvedores de software possam criar sistemas que tenham vidas longas e lucrativas.