Domain-Driven Design

ATACANDO AS COMPLEXIDADES NO CORAÇÃO DO SOFTWARE

Eric Evans

3ª Edição Revisada

ALTA BOOKS
EDITORA
Rio de Janeiro, 2010

SUMÁRIO

INTRODUÇÃO

São várias as coisas que tornaram complexo o desenvolvimento de softwares. Mas o coração dessa complexidade está na dificuldade essencial do próprio domínio-problema. Se você estiver tentando adicionar automação a uma empresa humana complicada, seu software não tem como fugir dessa complexidade – tudo que ele pode fazer é controlá-la.

A chave para controlar complexidades é um bom modelo de domínio, um modelo que vá além da visão superficial de um domínio introduzindo uma estrutura subjacente, que ofereça aos desenvolvedores de software o máximo de aproveitamento que eles procuram. Um bom modelo de domínio pode ser extremamente valioso, mas não é algo fácil de se fazer. Poucas pessoas o podem fazer bem, e é difícil ensinar.

Eric Evans é uma dessas poucas pessoas que conseguem criar modelos de domínio bem. Descobri isso trabalhando com ele – um daqueles maravilhosos momentos em que você encontra um cliente mais capacitado que você. Nossa colaboração foi curta, mas um grande prazer. Desde então, temos mantido contato, e pude acompanhar lentamente a gestação deste livro.

Valeu a pena esperar.

Este livro evoluiu e se transformou em uma obra que satisfaz uma grande ambição: descrever e construir um vocabulário sobre a arte da modelagem de domínios. Oferecer uma estrutura de referência através da qual possamos explicar essa atividade bem como ensinar essa capacidade de difícil aprendizado. Este é um livro que me proporcionou muitas novas ideias à medida que foi tomando forma, e ficaria bastante surpreso se mesmo pessoas já altamente experientes em modelagem conceitual não conseguirem tirar várias novas ideias de sua leitura.

Eric também solidifica muitas das coisas que aprendemos ao longo dos anos. Primeiro, na modelagem de domínios, não se devem separar conceitos de implementação. Um modelador de domínios eficaz tem a capacidade não só de usar uma lousa branca com um profissional da área contábil, mas também escrever Java com um programador. Em parte, isto é verdade porque você não pode construir um modelo conceitual *útil* sem considerar as questões de implementação. Mas o principal motivo pelo qual conceitos e implementação pertencem um ao outro é que o maior valor de um modelo de domínio está no fato de ele proporcionar uma *linguagem onipresente* (ou, em inglês, *ubiquitous language*, linguagem comum, ubíqua, que está em toda parte, daí onipresente) que serve de união para especialistas e tecnólogos em domínios.

Outra lição que você vai aprender neste livro é que os modelos de domínio não são modelados primeiro e, depois, implementados. Como muitas pessoas, acabei rejeitando o pensamento antiquado de "projetar e depois construir". Mas a lição aprendida com a experiência de Eric é que os modelos de domínio verdadeiramente bons evoluem com o tempo, e até mesmo os modeladores mais experientes acreditam que adquirem suas melhores ideias após o lançamento inicial de um sistema.

Acredito, e espero, que este seja um livro de grande influência. Um livro que acrescente estrutura e coesão a um campo bastante incerto ensinando ao mesmo tempo muitas pessoas a usarem uma valiosa ferramenta. Os modelos de domínios podem trazer grandes consequências para o controle do desenvolvimento de software – sejam quais forem a linguagem e o ambiente em que eles sejam implementados.

Um último pensamento, de extrema importância. Uma das coisas que mais respeito neste livro é que Eric não tem medo de falar sobre os momentos em que *não* obteve sucesso. A maioria dos autores prefere manter um ar de onipotência desinteressada. Eric deixa claro que, como a grande maioria, ele já teve sucessos e decepções. O importante é que ele pode aprender com ambos – e, para nós, o mais importante é que ele pode transmitir suas lições.

Martin Fowler

PREFÁCIO

Há pelo menos 20 anos, os grandes designers de software reconhecem que a modelagem e o design de domínios são tópicos fundamentais; ainda assim, muito pouco tem sido escrito sobre o que precisa ser feito ou como fazê-lo. Embora nunca tenha sido claramente formulada, uma filosofia surgiu como uma subcorrente na comunidade de objetos, uma filosofia que chamo de *domain-driven design* ou design dirigido (conduzido) por domínios ou simplesmente DDD.

Passei a última década desenvolvendo sistemas complexos em vários domínios de negócios e técnicos. Em meu trabalho, tentei usar as melhores práticas existentes nos processos de design e desenvolvimento, já que estas surgiram com os líderes em desenvolvimento orientado a objetos. Alguns projetos meus tiveram bastante sucesso; alguns fracassaram. Uma característica comum aos sucessos obtidos foi um modelo de domínio rico que evoluiu através de iterações de design e passaram a fazer parte do tecido que compunha o projeto.

Este livro oferece um framework para que se possam tomar decisões de design e um vocabulário técnico para discutir design de domínios. Ele é uma síntese das melhores práticas amplamente aceitas e minhas próprias opiniões e experiências. Equipes de desenvolvimento de software que se deparam com domínios complexos podem se valer desse framework para abordar o DDD de forma sistemática.

Contrastando três projetos

Três projetos se destacam em minha memória como exemplos vivos de como a prática do design de domínios pode afetar drasticamente os resultados de um desenvolvimento. Embora todos os três projetos tenham gerado softwares de grande utilidade, somente um atingiu seus objetivos ambiciosos e produziu um software complexo que continuou a evoluir até satisfazer as contínuas necessidades da organização.

Pude presenciar um projeto decolando rapidamente, oferecendo um sistema comercial simples e de grande utilidade baseado na Web. Os desenvolvedores guiavam-se com base em suas próprias experiências, mas isso nunca foi um obstáculo para eles, pois softwares simples podem ser escritos com pouquíssima atenção dada ao design. Em consequência deste sucesso inicial, as expectativas foram altas com relação a um futuro desenvolvimento. É aí que me foi solicitado trabalhar na segunda versão. Quando examinei de perto, vi que faltava um

modelo de domínio, ou até mesmo uma linguagem comum no projeto, e restava apenas um design desestruturado. Os líderes do projeto não concordaram com minha avaliação, e rejeitei o serviço. Um ano depois, a equipe se viu em apuros e impossibilitada de lançar uma segunda versão. Embora o uso que fizessem da tecnologia não fosse exemplar, foi a lógica de negócio que os arrebatou. O seu primeiro lançamento havia se solidificado prematuramente transformando-se em um legado que exigia alta manutenção.

Para enfrentar essa complexidade é necessária uma abordagem mais séria com relação ao design da lógica de domínios. No início de minha carreira, tive a sorte de acabar trabalhando em um projeto que enfatizava o design de domínios. Esse projeto, em um domínio pelo menos tão complexo quanto o primeiro, também começou com um sucesso inicial modesto, lançando um aplicativo simples para comerciantes institucionais. Mas, neste caso, o lançamento inicial foi seguido de sucessivas acelerações de desenvolvimento. Cada iteração abria novas e excelentes opções para integração e elaboração de funcionalidades do lançamento anterior. A equipe pôde responder às necessidades dos clientes com flexibilidade e uma capacidade cada vez maior. Essa trajetória ascendente estava diretamente ligada a um modelo de domínios incisivo, repetidamente refinado e expresso em códigos. À medida que a equipe aprendia com o domínio, o modelo se aprofundava. A qualidade de comunicação melhorou não só entre os desenvolvedores, mas também entre desenvolvedores e especialistas do domínio, e o design – longe de impor um fardo de manutenção ainda mais pesado – tornou-se mais fácil de modificar e expandir.

Infelizmente, os projetos não atingem um ciclo tão virtuoso assim simplesmente enfrentando-se os modelos com seriedade. Um de meus antigos projetos começou com ambiciosas aspirações para construir um sistema corporativo global baseado em um modelo de domínios, mas, após anos de decepção, o projeto baixou suas expectativas e se acomodou no convencional. A equipe dispunha de boas ferramentas e um bom conhecimento do negócio, e deu especial atenção à modelagem. Mas uma separação mal elaborada entre os papéis desempenhados pelos desenvolvedores desassociou a modelagem da implementação de forma que o design acabou não refletindo a profunda análise que estava em andamento. De qualquer forma, o projeto de objetos de negócios detalhados não foi vigoroso o suficiente para sustentar sua combinação em aplicativos mais elaborados. A realização de várias iterações não gerou nenhum avanço nos códigos, devido a um nível de conhecimento desigual entre os desenvolvedores, que não tinham nenhuma consciência do corpo informal de estilo e técnica para se criarem objetos baseados em modelos que também funcionam como um software prático e operacional. Com o passar dos meses, o trabalho de desenvolvimento se atolou na complexidade e a equipe perdeu sua visão coesa do sistema. Após anos de esforço, o projeto acabou produzindo um software modesto e útil, mas a equipe abandonara suas ambições iniciais e o foco do modelo.

O desafio da complexidade

Muitas coisas podem fazer um projeto perder seu rumo: a burocracia, objetivos pouco claros e a falta de recursos, para citar apenas alguns. Mas é a abordagem dada ao design que, em grande parte, determina até que ponto os softwares podem se tornar complexos. Quando a complexidade foge ao controle, os desenvolvedores já não podem entender o software bem o suficiente para alterá-lo ou expandi-lo com facilidade e segurança. Por outro lado, um bom design pode criar oportunidades para explorar essas características complexas.

Alguns fatores relativos ao design são tecnológicos. Boa parte dos esforços tem sido direcionada ao design de redes, bancos de dados e outras dimensões técnicas de softwares. Muitos livros têm sido escritos sobre como resolver esses problemas. Legiões de desenvolvedores têm cultivado suas habilidades e acompanhado cada avanço técnico.

Contudo, a complexidade mais significativa de muitos aplicativos não é técnica. Ela está no próprio domínio, a atividade ou negócio do usuário. Quando essa complexidade de domínio não é tratada no design, não fará diferença se a tecnologia de infraestrutura foi bem concebida. Um design de sucesso deve sistematicamente trabalhar com este aspecto fundamental do software.

São duas as premissas deste livro:

1. Na maioria dos projetos de software, o principal foco deve ser o domínio e a lógica do domínio.

2. Designs de domínios complexos devem se basear em um modelo.

O DDD é uma maneira de pensar e um conjunto de prioridades, voltado para a aceleração de projetos de software que têm que trabalhar com domínios complicados. Para atingir este objetivo, este livro apresenta um conjunto completo de práticas, técnicas e princípios de design.

Design *versus* processo de desenvolvimento

Livros de design. Livros de processos. Eles raramente fazem referência um ao outro. Cada tópico já é complexo por si só. Este é um livro de design, mas acredito que design e processos sejam inseparáveis. Os conceitos de design devem ser implementados com sucesso ou acabarão se tornando uma discussão acadêmica.

Ao aprenderem técnicas de design, as pessoas se sentem entusiasmadas com as possibilidades existentes. Logo em seguida, baixam sobre elas as realidades confusas de um projeto real. Elas não conseguem enquadrar as novas ideias de design com a tecnologia que devem usar. Ou não sabem quando deixar de lado um determinado aspecto do design em função do tempo disponível, ou quando lutar com

unhas e dentes em busca de uma solução adequada. Os desenvolvedores podem e devem conversar uns com os outros de forma abstrata sobre a aplicação dos princípios do design, mas é mais natural falar sobre como a realidade acontece. Por isso, embora este seja um livro de design, vou passar por esse limite artificial e ir direto aos processos, quando for necessário. Isso ajudará a inserir os princípios de design dentro do contexto.

Este livro não está associado a uma determinada metodologia, mas é orientado a uma nova família de "processos de desenvolvimento Agile". Especificamente, ele considera que existem duas práticas em ação no projeto. Essas duas práticas são pré-requisitos para que se possa aplicar a abordagem neste livro.

1. *O desenvolvimento é iterativo*. O desenvolvimento iterativo tem sido defendido e praticado há décadas, e é a pedra fundamental dos métodos de desenvolvimento Agile. Existem várias boas discussões nos textos referentes a desenvolvimento Agile e Extreme Programming (ou XP), entre elas, *Surviving Object-Oriented Projects* (Cockburn 1998) e *Extreme Programming Explained* (Beck 1999).

2. *Desenvolvedores e especialistas em domínio têm uma relação íntima*. O DDD compacta uma grande quantidade de conhecimento em um modelo que reflete uma visão profunda do domínio e um enfoque nos conceitos principais. É uma colaboração entre quem conhece o domínio e quem sabe como construir softwares.

Como o desenvolvimento é iterativo, essa colaboração deve continuar durante toda a vida do projeto.

Extreme Programming, concebido por Kent Beck, Ward Cunningham e outros (veja *Extreme Programming Explained* [Beck 2000]), é o mais proeminente dos processos Agile e aquele com que tenho trabalhado com mais frequência. Durante todo este livro, para concretizar explicações, vou usar o XP como base para discussões sobre a interação entre design e processos. Os princípios ilustrados são facilmente adaptados a outros processos Agile.

Nos últimos anos, tem havido uma rebelião contra metodologias de desenvolvimento mais elaboradas que sobrecarregam os projetos com documentos inúteis e estáticos além de planejamento e design obsessivos e "abertos" demais. Ao contrário destes, os processos Agile, tais como o XP, enfatizam a capacidade de lidar com mudanças e incertezas.

O Extreme Programming reconhece a importância das decisões de design, mas resiste bastante ao design "aberto". Em vez disso, ele dedica um esforço admirável a comunicar e melhorar a capacidade que o projeto tem de mudar de rumo rapidamente. Com essa capacidade de reação, os desenvolvedores podem utilizar "aquilo que há

de mais simples e funcional" em qualquer etapa de um projeto e, em seguida, fazer contínuas refatorações, realizando várias pequenas melhorias no design, chegando finalmente a um design que satisfaça as verdadeiras necessidades do cliente.

Esse minimalismo tem sido um antídoto de grande necessidade para alguns excessos cometidos pelos entusiastas do design. Alguns projetos têm ficado atolados por documentos incômodos que de pouco valem. Eles têm sofrido de "paralisia analítica", onde os membros das equipes têm tanto medo de um design imperfeito que acabam não progredindo. Algo tinha que mudar.

Infelizmente, algumas dessas ideias de processos podem ser mal interpretadas. Cada pessoa tem uma definição diferente do que vem a ser "mais simples". A refatoração contínua é uma série de pequenos re-designs; desenvolvedores que não tenham solidificado os princípios do design vão gerar um banco de códigos difícil de entender ou mudar – o contrário da agilidade. Embora o medo de exigências não antecipadas geralmente leve ao exagero na área de engenharia, a tentativa de se evitar o exagero na engenharia pode se transformar em outro medo: o medo de raciocinar profundamente sobre o design.

Na verdade, o XP funciona melhor para desenvolvedores com um sentido de design aguçado. O processo do XP considera que seja possível melhorar um design através da refatoração, e que isso seja feito com frequência e rapidez. Mas as antigas opções de design facilitam ou dificultam a própria refatoração. O processo do XP tenta aumentar a comunicação das equipes, mas as opções de modelo e design clareiam ou confundem a comunicação.

Este livro une a prática de design e desenvolvimento ilustrando como o DDD (*domain-driven design*) e o desenvolvimento Agile reforçam-se um ao outro. Uma abordagem sofisticada com relação à modelagem de domínios dentro do contexto de um processo de desenvolvimento Agile acelera o desenvolvimento. A interpelação de processo com desenvolvimento de domínios torna essa abordagem mais prática que qualquer outro tratamento de design "puro" lançado no vácuo.

A estrutura deste livro

Este livro é dividido em quatro seções principais:

Parte I: Colocando o modelo de domínios em ação apresenta os objetivos básicos do desenvolvimento dirigido por domínio; esses objetivos motivam as práticas nas seções posteriores. Como existem tantas abordagens com relação ao desenvolvimento de softwares, a Parte I define termos e oferece uma visão geral das implicações do uso do modelo de domínios para conduzir a comunicação e o design.

Parte II: Os tijolos da construção de um design baseado em modelos condensa o principal das melhores práticas utilizadas na modelagem de domí-

nios orientada a objetos transformando-as em um conjunto de tijolos básicos. Essa seção se concentra em transpor o espaço existente entre modelos e softwares práticos e operacionais. O compartilhamento desses padrões convencionais põe ordem ao design. Os membros das equipes podem facilmente entender o trabalho uns dos outros. O uso de padrões convencionais também contribui com terminologias para uma linguagem em comum, que todos os membros das equipes possam utilizar para discutir sobre modelos e decisões de design.

Mas o principal objetivo dessa seção é concentrar-se nos tipos de decisões que mantêm o modelo e a implementação alinhados um com o outro, cada qual reforçando a eficácia do outro. Esse alinhamento requer atenção aos detalhes de cada elemento. A cuidadosa elaboração nessa minúscula escala dá aos desenvolvedores a solidez do alicerce a partir do qual eles podem aplicar as abordagens de modelagem apresentadas nas Partes III e IV.

Parte III: A refatoração para um melhor entendimento vai além dos tijolos de construção com o desafio de colocá-los um sobre o outro e transformá-los em modelos práticos que paguem o investimento. Em vez de ir direto a princípios esotéricos de design, essa seção enfatiza o processo de descobrimento. Os grandes modelos não surgem imediatamente; eles requerem um profundo entendimento do domínio. Esse entendimento provém de um mergulho a fundo no assunto, implementando um design inicial baseado em um modelo provavelmente ingênuo e, em seguida, transformando-o várias vezes. A cada nova visão adquirida pela equipe, o modelo é transformado para revelar esse conhecimento mais aprofundado e o código é refatorado para refletir esse modelo mais profundo e disponibilizar seu potencial para o aplicativo. É aí que, de vez em quando, esse descascar de batatas leva a uma oportunidade de se chegar a um modelo bem mais profundo, sustentado por várias mudanças no design.

A exploração é inerentemente livre, mas não tem que ser aleatória. A Parte III mergulha nos princípios de modelagem que podem conduzir as escolhas ao longo do caminho e as técnicas que ajudam a direcionar a pesquisa.

Parte IV: Design estratégico lida com as situações que surgem em sistemas complexos, organizações maiores e interações com sistemas externos e sistemas legado. Essa seção explora três princípios que se aplicam ao sistema como um todo: contexto, destilação e estrutura em larga escala. As decisões estratégicas de design são tomadas pelas equipes, ou até mesmo entre equipes. O design

estratégico permite que os objetivos da Parte I sejam realizados em uma escala mais larga, para um grande sistema ou um aplicativo que atenda a uma rede corporativa em expansão.

Durante todo o livro, as discussões são ilustradas não só com problemas supersimplificados, mas com exemplos realistas adaptados a partir de projetos verdadeiros.

Boa parte do livro é escrita como um conjunto de "padrões". Os leitores provavelmente conseguirão entender o material sem se preocupar com esse artifício, mas quem estiver interessado no estilo e no formato dos padrões pode recorrer à leitura do apêndice.

A quem se destina este livro

Este livro é escrito principalmente para desenvolvedores de softwares orientados a objetos. A maioria dos membros de uma equipe de projetos de softwares pode tirar proveito de algumas partes do livro. Ele fará mais sentido para aqueles que estejam atualmente envolvidos em um projeto, tentando aplicar algumas dessas coisas no decorrer de seus trabalhos, e para aqueles que já possuem experiência profunda com esse tipo de projeto.

Para tirar maior proveito deste livro, é necessário ter algum conhecimento de modelagem orientada a objetos. Os exemplos incluem diagramas UML e códigos Java; portanto, é importante ter a capacidade de ler essas linguagens em nível básico, mas é desnecessário dominar os detalhes de cada uma. O conhecimento de Extreme Programming dá uma maior perspectiva às discussões sobre processos de desenvolvimento, mas o material deve ser compreensível também para quem não tem conhecimento anterior sobre o assunto.

Para desenvolvedores de software intermediários – leitores que já sabem alguma coisa sobre design orientado a objetos e possivelmente já leram um ou mais livros sobre design de softwares – este livro vai preencher as lacunas e oferecer perspectivas sobre como a modelagem de objetos se enquadra na vida real em um projeto de software. Este livro ajuda desenvolvedores intermediários a aprender como aplicar sofisticadas técnicas de modelagem e design em problemas práticos.

Desenvolvedores de software avançados ou experientes vão se interessar pela estrutura abrangente do livro para lidar com domínios. Essa abordagem sistemática com relação ao design ajudará líderes técnicos a conduzir suas equipes ao longo deste caminho. Além disso, a terminologia coerente utilizada durante todo o livro ajudará os desenvolvedores avançados a se comunicar com seus colegas.

Este livro é uma narrativa, e pode ser lido do início ao fim, ou a partir de qualquer capítulo. Leitores de vários ramos talvez prefiram seguir diferentes caminhos ao longo do livro, mas recomendo que todos os leitores comecem com a introdução à Parte I, bem como o Capítulo 1. Fora destes, as partes principais são provavel-

mente os Capítulos 2, 3, 9 e 14. Quem já possui certo conhecimento de um tópico não deverá ter dificuldade em assimilar os principais pontos lendo os cabeçalhos e o texto em negrito. Leitores mais avançados talvez prefiram ler rapidamente as Partes I e II e ir para o seu ponto de maior interesse, que são as partes III e IV.

Além dessa leitura fundamental, analistas e gerentes de projeto relativamente técnicos também tirarão proveito com a leitura do livro. Analistas podem usar a associação feita entre modelo e design para criar contribuições mais eficientes no contexto de um projeto Agile. Analistas talvez possam também utilizar alguns dos princípios de design estratégico para melhor enfocar e organizar seu trabalho.

Gerentes de projeto provavelmente estarão interessados na ênfase dada em como fazer uma equipe mais eficiente e mais enfocada no design de softwares que sejam significativos para os especialistas e usuários de negócios. E como as decisões de design estratégico estão inter-relacionadas com a organização da equipe e os estilos de trabalho, essas decisões de design necessariamente envolvem a liderança do projeto e têm grande impacto sobre a trajetória do projeto.

Uma equipe dirigida por domínios

Embora um desenvolvedor que entenda sobre DDD possa adquirir excelentes técnicas e perspectivas de design, os maiores ganhos acontecem quando uma equipe se une para aplicar uma abordagem de DDD e trazer o modelo do domínio para o centro de discussão do projeto. Ao fazer isso, os membros da equipe compartilham uma linguagem que enriquece sua comunicação e a mantém ligada ao software. Eles conseguem produzir uma implementação lúcida em compasso com o modelo, alavancando o desenvolvimento do aplicativo. Eles compartilham um mapa da relação existente entre o trabalho de design de diversas equipes concentrando sistematicamente sua atenção nas características mais distintas e valiosas para a organização.

O DDD é um desafio técnico difícil que pode ser compensado por grandes oportunidades de abertura exatamente quando a maioria dos projetos de software começa a se consolidar e a se transformar em um legado.

AGRADECIMENTOS

De uma forma ou de outra, venho trabalhando neste livro há mais de quatro anos, e muitas pessoas me ajudaram e apoiaram ao longo do caminho.

Agradeço aquelas pessoas que leram os manuscritos e fizeram seus comentários. Este livro simplesmente não teria sido possível sem esse retorno. Alguns deram às suas críticas uma atenção especialmente generosa. O Silicon Valley Patterns Group, sob o comando de Russ Rufer e Tracy Bialek, passou sete semanas examinando minuciosamente o primeiro rascunho completo do livro. O grupo de leitura da University of Illinois, sob o comando de Ralph Johnson, também passou várias semanas revisando uma das versões posteriores. Foi profundo o efeito proveniente das longas e animadas discussões desses grupos. Kyle Brown e Martin Fowler contribuíram com comentários detalhados, valiosas opiniões e incomensurável apoio moral (enquanto aguardávamos para saborear um peixe). Os comentários feitos por Ward Cunningham me ajudaram a trabalhar em alguns pontos fracos importantes. Alistair Cockburn, assim como Hilary Evans, me incentivou desde o início e me ajudou a descobrir o caminho a tomar ao longo do processo de publicação. David Siegel e Eugene Wallingford me ajudaram a evitar situações embaraçosas nas partes mais técnicas. Vibhu Mohindra e Vladimir Gitlevich analisaram minuciosamente todos os exemplos de códigos.

Rob Mee leu parte das minhas primeiras explorações do material e me ajudou a colher ideias enquanto eu procurava uma forma de transmitir este estilo de design, ajudando-me também a analisar uma versão posterior.

John Kerievsky é responsável por uma das principais viradas no desenvolvimento deste livro: ele me persuadiu a experimentar o formato de padrão "alexandrino", que se tornou fundamental para a organização deste livro. Ele também me ajudou a reunir a parte do material agora na Parte II e dar-lhe uma forma coerente pela primeira vez, durante o processo intensivo de "acompanhamento" que precedeu a conferência PLoP em 1999. Esta foi a semente em torno da qual boa parte do restante do livro germinou.

Agradeço também a Awad Faddoul pelas centenas de horas que passei sentado escrevendo em seu maravilhoso café. Esse lugar, aliado a muito windsurf, ajudou-me a continuar a caminhada.

Sou grato também a Martine Jousset, Richard Paselk e Ross Venables pela criação de belas fotografias para ilustração de alguns conceitos importantes (veja créditos de fotografias na página 491).

Antes de poder conceber este livro, tive que formar minha visão e compreensão sobre desenvolvimento de softwares. Essa formação deve muito à generosidade de algumas pessoas brilhantes que agiram como mentores informais e amigos para mim. David Spiegel, Eric Gold e Iseult White, cada qual de sua forma, me ajudaram a desenvolver minha forma de pensar em design de softwares. Ao mesmo tempo, Bruce Gordon, Richard Freyberg e Dr. Judith Segal, também de formas bastante diferentes, me ajudaram a encontrar meu caminho no mundo dos projetos bem-sucedidos.

Minhas emoções próprias naturalmente se desenvolveram a partir de um conjunto de ideias que pairavam no ar naquela época. Algumas dessas contribuições aparecerão de forma clara no texto principal e, onde possível, serão feitas referências a elas. Outras são tão fundamentais que nem mesmo percebo sua influência sobre mim.

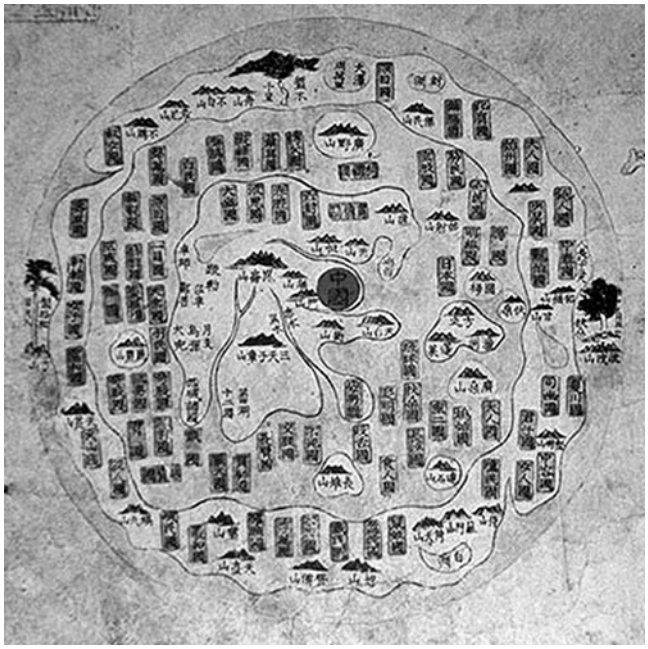
Dr. Bala Subramanium, orientador de minha tese de mestrado, me despertou para a modelagem matemática, que aplicamos à cinética das reações químicas. Modelagem é modelagem, e esse trabalho fez parte do caminho que levou a este livro.

Muito antes disso, minha forma de pensar foi moldada por meus pais, Carol e Gary Evans. E alguns professores especiais despertaram meu interesse e me ajudaram a construir minhas bases, principalmente Dale Currier (professor de matemática do ensino médio), Mary Brown (professora de redação e inglês do ensino médio) e Josephine McGlamery (professora de ciências da 6ª série).

Por último, agradeço meus amigos e minha família, e a Fernando De Leon, por seu incentivo ao longo de todo o caminho.

I

Colocando o modelo de domínios em ação



Este mapa chinês do século 18 representa o mundo todo. No centro, e ocupando a maior parte do espaço, está a China, cercada por representações espalhadas de outros países. Este é um modelo do mundo adequado àquela sociedade, que intencionalmente havia sido fechada. A visão de mundo que o mapa representa não deve ter ajudado a lidar com estrangeiros. Certamente, ela não se aplicaria à China moderna. Mapas são modelos, e cada modelo representa algum aspecto da realidade com uma ideia que seja de interesse. Um modelo é uma simplificação. Ele é uma interpretação da realidade que destaca os aspectos relevantes para resolver o problema que se tem em mãos ignorando os detalhes estranhos.

Cada programa de software está relacionado a alguma atividade ou interesse do seu usuário. Essa área de assunto em que o usuário aplica o programa é o *domínio* do software. Alguns domínios envolvem o mundo físico: o domínio de um programa de reservas de passagens aéreas envolve pessoas reais entrando em aeronaves reais. Alguns domínios são intangíveis: o domínio de um programa de contabilidade são o dinheiro e as finanças. Os domínios dos softwares geralmente têm pouco a ver com computadores, embora haja exceções: o domínio de um sistema de controle de códigos-fonte é o próprio desenvolvimento de softwares.

Para criar softwares que tenham valor para as atividades desempenhadas pelos usuários, a equipe de desenvolvimento deve trazer consigo um conjunto de conhecimentos relacionados a essas atividades. A quantidade de conhecimento necessária pode ser estarrecedora. O volume e a complexidade de informações podem ser imensos. Modelos são ferramentas para atacar essa sobrecarga. Um modelo é uma forma de conhecimento seletivamente simplificada e conscien-

temente estruturada. Um modelo adequado faz com que as informações tenham sentido e se concentra em um problema.

Um modelo de domínio não é um diagrama específico; ele é a ideia que o diagrama pretende transmitir. Ele não é simplesmente o conhecimento existente na cabeça de um especialista em domínios; *ele é uma abstração rigorosamente organizada e seletiva daquele conhecimento*. Um diagrama pode representar e comunicar um modelo, assim como acontece com um código cuidadosamente escrito, assim como uma frase em português.

A modelagem de domínios não é uma questão de se criar um modelo o mais "realista" possível. Mesmo em um domínio de coisas tangíveis da vida real, nosso modelo é uma criação artificial. Ele também não é simplesmente a construção de um mecanismo de software que fornece os resultados necessários. Ele mais se assemelha ao processo de criação de um filme, representando livremente a realidade com uma determinada finalidade. Até mesmo um filme-documentário não mostra uma vida real inédita. Assim como um cineasta seleciona aspectos da experiência e os apresenta de forma idiossincrática para contar uma história ou se fazer entender, um modelador de domínios escolhe um modelo em particular pela sua utilidade.

A utilidade de um modelo no Domain-Driven Design

No DDD (*domain-driven design*), são três as utilidades básicas que determinam a escolha de um modelo.

1. *O modelo e o coração do design dão forma um ao outro*. É essa ligação íntima entre o modelo e a implementação que torna o modelo relevante e garante que a análise a ele dedicada se aplique ao produto final, um programa de execução. Essa ligação de modelo e implementação também ajuda durante a manutenção e contínuo desenvolvimento, pois o código pode ser interpretado com base na compreensão do modelo. (Veja o Capítulo 3).

2. *O modelo é a espinha dorsal de uma linguagem utilizada por todos os membros da equipe*. Devido à ligação de modelo e implementação, os desenvolvedores podem conversar sobre o programa nessa linguagem. Eles podem comunicar-se com os especialistas do domínio sem a necessidade de tradução. E como a linguagem é baseada no modelo, nossa capacidade linguística natural pode ser usada para refinar o próprio modelo. (Veja o Capítulo 2).

3. *O modelo é um conhecimento destilado*. O modelo é a forma aceita pela equipe de estruturar o conhecimento do domínio e distinguir os elementos de maior interesse. Um modelo capta a forma que escolhemos para pensar no domínio à medida que selecionamos termos, interpretamos conceitos e