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 idéia 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 idéia 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 lingüística 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

os relacionamos. A linguagem compartilhada permite que os desenvolvedores e os especialistas do domínio trabalhem efetivamente em conjunto à medida que colocam informações nessa forma. A ligação de modelo e implementação faz com que as experiências obtidas com versões anteriores do software sejam aplicáveis como feedback para o processo de modelagem. (Veja o Capítulo 1).

Os três próximos capítulos examinam o significado e o valor de cada uma dessas contribuições, uma de cada vez, e como elas estão interligadas. O uso de um modelo dessa forma pode sustentar o desenvolvimento do software com grandes funcionalidades que, de outra forma, precisariam de investimentos maciços de desenvolvimento na medida dos acontecimentos.

O coração do software

O coração do software está na sua capacidade de resolver problemas relacionados ao domínio para o seu usuário. Todas as outras características, por mais vitais que possam ser, se apóiam nessa finalidade básica. Quando o domínio é complexo, esta é uma tarefa difícil, exigindo a concentração de esforços de pessoas talentosas e capacitadas. Os desenvolvedores têm que mergulhar a fundo no domínio para adquirir conhecimentos sobre o negócio. É preciso afiar sua capacidade de modelagem e dominar design de domínios.

Todavia, estas não são as prioridades na maioria dos projetos de software. A maioria dos desenvolvedores talentosos não tem muito interesse em aprender sobre um domínio específico em que eles estejam trabalhando, muito menos firmar um compromisso em expandir suas capacidades de modelagem de domínios. Pessoas técnicas gostam de problemas quantificáveis que exercitem suas capacidades técnicas. O trabalho com domínios é confuso e exige muitos novos conhecimentos complicados que parecem não acrescentar nada à capacidade de um cientista da computação.

Em vez disso, o talento técnico é direcionado para o trabalho com estruturas elaboradas, na tentativa de resolver problemas relacionados a domínios usando a tecnologia. Aprender e modelar domínios ficam a cargo dos outros. A complexidade existente no coração de um software deve ser enfrentada cara a cara. Qualquer tentativa de se fazer o contrário é arriscar a irrelevância.

Em uma entrevista dada a um programa de TV, o humorista John Cleese contou uma história de um evento que aconteceu durante a filmagem de *Monty Python and the Holy Grail (Monty Pyhton em busca do cálice sagrado)*. Eles já haviam filmado uma determinada cena várias vezes, mas, de alguma forma, ela não era engraçada. Finalmente, ele fez uma pausa e consultou um amigo comediante, Michael Palin (o

outro ator da cena), e fizeram uma pequena variação. Foi filmado mais uma tomada, que dessa vez ficou engraçado, e eles deram o dia por encerrado.

No dia seguinte, Cleese estava dando uma olhada no rascunho que o editor do filme havia feito sobre o trabalho do dia anterior. Ao chegar à cena com a qual eles haviam lutado várias vezes, Cleese não achou nada engraçado; fora usada uma das tomadas anteriores.

Ele então perguntou ao editor do filme porque não havia sido usada a última tomada, conforme as instruções. – Não pude usá-la. Alguém entrou e apareceu na cena, respondeu o editor. Cleese assistiu à cena várias vezes, mas não conseguiu ver nada de errado. Finalmente, o editor pausou o filme e apontou para a manga de um casaco visível por alguns instantes no canto da imagem.

O editor do filme estava concentrado na perfeita execução de sua especialidade. Estava preocupado que outros editores que assistissem a seu filme julgassem seu trabalho com base em sua perfeição técnica. No processo, o coração da cena havia sido perdido ("The Late Late Show with Craig Kilborn", CBS, setembro de 2001).

Felizmente, a cena engraçada foi restaurada por um diretor que entendia de comédia. Exatamente da mesma forma, os líderes de uma equipe que entendem os fundamentos de um domínio podem colocar seu projeto de software de volta em seu rumo quando o desenvolvimento de um modelo que reflete um profundo entendimento se perde em meio à confusão.

Este livro mostra que o desenvolvimento de domínios traz oportunidades para cultivar capacidades bastante sofisticadas de design. A confusão da maioria dos domínios de software é, na verdade, um interessante desafio técnico. De fato, em várias disciplinas científicas, "complexidade" é um dos tópicos atuais mais motivantes, pois os pesquisadores tentam controlar a confusão da vida real. Um desenvolvedor de software tem essa mesma perspectiva quando se depara com um domínio complicado e que nunca foi formalizado. Criar um modelo lúcido que elimine essa complexidade é algo excitante.

Existem maneiras sistemáticas de pensar que podem ser empregadas pelos desenvolvedores na busca por novas visões e pela geração de modelos eficazes. Existem técnicas de design que podem dar ordem a um aplicativo de software em expansão. O cultivo dessas capacidades torna um desenvolvedor muito mais valioso, até mesmo em um domínio inicialmente pouco familiar.

UM

Assimilando o conhecimento

Alguns anos atrás, decidi criar uma ferramenta de software especializada para o projeto de uma placa de circuito impresso (PCI). Uma armadilha: não sabia nada sobre hardware eletrônico. Obviamente, tive acesso a alguns projetistas de PCIs, mas, normalmente, bastavam 3 minutos para que eles deixassem minha cabeça pegando fogo. Como é que eu poderia conseguir entender o suficiente sobre o assunto para escrever esse software? Certamente, eu não ia me tornar um engenheiro eletricista antes do prazo de entrega do serviço!

Tentamos fazer com que os projetistas de PCIs me dissessem exatamente o que o software deveria fazer. Péssima idéia. Eles eram ótimos projetistas de circuitos, mas suas idéias sobre softwares geralmente envolviam a leitura de um arquivo ASCII, sua classificação, sua escrita com algumas anotações e a geração de um relatório. Isso obviamente não levaria ao grande avanço em produtividade que eles procuravam.

Os primeiros encontros foram desmotivantes, mas havia um sinal de esperança nos relatórios que eles pediram. Eles sempre envolviam "redes" e vários detalhes sobre elas. Uma rede, nesse domínio, é essencialmente um condutor de fios que pode conectar qualquer número de componentes de uma PCI e transmitir um sinal elétrico a tudo a que ele esteja conectado. Tínhamos então o primeiro elemento do modelo do domínio.

Figura 1.1

Comecei a desenhar diagramas para eles de acordo com o que eles queriam que o software fizesse. Usei uma variante informal de diagramas de interação de objetos para traçar os cenários.

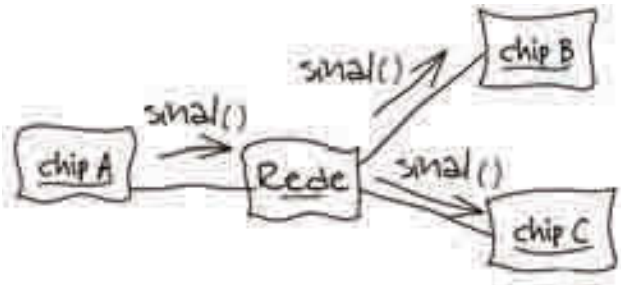

Figura 1.2

Especialista em PCI 1: Os componentes não teriam que ser chips.

Desenvolvedor (eu): Então eu deveria chamá-los apenas de "componentes"?

Especialista 1: Nós o chamamos de "instâncias de componentes". Pode haver vários componentes do mesmo tipo.

Especialista 2: A caixa da "rede" parece ser exatamente uma instância de componente.

Especialista 1: Ele não está usando a nossa notação. Para eles, tudo é uma caixa, suponho.

Desenvolvedor: Confesso que sim. Acho melhor eu explicar essa notação um pouco mais.

Eles me corrigiam constantemente, e à medida que o faziam, comecei a aprender. Eliminamos colisões e ambigüidades em suas terminologias e as diferenças entre suas opiniões técnicas, e eles aprenderam. Começaram a explicar as coisas com mais precisão e consistência, e começamos a desenvolver um modelo juntos.

Especialista 1: Não basta dizer que um sinal chega a um ref-des, temos que saber qual pino.

Desenvolvedor: Ref-des?

Especialista 2: O mesmo que instância de componentes. Ref-des é o nome usado em uma ferramenta que utilizamos.

Especialista 1: De qualquer forma, uma rede conecta um determinado pino de uma instância a um determinado pino de outra.

Desenvolvedor: Quer dizer então que um pino pertence a apenas uma instância de componentes e se conecta a apenas uma rede?

Especialista 1: Sim, exatamente.

Especialista 2: Além disso, cada rede tem uma topologia, um arranjo que determina a forma em que os elementos da rede se conectam.

Desenvolvedor: OK, que tal isso?

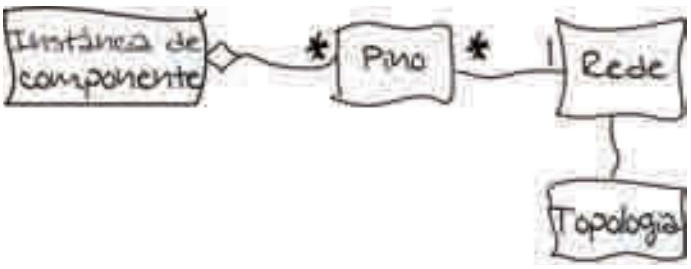

Figura 1.3

Para enfocar nossa exploração, limitamo-nos, por um tempo, a estudar um recurso específico. Uma "simulação de teste" rastrearia a propagação de um sinal para detectar possíveis locais de certos tipos de problemas no projeto.

Desenvolvedor: Entendo como o sinal é transmitido pela **Rede** a todos os **Pinos** ligados, mas como é que ele vai além disso? Será que a **Topologia** tem algo a ver com isso?

Especialista 2: Não. O componente empurra o sinal.

Desenvolvedor: Certamente não podemos modelar o comportamento interno de um chip. Seria muito complicado.

Especialista 2: Não é necessário. Podemos usar uma simplificação. Apenas uma lista de empurrões dados através do componente de certos **Pinos** para certos **Pinos**.

Desenvolvedor: Algo mais ou menos assim?

[Com consideráveis tentativas e erros, juntos esboçamos um cenário].

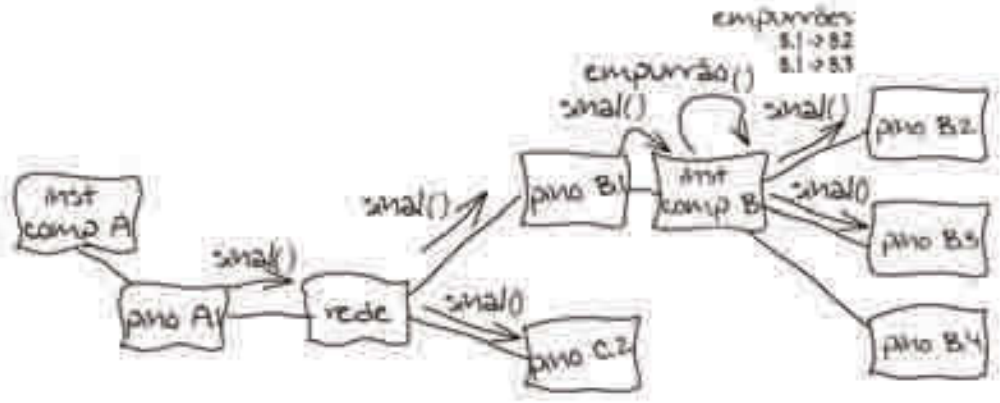

Figura 1.4

Desenvolvedor: Mas o que exatamente você precisa saber com base nessa computação?

Especialista 2: Estamos interessados nos atrasos prolongados de sinal – por exemplo, qualquer trajeto de sinais que tenha sido mais do que dois ou três pulsos. É uma regra prática. Se o trajeto for muito longo, o sinal pode não chegar durante o ciclo do relógio.

Desenvolvedor: Mais do que três pulsos... Então precisamos calcular a extensão dos trajetos. E o que conta como sendo um pulso?

Especialista 2: Cada vez que o sinal passa por uma **Rede**, é um pulso.

Desenvolvedor: Então poderíamos prolongar o número de pulsos, e uma **Rede** poderia incrementá-lo, da seguinte forma:

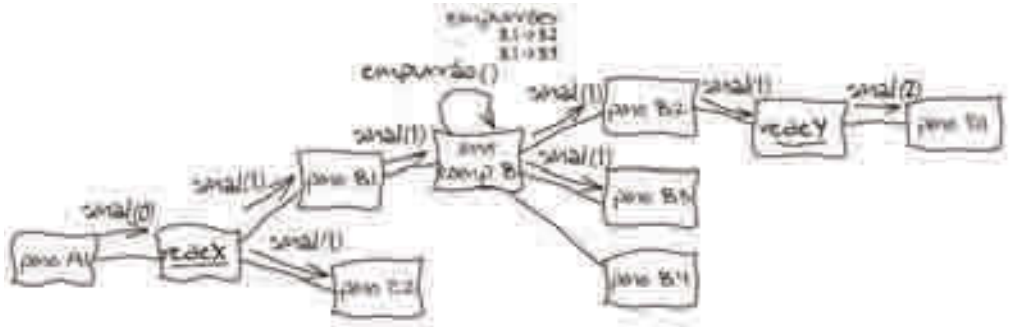

Figura 1.5

Desenvolvedor: A única parte que não ficou clara para mim é de onde vêm os "empurrões". Esses dados são armazenados para cada **Instância de componente**?

Especialista 2: Os empurrões seriam os mesmos para todas as instâncias de componente.

Desenvolvedor: Então o tipo de componente determina os empurrões. Eles seriam os mesmos para cada instância?

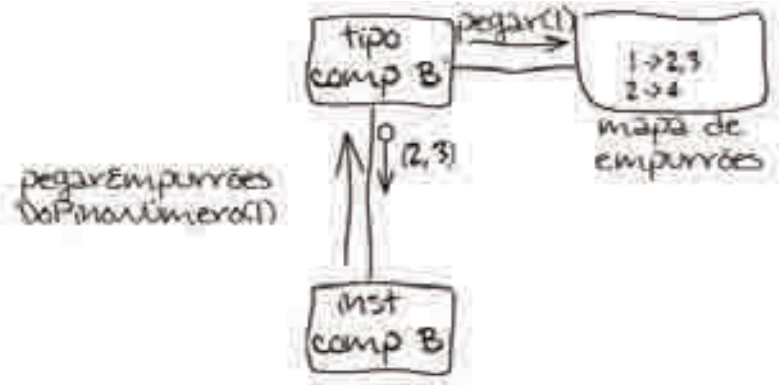

Figura 1.6

Especialista 2: Não sei exatamente o que significa cada uma das partes, mas imagino que o armazenamento de empurrões de cada componente teria um aspecto parecido com isso.

Desenvolvedor: Desculpe. Coloquei detalhes demais. Estava só pensando... Agora, então, onde entra a **Topologia**?

Especialista 1: Ela não é usada para a simulação de teste.

Desenvolvedor: Então vou deixá-la de fora por enquanto, OK? Podemos trazê-la de volta quando chegarmos a essas características.

E assim as coisas caminharam (com muito mais tropeções do que mostrados aqui). Colhendo e refinando idéias, questionando e explicando, o modelo se desenvolveu junto com o meu entendimento do domínio e o entendimento que eles tinham de como o modelo atuaria na solução. Um diagrama de classes para representar o modelo inicial tem a seguinte aspecto:

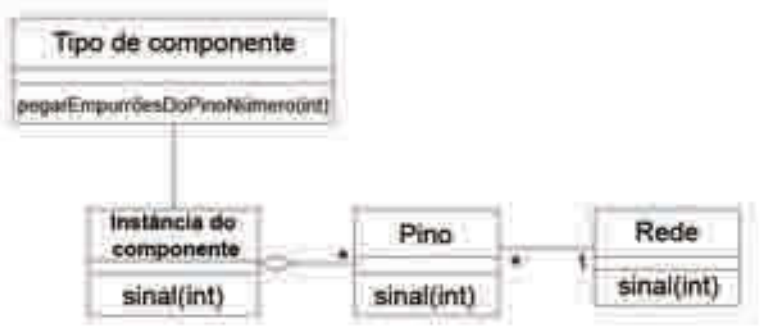

Figura 1.7

Depois de mais alguns dias, percebi que entendia o suficiente para tentar algum tipo de código. Escrevi um protótipo bastante simples, movido por um framework de teste automatizado. Evitei toda a infra-estrutura. Não havia nenhuma persistência e nenhuma interface com o usuário (IU). Isso me permitiu concentrar no comportamento. Consegui demonstrar uma simulação de teste simples em poucos dias. Embora utilizasse dados fictícios e escrevesse texto puro no console, ela estava fazendo a computação real das extensões de trajetos utilizando objetos Java. Esses objetos Java refletiam um modelo compartilhado pelos especialistas do domínio e eu.

A solidificação desse protótipo tornou mais claro para os especialistas do domínio o que o modelo significava e que relação havia entre ele e o software operacional. A partir deste ponto, as discussões sobre o nosso modelo se tornaram mais interativas, pois eles puderam ver como incorporei meu recém-adquirido conhecimento no modelo e, depois, no software. E eles tiveram um feedback concreto a partir do protótipo para avaliar seus próprios pensamentos.

Embutido naquele modelo, que naturalmente se tornou muito mais complicado do que o mostrado aqui, estava o conhecimento sobre o domínio de PCIs relevante para os problemas que estávamos resolvendo. Ele consolidou muitos sinônimos e pequenas variações de descrição. E excluiu centenas de fatos que os engenheiros entendiam, mas que não eram diretamente relevantes, tais como os verdadeiros recursos digitais dos componentes. Um especialista em software como eu poderia observar os diagramas e, em questão de minutos, começar a entender do que se tratava o software, pois ele teria uma estrutura para organizar novas informações e aprender com maior rapidez, para fazer deduções melhores sobre o que era importante ou não, e comunicar-se melhor com os engenheiros de PCIs.

À medida que os engenheiros descreviam os novos recursos de que precisavam, fiz com que eles me conduzissem através dos cenários de interação dos objetos. Quando os objetos do modelo não nos podiam conduzir através de um cenário importante, idealizávamos novos cenários ou os alterávamos, assimilando o conhecimento. Refinamos o modelo; o código evoluiu. Alguns meses mais tarde, os engenheiros de PCIs contavam com uma excelente ferramenta que excedia suas expectativas.

Os ingredientes de uma modelagem eficaz

Algumas coisas que fizemos levaram ao sucesso que acabei de descrever.

1. *Ligando o modelo e a implementação*. Esse protótipo rudimentar forjou o elo essencial logo no início, e ele foi mantido em todas as iterações posteriores.

2. *Cultivando uma linguagem baseada no modelo*. Inicialmente, os engenheiros tiveram que me explicar questões elementares sobre PCIs, e tive que explicar o que significavam diagramas de classes. Mas à medida que o projeto avançava, qualquer um de nós podia tirar os termos diretamente do modelo, organizá-los em frases consistentes com a estrutura do modelo e ser entendido sem ambigüidades e sem tradução.

3. *Desenvolvendo um modelo rico em conhecimento*. Os objetos tinham um comportamento e impunham regras. O modelo não era simplesmente um esquema de dados; ele era necessário e importante para a resolução de um problema complexo. E captava vários tipos de conhecimento.

4. *Destilando o modelo*. Conceitos importantes foram acrescentados ao modelo à medida que ele se tornava mais completo, mas conceitos igualmente importantes foram eliminados quando provavam ser inúteis ou secundários. Quando um conceito desnecessário estava ligado a um conceito necessário, achava-se um novo modelo que distinguisse o conceito essencial para que o outro pudesse ser eliminado.

5. *Colhendo idéias e experimentando*. A linguagem, aliada a esboços e uma atitude de geração de idéias, transformou nossas discussões em laboratórios do modelo, em que centenas de variações experimentais podiam ser exercitadas, tentadas e julgadas. À medida que a equipe passava por cenários, as expressões faladas proporcionavam um rápido teste de viabilidade de um modelo proposto, pois o ouvido podia rapidamente detectar a clareza e a facilidade ou a estranheza da expressão.

É a criatividade das idéias e da experimentação maciça, alavancadas através de uma linguagem baseada em modelos e disciplinada pelo ciclo de feedback através da implementação, que possibilita encontrar um modelo rico em conhecimento e destilá-lo. Esse tipo de *assimilação do conhecimento* transforma o conhecimento da equipe em modelos valiosos.

Assimilando o conhecimento

Analistas financeiros digerem números. Eles analisam centenas de números detalhadamente, combinando e recombinando-os procurando pelo significado que existe por trás, em busca de uma apresentação simples que destaque o que é realmente importante – um entendimento que pode ser a base de uma decisão financeira.

Modeladores de domínio eficazes digerem conhecimento. Eles tomam uma torrente de informações e saem em busca daquilo que é relevante. Experimentam uma idéia de organização após a outra, buscando a visão simples que dê sentido à massa. Muitos modelos são tentados, rejeitados ou transformados. O sucesso vem em um conjunto emergente de conceitos abstratos e que dá sentido a todos os detalhes. Essa destilação é uma expressão rigorosa do conhecimento específico considerado mais relevante.

A assimilação do conhecimento não é uma atividade solidária. Uma equipe de desenvolvedores e especialistas do domínio trabalha em conjunto, normalmente sob o comando de desenvolvedores. Juntos, eles coletam informações e as digerem dando-lhes uma forma útil. O material puro vem das mentes dos especialistas do domínio, de usuários de sistemas existentes, de experiências anteriores da equipe técnica com um sistema legado relacionado a ele ou de outro projeto no mesmo domínio. Ele vem na forma de documentos escritos para o projeto ou utilizados no negócio, e muitas, muitas conversas. As versões iniciais, ou protótipos, dão retorno das experiências para a equipe e mudam as interpretações.

No antigo método da cascata, os especialistas de um negócio conversam com os analistas, e os analistas digerem, assimilam e passam o resultado para os programadores, que codificam o software. Essa abordagem falha porque lhe falta feedback. Os analistas têm responsabilidade total por criar o modelo, baseados somente nas

informações fornecidas pelos especialistas daquele negócio. Eles não têm a oportunidade de aprender com os programadores ou adquirir experiência com versões iniciais do software. O conhecimento escoa em uma direção, mas não acumula.

Outros projetos utilizam um processo iterativo, mas deixam de acumular conhecimento porque não o assimilam. Os desenvolvedores fazem com que os especialistas descrevam um recurso desejado e, em seguida, partem para construí-lo. Mostram aos especialistas o resultado e perguntam o que devem fazer em seguida. Se os programadores praticarem a refatoração, eles poderão manter o software limpo o suficiente para continuar a expandi-lo. Mas, se não estiverem interessados no domínio, os programadores aprendem somente o que o aplicativo deve fazer, e não os princípios existentes por trás dele. Softwares úteis podem ser construídos dessa forma, mas o projeto nunca chegará ao ponto em que novos recursos poderosos se revelam como corolários aos antigos recursos.

Bons programadores naturalmente vão começar a assimilar e desenvolver um modelo que possa executar mais serviços. Mas, quando isso acontece somente em um ambiente técnico, sem colaboração com especialistas do domínio, os conceitos são ingênuos. A superficialidade do conhecimento produz softwares que fazem serviços básicos, mas que não contam com uma ligação profunda com a forma de pensar de um especialista naquele domínio.

A interação entre os membros da equipe muda à medida que todos os membros digerem aquele modelo em conjunto. O constante refinamento do modelo do domínio força os desenvolvedores a aprender os princípios importantes do negócio ao qual estão dando assistência, em vez de gerar funções mecanicamente. Os especialistas do domínio geralmente refinam seu próprio entendimento sendo forçados a destilar o que sabem em sua essência e passam a entender o rigor conceitual exigido pelos projetos de software.

Tudo isso faz com que os membros da equipe sejam mais competentes ao digerir o conhecimento. Eles eliminam aquilo que é estranho. E dão uma forma cada vez mais útil ao modelo. Como analistas e programadores estão contribuindo para ele, ele se torna limpo, organizado e assimilado, para que possa servir de alavanca para a implementação. Como os especialistas naquele domínio estão contribuindo para ele, o modelo reflete um profundo conhecimento do negócio. As abstrações são princípios reais daquele negócio.

À medida que o modelo avança, ele se torna uma ferramenta para organizar as informações que continuam a fluir pelo projeto. O modelo se concentra na análise dos requisitos, interagindo intimamente com a programação e o design. E, em um círculo virtuoso, ele aprofunda a visão que os membros da equipe têm sobre o domínio, permitindo que eles vejam com mais clareza levando a um refinamento ainda maior do modelo. Esses modelos nunca são perfeitos; eles evoluem. Devem ser práticos e úteis em dar sentido ao domínio. Devem ser rigorosos o suficiente para tornar o aplicativo simples de implementar e entender.

Aprendizado contínuo

Quando decidimos escrever um software, nunca sabemos o suficiente. O conhecimento sobre o projeto é fragmentado, disperso entre muitas pessoas e documentos, e misturado com outras informações de forma que nem mesmo sabemos qual conhecimento realmente se faz necessário. Domínios aparentemente menos assustadores podem enganar: não percebemos o quanto não sabemos. Essa ignorância nos leva a fazer considerações falsas.

Enquanto isso, todos os projetos deixam o conhecimento escapar. As pessoas que tiverem aprendido alguma coisa continuam. A reorganização dispersa a equipe e o conhecimento é novamente fragmentado. Subsistemas fundamentais são divididos de forma que códigos sejam produzidos, mas não conhecimento. E com as abordagens típicas de design, códigos e documentos não expressam de forma utilizável esse conhecimento arduamente adquirido, e quando a tradição oral é interrompida por qualquer motivo, o conhecimento se perde.

Equipes altamente produtivas aumentam seu conhecimento conscientemente, praticando o *aprendizado contínuo* (Kerievsky 2003). Para os desenvolvedores, isso significa melhorar o conhecimento técnico, junto com a capacidade geral de modelagem de domínios (tais como as apresentadas neste livro). Mas também inclui um aprendizado sério sobre o domínio específico com o qual estão trabalhando.

Esses membros autodidatas de uma equipe formam um núcleo sólido de pessoas para se concentrarem nas tarefas de desenvolvimento que envolvem as áreas mais críticas. (Para mais informações sobre este assunto, veja o Capítulo 15). O conhecimento acumulado nas mentes dessa equipe-núcleo faz com que o conhecimento seja assimilado com maior eficácia.

Neste exato momento, pare e faça uma pergunta a si mesmo. Você aprendeu alguma coisa sobre o processo de criação de um projeto de PCI? Embora este exemplo tenha sido um tratamento superficial daquele domínio, deveria haver algum aprendizado quando um modelo de domínio é discutido. Aprendi muito. Não aprendi a ser um engenheiro de PCIs. Este não era o objetivo. Aprendi a conversar com especialistas em PCIs, entender os principais conceitos relevantes para o aplicativo e avaliar o que estávamos construindo.

De fato, nossa equipe acabou descobrindo que a simulação de teste tinha uma baixa prioridade para o desenvolvimento, e esse recurso acabou sendo descartado por completo. Com ele, foram-se partes do modelo que captavam o entendimento da transmissão de sinais através dos componentes e da contagem de pulsos. O fundamental do aplicativo mostrou estar em outro lugar, e o modelo mudou para trazer esses aspectos para o centro do palco. Os especialistas daquele domínio haviam aprendido mais e esclareceram melhor o objetivo do aplicativo. (O Capítulo 15 discute essas questões mais a fundo).

Mesmo assim, o trabalho inicial foi essencial. Elementos importantíssimos para o modelo foram retidos, mas o mais importante é que o trabalho colocou em ação

o processo de assimilação do conhecimento que fez com que todo o trabalho posterior fosse eficaz: o conhecimento adquirido pelos membros da equipe, desenvolvedores e especialistas do domínio; o início de uma linguagem compartilhada; e o fechamento de um ciclo de feedback através da implementação. Uma viagem de descoberta tem que começar em algum lugar.

Design rico em conhecimentos

O tipo de conhecimento captado em um modelo, como o exemplo do PCI, vai além de "identificar substantivos". As atividades e regras de negócios são tão fundamentais para um domínio quanto as entidades envolvidas; qualquer domínio terá várias categorias de conceitos. A assimilação do conhecimento gera modelos que refletem este tipo de visão. Em paralelo com as mudanças do modelo, os desenvolvedores fazem a refatoração da implementação para expressar o modelo, dando ao aplicativo uma utilização para aquele conhecimento.

É com este movimento que vai além das entidades e dos valores que a assimilação do conhecimento se torna intensa, pois pode haver inconsistências reais entre regras de negócios. Os especialistas do domínio geralmente não estão cientes da complexidade de seus processos mentais à medida que, no decorrer de seu trabalho, eles passam por todas essas regras, reconciliam contradições e preenchem lacunas usando o bom senso. Um software não pode fazer isso. É através da assimilação do conhecimento em um trabalho de colaboração íntimo com especialistas em software que as regras são esclarecidas, acrescentadas, reconciliadas ou descartadas.

Exemplo

Extraindo um conceito oculto

Vamos começar com um modelo de domínio bastante simples que poderia ser a base de um aplicativo para a reserva de cargas na viagem de um navio.

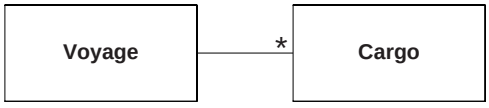

Figura 1.8

Podemos dizer que a responsabilidade do aplicativo de reservas é associar cada **Carga (Cargo)** a uma **Viagem (Voyage)**, registrando e rastreando essa relação. Até aí, tudo bem. Em algum lugar do código do aplicativo, poderia existir um método como o seguinte:

```
public int makeBooking(Cargo cargo, Voyage voyage) {
   int confirmation = orderConfirmationSequence.next();
   voyage.addCargo(cargo, confirmation);
   return confirmation;
}
```

Como sempre há cancelamentos de última hora, a prática adotada no setor de transporte naval é aceitar mais cargas do que uma determinada embarcação pode transportar em uma viagem. Isso recebe o nome de "overbooking".

Às vezes, é usada uma simples porcentagem da capacidade total, como a reserva de 110% da capacidade. Em outros casos, são aplicadas regras complexas, favorecendo clientes importantes ou determinados tipos de carga.

Esta é uma estratégica básica no domínio do transporte naval que seria conhecida de qualquer comerciante que pertença a este setor, mas pode não ser entendida por todo o pessoal técnico de uma equipe de software.

O documento de requisitos contém a seguinte linha:

Considere 10% de overbooking.

O diagrama de classes e o código agora têm o seguinte aspecto:

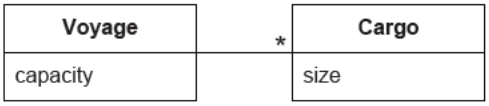

Figura 1.9

```
public int makeBooking(Cargo cargo, Voyage voyage) {
   double maxBooking = voyage.capacity() * 1.1;
   if ((voyage.bookedCargoSize() + cargo.size()) > maxBooking)
      return -1;
   int confirmation = orderConfirmationSequence.next();
   voyage.addCargo(cargo, confirmation);
   return confirmation;
}
```

Agora, uma importante regra de negócio está oculta como uma cláusula de proteção em um método do aplicativo. Mais adiante, no Capítulo 4, vamos examinar o princípio da ARQUITETURA EM CAMADAS, que nos orientará na transferência da regra de overbooking para um objeto de domínio, mas, por enquanto, vamos nos concentrar em como poderíamos tornar este conhecimento mais explícito e acessível a todos que participam do projeto. Isso nos levará a uma solução semelhante.

1. Conforme escrito, é improvável que qualquer especialista de negócio possa ler este código e verificar a regra, mesmo com a ajuda de um desenvolvedor.

2. Seria difícil para uma pessoa técnica, que não pertença à área de negócios, fazer uma ligação entre o texto de requisitos e o código.

Se a regra fosse mais complexa, tudo isso estaria em perigo.

Podemos mudar o design para melhor captar esse conhecimento. A regra do overbooking é uma política. *Política* é outro nome dado ao padrão de design conhecido como STRATEGY (Gamma et al. 1995). Ela é geralmente motivada pela necessidade de substituir regras diferentes, o que não é necessário aqui, pelo que saibamos. Mas o conceito que estamos tentando captar não se enquadra no *signi-*

ficado de política, que é uma motivação igualmente importante no DDD. (Veja o Capítulo 12 "Relacionando padrões de projeto com o modelo").

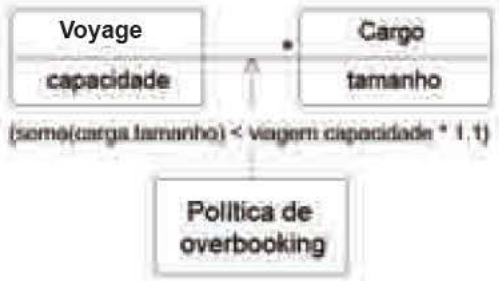

Figura 1.10

Agora, o código fica:

```
public int makeBooking(Cargo cargo, Voyage voyage) {
    if (!overbookingPolicy.isAllowed(cargo, voyage)) return -1;
    int confirmation = orderConfirmationSequence.next();
    voyage.addCargo(cargo, confirmation);
    return confirmation;
}
```

A nova classe **Política de overbooking** contém o seguinte método:

```
public boolean isAllowed(Cargo cargo, Voyage voyage) {
    return (cargo.size() + voyage.bookedCargoSize()) <=
    (voyage.capacity() * 1.1);
}
```

Fica claro para todos que o overbooking é uma política distinta, e a implementação dessa regra é explícita e separada.

No entanto, *não estou recomendando que um design tão elaborado assim seja aplicado a todos os detalhes do domínio.* O Capítulo 15, "Destilação", vai mais a fundo em como se concentrar naquilo que é importante e minimizar ou separar o restante. Este exemplo tem o objetivo de mostrar que um modelo de domínio e o design correspondente podem ser usados para firmar e compartilhar conhecimento. Um design mais explícito apresenta as seguintes vantagens:

1. Para que o design chegue a esse estágio, os programadores e todas as outras pessoas envolvidas terão de entender a natureza do overbooking como uma regra de negócio distinta e importante, e não simplesmente um cálculo obscuro.

2. Os programadores podem mostrar aos especialistas daquele negócio artefatos técnicos, até mesmo códigos, que devem ser inteligíveis para os especialistas do domínio (com um pouco de ajuda), fechando assim o ciclo de feedback.

Modelos profundos

Os modelos mais úteis raramente ficam na superfície. À medida que começamos a entender o domínio e as necessidades do aplicativo, geralmente descartamos elementos de modelos superficiais que pareciam importantes no começo, ou mudamos sua perspectiva. Surgem abstrações sutis que não nos teriam ocorrido no início, mas que vão direto ao coração do problema.

O exemplo anterior está baseado livremente em um dos projetos em que vou me basear para vários exemplos dados durante o livro: um sistema de transporte de cargas em contêineres. Os exemplos dados neste livro serão acessíveis a especialistas que não lidam com transporte de cargas. Mas, em um projeto real, onde o aprendizado contínuo prepara os membros da equipe, os modelos de utilidade e clareza geralmente exigem sofisticação tanto no domínio quanto na técnica de modelagem.

Nesse projeto, pelo fato de o transporte começar com o ato da reserva da carga, desenvolvemos um modelo que nos permitisse descrever a carga, seu itinerário e assim por diante. Tudo isso era necessário e útil, mas mesmo assim os especialistas do domínio se sentiam insatisfeitos. Havia uma forma em que eles enxergavam seu negócio que ainda não havíamos captado.

Por fim, depois de meses de assimilação de conhecimento, percebemos que o manuseio da carga, o carregamento e o descarregamento físico, os deslocamentos de um lugar para outro, eram em grande parte realizados por subempreiteiros ou pessoal operacional da empresa. Na visão dos nossos especialistas em transporte de carga, havia uma série de transferências de responsabilidade entre partes. Um processo governava essa transferência de responsabilidade legal e prática, do transportador (navio) para alguma transportadora local, de uma transportadora para outra, e finalmente para o consignatário. Geralmente, a carga ficava em um armazém enquanto estavam sendo dados outros passos importantes. Outras vezes, a carga passava por etapas físicas complexas que não eram relevantes para as decisões comerciais da empresa de transporte. Em lugar da logística do itinerário, o que assumia a frente eram os documentos legais tais como o reconhecimento de embarque, e processos que levavam à liberação de pagamentos.

Essa visão mais profunda do negócio referente ao transporte de cargas não levou à remoção do objeto Itinerário, mas o modelo mudou profundamente. Nossa visão de transporte de cargas deixou de ser a movimentação de contêineres de um lugar para outro e passou para a transferência de responsabilidades pela carga de entidade em entidade. Os recursos necessários para o controle dessas transferências de responsabilidade já não estavam estranhamente ligados às operações de carregamento, mas se sustentavam por um modelo que surgiu do entendimento da relação significativa entre essas operações e essas responsabilidades.

A assimilação do conhecimento é uma exploração, e é impossível saber onde se pode chegar.