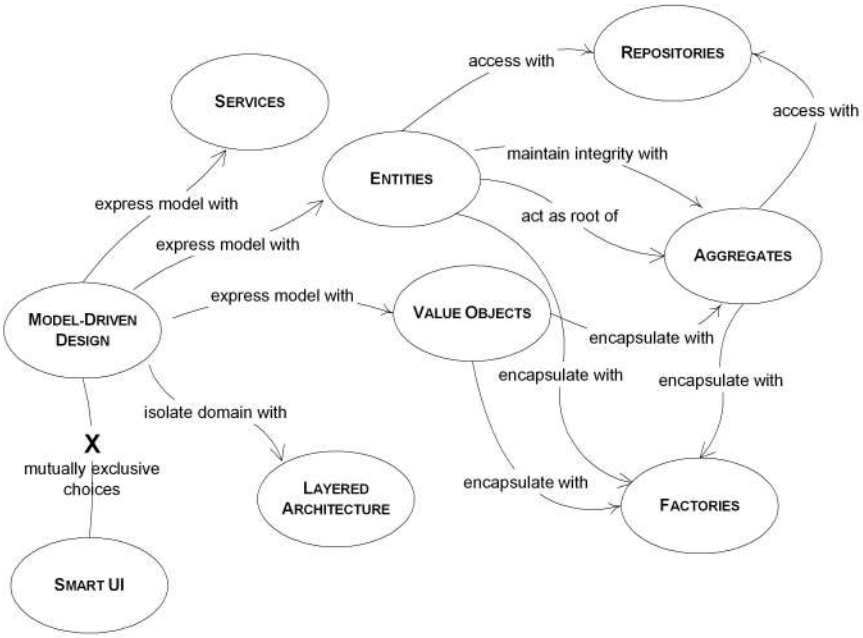
A Summary of Eric Evans' ***Domain-Driven Design***

Domain-Driven Design *Quickly*



by Abel Avram & Floyd Marinescu

edited by: Dan Bergh Johnsson, Vladimir Gitlevich

InfoQ *Enterprise Software Development Series*

LIVRE ONLINE EDIÇÃO

(Versão online gratuita não-imprimíveis)

Se você gosta do livro, por favor apoio
o autor e InfoQ por

comprar o livro impresso:

http://www.lulu.com/content/325231

(Apenas US $ 30.00)

Trazido a você
cortesia de

Este livro é distribuído gratuitamente no InfoQ.com, se você recebeu este livro de qualquer outra fonte, em seguida, por favor, ajudem o autor eo
publisher registrando em InfoQ.com.

Visite a página inicial para este livro em:

http://www.infoq.com/minibooks/domain-
-driven design rapidamente

Domínio-Driven projeto

Rapidamente

C4Media, Publisher da Comunidade de Desenvolvimento de Software Empresa InfoQ.com

Parte da série InfoQ Empresa de Desenvolvimento de Software de livros.

Para obter informações ou solicitar este ou outros livros InfoQ, entre em contato books@c4media.com.

Créditos de produção:
DDD Resumo por: Abel Avram Editor Geral: arte
Floyd Marinescu capa: Gene Steffanson

Composição: Laura Brown e Melissa Tessier especiais graças a
Eric Evans.

Internacionais de Catalogação na Publicação de dados:

ISBN: 978-1-4116-0925-9

Impresso nos Estados Unidos da América

10 9 8 7 6 5 3 1 2

Conteúdo

Prefácio: Por DDD rapidamente?

Eu ouvi pela primeira vez sobre Domain Driven Projeto e conheceu Eric Evans em um pequeno grupo de arquitetos em um cume da montanha organizado por Bruce Eckel, no verão de 2005. A cúpula foi assistido por um número de pessoas que eu respeito, incluindo Martin Fowler, Rod Johnson, Cameron Purdy, Randy Stafford, e Gregor Hohpe.

O grupo parecia bastante impressionado com a visão de Domain Driven Design, e estava ansioso para aprender mais sobre ele. Eu também tenho a sensação de que todo mundo desejava que estes conceitos eram mais mainstream. Quando eu percebi como Eric utilizado o modelo de domínio para discutir soluções para alguns dos vários desafios técnicos, o grupo discutiu, e como muita ênfase que ele colocado no domínio do negócio em vez de hype específicos de tecnologia, eu soube imediatamente que esta visão é aquela que a comunidade extremamente necessário.

Nós, na comunidade de desenvolvimento empresarial, especialmente a comunidade de desenvolvimento web, têm sido manchada por anos de hype que nos levou longe de desenvolvimento de software orientado adequada objeto. Na comunidade Java, boa modelagem de domínio foi perdido no hype de EJB e os contentores / modelos de componentes de 1999-2004. Felizmente, as mudanças na tecnologia e as experiências coletivas da comunidade de desenvolvimento de software estão nos movendo em direção paradigmas tradicionais orientadas a objeto. No entanto, a comunidade está faltando uma visão clara de como aplicar a orientação a objetos em escala empresarial, que é por isso que eu acho DDD é importante.

Infelizmente, fora de um pequeno grupo dos mais arquitetos seniores, percebi que muito poucas pessoas estavam cientes de DDD, razão pela qual InfoQ encomendou a elaboração deste livro.

A minha esperança é que através da publicação de um resumo, resumo rapidamente legível e introdução aos fundamentos de DDD e torná-lo livremente para download na InfoQ com uma versão de impressão barato de bolso, essa visão pode se tornar mainstream.

Este livro faz **não introduzir quaisquer novos conceitos; ele tenta resumir de forma** concisa a essência do que DDD é, atraindo principalmente em livro original Eric Evans' sobre o assunto, bem como outras fontes, uma vez publicados, como Jimmy Nilsson de *aplicando DDD*
e vários fóruns de discussão DDD. O livro vai lhe dar um curso sobre os fundamentos da DDD, mas não é nenhum substituto para os numerosos exemplos e estudos de caso fornecidos no livro de Eric ou o hands-on exemplos fornecidos no livro de Jimmy. Eu altamente incentivá-lo a ler esses dois excelentes trabalhos. Entretanto, se você concorda que a comunidade precisa DDD de fazer parte da nossa consciência de grupo, por favor, que as pessoas saibam sobre este livro e o trabalho de Eric.

Floyd Marinescu

Co-fundador e editor chefe da InfoQ.com

Introdução

Software é um instrumento criado para nos ajudar a lidar com as complexidades da nossa vida moderna. Software é apenas um meio para um fim, e normalmente isso é algo muito prático e real. Por exemplo, podemos usar o software para controle de tráfego aéreo, e isso está diretamente relacionado com o mundo que nos rodeia. Queremos voar de um lugar para outro, e fazemos isso utilizando máquinas sofisticadas, por isso criamos software para coordenar a fuga de milhares de aviões que acontecerá a estar no ar a qualquer momento.

Software tem que ser prático e útil; caso contrário, não iria investir tanto tempo e recursos para a sua criação. Isso torna extremamente ligado a um certo aspecto de nossas vidas. Um pacote útil de software não pode ser dissociado do que esfera da realidade, o domínio é suposto para nos ajudar a gerir. Pelo contrário, o software está profundamente envolvido com ele.

design de software é uma arte, e como qualquer arte que não pode ser ensinado e aprendido como uma ciência exacta, por meio de teoremas e fórmulas. Podemos descobrir princípios e técnicas úteis para ser aplicado em todo o processo de criação de software, mas provavelmente nunca vai ser capaz de fornecer um caminho exato para seguir a partir da necessidade do mundo real para o módulo de código utilizado para servir a essa necessidade. Como uma imagem ou um edifício, um produto de software incluirá o toque pessoal de quem projetou e desenvolveu-lo, algo do carisma e talento (ou a falta dela) dos que contribuíram para a sua criação e crescimento.

Existem maneiras diferentes de abordar design de software. Durante os últimos 20 anos, a indústria de software tem conhecido e utilizado vários métodos para criar seus produtos, cada um com suas vantagens e desvantagens. O propósito deste livro é se concentrar em um projeto

método que surgiu e evoluiu ao longo das últimas duas décadas, mas se cristalizou mais claramente durante os últimos anos: Domain-Driven Design. Eric Evans fez uma grande contribuição para este assunto por escrito para baixo em um livro grande parte do conhecimento acumulado sobre Domain-Driven Design. Para uma apresentação mais detalhada deste tópico, recomendamos a leitura de seu livro “Domain-Driven Edifício: Roubada de Complexidade no coração de Software”, publicado pela Addison-Wesley, ISBN: 0-321-12521-5.

Muitas informações valiosas também pode ser aprendido, seguindo o design grupo de discussão Domain Driven em:

http://groups.yahoo.com/group/domaindrivendesign

Este livro é apenas uma introdução ao tema, destinado a dar-lhe rapidamente uma fundamental, mas não uma compreensão detalhada do Domain Driven projeto. Nós apenas queremos abrir o apetite para um bom design de software com os princípios e diretrizes utilizadas no mundo do Domain-Driven Design.

Versão online grátis.

Apoiar este trabalho, comprar a cópia impressa:
http://www.infoq.com/minibooks/domain-drivendesign-quickly

1

O que é de domínio-Driven projeto

S desenvolvimento oftware é mais frequentemente aplicada a automatização de processos que existem no mundo real, ou fornecimento de soluções para problemas reais de negócios; Os processos de negócio a ser automatizado ou problemas do mundo real que o software é o domínio do software. Devemos entender desde o início que o software é originado a partir e profundamente relacionado com este domínio.

Software é composto de código. Podemos ser tentados a gastar muito tempo com o código e exibir o software simplesmente como objetos e métodos.

Considere fabricação de automóveis como uma metáfora. Os trabalhadores envolvidos na fabricação de automóveis podem se especializar na produção de peças do carro, mas ao fazê-lo muitas vezes eles têm uma visão limitada de todo o processo de fabricação do carro. Eles começar a ver o carro como uma enorme coleção de peças que precisam se encaixam, mas um carro é muito mais do que isso. Um bom carro começa com uma visão. Ela começa com as especificações cuidadosamente escritas. E continua com design. Lotes e lotes do design. Meses, talvez anos de tempo gasto em design, mudando e refiná-lo até que ele atinja a perfeição, até que reflete a visão original. O projeto de processamento não é tudo no papel. Muito do que inclui fazer modelos do carro, e testá-las sob certas condições, para ver se eles funcionam. O projeto é modificado com base nos resultados dos testes. O carro é enviado para a produção, eventualmente,

desenvolvimento de software é semelhante. Nós não podemos apenas sentar e escrever código. Podemos fazer isso, e ele funciona bem para casos triviais. Mas não podemos criar software complexo como aquele.

A fim de criar um bom software, você tem que saber o que o software está em causa. Você não pode criar um sistema de software bancário, a menos que você tem uma boa compreensão do que banking é tudo, é preciso entender o *domínio* da banca.

É possível criar software bancário complexo sem um bom conhecimento de domínio? De jeito nenhum. Nunca. Quem sabe bancário? O arquiteto de software? Não. Ele só usa o banco para manter seu dinheiro seguro e disponível quando ele precisa deles. O analista de software? Na verdade não. Ele sabe analisar um determinado tema, quando ele é dado todos os ingredientes necessários. O desenvolvedor? Esqueça. Quem então? Os banqueiros, é claro. O sistema bancário é muito bem compreendida pelas pessoas dentro, por seus especialistas. Eles sabem todos os detalhes, todas as capturas, todos os possíveis problemas, todas as regras. Este é o lugar onde devemos sempre começar: o domínio.

Quando começamos um projeto de software, devemos concentrar-nos do domínio que está operando dentro. Todo o propósito do software é aumentar a um domínio específico. Para ser capaz de fazer isso, o software tem que se encaixar harmoniosamente com o domínio foi criado para. Caso contrário, ele irá introduzir tensão para o domínio, mau funcionamento de provocar, danos, e até mesmo o caos desabafar.

Como podemos fazer o ajuste software harmoniosamente com o domínio? A melhor maneira de fazer isso é tornar o software um reflexo do domínio. Software precisa incorporar os conceitos e elementos do domínio do núcleo, e precisamente perceber as relações entre eles. Software tem para modelar o domínio.

Alguém sem o conhecimento do sistema bancário deve ser capaz de aprender muito apenas lendo o código em um modelo de domínio. Isto é essencial. Software que não tem suas raízes plantadas profundamente no domínio não reagem bem às mudanças ao longo do tempo.

Então, vamos começar com o domínio. Então o que? Um domínio é algo deste mundo. Ele não pode simplesmente ser tomada e derramado sobre o

teclado no computador para se tornar código. Precisamos criar uma abstração do domínio. Nós aprendemos muito sobre um domínio ao falar com os peritos do domínio. Mas este conhecimento matéria não vai ser facilmente transformado em construções de software, a menos que nós construímos uma abstração dele, um plano em nossas mentes. No início, o projeto é sempre incompleta. Mas com o tempo, enquanto trabalhava nele, nós torná-lo melhor, e torna-se cada vez mais claro para nós. O que é essa abstração? É um modelo, um modelo do domínio. De acordo com Eric Evans, um modelo de domínio não é um diagrama em particular; é a ideia de que o diagrama se pretende transmitir. Não é apenas o conhecimento na cabeça de um especialista de domínio; é uma abstracção rigorosamente organizado e selectiva do que o conhecimento. Um diagrama pode representar e comunicar um modelo, como pode código cuidadosamente escrito,

O modelo é a nossa representação interna do domínio de destino, e é muito necessário em todo o processo de desenvolvimento de design e. Durante o processo de design nos lembramos e fazer muitas referências ao modelo. O mundo em torno de nós é demais para a cabeça de manusear. Mesmo um domínio específico poderia ser mais do que uma mente humana pode manipular ao mesmo tempo. Precisamos organizar a informação, para sistematizá-lo, dividi-lo em pedaços menores, para agrupar as peças em módulos lógicos, e tomar um de cada vez e lidar com ele. Nós ainda precisa deixar algumas partes do fora do domínio. Um domínio contém apenas demasiada informação para incluir tudo no modelo. E muito do que não é sequer necessário para ser considerado. Este é um desafio por si só. O que manter eo que jogar fora? É parte do projeto, o processo de criação de software. O software bancário vai certamente manter o controle de endereço do cliente, mas não deve se preocupar com a cor dos olhos do cliente. Isso é um caso óbvio, mas outros exemplos pode não ser tão óbvio.

Um modelo é uma parte essencial do design de software. Precisamos dele, a fim de ser capaz de lidar com a complexidade. Todo nosso processo de pensamento sobre o domínio é sintetizada a esse modelo. Isso é bom, mas tem que sair da nossa cabeça. Não é muito útil se ele permanece lá, não é? Precisamos comunicar este modelo com especialistas de domínio, com colegas designers, e com os desenvolvedores. o

modelo é a essência do software, mas precisamos criar maneiras de expressá-lo, para se comunicar com os outros. Nós não estamos sozinhos neste processo, por isso precisamos de compartilhar conhecimento e informação, e precisamos fazê-lo bem, precisamente, completamente e sem ambigüidade. Existem diferentes maneiras de fazer isso. Um deles é gráficas: diagramas, casos de uso, desenhos, imagens, etc. Outro está escrevendo. Anotamos a nossa visão sobre o domínio. Outra é a linguagem. Nós podemos e devemos criar uma linguagem para comunicar questões específicas sobre o domínio. Vamos detalhar todos estes mais tarde, mas o ponto principal é que *precisamos nos comunicar o modelo*.

Quando tivermos um modelo expresso, podemos começar a fazer projeto do código. Isso é diferente do design de software. design de software é como criar a arquitetura de uma casa, é sobre a imagem grande. Por outro lado, design código está trabalhando nos detalhes, como a localização de uma pintura em uma determinada parede. projeto do código também é muito importante, mas não tão fundamental como design de software. Um erro de design de código é geralmente mais facilmente corrigido, enquanto os erros de design de software são muito mais caro para consertar. É uma coisa para mover uma pintura mais à esquerda, e uma coisa completamente diferente para derrubar um lado da casa, a fim de fazê-lo de forma diferente. No entanto, o produto final não será bom sem projeto do código bom. Aqui padrões de projeto de código vir a calhar, e eles devem ser aplicadas quando necessário. Boa técnicas de codificação ajuda para criar limpo,

Existem diferentes abordagens para design de software. Um deles é o método de projeto cachoeira. Este método envolve uma série de etapas. Os especialistas em negócios colocar-se um conjunto de requisitos que são comunicados aos analistas de negócios. Os analistas criar um modelo com base nesses requisitos, e passar os resultados para os desenvolvedores, que começam a codificação com base no que eles têm recebido. É um fluxo de uma forma de conhecimento. Embora esta tenha sido uma abordagem tradicional em design de software, e tem sido utilizado com um certo nível de sucesso ao longo dos anos, ele tem suas falhas e limites. O principal problema é que não há feedback dos analistas para os especialistas em negócios ou a partir dos desenvolvedores para os analistas.

Outra abordagem é as metodologias ágeis, como Extreme Programming (XP). Estas metodologias são um movimento coletivo contra a abordagem em cascata, decorrente das dificuldades de tentar chegar a todos os requisitos iniciais, particularmente à luz da mudança de requisitos. É realmente duro para criar um modelo completo que abrange todos os aspectos de um inicial de domínio. Ele tem um monte de pensar, e muitas vezes você simplesmente não consegue ver todas as questões envolvidas desde o início, nem pode prever alguns dos efeitos colaterais negativos ou erros de seu projeto. Outro problema tentativas ágeis para resolver é a chamada “paralisia da análise”, com membros da equipe tanto medo de fazer quaisquer decisões de design que eles fazem nenhum progresso. Embora os defensores Ágeis reconhecem a importância da decisão de projeto, eles resistem projeto inicial.

Os métodos ágeis têm seus próprios problemas e limitações; eles defendem simplicidade, mas todo mundo tem sua própria visão do que

que significa. Além disso, refatoração contínuo feito por desenvolvedores sem princípios de design sólidos irá produzir código que é difícil de entender ou mudar. E enquanto a abordagem em cascata pode levar a overengineering, o medo de overengineering pode levar a outro medo: o medo de fazer uma profunda e cuidadosamente pensada design.

Este livro apresenta os princípios de design orientado domínio, que, quando aplicado pode aumentar significativamente qualquer capacidade processo de desenvolvimento para modelar e implementar os problemas complexos no domínio de forma sustentável. Domain Driven Design Design combina e prática de desenvolvimento, e mostra como design e desenvolvimento podem trabalhar juntos para criar uma solução melhor. Um bom design vai acelerar o desenvolvimento, enquanto o feedback proveniente do processo de desenvolvimento irá melhorar o design.

Construção do Conhecimento Domínio

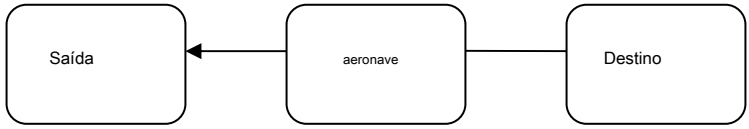
Vamos considerar o exemplo de um projeto de sistema de controle de vôo de avião, e como o conhecimento de domínio pode ser construída.

Milhares de aviões estão no ar em um dado momento em todo o planeta. Eles estão voando seus próprios caminhos para os seus destinos, e é muito importante para se certificar de que eles não colidem no ar. Não vamos tentar elaborar sobre todo o sistema de controle de tráfego, mas em um subconjunto menor que é um sistema de monitoramento de vôo. O projeto proposto é um sistema de monitoramento que acompanha cada vôo sobre uma determinada área, determina se o voo segue sua suposta rota ou não, e se existe a possibilidade de uma colisão.

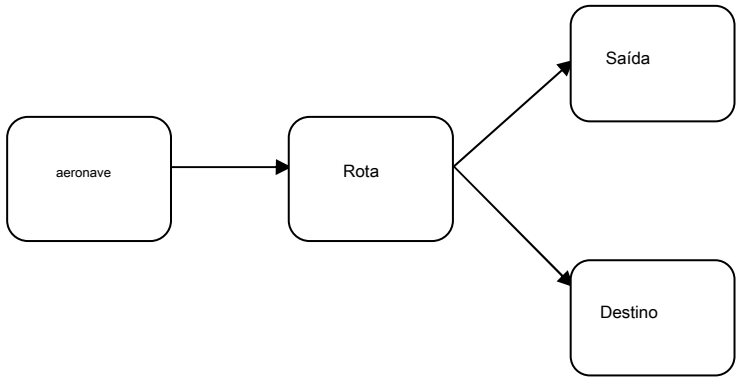
Onde é que vamos começar a partir de uma perspectiva de desenvolvimento de software? Na seção anterior dissemos que deve começar pela compreensão do domínio, que neste caso é o monitoramento do tráfego aéreo. controladores de tráfego aéreo são os especialistas deste domínio. Mas os controladores não são os projetistas de sistemas ou especialistas de software. Você não pode esperar que eles para entregar-lhe uma descrição completa de seu domínio do problema.

Os controladores de tráfego aéreo tem vasto conhecimento sobre o seu domínio, mas, a fim de ser capaz de construir um modelo que você precisa para extrair a informação essencial e generalizá-lo. Quando você começar a falar com eles, você vai ouvir muito sobre aviões decolando e desembarque, aeronaves em pleno ar e do perigo de colisão, aviões de espera antes de serem autorizados a terra, etc. Para encontrar ordem nesta quantidade aparentemente caótico de informações, precisamos começar em algum lugar.

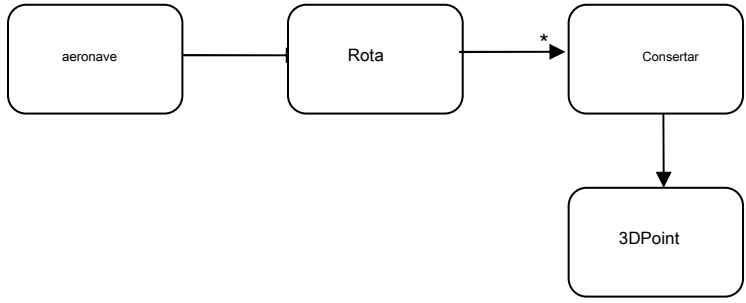
O controlador e você concorda que cada aeronave tem uma partida e um aeródromo de destino. Portanto, temos uma aeronave, uma partida e um destino, como mostrado na figura abaixo.



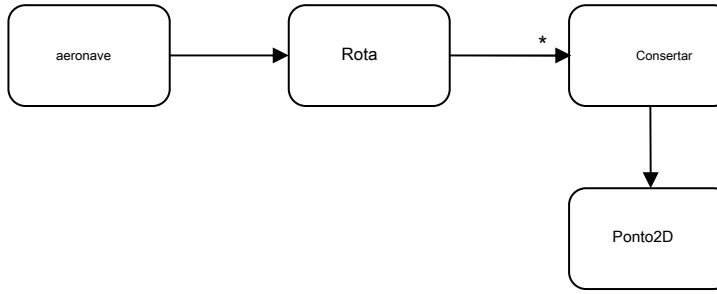
OK, o avião decola de algum lugar e toca baixo em outro. Mas o que acontece no ar? O caminho de voo vai? Na verdade, estamos mais interessados no que acontece enquanto ele está AIRBORN. O controlador diz que cada plano é atribuído um plano de vôo que é suposto para descrever a viagem aérea inteira. Enquanto ouvir sobre um plano de vôo, você pode pensar em sua mente que este é sobre o caminho percorrido pelo avião durante o vôo. Depois de uma discussão mais aprofundada, você ouve uma palavra interessante: rota. Ele imediatamente chama a sua atenção, e por uma boa razão. A rota contém um conceito importante da viagem vôo. Isso é o que os aviões fazer durante o vôo, eles seguem uma rota. É óbvio que a partida e de destino pontos da aeronave são também os pontos inicial e final da rota. Então,



Falando com o controlador sobre as rotas aviões seguem, você descobre que, na verdade, a rota é composta de pequenos segmentos, que juntos constituem uma espécie de linha torta de partida até o destino. A linha é suposto passar através de pontos fixos predeterminados. Assim, uma rota pode ser considerado como uma série de correções consecutivas. Neste ponto, você já não vê a partida e de destino como os pontos terminais da rota, mas apenas mais duas dessas correções. Este é provavelmente muito diferente de como o controlador de vê-los, mas é uma abstração necessário que ajuda mais tarde. As mudanças resultantes com base nestas descobertas são:



O diagrama mostra um outro elemento, o facto de cada solução é um ponto no espaço seguido da rota, e expressa-se como um ponto tridimensional. Mas quando você fala com o controlador, você vai descobrir que ele não vê dessa forma. Na verdade, ele vê a rota como a projeção na terra do vôo de avião. As correções são apenas pontos da superfície da Terra é determinado unicamente pela sua latitude e longitude. Assim, o diagrama correta é:



O que está realmente acontecendo aqui? Você e os especialistas de domínio estão falando, você está trocando conhecimento. Você começar a fazer perguntas, e eles respondem. Enquanto eles fazem isso, eles cavam conceitos essenciais para fora do domínio do tráfego aéreo. Esses conceitos podem sair rude e desorganizado, mas no entanto eles são essenciais para a compreensão do domínio. Você precisa aprender o máximo possível sobre o domínio dos peritos. E, colocando as perguntas certas, e processar a informação da maneira certa, você e os peritos vão começar a esboçar uma visão do domínio, um modelo de domínio. Esta visão não é completa nem correto, mas é o começo que você precisa. Tente descobrir os conceitos essenciais do domínio.

Esta é uma parte importante do design. Normalmente há longas discussões entre arquitetos de software e desenvolvedores e os especialistas de domínio. Os especialistas em software quer extrair conhecimento dos especialistas de domínio, e eles também tem que transformá-lo em uma forma útil. Em algum momento, eles podem querer criar um protótipo para ver como ele funciona até agora. Ao fazer isso eles podem encontrar alguns problemas com seu modelo ou sua abordagem, e pode querer

para mudar o modelo. o

a comunicação não é apenas uma maneira, com os especialistas de domínio para o arquiteto de software e mais para os desenvolvedores. Há também feedback, que ajuda a criar um modelo melhor, e uma compreensão mais clara e mais correta do domínio. Especialistas de domínio sabe sua área de atuação bem, mas eles organizar e usar seus conhecimentos de uma maneira específica, o que nem sempre é o melhor a ser implementado em um sistema de software. A mente analítica do designer de software ajuda a alguns desenterrar dos conceitos-chave do

domínio durante discussões com especialistas de domínio, e também ajudar a construir uma estrutura para discussões futuras, como veremos no próximo capítulo. Nós, os especialistas em software (arquitetos de software e desenvolvedores) e os especialistas de domínio, está criando o modelo do domínio juntos, eo modelo é o lugar onde essas duas áreas de especialização encontram. Isto pode parecer um processo muito demorado, e é, mas é assim que deve ser, porque no final o propósito do software é a de resolver problemas de negócios em um domínio da vida real, por isso tem que combinam perfeitamente com o domínio.

2

The Ubiquitous Idioma

A necessidade de uma linguagem comum

Tele capítulo anterior fez o caso de que é absolutamente necessário desenvolver um modelo do domínio por ter o os especialistas em software trabalho com os especialistas de domínio; Contudo, naquela abordagem geralmente tem algumas dificuldades iniciais devido a uma barreira de comunicação fundamental. Os desenvolvedores têm suas mentes cheias de classes, métodos, algoritmos, padrões, e tendem a sempre fazer um jogo entre um conceito vida real e um artefato de programação. Eles querem ver o que classes de objetos para criar e quais as relações com o modelo entre eles. Eles pensam em termos de herança, polimorfismo, OOP, etc. E eles falam assim o tempo todo. E é normal para eles a fazê-lo. Os desenvolvedores serão desenvolvedores. Mas os especialistas do domínio geralmente não sabe nada sobre nada disso. Eles não têm idéia sobre as bibliotecas de software, frameworks, persistência, em muitos casos, nem mesmo os bancos de dados. Eles sabem sobre a sua área específica de especialização.

No exemplo a monitorização de tráfego aéreo, os especialistas de domínio saber sobre aviões, sobre rotas, altitudes, longitudes e latitudes, eles sabem sobre desvios de rota normal, cerca de trajetórias de avião. E eles falam sobre essas coisas em seu próprio jargão, o que por vezes não é tão simples de seguir por um estranho.

Para superar essa diferença no estilo de comunicação, quando construir o modelo, temos de comunicar à troca de ideias sobre o modelo, sobre os elementos envolvidos no modelo, como nós conectá-los, o que é relevante eo que não é. Comunicação a este nível é fundamental para o sucesso do projeto. Se alguém diz alguma coisa, eo outro não entende ou, pior ainda, entende outra coisa, quais são as chances do projeto ter sucesso?

Um projeto enfrenta sérios problemas quando os membros da equipe não compartilham uma linguagem comum para discutir o domínio. Especialistas de domínio usar seu jargão, enquanto os membros da equipa técnica têm sua própria língua atento para discutir o domínio em termos de design.

A terminologia de discussões do dia-a-dia é desligado da terminologia incorporada no código (em última análise, o produto mais importante de um projecto de software). E até mesmo a mesma pessoa usa uma linguagem diferente na fala e na escrita, de modo que as expressões mais incisivas do domínio muitas vezes surgem de uma forma transitória que nunca é capturado no código ou até mesmo por escrito.

Durante estas sessões de comunicação, tradução é muitas vezes usado para deixar os outros entender o que alguns conceitos são. Os desenvolvedores podem tentar explicar alguns padrões de projeto usando a linguagem de um leigo, e às vezes sem sucesso. Os especialistas do domínio vai se esforçar para levar para casa algumas das suas ideias, provavelmente criando um novo jargão. Durante este sofre de comunicação de processo, e este tipo de tradução não ajuda o processo de construção do conhecimento.

Nós tendemos a usar nossos próprios dialetos durante estas sessões de design, mas nenhum desses dialetos pode ser uma linguagem comum porque nenhum serve as necessidades de todos.

Definitivamente precisamos de falar a mesma língua quando nos encontramos para falar sobre o modelo e para defini-lo. Que língua é que vai ser? linguagem dos desenvolvedores? linguagem dos especialistas do domínio? Algo entre os dois?

Um dos princípios fundamentais de Domain-Driven Design é usar uma linguagem baseada no modelo. Uma vez que o modelo é o terreno comum, o lugar onde o software atenda o domínio, é apropriado usá-lo como o chão de construção para este idioma.

Use o modelo de como a espinha dorsal de uma linguagem. Solicitar que a equipe de usar a linguagem de forma consistente em todas as comunicações, e também no código. Enquanto partilha de conhecimentos e martelar o modelo, a equipe usa a fala, escrita e diagramas. Certifique-se este idioma aparece de forma consistente em todas as formas de comunicação utilizadas pela equipe; por esta razão, a linguagem é chamada de Linguagem Ubíqua.

The Ubiquitous Idioma conecta todas as partes do design, e cria a premissa para a equipe de design para funcionar bem. Leva semanas e até meses para projetos projeto de grande escala a tomar forma. Os membros da equipe descobrem que alguns dos conceitos iniciais estão incorretos ou inadequadamente utilizados, ou eles descobrem novos elementos do projeto que precisam ser considerados e se encaixar no design geral. Tudo isso não é possível sem uma linguagem comum.

Línguas não aparecem durante a noite. É preciso muito trabalho e um monte de foco para se certificar de que os principais elementos da linguagem são trazidos à luz. Precisamos encontrar esses conceitos-chave que definem o domínio eo design, e encontrar palavras correspondentes para eles, e começar a usá-los. Alguns deles são facilmente visualizados, mas alguns são mais difíceis.

Resolver dificuldades experimentando com expressões alternativas, que refletem modelos alternativos. Em seguida, refatorar o código, renomear classes, métodos e módulos de acordo com o novo modelo. Resolve confusão sobre os termos em conversa, apenas na maneira chegamos a acordo sobre o significado das palavras comuns.

Construção de uma linguagem como que tem um resultado claro: o modelo ea linguagem são fortemente interligados uns com os outros. A mudança na linguagem deve tornar-se uma mudança para o modelo.

Especialistas de domínio deve opor-se a termos ou estruturas que são difíceis ou inadequadas para transmitir conhecimento de domínio. Se os especialistas de domínio não pode entender alguma coisa no modelo ou a linguagem, então é mais provável que haja algo de errado com ele. Por outro lado, os desenvolvedores devem estar atentos para a ambigüidade ou inconsistência que tendem a aparecer no design.

Criando a Linguagem Ubíqua

Como podemos começar a construir uma linguagem? Aqui está um diálogo hipotético entre um desenvolvedor de software e um especialista de domínio no projeto de monitoramento de tráfego aéreo. Atente para as palavras que aparecem em negrito.

Desenvolvedor: Queremos monitorar o tráfego aéreo. Por onde começamos?

Especialista: Vamos começar com o básico. Todo este tráfego é composta de **aviões.** Cada avião decola de um **saída** lugar, e terras em um **destino** Lugar, colocar.

Desenvolvedor: Isso é fácil. Quando se voa, o avião pode simplesmente escolher qualquer caminho de ar dos pilotos gosta? É até eles para decidir qual caminho que devem seguir, enquanto eles chegar ao destino?

Especialista: Ah não. Os pilotos recebem um **rota** eles devem seguir. E eles devem ficar nessa rota mais próximo possível.

Desenvolvedor: Estou pensando desta **rota** como um caminho 3D no ar. Se usarmos um sistema cartesiano de coordenadas, então o **rota** é simplesmente uma série de pontos 3D.

Especialista: Acho que não. Nós não vemos **rota** dessa maneira. o **rota** é, na verdade, a projecção no solo do percurso de ar esperada do avião. o **rota** passa por uma série de pontos no solo determinada pela sua **latitude** e **longitude.**

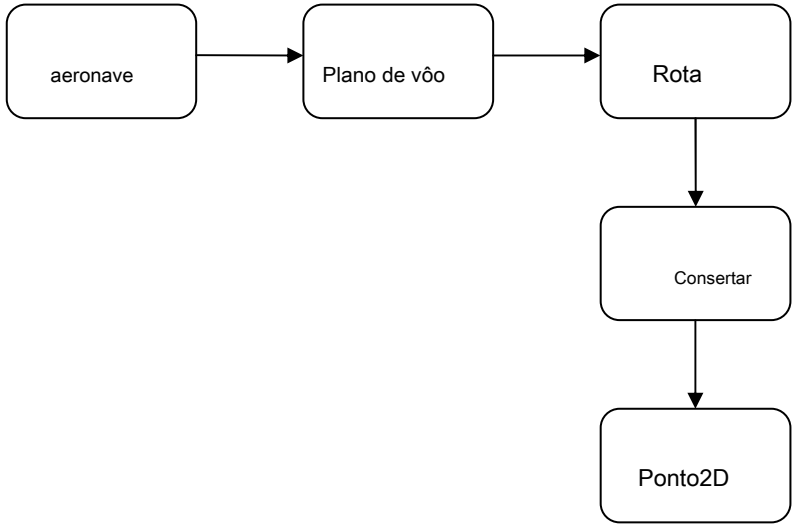
Desenvolvedor: OK, então vamos chamar cada um desses pontos um **consertar,** porque é um ponto fixo da superfície da Terra. E nós vamos usar, em seguida, uma série de pontos 2D para descrever o caminho. E, a propósito, o **saída e destino** são apenas **Conserta.** Não devemos considerá-los como conceitos separados. o **rota** chega ao destino, uma vez que atinge qualquer outro **consertar.** O avião deve seguir a rota, mas isso significa que ele pode voar tão alto ou tão baixo como ele gosta?

Especialista: Não. O **altitude** que um avião é ter em um determinado momento, também está estabelecido no **plano de vôo.**

Desenvolvedor: plano de vôo? O que é isso?

Especialista: Antes de deixar o aeroporto, os pilotos recebem uma detalhada **plano de vôo** que inclui todos os tipos de informações sobre o **voar: a rota,** cruzeiro **altitude,** o Cruzeiro **Rapidez,** o tipo de **avião,** até mesmo informações sobre os membros da tripulação.

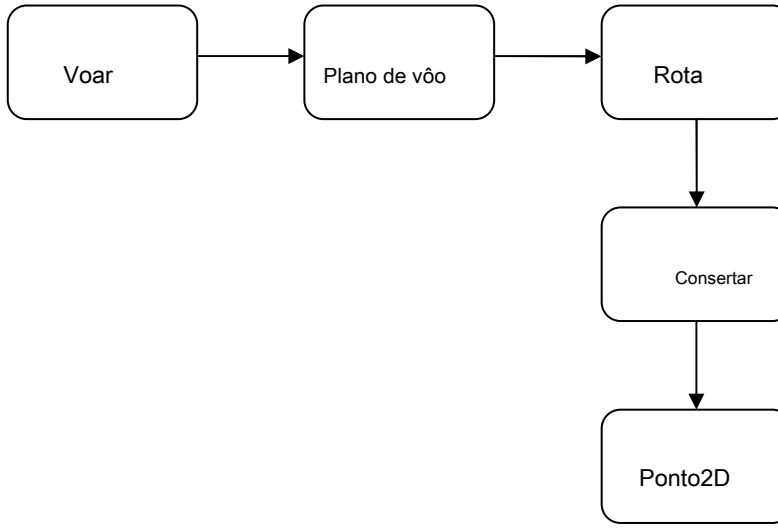
Desenvolvedor: Hmm, o **plano de vôo** parece bastante importante para mim. Vamos incluí-lo no modelo.



Desenvolvedor: Isso é melhor. Agora que eu estou olhando para ele, percebo alguma coisa. Quando estamos monitorando o tráfego aéreo, não estamos realmente interessados nos planos si mesmos, se eles são brancos ou azul, ou se eles são a Boeing ou a Airbus. Estamos interessados em sua
voar. Isso é o que estamos realmente seguindo e de medição. Acho que devemos mudar o modelo um pouco para ser mais preciso.

Observe como esta equipe, falando sobre o tráfego aéreo monitoramento de domínio e em torno de seu modelo incipiente, está lentamente criando uma linguagem composta pelas palavras em negrito. Observe também como que a linguagem muda o modelo!

No entanto, na vida real tal diálogo é muito mais detalhado, e as pessoas muitas vezes falam sobre coisas indiretamente, ou entrar em muitos detalhes, ou escolher os conceitos errados; isso pode fazer a subir com a língua muito difícil. Para começar a resolver isso, todos os membros da equipe devem estar cientes da necessidade de criar uma linguagem comum e deve ser lembrado para manter o foco no que é essencial, e usar a linguagem sempre que necessário. Devemos usar nosso próprio jargão durante essas sessões o mínimo possível, e devemos usar a Linguagem Ubíqua, porque isso nos ajuda a comunicar de forma clara e precisa.



Também é altamente recomendado para os desenvolvedores para implementar os principais conceitos do modelo no código. Uma classe pode ser escrito para Route e outro para Fix. A classe Fix poderia herdar de uma classe Ponto2D, ou pode conter um Ponto2D como seu atributo principal. Isso depende de outros fatores que serão discutidos mais tarde. Ao criar classes para os conceitos do modelo correspondentes, estamos mapeamento entre o modelo e o código e entre a linguagem eo código. Isto é muito útil, pois torna o código mais legível, e torna reproduzir o modelo. Ter o código de expressar o modelo compensa mais tarde no projeto, quando o modelo cresce grande, e quando mudanças no código pode ter consequências indesejáveis se o código não foi devidamente projetado.

Vimos como a linguagem é compartilhada por toda a equipe, e também como ele ajuda a construir o conhecimento e criar o modelo. O que devemos usar para a linguagem? discurso justo? Temos diagramas usados. O quê mais? Escrevendo?

Alguns podem dizer que UML é bom o suficiente para construir um modelo em cima. E de fato é uma grande ferramenta para anotar os principais conceitos como classes, e para expressar relações entre eles. Você pode desenhar quatro ou cinco classes em um bloco de notas, anote sua

nomes, e mostrar as relações entre eles. É muito fácil para que todos possam acompanhar o que você está pensando, e uma expressão gráfica de uma idéia é fácil de entender. Todos compartilham instantaneamente a mesma visão sobre um determinado tópico, e torna-se mais simples para se comunicar com base nisso. Quando novas idéias surgem, eo diagrama é modificada para refletir a mudança conceitual.

diagramas UML são muito úteis quando o número de elementos envolvidos é pequena. Mas a UML pode crescer como cogumelos depois de uma chuva de verão agradável. O que você faz quando você tem centenas de aulas de encher uma folha de papel, enquanto Mississippi? É difícil de ler, mesmo pelos especialistas em software, não para especialistas em domínio mencionar. Eles não vão entender muito do que quando fica grande, e fá-lo mesmo para projetos de médio porte.

Além disso, UML é bom em aulas expressando, seus atributos e relações entre eles. Mas o comportamento das classes e as restrições não são expressas com tanta facilidade. Para que os resorts UML para texto colocados como notas no diagrama. Então UML não pode transmitir dois aspectos importantes de um modelo: o significado dos conceitos que ela representa e que os objetos são suposto fazer. Mas isso é OK, desde que nós podemos adicionar outras ferramentas de comunicação para fazê-lo.

Podemos usar documentos. Uma maneira conveniente de comunicar o modelo é fazer algumas pequenas diagramas cada um contendo um subconjunto do modelo. Esses diagramas iria conter várias classes, e a relação entre eles. Que já inclui uma parte boa dos conceitos envolvidos. Então, podemos adicionar texto ao diagrama. O texto irá explicar o comportamento e as restrições que o diagrama não pode. Cada tais tentativas subseção para explicar um aspecto importante do domínio, ele aponta um “destaque” para iluminar uma parte do domínio.

Esses documentos podem ser ainda desenhado à mão, porque que transmite a sensação de que eles são temporários, e pode ser alterado no futuro próximo, o que é verdade, porque o modelo é alterado muitas vezes no início, antes que ele atinja um estado mais estável.

Pode ser tentador para tentar criar um grande diagrama ao longo de todo o modelo. No entanto, na maioria das vezes esses diagramas são quase impossíveis de colocar juntos. E além disso, mesmo se você conseguir fazer esse diagrama unificada, será tão confuso que não vai transmitir a compreender melhor então que o conjunto de pequenos diagramas.

Desconfie de documentos longos. Leva muito tempo para escrevê-los, e eles podem se tornar obsoletos antes que eles estão acabados. Os documentos devem estar em sincronia com o modelo. documentos antigos, que utilizam o idioma errado e não refletem o modelo não são muito úteis. Tente evitá-los quando possível.

Também é possível se comunicar usando o código. Esta abordagem é amplamente defendida pela comunidade XP. Bem código escrito pode ser muito comunicativa. Embora o comportamento expresso através de um método é claro, é o nome do método tão claro como o corpo? Afirmações de falar teste para si, mas como sobre os nomes de variáveis e estrutura geral do código? eles estão contando a história toda, alto e bom som? Código, que faz funcionalmente a coisa certa, não expressam necessariamente a coisa certa. Escrevendo um modelo em código é muito difícil.

Há outras maneiras de se comunicar durante o projeto. Não é o propósito deste livro para apresentar todos eles. Uma coisa não deixa de ser claro: a equipe de design, composta por arquitetos de software, desenvolvedores e especialistas de domínio, precisa de uma linguagem que unifica suas ações, e ajuda-los a criar um modelo e expressar esse modelo com o código.

Versão online grátis.

Apoiar este trabalho, comprar a cópia impressa:
http://www.infoq.com/minibooks/domain-drivendesign-quickly

3

MODEL- D despedaçado D ESIGN

Tele capítulos anteriores sublinharam a importância de uma abordagem de desenvolvimento de software que está centrada no domínio do negócio. Nós disse que é de fundamental importância para criar um modelo que está profundamente enraizado no domínio, e deve refletir os conceitos essenciais do domínio com grande precisão. The Ubiquitous Idioma devem ser inteiramente exercidos durante todo o processo de modelagem, a fim de facilitar a comunicação entre os especialistas em software e os especialistas de domínio, e descobrir conceitos do domínio-chave que devem ser utilizados no modelo. O objetivo deste processo de modelagem é criar um modelo de bom. O próximo passo é implementar o modelo em código. Esta é uma fase igualmente importante do processo de desenvolvimento de software. Tendo criado um grande modelo, mas não para transferi-lo corretamente em código vai acabar em software de qualidade questionável.

Acontece que os analistas de software trabalham com especialistas do domínio de negócio há meses, descobrir os elementos fundamentais do domínio, enfatizar os relationshipbs entre eles, e criar um modelo correto, que capta com precisão o domínio. Em seguida, o modelo é repassado para os desenvolvedores de software. Os desenvolvedores podem olhar para o modelo e descobrir que alguns dos conceitos ou relações encontradas nele não podem ser adequadamente expressas em código. Então eles usam o modelo como a fonte original de inspiração, mas eles criam seu próprio projeto que toma emprestado algumas das idéias a partir do modelo, e adiciona alguns dos seus próprios. O processo de desenvolvimento continua ainda mais, e mais aulas são adicionados ao código, ampliando o fosso entre o modelo original eo

implementação final. O bom resultado final não está garantida. Bons desenvolvedores pode reunir um produto que funciona, mas será que vai suportar os ensaios de tempo? Será que vai ser facilmente extensível? Será que vai ser fácil manutenção?

Qualquer domínio pode ser expressa com muitos modelos, e qualquer modelo pode ser expresso de várias maneiras em código. Para cada problema particular não pode haver mais de uma solução. Qual deles é que vamos escolher? Ter um modelo analiticamente correta não significa que o modelo pode ser directamente expressos em código. Ou talvez a sua implementação vai quebrar alguns princípios de design de software, o que não é aconselhável. É importante escolher um modelo que pode ser facilmente e com precisão colocar em código. A questão básica aqui é: como é que vamos abordar a transição de modelo para código?

Uma das técnicas de design recomendado é o chamado *modelo de análise,* que é visto como separado do projeto do código e normalmente é feito por pessoas diferentes. O modelo de análise é o resultado da análise de domínio de negócio, resultando em um modelo que não tem nenhuma consideração para o software usado para a implementação. modelo tal é usado para entender o domínio. Um certo nível de conhecimento é construído, eo modelo resultante pode ser analiticamente correta. Software não é levado em conta nesta fase, pois é considerado um fator de confusão. Este modelo atinge os desenvolvedores que são supostamente para fazer o projeto. Como o modelo não foi construído com princípios de design em mente, ele provavelmente não vai servir bem o efeito. Os desenvolvedores terão que adaptá-lo, ou para criar um projeto separado. E já não há um mapeamento entre o modelo eo código. O resultado é que os modelos de análise são logo abandonado depois de codificação começa.

Um dos principais problemas com esta abordagem é que os analistas não se pode prever alguns dos defeitos de seu modelo, e todos os meandros do domínio. Os analistas podem ter ido em muitos detalhes com alguns dos componentes do modelo, e não ter detalhado o suficiente outros. detalhes muito importantes são descobertos durante o processo de concepção e implementação. Um modelo que é verdadeiro para o domínio poderá vir a ter sérios problemas com persistência objeto ou comportamento desempenho inaceitável.

Os desenvolvedores serão obrigados a tomar algumas decisões por conta própria, e vai fazer alterações de design, a fim de resolver um problema real que não foi considerado quando o modelo foi criado. Eles criam um design que foge do modelo, tornando-o menos relevante.

Se os analistas trabalhar de forma independente, eles acabarão por criar um modelo. Quando este modelo é passado para os designers, alguns dos conhecimentos dos analistas sobre o domínio eo modelo está perdido. Enquanto o modelo pode ser expresso em diagramas e escrita, as chances são os designers não vai entender todo o significado do modelo, ou as relações entre alguns objetos, ou seu comportamento. Há detalhes em um modelo que não são facilmente expressos em um diagrama, e não podem ser totalmente apresentado mesmo por escrito. Os desenvolvedores terão um tempo difícil descobrir-los. Em alguns casos eles vão fazer algumas suposições sobre o comportamento desejado, e é possível para que façam os errados, resultando em mau funcionamento do programa.

Os analistas têm suas próprias reuniões fechadas onde muitas coisas são discutidas sobre o domínio, e há um monte de partilha de conhecimentos. Eles criam um modelo que deve conter todas as informações em uma forma condensada, e os desenvolvedores têm que assimilar tudo isso através da leitura dos documentos fornecidos a eles. Seria muito mais produtivo se os desenvolvedores poderiam juntar-se as reuniões de analistas e ter, assim, alcançar uma visão clara e completa do domínio e o modelo antes de começar a projetar o código.

A melhor abordagem é relacionar intimamente modelagem e design de domínio. O modelo deve ser construído com um aberto olho para as considerações de software e de design. Os desenvolvedores devem ser incluídos no processo de modelagem. A idéia principal é escolher um modelo que pode ser expressa adequadamente em software, para que o processo de design é simples e baseada no modelo. Firmemente relacionando o código para um modelo subjacente dá o significado de código e torna o modelo relevante.

Obtendo os desenvolvedores envolvidos fornece feedback. Ele garante que o modelo pode ser implementado em software. E se

algo está errado, ela é identificada em um estágio inicial, e que o problema pode ser facilmente corrigido.

Aqueles que escrevem o código deve saber o modelo muito bem, e deve sentir-se responsável pela sua integridade. Eles devem perceber que uma alteração ao código implica uma mudança para o modelo; caso contrário eles vão refatorar o código para o ponto em que já não expressa o modelo original. Se o analista é separado do processo de implementação, ele logo vai perder a sua preocupação com as limitações introduzidas pelo desenvolvimento. O resultado é um modelo que não é prático.

Qualquer pessoa técnica que contribua para o modelo deve passar algum tempo tocar no código, qualquer que seja papel primordial que ele ou ela desempenha no projeto. Qualquer pessoa responsável por alterar o código deve aprender a expressar um modelo através do código. Cada desenvolvedor deve ser envolvido em algum nível de discussão sobre o modelo e ter contato com especialistas de domínio. Aqueles que contribuem de diferentes maneiras deve conscientemente envolver aqueles que tocar o código em uma troca dinâmica de ideias modelo através da Linguagem Ubíqua.

Se o design, ou algum parte central do mesmo, não mapeia para o modelo de domínio, esse modelo é de pouco valor, e da exactidão do software é suspeito. Ao mesmo tempo, os mapeamentos complexos entre os modelos e funções de design são difíceis de entender e, na prática, impossível de manter, como as alterações de design. A divisão mortal abre entre análise e projeto de modo que conhecimento adquirido em cada uma dessas atividades não alimentar para o outro.

Projetar uma parte do sistema de software para refletir o modelo de domínio de uma maneira muito literal, de modo que o mapeamento é óbvia. Revisitar o modelo e modificá-lo para ser implementado de forma mais natural em software, mesmo que você procurar para que reflita uma visão mais profunda do domínio. Exigir um modelo único que serve ambas as finalidades bem, para além de suportar um fluente ubíquos Língua.

Desenhe a partir do modelo da terminologia utilizada na concepção e a atribuição básica de responsabilidades. O código torna-se uma expressão do modelo, de modo que uma alteração no código pode ser um

mudar para o modelo. Seu efeito deve ondulação através do resto das atividades do projeto em conformidade.

Para amarrar firmemente a implementação de um modelo geralmente requer ferramentas de desenvolvimento de software e linguagens que suportam um paradigma de modelagem, tais como programação orientada a objeto.

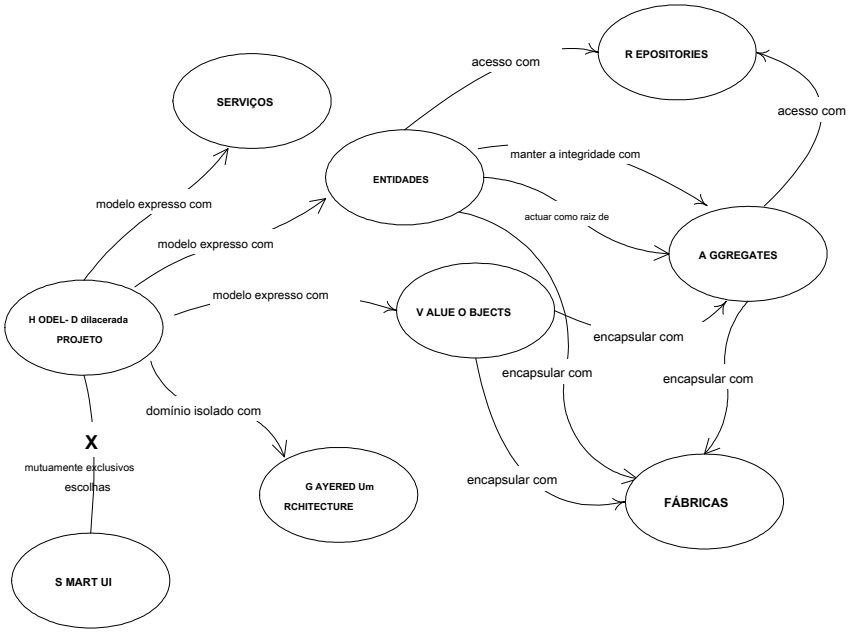
Programação orientada a objetos é adequado para o modelo implementação, porque eles são ambos baseados no mesmo paradigma. programação orientada por objectos fornece as classes de objectos e associações de classes, as instâncias de objectos, e de mensagens entre eles. linguagens OOP tornam possível criar mapeamentos diretos entre o modelo objetos com seus relacionamentos, e os seus homólogos de programação.

linguagens procedurais oferecer suporte limitado para o projeto orientado a modelos. Tais linguagens não oferecem as construções necessárias para implementar componentes-chave de um modelo. Alguns dizem que OOP pode ser feito com uma linguagem procedural, como C, e de fato, algumas das funcionalidades pode ser reproduzido assim. Os objectos podem ser simulado como estruturas de dados. Tais estruturas não contêm o comportamento do objecto, e que tem de ser adicionada separadamente como funções. O significado de tais dados só existe na mente de desenvolvedor, porque o código em si não é explícita. Um programa escrito em uma linguagem procedural é geralmente percebido como um conjunto de funções, um chamando o outro, e trabalhar em conjunto para alcançar um determinado resultado. Tal programa não pode facilmente encapsular conexões conceituais, fazendo mapeamento entre domínio e código difícil de ser realizado.

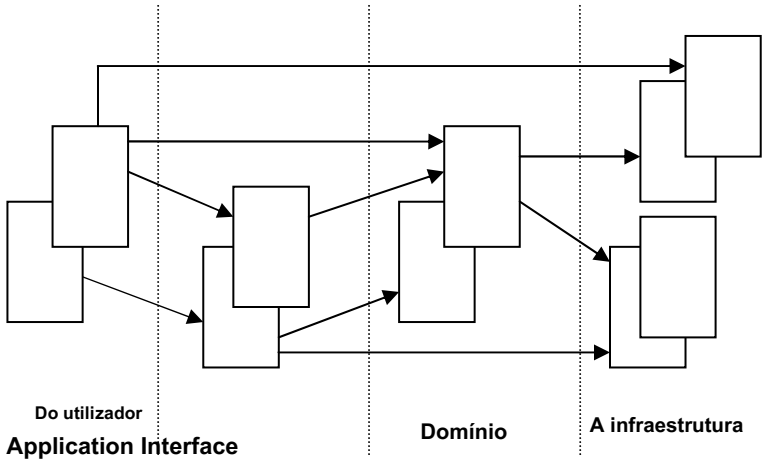
Alguns domínios específicos, como a matemática, pode ser facilmente modelado e implementado usando programação procedural, porque muitas teorias matemáticas são simplesmente endereçado usando chamadas de função e estruturas de dados, porque é principalmente sobre cálculos. domínios mais complexos não são apenas um conjunto de conceitos abstratos envolvendo cálculos, e não pode ser reduzido a um conjunto de algoritmos, linguagens tão processuais ficam aquém da tarefa de expressar os respectivos modelos. Por essa razão, a programação processual não é recomendado para o projeto orientado a modelos.

Os blocos de construção de Design Um Model-Driven

As seções a seguir deste capítulo irá apresentar os padrões mais importantes para ser usado no design orientado a modelos. O objetivo desses padrões é apresentar alguns dos elementos-chave da modelagem de objetos e design de software do ponto de vista de Domain-Driven Design. O diagrama a seguir é um mapa dos padrões apresentado e a relações entre eles.



Layered Architecture



Quando criamos um aplicativo de software, uma grande parte da aplicação não está directamente relacionada com o domínio, mas é parte da infra-estrutura ou serve o software em si. É possível e ok para a parte de domínio de um aplicativo para ser muito pequeno em comparação com o resto, já que uma aplicação típica contém um monte de código relacionado ao acesso de dados, arquivo ou rede de acesso, interfaces de usuário, etc.

Em um programa orientado a objetos, UI, banco de dados, e outro código de apoio muitas vezes é escrita diretamente para os objetos de negócios. lógica de negócios adicional é incorporado no comportamento de widgets de interface do usuário e scripts do banco de dados. Isto algumas vezes acontece porque é a maneira mais fácil de fazer as coisas funcionarem rapidamente.

No entanto, quando código de domínio relacionado é misturada com as outras camadas, torna-se extremamente difícil de ver e pensar. mudanças superficiais na interface do usuário pode realmente mudar a lógica de negócios. Para alterar uma regra de negócio pode exigir o rastreamento minucioso de código de interface do usuário, código de banco de dados ou outros elementos do programa. Implementando objetos coerentes, baseadas em modelos se torna impraticável. Teste automatizado é incómodo. Com todas as tecnologias e lógica envolvidos

em cada atividade, um programa deve ser mantido muito simples ou torna-se impossível de entender.

Portanto, particionar um programa complexo em camadas. Desenvolver um projeto dentro de cada camada que é coesa e que depende apenas das camadas abaixo. Siga padrões de arquitetura padrão para fornecer baixo acoplamento para as camadas acima. Concentra-se todo o código relacionado com o modelo de domínio de uma camada e isolá-lo a partir da interface do utilizador, aplicação, e o código de infra-estrutura. Os objetos de domínio, sem a responsabilidade de exibir-se, armazenar-se, gerenciamento de tarefas de aplicação, e assim por diante, pode ser focado em expressar o modelo de domínio. Isso permite que um modelo de evoluir para ser o bastante rico e suficientemente clara para capturar conhecimento do negócio essencial e colocá-lo para o trabalho.

A solução arquitectónica comum para projetos-driven de domínio contêm quatro co camadas nceptual:

User Interface (camada de apresentação)	Responsável por apresentar informações para o usuário e interpretar os comandos do usuário.
Camada de aplicação	Esta é uma camada fina que coordena a actividade da aplicação. Ele não contém lógica de negócios. Não segure o estado dos objetos de negócios, mas ele pode armazenar o estado de um progresso da tarefa de aplicação.
Camada de domínio	Esta camada contém informações sobre o domínio. Este é o coração do software de negócios. O estado de objetos de negócios é realizado aqui. Persistência dos objetos de negócios e, possivelmente, seu estado é delegada para a camada de infra-estrutura.
Camada de infra-estrutura	Esta camada actua como uma biblioteca de suporte em todas as outras camadas. Ele fornece comunicação entre as camadas, implementos de persistência para objectos de negócios, contém bibliotecas de suporte para a camada de interface de utilizador, etc.

É importante para dividir uma aplicação em camadas separadas, e estabelecer regras de interacções entre as camadas. Se o código não for claramente separados em camadas, que em breve se tornará tão emaranhados que se torna muito difícil de gerir mudanças. Uma simples mudança de uma seção do código pode ter resultados inesperados e indesejáveis em outras seções. A camada de domínio

deve ser focado em questões de domínio do núcleo. Não deve ser envolvido em atividades de infra-estrutura. A interface do usuário não deve nem ser firmemente ligado à lógica de negócios, nem para as tarefas que normalmente pertencem à camada de infra-estrutura. Uma camada de aplicação é necessária em muitos casos. Tem que haver um gerente sobre a lógica de negócios que supervisiona e coordena a actividade global da aplicação.

Por exemplo, uma interação típica da aplicação, domínio e infra-estrutura poderia ser assim. O usuário quer reservar uma rota de vôos, e pede um serviço de aplicação na camada de aplicação para fazê-lo. A camada de aplicativo obtém os objetos de domínio relevantes da infra-estrutura e invoca métodos relevantes sobre eles, por exemplo, para verificar as margens de segurança para outros voos já reservados. Uma vez que os objetos de domínio fizeram todas as verificações e atualiza o seu status para “decidiu”, o serviço de aplicação persistir os objetos para a infra-estrutura.

Entidades

Há uma categoria de objetos que parecem ter uma identidade, que permanece o mesmo durante todo os estados do software. Para estes objetos não são os atributos que importa, mas um fio de continuidade e identidade, que abrange a vida de um sistema e pode se estender para além dela. Tais objetos são chamados de Entidades

linguagens OOP manter instâncias de objetos na memória, e eles associam uma referência ou um endereço de memória para cada objeto. Esta referência é única para cada objeto em um determinado momento do tempo, mas não há nenhuma garantia de que ele vai ficar assim por um período indefinido de tempo. Na verdade, o contrário é verdadeiro. Os objetos são constantemente deslocado para fora e volta para a memória, eles são serializados e enviados pela rede e recriado na outra extremidade, ou eles são destruídos. Esta referência, que se destaca como uma identidade para o ambiente de execução do programa, não é a identidade que estamos a falar. Se existe uma classe que detém tempo

informações, como a temperatura, é perfeitamente possível ter dois casos distintos da classe respectiva, ambas contendo o mesmo valor. Os objetos são perfeitamente iguais e intercambiáveis uns com os outros, mas eles têm diferentes referências. Eles não são entidades.

Se fôssemos para implementar o conceito de uma pessoa usando um programa de software, provavelmente criar uma classe Pessoa com uma série de atributos: nome, data de nascimento, local de nascimento, etc, são qualquer um desses atributos a identidade da pessoa ? O nome não pode ser a identidade porque pode haver mais pessoas com o mesmo nome. Não podia distinguir entre a pessoas com o mesmo nome, se fôssemos levar em conta apenas o seu nome. Nós não podemos usar a data de nascimento também, porque há muitas pessoas nascidas no mesmo dia. O mesmo se aplica para o lugar de nascimento. Um objeto deve ser distinguido de outros objetos mesmo que eles possam ter os mesmos atributos. confusão de identidade pode levar a corrupção de dados.

Considere um sytem contabilidade bancária. Cada conta tem o seu próprio número. Uma conta pode ser precisamente identificado pelo seu número. Este número é mantido durante toda a vida do sistema e garante a continuidade. O número da conta pode existir como um objeto na memória, ou pode ser destruído na memória e enviados para o banco de dados. Ele também pode ser arquivado quando a conta está fechada, mas ainda existe em algum lugar, desde que haja algum interesse em mantê-lo ao redor. Não importa o que a representação é preciso, o número permanece o mesmo.

Portanto, entidades de implementação em software significa a criação de identidade. Para uma pessoa que pode ser uma combinação de atributos: nome, data de nascimento, local de nascimento, nome dos pais, endereço atual. O número da Segurança Social também é usado nos EUA para criar identidade. Para uma conta bancária no número de conta parece ser suficiente para a sua identidade. Normalmente, a identidade ou é um atributo do objecto, uma combinação de atributos, um atributo especialmente criado para preservar e expressar a identidade, ou mesmo um comportamento. É importante para os dois objetos com diferentes identidades para ser para ser facilmente distinguidos pelo sistema, e dois objetos com o

mesma identidade a ser considerado o mesmo pelo sistema. Se essa condição não for atendida, em seguida, todo o sistema pode ser corrompido.

Existem maneiras diferentes de criar uma identidade única para cada objeto. O ID pode ser gerado automaticamente por um módulo, e usado internamente no software sem torná-lo visível para o usuário. Pode ser uma chave primária em uma tabela de banco de dados, o que é garantido que ser exclusivo no banco de dados. Sempre que o objeto é recuperado do banco de dados, o seu ID é recuperada e recriado na memória. O ID pode ser criado pelo usuário, como acontece com os códigos associados aos aeroportos. Cada aeroporto tem um ID de string único, que é reconhecido internacionalmente e utilizada pelas agências de viagens em todo o mundo para identificar os aeroportos em seus horários de viagem. Outra solução é usar os atributos do objeto para criar o ID, e quando isso não for suficiente, um outro atributo pode ser adicionado para ajudar a identificar o respectivo objeto.

Quando um objeto se distingue pela sua identidade, ao invés de seus atributos, fazem deste primário para a sua definição no modelo. Mantenha a simples definição de classe e focada na vida de continuidade do ciclo e identidade. Definir um meio de distinguir cada objeto independentemente da sua forma ou a história. Esteja atento aos requisitos que a chamada para a correspondência de objetos por atributos. Definir uma operação que é garantido para produzir um resultado único para cada objeto, possivelmente, anexando um símbolo que é garantido único. Este meio de identificação pode vir do lado de fora, ou pode ser um identificador arbitrário criado por e para o sistema, mas devem corresponder aos distinções de identidade no modelo. O modelo deve definir o que significa ser a mesma coisa.

Entidades são objetos importantes de um modelo de domínio, e eles devem ser considerados desde o início do processo de modelagem. Também é importante para determinar se um objeto precisa ser uma entidade ou não, o que é discutido no próximo padrão.

Valor Objects

Temos discutido entidades e a importância de reconhecer entidades início durante a fase de modelagem. Entidades são objetos necessários em um modelo de domínio. Devemos fazer todos os objetos entidades? Deve cada objeto tem uma identidade?

Podemos ser tentados a fazer todos os objetos entidades. As entidades podem ser rastreadas. Mas rastreamento e criação de identidade vem com um custo. Precisamos ter certeza de que cada instância tem a sua identidade única, e acompanhamento de identidade não é muito simples. Ele tem um monte de pensamento cuidadoso para decidir o que faz com que uma identidade, porque uma decisão errada levaria a objetos com a mesma identidade, algo que não é desejado. Há também implicações de desempenho em fazer todos os objetos entidades. Tem de haver um exemplo para cada objecto. Se o Cliente é um objeto de entidade, em seguida, uma instância desse objeto, o que representa um cliente bancária específica, não pode ser reutilizado para operações de conta correspondentes a outros clientes. O resultado é que tal uma instância tem que ser criado para cada cliente.

Vamos considerar um aplicativo de desenho. O usuário é apresentado uma tela e ele pode tirar quaisquer pontos e linhas de qualquer espessura, estilo e cor. É útil para criar uma classe de objeto chamado Point, eo programa poderia criar uma instância dessa classe para cada ponto na tela. ponto Tal deverá conter dois atributos associados a tela ou lona coordenadas. É necessário considerar cada ponto como tendo uma identidade? Será que ela tem continuidade? Parece que a única coisa que importa para tal objeto é suas coordenadas.

Há casos em que precisamos conter alguns atributos de um elemento de domínio. Nós não estamos interessados em que objeto é, mas o que os atributos que tem. Um objeto que é usado para descrever certos aspectos de um domínio, e que não tem identidade, é nomeado valor dos objetos.

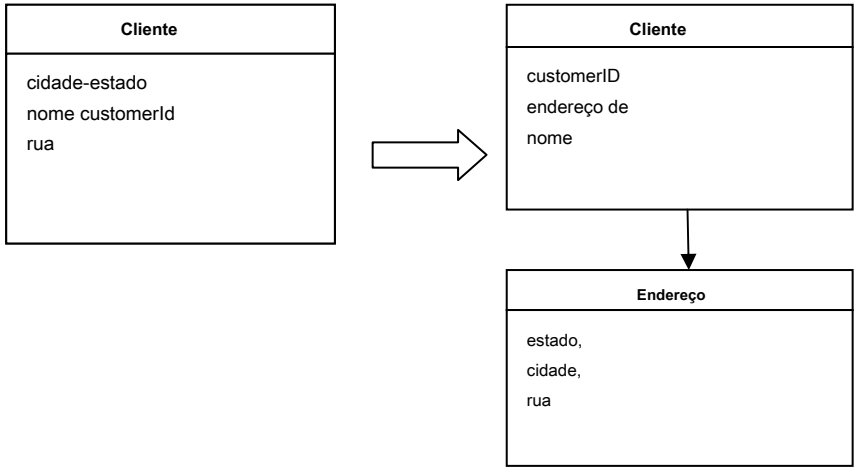
É necessário distinguir entre objetos de entidade e objetos de valor. Não é útil para fazer todas as entidades objeto para o bem da uniformidade. Na verdade, é recomendado para selecionar como entidades apenas os objetos que estejam em conformidade com a definição da entidade. E fazer o resto do objetos de valor objetos. (Vamos apresentar um outro tipo de objeto na próxima seção, mas vamos supor que temos apenas objetos de entidade e objetos de valor, por enquanto.) Isto irá simplificar o design, e haverá algumas outras consequências positivas.

Não tendo nenhuma identidade, objetos de valor podem ser facilmente criados e descartados. cuidados ninguém sobre a criação de uma identidade, e o coletor de lixo cuida do objeto quando não é mais referenciado por qualquer outro objeto. Isto simplifica o design muito.

É altamente recomendável que os objetos de valor ser imutável. Eles são criados com um construtor, e nunca modificado durante o seu tempo de vida. Quando você quiser um valor diferente para o objeto, você simplesmente criar outra. Isto tem consequências importantes para o design. Sendo imutável, e não tendo nenhuma identidade, objetos de valor pode ser compartilhado. Isso pode ser imperativo para alguns projetos. Objetos imutáveis são compartilháveis com importantes implicações de desempenho. Eles integridade também se manifestar, a integridade dos dados ie. Imagine o que isso significaria para compartilhar um objeto que não é imutável. Um sistema de reservas de viagens aéreas pode criar objetos para cada voo. Um dos atributos poderia ser o código de voo. Uma livros cliente um vôo para um determinado destino. Outro cliente quer reservar o mesmo vôo. O sistema escolhe para reutilizar o objeto que contém o código de voo, porque é sobre o mesmo vôo. Enquanto isso, o cliente muda de idéia e decide tomar um vôo diferente. O sistema altera o código de voo, porque este não é imutável. O resultado é que o código de voo do primeiro cliente muda também.

Uma regra de ouro é: se objetos de valor são compartilháveis, eles devem ser imutáveis. Objetos de valor deve ser mantido fino e simples. Quando um objeto de valor é necessária por outra parte, ele pode ser simplesmente passados por valor, ou uma cópia do mesmo pode ser criado e dado. Fazer uma cópia de um objeto de valor é simples, e geralmente sem

quaisquer consequências. Se não houver identidade, você pode fazer quantas cópias quiser, e destruir todos eles quando necessário.



Valor Os objetos podem conter outros objetos de valor, e eles podem até mesmo conter referências a entidades. Embora objetos de valor são usados para simplesmente conter atributos de um objeto de domínio, isso não significa que ele deve conter uma longa lista com todos os atributos. Os atributos podem ser agrupados em diferentes objetos. Atributos escolhido para fazer um objeto de valor deve formar um todo conceitual. Um cliente está associado a um nome, uma rua, uma cidade, e um estado. É melhor para conter as informações de endereço em um objeto separado, e o objeto cliente irá incluir uma referência a tal objeto. Rua, cidade, estado deve ter um objeto de seu próprio, o endereço, porque pertencem conceitualmente juntos, ao invés de ser atributos distintos de cliente, como mostrado no diagrama abaixo.

Serviços

Quando analisamos o domínio e tentar definir os principais objetos que compõem o modelo, descobrimos que alguns aspectos do domínio não são facilmente mapeados para objetos. Os objectos são geralmente consideradas como tendo atributos, um estado interno que é gerido pelo objecto, e exibem um comportamento. Quando desenvolvemos a linguagem ubíqua, os principais conceitos do domínio são introduzidos na língua, e os substantivos da língua são facilmente mapeados para objetos. Os verbos da língua, associados aos seus nomes correspondentes tornam-se parte do comportamento desses objetos. Mas existem algumas acções no domínio, alguns verbos, que não parecem pertencer a qualquer objeto. Eles representam um comportamento importante do domínio, de modo que não pode ser negligenciada ou simplesmente incorporados em algumas das entidades ou objetos de valor. Adicionando tal comportamento a um objeto iria estragar o objeto, tornando-se para a funcionalidade que não pertence a ele. No entanto, usando uma linguagem orientada a objetos, temos que usar um objeto para esta finalidade. Não podemos ter apenas uma função separada por conta própria. Tem que ser ligado a algum objeto. Muitas vezes este tipo de funções de comportamento através de vários objetos, talvez de diferentes classes. Por exemplo, para transferir dinheiro de uma conta para outra; deve essa função ser na conta de envio ou a conta receptora? Ela se sente tão perdido em qualquer um.
Muitas vezes este tipo de funções de comportamento através de vários objetos, talvez de diferentes classes. Por exemplo, para transferir dinheiro de uma conta para outra; deve essa função ser na conta de envio ou a conta receptora? Ela se sente tão perdido em qualquer um.
Muitas vezes este tipo de funções de comportamento através de vários objetos, talvez de diferentes classes. Por exemplo, para transferir dinheiro de uma conta para outra; deve essa função ser na conta de envio ou a conta receptora? Ela se sente tão perdido em qualquer um.

Quando tal comportamento um é reconhecida no domínio, a melhor prática é declará-la como um serviço. Tal objeto não tem um estado interno, e seu objetivo é simplesmente fornecer funcionalidade para o domínio. A assistência fornecida por um serviço pode ser um dos mais importantes, e um grupo de serviço pode relacionada funcionalidade que serve as entidades e o Valor de Objectos. É muito melhor para declarar o serviço explicitamente, porque cria uma distinção clara no domínio, ele encapsula um conceito. Ele cria confusão para incorporar essa funcionalidade em uma entidade ou valor de objeto, porque não vai ser claro o que esses objetos representam.

Serviços atuar como interfaces que fornecem operações. Serviços são comuns em estruturas técnicas, mas eles podem ser usados na camada de domínio também. Um serviço não é sobre o objeto a realização do serviço, mas está relacionado aos objetos as operações são realizadas em / para. Desta forma, um serviço geralmente se torna um ponto de conexão para muitos objetos. Esta é uma das razões por que o comportamento que pertence naturalmente a um serviço não devem ser incluídos em objetos de domínio. Se tal funcionalidade está incluída no objetos de domínio, uma densa rede de associações é criada entre eles e os objetos que são o beneficiário das operações. Um alto grau de acoplamento entre muitos objetos é um sinal de má concepção, porque torna o código difícil de ler e compreender, e mais importante, torna-se difícil de mudar.

Um serviço não deve substituir a operação que normalmente pertence em objetos de domínio. Não devemos criar um serviço para cada operação necessária. Mas quando tal operação se destaca como um conceito importante no domínio, um serviço deve ser criado para ele. Há três características de um serviço:

1. A operação realizada pelo Serviço refere-se a um conceito de domínio que não faz naturalmente parte de uma entidade ou valor objeto.

2. A operação realizada refere-se a outros objectos no domínio.

3. O funcionamento é sem estado.

Quando um processo significativo ou transformação no domínio não é uma responsabilidade natural de uma Entidade ou Valor Object, adicione uma operação para o modelo como uma interface independente declarado como um serviço. Definir a interface em termos da linguagem do modelo e certifique-se o nome da operação é parte do Ubiquitous Language. Faça o apátrida Service.

Ao usar os Serviços, é importante manter a camada de domínio isolado. É fácil ficar confuso entre os serviços que pertencem à camada de domínio, e as pertencentes à infra-estrutura. Também pode haver serviços na camada de aplicação

que acrescenta um nível adicional de complexidade. Esses serviços são ainda mais difíceis de separar os seus homólogos que residem na camada de domínio. Enquanto trabalhava no modelo e durante a fase de design, precisamos ter certeza de que os restos nível de domínio isolado dos outros níveis.

Ambos os aplicativos e domínio serviços são geralmente construídas em cima de Entidades de domínio e Valores fornecendo funcionalidade necessária diretamente relacionados a esses objetos. Decidir a camada de um Serviço pertence é difícil. Se a operação realizada conceptualmente pertence à camada de aplicação, em seguida, o serviço deve ser ali colocado. Se a operação é de cerca de objetos de domínio, e está estritamente relacionada com o domínio, cumprindo uma necessidade de domínio, então ele deve pertencer à camada de domínio.

Vamos considerar um exemplo prático, um aplicativo de relatórios web. Os relatórios fazem uso de dados armazenados em um banco de dados, e eles são gerados com base em modelos. O resultado final é uma página HTML que é mostrado para o usuário em um navegador web.

A camada de interface do usuário é incorporado em páginas da web e permite que o usuário de login, para selecionar o relatório desejado e clique em um botão para solicitá-lo. A camada de aplicação é uma camada fina que se encontra entre a interface de utilizador, o domínio e a infra-estrutura. Ele interage com a infra-estrutura de banco de dados durante as operações de login, e interage com a camada de domínio quando ele precisa para criar relatórios. A camada de domínio irá conter o núcleo do domínio, objectos directamente relacionadas com os relatórios. Dois desses objetos são Relatório e modelo, que os relatórios são baseados em. A camada de infra-estrutura irá suportar o acesso de banco de dados e acesso a arquivos.

Quando um usuário seleciona um relatório a ser criado, na verdade ele seleciona o nome do relatório a partir de uma lista de nomes. Esta é a reportId, uma corda. Alguns outros parâmetros são passados, como os itens mostrados no relatório e o intervalo de tempo dos dados incluídos no relatório. Mas vamos citar apenas os reportId pela simplicidade. Este nome é passado através da camada de aplicação para a camada de domínio. A camada de domínio é responsável pela criação e retornando o relatório que está sendo dado o seu nome. Como os relatórios são baseados em modelos, um serviço poderia ser criado, e seu objetivo seria

para obter o molde que corresponde a uma reportId. Este modelo é armazenado em um arquivo ou no banco de dados. Não é apropriado para colocar uma tal operação no próprio objeto Relatório. Ele não pertence ao objeto Gabarito quer. Então, criamos um serviço separado cujo objetivo é recuperar um modelo de relatório com base na identificação do relatório. Este seria um serviço localizado na camada de domínio. Ele faria uso da infra-estrutura de arquivo para recuperar o modelo a partir do disco.

módulos

Para uma aplicação grande e complexo, o modelo tende a crescer mais e mais. O modelo chega a um ponto onde é difícil falar sobre como um todo, e compreender as relações e interações entre as diferentes partes se torna difícil. Por essa razão, é necessário organizar o modelo em módulos. Os módulos são utilizados como um método de organizar conceitos e tarefas relacionadas, a fim de reduzir a complexidade.

Os módulos são amplamente utilizados na maioria dos projetos. É mais fácil para obter a imagem de um grande modelo, se você olhar para os módulos que ele contém, em seguida, as relações entre os módulos. Depois da interacção entre os módulos é entendido, pode-se começar a descobrir os detalhes do interior de um módulo. É uma maneira simples e eficiente para gerenciar a complexidade.

Outra razão para usar módulos está relacionada com a qualidade do código. É amplamente aceito que o código do software deve ter um elevado nível de coesão e um baixo nível de acoplamento. Enquanto a coesão começa no nível da classe e método, ele pode ser aplicado no nível de módulo. Recomenda-se a aulas em grupo altamente relacionada em módulos para fornecer a máxima coesão possível. Existem vários tipos de coesão. Dois dos mais utilizados são *coesão comunicacional*

e *coesão funcional.* Communicational coesão é conseguida quando as peças do módulo de operar sobre os mesmos dados. Faz sentido para agrupá-los, porque não há uma relação forte

entre eles. A coesão funcional é atingida quando todas as partes do conjunto de trabalho módulo para executar uma tarefa bem definida. Este é considerado o melhor tipo de coesão.

Usando módulos no design é uma maneira de aumentar o acoplamento coesão ea diminuição. Os módulos devem ser constituída por elementos que funcionalmente ou logicamente pertencem um ao outro, assegurando a coesão. Os módulos devem ter interfaces que são acessados por outros módulos bem definidos. Em vez de chamar três objetos de um módulo, é melhor o acesso de uma interface, uma vez que reduz o acoplamento. acoplamento baixo reduz a complexidade, e aumenta a facilidade de manutenção. É mais fácil entender como funciona um sistema quando há poucas conexões entre os módulos que executam tarefas bem definidas, do que quando cada módulo tem muitas ligações a todos os outros módulos.

Escolha Os módulos que contam a história do sistema e conter um conjunto coeso de conceitos. Isso muitas vezes resulta baixo acoplamento entre os módulos, mas se não procurar uma maneira de mudar o modelo de separar os conceitos, ou um conceito que pode ser a base de um módulo que traria os elementos juntos de uma forma significativa esquecido. Buscar baixo de engate, no sentido de que os conceitos podem ser compreendidos e fundamentados sobre independentemente um do outro. Refinar o modelo até que divisórias de acordo com conceitos do domínio de alto nível eo código correspondente é desacoplada também.

Dê os nomes módulos que se tornam parte do Ubiquitous Language. Módulos e seus nomes devem refletir uma visão sobre o domínio.

Designers estão acostumados a criação de módulos desde o início. Eles são partes comuns dos nossos projetos. Depois que a função do módulo é decidido, ele geralmente permanece inalterado, enquanto os internos do módulo pode mudar muita coisa. Recomenda-se ter alguma flexibilidade, e permitir que os módulos de evoluir com o projeto, e não deve ser mantido congelado. É verdade que o módulo refatoração pode ser mais caro do que uma refatoração classe, mas quando um erro design do módulo é encontrado, ele é melhor para enfrentá-lo, alterando o módulo, em seguida, encontrando maneiras de contornar isso.

agregados

Os últimos três padrões neste capítulo irá lidar com um desafio de modelagem diferente, uma relacionada com o ciclo de vida dos objetos de domínio. objetos de domínio passar por um conjunto de estados durante o seu tempo de vida. Eles são criados, colocados na memória e usada em cálculos, e eles são destruídos. Em alguns casos, eles são salvos em locais permanentes, como um banco de dados, onde podem ser recuperados a partir de algum tempo mais tarde, ou eles podem ser arquivados. Em algum momento eles podem ser completamente apagados do sistema, incluindo banco de dados eo armazenamento de arquivos.

Gerenciamento do ciclo de vida de um objeto de domínio constitui um desafio em si mesmo, e se não for feito corretamente, ele pode ter um impacto negativo sobre o modelo de domínio. Vamos apresentar três padrões que nos ajudam a lidar com isso. Agregada é um padrão de domínio usado para definir a propriedade objeto e limites. Fábricas e Repositórios são dois padrões de design que nos ajudar a lidar com a criação e armazenamento de objetos. Vamos começar falando sobre Agregados.

Um modelo pode conter um grande número de objetos de domínio. Não importa o quanto consideração que colocamos no projeto, acontece que muitos objetos estão associados uns com os outros, criando uma complexa rede de relacionamentos. Existem vários tipos de associações. Para cada associação traversable no modelo, tem de ser mecanismo de software que impõe que correspondente. associações reais entre objeto de domínio acabar no código, e muitas vezes até mesmo no banco de dados. Um relacionamento um-para-um entre um cliente e da conta bancária aberta em seu nome é expresso como uma referência entre dois objetos, e implica uma relação entre duas tabelas de banco de dados, o que mantém os clientes e aquele que mantém as contas.

Os desafios de modelos são mais frequentemente não para torná-los o suficiente completa, mas para torná-los mais simples e

compreensível quanto possível. Na maioria das vezes ele paga de eliminar ou relações Simplifique a partir do modelo. Isto é, a menos que eles incorporar conhecimento profundo do domínio.

A associação de um-para-muitos é mais complexa porque envolve muitos objetos que se tornam relacionado. Esta relação pode ser simplificado por transformá-la em uma associação entre um objeto e uma coleção de outros objetos, embora nem sempre é possível.

Há muitos-para-muitos associações e um grande número deles são bidirecionais. Isso aumenta a complexidade muito, tornando o gerenciamento do ciclo de vida de tais objetos bastante difícil. O número de associações deve ser reduzida, tanto quanto possível. Em primeiro lugar, as associações que não são essenciais para o modelo deve ser removido. Podem existir no domínio, mas eles não são necessários em nosso modelo, para tirá-los. Em segundo lugar, a multiplicidade pode ser reduzida por adição de uma restrição. Se muitos objetos satisfazer um relacionamento, é possível que apenas um vai fazê-lo se a restrição de direito é imposto sobre o relacionamento. Em terceiro lugar, muitas associações vezes bidirecionais pode ser transformado em uns unidirecionais. Cada carro tem um motor, e cada motor tem um carro, onde ele é executado. A relação é bidireccional,

Depois de reduzir e simplificar as associações entre os objetos, ainda pode acabar com muitos relacionamentos. Um sistema bancário detém dados e processos dos clientes. Esses dados incluem cliente dados pessoais,

como nome, endereço, telefone, trabalho

descrição e conta dados: número da conta, equilíbrio, operações realizadas, etc. Quando os arquivos de sistema ou completamente exclui as informações sobre um cliente, tem que se certificar de que todas as referências são removidos. Se muitos objetos manter tais referências, é difícil garantir que todos eles são removidos. Além disso, quando alguns dados muda para um cliente, o sistema tem que se certificar de que está devidamente atualizado durante todo o sistema e integridade dos dados é garantida. Isso geralmente é deixado para ser tratado a nível de banco de dados. As transacções são usadas para reforçar a integridade dos dados.

Mas se o modelo não foi cuidadosamente projetado, haverá um alto grau de contenção de banco de dados, resultando em mau desempenho. Enquanto as operações de banco de dados desempenham um papel vital em tais operações, é desejável para resolver alguns dos problemas relacionados com a integridade dos dados diretamente no modelo.

Também é necessário para ser capaz de cumprir as invariantes. Os invariantes são essas regras que têm de ser mantidos sempre que muda de dados. Isto é difícil de perceber quando muitos objetos manter referências a mudar objetos de dados.

É difícil garantir a consistência de mudanças a objetos em um modelo com associações complexas. Muitas vezes as invariantes aplica a objetos intimamente relacionados, não apenas aqueles discretos. No entanto, esquemas de bloqueio cautelosos causar vários usuários de interferir inutilmente uns com os outros e fazer um sistema inutilizável.

Portanto, uso agregados. Um agregado é um grupo de objectos associados que são considerados como uma unidade em relação a alterações de dados. O agregado é delimitada por uma fronteira que separa os objetos dentro daqueles fora. Cada agregado tem uma raiz. A raiz é uma entidade, e é o único objeto acessível a partir do exterior. A raiz pode conter referências a qualquer dos objectos agregados, e os outros objectos pode conter referências uns aos outros, mas um objecto exterior pode conter referências apenas para o objecto de raiz. Se há outras entidades dentro do limite, a identidade dessas entidades é local, fazendo sentido somente dentro do agregado.

Como é o agregado garantindo a integridade dos dados e fazer cumprir as invariantes? Uma vez que outros objetos podem conter referências apenas para a raiz, isso significa que eles não podem mudar diretamente os outros objetos no agregado. Tudo o que eles podem fazer é mudar a raiz, ou pedir a raiz para executar algumas ações. E a raiz será capaz de mudar os outros objetos, mas isso é uma operação contida no interior do agregado, e é controlável. Se a raiz é excluído e removido da memória, todos os outros objetos do agregado serão apagados também, porque não há nenhuma outra referência de objeto de retenção a qualquer um deles. Quando qualquer alteração for feita para a raiz que afeta indiretamente os outros objetos no

agregada, é simples de fazer cumprir as invariantes porque a raiz vai fazer isso. É muito mais difícil fazê-lo quando objetos externos têm acesso direto aos internos e alterá-las. Garantir o cumprimento das invariantes em tal circunstância um envolve colocar alguma lógica em objetos externos para lidar com ele, o que não é desejável.

É possível que a raiz para passar referências transitórias de objetos internos para os externos, com a condição de que os objetos externos não segurar a referência após a operação estiver concluída. Uma maneira simples de fazer isso é passar cópias do Valor Objects para objetos externos. Realmente não importa o que acontece com esses objetos, porque não vai afetar a integridade do agregado de qualquer forma.

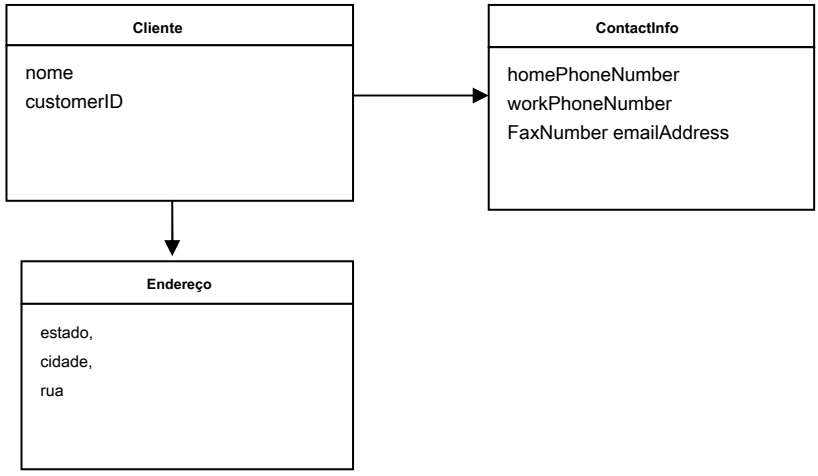
Se objetos de um agregado são armazenados em um banco de dados, apenas a raiz deve ser obtidas através de consultas. Os outros objectos devem ser obtido através de associações de atravessamento.

Objetos dentro de um agregado devem ser autorizados a realizar referências a raízes de outros agregados.

A raiz Entidade tem identidade global, e é responsável por manter os invariantes. Entidades internos têm identidade local.

Agrupar as entidades e objetos de valor em agregados e definir os limites em torno de cada. Escolha uma entidade para ser a raiz de cada agregado, e controlar todo o acesso aos objetos dentro do limite através da raiz. Permitir objetos externos para manter referências a apenas a raiz. referências transitórias para membros internos pode ser passado para fora para uso dentro de apenas uma única operação. Porque os controles de raiz acessar, não podem ser surpreendidos por mudanças para os internos. Este arranjo torna prático para fazer cumprir todas as invariantes para objetos no agregado e para o agregado como um todo, em qualquer mudança de estado.

Um exemplo simples de uma agregação é mostrado no diagrama seguinte. O cliente é a raiz do agregado, e todos os outros objetos são internos. Se o endereço for necessário, uma cópia do mesmo podem ser passados para objetos externos.



Fábricas

Entidades e Agregados muitas vezes pode ser grande e complexo - demasiado complexo para criar no construtor da entidade raiz. De fato tentando construir um agregado complexo em sua constructure está em contradição com o que acontece muitas vezes no próprio domínio, onde as coisas são criadas por outras coisas (como eletrônicos são criados em em linhas de montagem). É como ter a própria configuração da impressora.

Quando um objeto cliente quer criar um outro objeto, ele chama seu construtor e, eventualmente, passa alguns parâmetros. Mas quando a construção objeto é um processo trabalhoso, criando o objeto envolve um monte de conhecimento sobre a estrutura interna do objeto, sobre as relações entre os objetos contidos, e as regras aplicadas a eles. Isto significa que cada cliente do objeto irá realizar conhecimentos específicos sobre o objeto construído. Isso quebra o encapsulamento do domínio objetos e dos agregados. Se o cliente pertence à camada de aplicação, uma parte da camada de domínio foi movido para fora, bagunçar o projeto inteiro. Na vida real, é como nos é dado plástico, borracha, metal, silicone, e estamos construindo nossa própria impressora. Não é impossível, mas é realmente vale a pena fazer isso?

Criação de um objeto pode ser uma grande operação em si, mas as operações de montagem complexas não se encaixam a responsabilidade dos objetos criados. Combinando tais responsabilidades podem produzir desenhos desajeitados que são difíceis de entender.

Portanto, um novo conceito é necessário ser introduzida, que ajuda a encapsular o processo de criação do objeto complexo. Isso é chamado **Fábrica.** Fábricas são usados para encapsular o conhecimento necessário para a criação do objeto, e eles são especialmente úteis para criar Agregados. Quando a raiz do agregado é criada, todos os objetos contidos pelo agregado são criados juntamente com ele, e todos os invariantes são aplicadas.

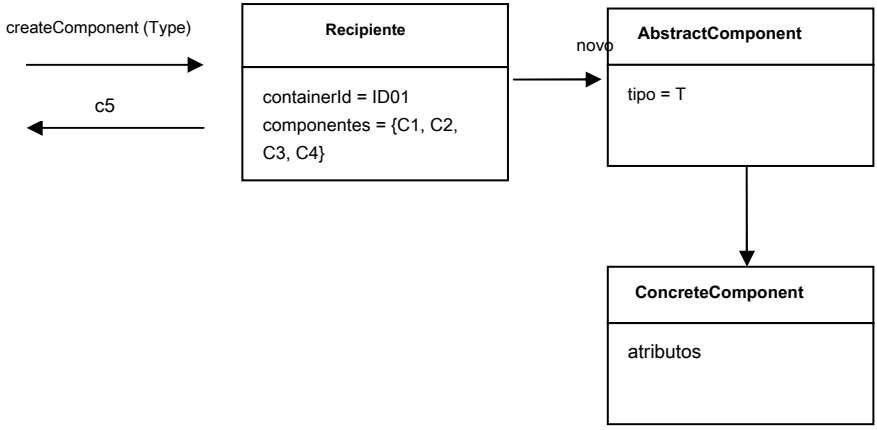
É importante para o processo de criação para ser atômica. Se não for, há uma chance para o processo de criação de ser meio caminho andado para alguns objetos, deixando-os em um estado indefinido. Isto é ainda mais verdadeiro para Agregados. Quando a raiz é criado, é necessário que todos os objectos sujeitos a invariantes são criados também. Caso contrário, as invariantes não pode ser imposta. Para imutável Valor Objetos isso significa que todos os atributos são inicializada para seu estado válido. Se um objeto não pode ser criado corretamente, uma exceção deve ser levantada, certificando-se que um valor inválido não é devolvido.

Portanto, transferir a responsabilidade para a criação de instâncias de objetos complexos e Agregados para um objeto separado, o que pode em si não tem nenhuma responsabilidade no modelo de domínio, mas ainda é parte do projeto de domínio. Fornecer uma interface que encapsula todas montagem complexa e que não requer o cliente para referenciar as classes concretas dos objetos que estão sendo instanciado. Criar Agregados inteiros como uma unidade, fazer valer os seus invariantes.

Existem vários padrões de projeto usados para implementar fábricas. O livro Design Patterns por Gamma et all. os descreve em detalhe, e apresenta estes dois padrões, entre outros: método de fábrica, Abstract Factory. Não vamos tentar apresentar os padrões de uma perspectiva de design, mas a partir de uma modelagem de domínio.

Um método de depósito é um método de objeto que contém e esconde o conhecimento necessário para criar um outro objecto. Isto é muito útil

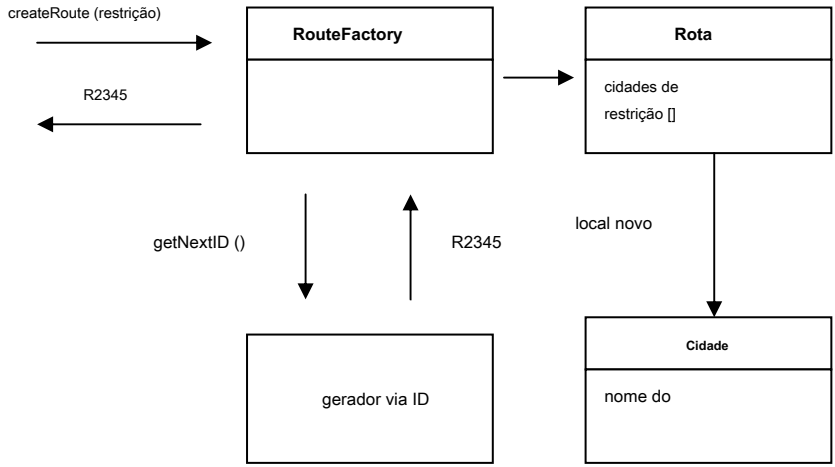
quando um cliente quer criar um objeto que pertence a um agregado. A solução é adicionar um método para a raiz agregado, que cuida da criação de objetos, impõe todos os invariantes, e retorna uma referência a esse objeto, ou uma cópia do mesmo.



O recipiente contém componentes e que são de um determinado tipo. É necessário que, quando tal componente é criado para automaticamente pertencem a um recipiente. O cliente chama o método createComponent (Type t) do recipiente. O recipiente instancia um novo componente. A classe concreta do componente é determinado com base no seu tipo. Após a sua criação, o componente é adicionado à coleção de componentes contidos pelo recipiente, e uma cópia do que é devolvido ao cliente.

Há momentos em que a construção de um objeto é mais complexo, ou quando a criação de um objeto envolve a criação de uma série de objetos. Por exemplo: a criação de um agregado. Escondendo as necessidades de construção internos de um agregado pode ser feito em um objeto de fábrica separada, que é dedicado a esta tarefa. Vamos considerar o exemplo de um módulo de programa que calcula a rota que pode ser seguido por um carro de partida até o destino a ser dado uma série de restrições. O usuário se conecta à web site executando o aplicativo e especifica um dos

seguintes restrições: a rota mais curta, a rota mais rápida, a rota mais barata. As rotas criadas podem ser anotados com informações do usuário que precisa ser salvo, para que possam ser posteriormente recuperados quando os logs de clientes novamente.



O gerador Route ID é usado para criar uma identidade única para cada rota que é necessário para uma entidade.

Ao criar um Factory, somos forçados a violar o encapsulamento de um objeto, o que deve ser feito com cuidado. Sempre que algo muda no objeto que tem um impacto sobre as regras de construção ou em algumas das invariantes, precisamos garantir que a fábrica é atualizado para suportar a nova condição. As fábricas estão fortemente relacionadas com os objetos que são criados. Isso pode ser uma fraqueza, mas também pode ser uma força. Um agregado contém uma série de objetos que estão intimamente relacionados. A construção de raiz está relacionada com a criação de outros objetos no agregado. Tem que haver alguma lógica que reúne um agregado. A lógica naturalmente não pertence a nenhum dos objetos, porque é sobre a construção de outros objetos. Parece apropriado usar uma classe de fábrica especial que é dada a tarefa de criar todo o agregado, e que conterá as regras, as restrições e as invariantes que têm de ser cumpridas para o agregado para ser válido. Os objetos serão

permanecer simples e vai servir o seu propósito específico, sem a confusão de lógica complexa construção.

Fábricas de entidade e Fábricas objeto de valor são diferentes. Os valores são geralmente objetos imutáveis, e todos os atributos necessários precisam ser produzidos no momento da criação. Quando o objeto é criado, ele tem que ser válido e final. Não vai mudar. As entidades não são imutáveis. Eles podem ser alteradas posteriormente, definindo alguns dos atributos com a menção de que todas as invariantes precisam ser respeitados. Outra diferença vem do fato de que as entidades precisam de identidade, enquanto objetos de valor não.

Há momentos em que a fábrica não é necessário, e um construtor simples é suficiente. Use um construtor quando:

- A construção não é complicado.
- A criação de um objeto não envolve a criação de outros, e todos os atributos necessários são passados através do construtor.
- O cliente está interessado na implementação, talvez quer escolher a estratégia utilizada.
- A classe é o tipo. Não há hierarquia envolvidos, então não há necessidade de escolher entre uma lista de implementações concretas.

Outra observação é que as fábricas precisa criar novos objetos a partir do zero, ou eles são obrigados a objetos Reconstituir que existiam anteriormente, mas foram provavelmente persistiu a um banco de dados. Trazendo Entidades volta para a memória de seu lugar de descanso em um banco de dados envolve um processo completamente diferente do que criar um novo. Uma diferença óbvia é que o novo objeto não precisa de uma nova identidade. O objeto já tem um. Violações dos invariantes são tratadas de maneira diferente. Quando um novo objeto é criado a partir do zero, qualquer violação dos invariantes acaba em uma exceção. Nós não podemos fazer isso com objetos recriados a partir de um banco de dados. Os objetos precisam ser reparado de alguma forma, para que eles possam ser funcional, caso contrário, há perda de dados.

repositórios

Em um projeto orientado a modelo, os objetos têm um ciclo de vida que começa com a criação e terminando com exclusão ou arquivamento. Um construtor ou uma fábrica cuida de criação do objeto. Todo o propósito de objetos criando é usá-los. Em uma linguagem orientada a objetos, é preciso manter uma referência a um objeto, a fim de ser capaz de usá-lo. Para ter referência tal, o cliente deve criar o objeto ou obtê-la a partir de outro, atravessando uma associação existente. Por exemplo, para obter um objeto de valor de um agregado, o cliente deve solicitá-lo a partir da raiz do agregado. O problema agora é que o cliente deve ter uma referência para a raiz. Para grandes aplicações, isso se torna um problema porque é preciso certificar-se de que o cliente tem sempre uma referência para o objeto necessário, ou para outro que tem uma referência para o respectivo objecto. Usando essa regra no projeto irá forçar os objetos para segurar uma série de referências que eles provavelmente não iria manter o contrário. Isso aumenta o acoplamento, criando uma série de associações que não são realmente necessários.

Para usar um meio objeto o objeto já foi criado. Se o objeto é a raiz de um agregado, então é uma entidade, e as chances são de que vai ser armazenado em um estado persistente em um banco de dados ou outra forma de persistência. Se for um objeto de valor, pode ser obtido a partir de uma Entidade atravessando uma associação. Acontece que uma grande quantidade de objetos podem ser obtidos diretamente do banco de dados. Isto resolve o problema de obter referência de objetos. Quando um cliente quer usar um objeto, ele acessa o banco de dados, recupera o objeto dele e usa-lo. Esta parece ser uma solução rápida e simples, mas tem impactos negativos sobre o design.

Bancos de dados são parte da infra-estrutura. A má solução é para que o cliente estar ciente dos detalhes necessários para acessar um banco de dados. Por exemplo, o cliente tem que criar consultas SQL para recuperar os dados desejados. A consulta de banco de dados pode retornar um conjunto de registros, expondo ainda mais de seus detalhes internos. Quando muitos clientes têm de criar objetos diretamente do banco de dados, verifica-se que

tal código está espalhado por todo o domínio. Nesse ponto, o modelo de domínio torna-se comprometida. Ele tem que lidar com muitos detalhes de infra-estrutura, em vez de lidar com conceitos do domínio. O que acontece se uma decisão é feita para alterar o banco de dados subjacente? Tudo o que necessita de código dispersos de ser alterado para ser capaz de acessar o novo armazenamento. Quando o código do cliente acessa um banco de dados diretamente, é possível que ele irá restaurar um objeto interno para um agregado. Isso quebra o encapsulamento do agregado com consequências desconhecidas.

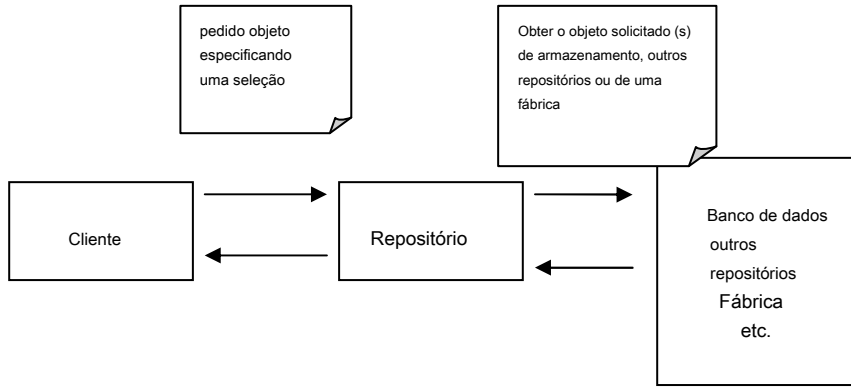
Um cliente precisa de um meio prático de adquirir referências a preexistente objetos de domínio. Se a infra-estrutura torna mais fácil para fazer isso, os desenvolvedores do cliente pode adicionar associações mais traversable, atrapalhando o modelo. Por outro lado, eles podem usar consultas para puxar os dados exatos de que precisam a partir do banco de dados, ou para puxar objetos alguns específico em vez de navegar a partir de raízes de agregação. Domínio lógica move-se em consultas e código de cliente, e as entidades e objetos de valor se tornam meros recipientes de dados. A complexidade técnica pura de aplicação de mais infra-estrutura de acesso à base de dados rapidamente inunda o código do cliente, o que leva os desenvolvedores a dumb-se a camada de domínio, o que torna o modelo irrelevante. O efeito geral é que o foco de domínio está perdido e que o projeto é comprometida.

Portanto, usar um repositório, cuja finalidade é encapsular toda a lógica necessária para obter referências de objeto. Os objetos de domínio não terá que lidar com a infra-estrutura para obter as referências necessárias para outros objetos do domínio. Eles só vão levá-los a partir do Repositório eo modelo é recuperar sua clareza e foco.

O Repositório pode armazenar referências a alguns dos objetos. Quando um objeto é criado, ele pode ser salvo no repositório, e recuperados de lá para ser usado mais tarde. Se o cliente solicitou um objeto a partir do Repositório, eo repositório não tê-lo, pode obtê-lo a partir do armazenamento. De qualquer maneira, o Repositório atua como um local de armazenamento para objetos globalmente acessíveis.

O Repositório pode também incluir uma estratégia. Pode aceder a um armazenamento de persistência ou outra baseada na Estratégia especificado. isto

podem utilizar diferentes locais de armazenamento para diferentes tipos de objetos. O efeito geral é que o modelo de domínio é dissociado da necessidade de armazenar objetos ou suas referências, e acessar a infra-estrutura de persistência subjacente.



Para cada tipo de objeto que precisa de acesso global, criar um objeto que pode fornecer a ilusão de uma coleção na memória de todos os objetos desse tipo. Configurar o acesso através de uma interface global, bem conhecido. Fornecer métodos para adicionar e remover objectos, os quais irão encapsular a inserção real ou remoção de dados no armazenador de dados. Fornecer métodos que selecionar objetos com base em alguns critérios e retornam objetos totalmente instanciado ou coleções de objetos cujos valores atributo cumprir os critérios,

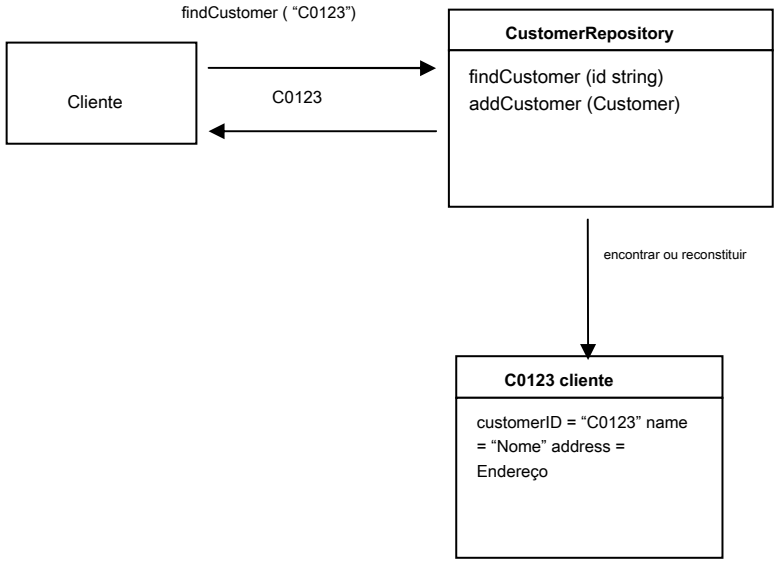
assim

encapsular o armazenamento real e consulta tecnologia. Fornecer repositórios somente para raízes agregado que realmente precisam de acesso direto. Manter o cliente focado no modelo, delegando todo o armazenamento de objetos e acesso aos Repositórios.

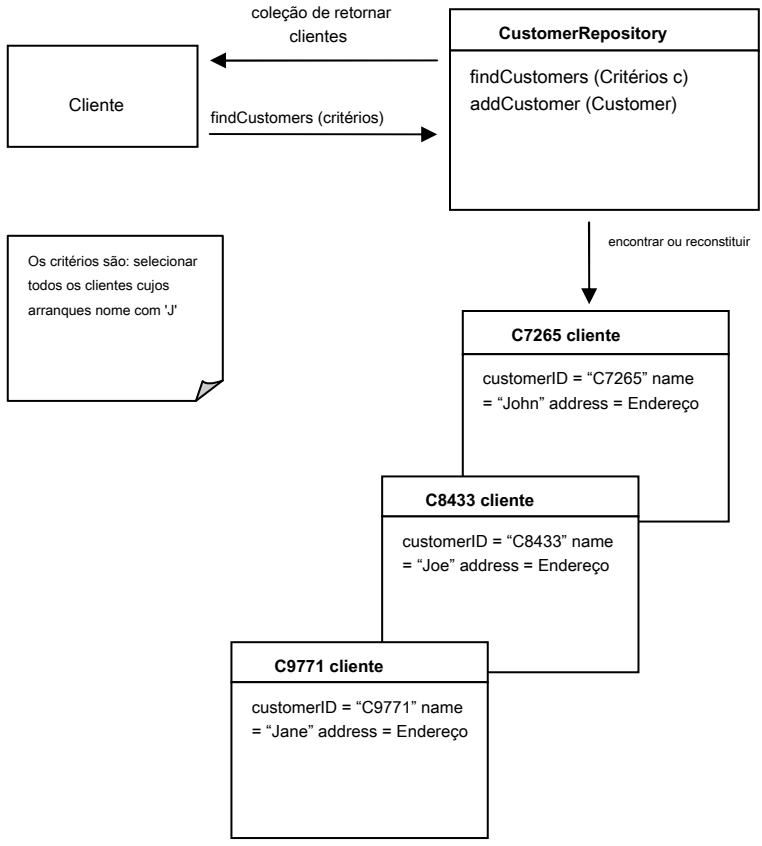
Um repositório pode conter informações detalhadas usadas para o acesso a infra-estrutura, mas sua interface deve ser simples. Um repositório deve ter um conjunto de métodos usados para recuperar objetos. O cliente chama esse método uma e passa um ou mais

parâmetros que representam os critérios de selecção utilizados para seleccionar um objecto ou um conjunto de objectos correspondentes. Uma entidade pode ser facilmente especificado, passando a sua identidade. Outros critérios de seleção podem ser composta por um conjunto de atributos de objeto. O Repositório irá comparar todos os objetos contra esse conjunto e irá retornar aqueles que satisfazem os critérios. A interface Repositório podem conter métodos utilizados para a realização de alguns cálculos suplementares, tais como o número de objectos de um determinado tipo.

Pode-se notar que a implementação de um repositório pode estar intimamente gostava da infra-estrutura, mas que a interface de repositório será modelo de domínio puro.

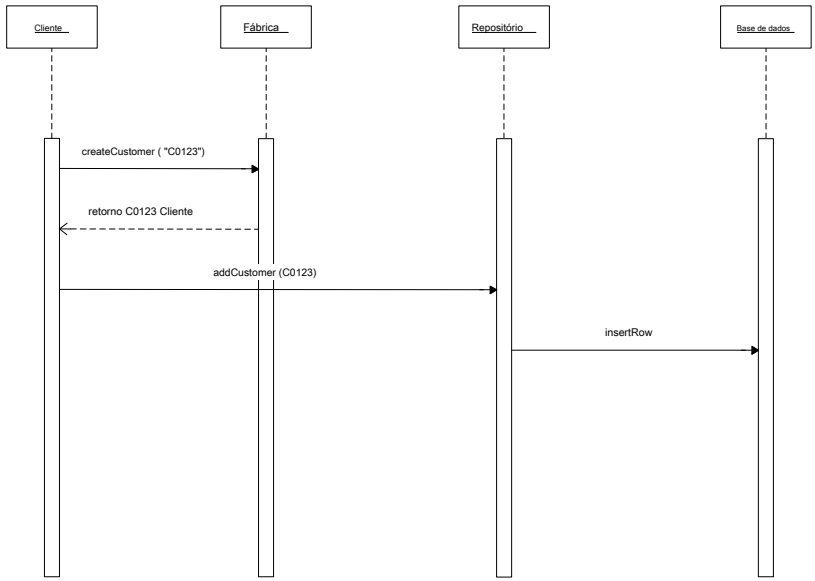


Outra opção é especificar um critério de seleção como uma especificação. A especificação permite a definição de um critério mais complexas, tais como a seguir:



Existe uma relação entre a fábrica e Repositório. Eles são ambos os padrões de design orientado a modelos, e ambos nos ajudam a gerenciar o ciclo de vida de objetos de domínio. Enquanto a fábrica está preocupado com a criação de objetos, o Repositório cuida de objetos já existentes. O Repositório pode cache de objetos localmente, mas na maioria das vezes ele precisa recuperá-los a partir de um armazenamento persistente. Os objectos são, quer criado utilizando um construtor ou eles são passados para uma fábrica para ser construído. Por esta razão, o repositório pode ser visto como uma fábrica, porque cria

objetos. Não é uma criação a partir do zero, mas uma reconstituição de um objeto que existia. Não devemos misturar um repositório com uma fábrica. A fábrica deve criar novos objetos, enquanto o repositório deve encontrar objetos já criados. Quando um novo objecto é para ser adicionado ao Repositório, deve ser criado pela primeira vez utilizando a fábrica, e em seguida, deve ser dado para o repositório que armazená-lo como no exemplo abaixo.



Outra forma como isso é observado é que as fábricas são “puro domínio”, mas que os repositórios podem conter links para a infra-estrutura, por exemplo, o banco de dados.

Versão online grátis.

Apoiar este trabalho, comprar a cópia impressa:
http://www.infoq.com/minibooks/domain-drivendesign-quickly

4

Refatoração Toward uma visão mais profunda

Refactoring contínua

So agora, temos falado sobre o domínio, e a importância de criar um modelo que expressa o domínio. Demos algumas orientações sobre as técnicas para ser usado para criar um modelo útil. O modelo tem de ser bem associado ao domínio que vem. Nós também disse que o projeto do código tem que ser feito em torno do modelo e do próprio modelo deve ser melhorado com base em decisões de design. Projetando sem um modelo pode levar a software que não é verdade para o domínio que serve, e não pode ter o comportamento esperado. Modelando sem feedback do projeto e sem desenvolvedores sendo leads envolvidos para um modelo que não é bem compreendido por aqueles que têm de implementá-lo, e pode não ser apropriado para as tecnologias utilizadas.

Durante o processo de design e desenvolvimento, temos de parar de vez em quando, e dar uma olhada no código. Pode ser o momento para uma refatoração. Refactoring é o processo de redesenhar o código para torná-lo melhor, sem alterar o comportamento do aplicativo. Refatoração geralmente é feito em pequenos passos, controláveis, com muito cuidado para que não interromper a funcionalidade ou introduzir alguns bugs. Afinal, o objetivo de refatoração é fazer com que o código melhor não pior. testes automatizados são de grande ajuda para garantir que nós não ter quebrado nada.

Há muitas maneiras de fazer refatoração de código. Existem padrões mesmo refatoração. Tais padrões representam uma abordagem automatizada para refatoração. Existem ferramentas desenvolvidas a partir desses padrões que fazem a vida do desenvolvedor mais fácil do que costumava ser. Sem essas ferramentas de refatoração pode ser muito difícil. Este tipo de refatoração ofertas mais com o código e sua qualidade.

Há um outro tipo de refatoração, um relacionado com o domínio e seu modelo. Às vezes, há uma nova visão sobre o domínio, algo se torna mais clara, ou uma relação entre dois elementos é descoberto. Tudo o que deve ser incluído no projeto através de refatoração. É muito importante ter o código expressivo que é fácil de ler e entender. A partir da leitura do código, deve ser capaz de dizer que o código faz, mas também por que ele faz isso. Só então o código realmente capturar a essência do modelo.

refactoring técnico, o que com base em padrões, pode ser organizado e estruturado. Refatoração para uma análise mais profunda não pode ser feito da mesma maneira. Não podemos criar padrões para ele. A complexidade de um modelo e a variedade de modelos não oferecem-nos a possibilidade de modelagem de abordagem de uma forma mecanicista. Um bom modelo é o resultado de um pensamento profundo, visão, experiência e talento.

Uma das primeiras coisas que são ensinadas sobre a modelagem é ler as especificações de negócios e procurar por substantivos e verbos. Os substantivos são convertidos para as aulas, enquanto os verbos se tornam métodos. Esta é uma simplificação, e vai levar a um modelo superficial. Todos os modelos são de pouca profundidade no início, mas devemos refatorar o modelo para mais e mais profundo insight.

O projeto tem que ser flexível. A dura resiste projeto refatoração. Código que não foi construída com flexibilidade em mente é o código difícil de trabalhar. Sempre que é necessária uma mudança, você verá o código lutando contra você, e coisas que devem ser reformulado facilmente levar muito tempo.

Usando um conjunto comprovado de blocos de construção básicos, juntamente com linguagem consistente traz alguma sanidade ao esforço de desenvolvimento. este

folhas o desafio de realmente encontrar um modelo incisiva, que capta preocupações sutis dos peritos do domínio e pode dirigir um design prático. Um modelo que sacode o superficial e capta o essencial é um modelo de profundidade. Isso deve tornar o software mais em sintonia com a forma como os especialistas em domínio pensar e mais sensível às necessidades do usuário.

Tradicionalmente, a refatoração é descrito em termos de transformações de código com motivações técnicas. Refatoração também pode ser motivado por uma visão sobre o domínio e um refinamento correspondentes do modelo ou a sua expressão em código.

modelos de domínio sofisticadas são raramente desenvolvido exceto através de um processo iterativo de refatoração, incluindo a implicação dos especialistas de domínio com desenvolvedores interessados em aprender sobre o domínio.

Traga conceitos chave Into Light

Refatoração é feito em pequenos passos. O resultado é também uma série de pequenas melhorias. Há momentos em que lotes de pequenas alterações adicionar muito pouco valor ao design, e há momentos em que poucas mudanças fazem muita diferença. É um grande avanço.

Começamos com uma grossa, modelo rasa. Em seguida, refiná-lo e o design baseado no conhecimento mais profundo sobre o domínio, em uma melhor compreensão das preocupações. Nós adicionamos novos conceitos e abstrações a ele. O projeto é então reformulado. Cada refinamento acrescenta mais clareza ao design. Isso cria por sua vez, as premissas para um avanço.

A Breakthrough muitas vezes envolve uma mudança no pensamento, na nossa maneira de ver o modelo. É também uma fonte de grande progresso no projeto, mas também tem algumas desvantagens. A descoberta pode implicar uma grande quantidade de refatoração. Isso significa tempo e recursos, algo que parece nunca ter o suficiente. Isso é também

arriscado, porque ampla refatoração pode introduzir mudanças comportamentais na aplicação.

Para alcançar um avanço, precisamos fazer os conceitos implícitos explícito. Quando falamos com os especialistas do domínio, trocamos um monte de idéias e conhecimentos. Alguns dos conceitos fazem o seu caminho para a Linguagem Ubíqua, mas alguns permanecem despercebidos no início. São conceitos implícitos, usados para explicar outros conceitos que já estão no modelo. Durante este processo de refinamento design, alguns desses conceitos implícitos chamar a nossa atenção. Descobrimos que alguns deles desempenham um papel fundamental no projeto. Nesse ponto, devemos fazer os respectivos conceitos explícito. Devemos criar classes e relacionamentos para eles. Quando isso acontece, podemos ter a chance de um avanço.

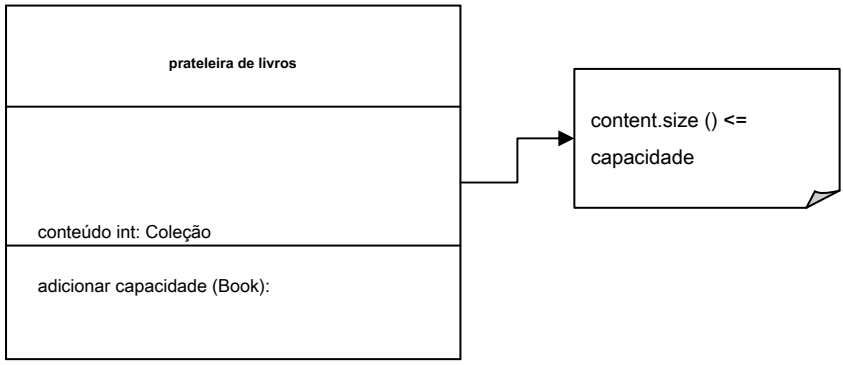
conceitos implícitos não deve permanecer assim. Se eles são conceitos de domínio, eles devem estar presentes no modelo e do design. Como podemos reconhecê-los? A primeira maneira de descobrir conceitos implícitos é ouvir o idioma. A linguagem que está usando durante a modelagem e design contém uma grande quantidade de informações sobre o domínio. No início pode não ser muito, ou algumas das informações não podem ser utilizadas corretamente. Alguns dos conceitos podem não ser totalmente compreendida, ou mesmo completamente mal interpretado. Isso tudo faz parte do aprendizado de um novo domínio. Mas como vamos construir a nossa Linguagem Ubíqua, os conceitos-chave fazem o seu caminho para ele. É aí que deve começar a olhar para os conceitos implícitos.

Às vezes seções do projeto pode não ser tão clara. Há um conjunto de relações que torna o caminho da computação difícil de seguir. Ou os procedimentos estão fazendo algo complicado que é difícil de entender. Esta é constrangimento no projeto. Este é um bom lugar para procurar por conceitos escondidos. Provavelmente algo está faltando. Se um conceito-chave que falta do quebra-cabeça, os outros vão ter que substituir sua funcionalidade. Isso vai engordar alguns objetos, acrescentando-lhes um comportamento que não é suposto estar lá. A clareza do projeto sofrerá. Tente ver se há um conceito faltando. Se for encontrado, torná-lo explícito. Refatorar o projeto para torná-lo mais simples e suppler.

Quando construí-lo conhecimento é possível executar em contradições. O que um especialista de domínio diz parecem contradizer o que mais confirma. Um requisito pode parecer contradizer o outro. Algumas das contradições não são realmente contradições, mas diferentes formas de ver a mesma coisa, ou simplesmente falta de precisão nas explicações. Devemos tentar conciliar contradições. Às vezes isso traz à luz importante conceitos. Mesmo se isso não acontecer, ainda é importante para manter tudo claro.

Outra maneira óbvia de cavar conceitos do modelo é a literatura de domínio uso. Há livros escritos sobre praticamente qualquer assunto possível. Eles contêm grande quantidade de conhecimentos sobre os respectivos domínios. Os livros não costumam conter modelos para os domínios que apresentam. A informação que eles contêm precisa ser processada, destilada e refinado. No entanto, as informações encontradas em livros é valioso, e oferece uma vista profunda do domínio.

Existem outros conceitos que são muito úteis quando explicitado: Restrição, Processos e Especificação. Uma restrição é uma maneira simples de expressar uma invariante. Aconteça o que acontecer com os dados do objeto, o invariante é respeitada. Isto é simplesmente feito, colocando a lógica invariável em uma restrição. O seguinte é um exemplo simples. Seu objetivo é explicar o conceito, não para representar a abordagem sugerida para um caso similar.



Podemos adicionar livros a uma estante, mas nunca devemos adicionar mais do que sua capacidade. Isto pode ser visto como parte do comportamento Bookshelf, como no seguinte código Java.

```
classe pública Bookshelf {
        capacidade de private int = 20;
        conteúdo colecção privada;
        public void add (livro Book) {
                se (content.size () + 1 <= capacidade) {
                        content.add (livro);
                } outro {
                throw new IllegalOperationException (
                “A estante atingiu o seu limite.”);
                }
        }
}
```

Nós podemos refatorar o código, extraindo a restrição em um método separado.

```
classe pública Bookshelf {
        capacidade de private int = 20;
        conteúdo colecção privada;
        public void add (livro Book) {
                if (isSpaceAvailable ()) {
                        content.add (livro);
                } outro {
                throw new IllegalOperationException (
                        “A estante atingiu o seu limite.”);

                }
        }
        boolean isSpaceAvailable privada () {
                voltar content.size () <capacidade;
        }
}
```

Colocar a restrição em um método separado tem a vantagem de tornar mais explícito. É fácil de ler e todo mundo vai notar que o método add () está sujeita a essa restrição. Também há espaço para o crescimento de adicionar mais lógica para os métodos se a restrição se torna mais complexo.

Os processos são geralmente expressos em código com procedimentos. Nós não usaremos uma abordagem processual, uma vez que estamos usando uma linguagem orientada a objetos, por isso precisamos de escolher um objeto para o processo, e adicionar um comportamento a ele. A melhor maneira de implementar processos é usar um serviço. Se existem diferentes maneiras de realizar o processo, então podemos encapsular o algoritmo em um objeto e usar uma estratégia. Nem todos os processos devem ser explicitadas. Se a Linguagem Ubíqua menciona especificamente o respectivo processo, então é hora para uma implementação explícita.

O último método para tornar os conceitos explícito de que estamos a tratar aqui é Specification. Basta dizer, a especificação é usada para testar um objeto para ver se satisfaz a determinados critérios.

A camada de domínio contém regras comerciais que se aplicam a entidades e objetos de valor. Essas regras são geralmente incorporados aos objetos que se aplicam. Algumas dessas regras são apenas um conjunto de perguntas cuja resposta é “sim” ou “não”. Essas regras podem ser expressas através de uma série de operações lógicas executadas em valores booleanos, eo resultado final é também um booleano. Um exemplo é o teste realizado em um objeto do cliente para ver se ele é elegível para um determinado crédito. A regra pode ser expressa como um método, chamado isEligible (), e pode ser anexado ao objeto Cliente. Mas esta regra não é um método simples que opera estritamente em dados do cliente. Avaliando a regra envolve verificar as credenciais do cliente, verificando se ele pagou suas dívidas no passado, a verificação para ver se ele tem saldos pendentes, etc. Tais regras de negócio pode ser grande e complexo, inchaço do objeto a tal ponto que já não serve o seu propósito original. Neste ponto, pode ser tentado a mover a regra inteira ao nível da aplicação, porque parece que se estende para além do nível de domínio. Na verdade, é tempo para uma refatoração.

A regra deve ser encapsulado em um objeto próprio, que se torna a especificação do cliente, e deve ser mantido na camada de domínio. O novo objeto irá conter uma série de métodos booleanos que teste se um determinado objeto do cliente é elegível para crédito ou não. Cada método desempenha o papel de um pequeno teste, e todos os métodos combinados dar a resposta à pergunta original. Se a regra de negócio não é compreendido em um objeto Specification, o código correspondente vai acabar sendo distribuídos por um número de objetos, tornando-se inconsistente.

A especificação é usada para objetos de teste para ver se eles cumprem alguma necessidade, ou se eles estão prontos para algum propósito. Ele também pode ser usado para seleccionar um determinado objecto a partir de uma colecção, ou como uma condição durante a criação de um objecto.

Muitas vezes, um único controlo Especificação se uma regra simples é satisfeito, e, em seguida, um número de tais especificações são combinadas em um um compósito que expressa o complexo de regra, como este:

```
= Clientes Cliente
customerRepository.findCustomer (customerIdentiy);

... Especificação customerEligibleForRefund = new Specification (


        nova CustomerPaidHisDebtsInThePast (),

        novos CustomerHasNoOutstandingBalances ());

if (customerEligibleForRefund.isSatisfiedBy (cliente) {

        refundService.issueRefundTo (cliente);

}
```

Testando regras simples é mais simples, e apenas com a leitura deste código é óbvio o que significa que um cliente é elegível para um reembolso.

5

Preservar Modelo Integrity

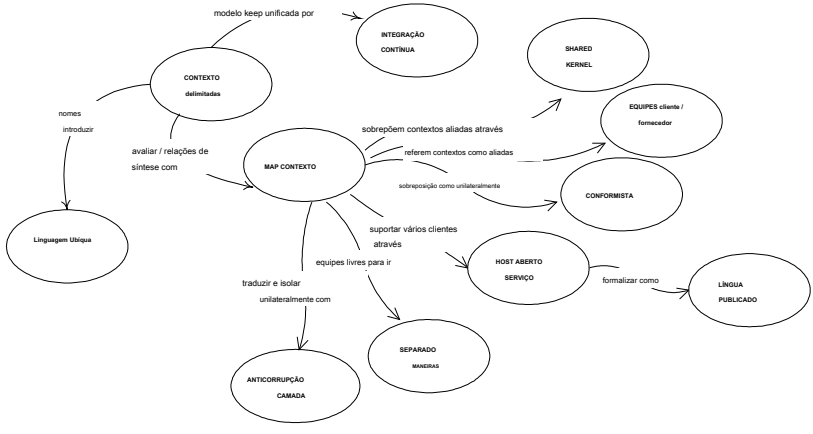
O seu capítulo é sobre os grandes projectos que requerem os esforços combinados de várias equipes. Somos confrontados com um conjunto diferente de desafios quando várias equipes, sob diferentes gestão e coordenação, são definidas na tarefa de desenvolver um projeto. projectos empresariais são geralmente grandes projectos, que empregam várias tecnologias e recursos. O projeto de tais projetos ainda deve ser baseada em um modelo de domínio, e temos de tomar medidas adequadas para garantir o sucesso do projeto.

Quando várias equipes trabalham em um projeto, o desenvolvimento do código é feita em paralelo, cada equipe sendo atribuída uma parte específica do modelo. As partes não são independentes, mas são mais ou menos interligadas. Todos eles começam com um modelo grande, e eles recebem uma parte dela para implementar. Vamos dizer que uma das equipes criou um módulo, e eles torná-lo disponível para outras equipes para usá-lo. Um desenvolvedor de outra equipe começa a usar o módulo, e descobre que está faltando alguma funcionalidade necessária para o seu próprio módulo. Ele adiciona a funcionalidade e verifica-in necessário o código para que ele possa ser usado por todos. O que ele pode não perceber é que esta é realmente uma mudança do modelo, e é bem possível que esta mudança irá interromper a funcionalidade do aplicativo. Isso pode acontecer facilmente, pois ninguém toma o tempo para entender completamente o modelo inteiro.

É tão fácil começar a partir de um bom modelo e progredir em direção a um único inconsistente. O primeiro requisito de um modelo é para ser consistente, com termos invariáveis e sem contradições. o

consistência interna de um modelo é chamada *unificação*. Um projeto da empresa poderia ter um modelo que abrange todo o domínio da empresa, sem contradições e sobreposição termos. Um modelo empresarial unificado é um ideal que não é facilmente realizado, e às vezes nem vale a pena tentar. Esses projetos precisam do esforço combinado de muitas equipes. As equipes precisam de um grande grau de independência no processo de desenvolvimento, porque eles não têm tempo para constantemente se reunir e discutir o design. A coordenação dessas equipes é uma tarefa difícil. Eles podem pertencer a diferentes departamentos e ter uma gestão separada. Quando o desenho do modelo evolui parcialmente de forma independente, estamos diante da possibilidade de integridade do modelo perde. Preservando a integridade do modelo, esforçando-se para manter um modelo unificado grande para todo o projeto da empresa não está indo para o trabalho. A solução não é tão óbvio, porque é o oposto de tudo o que aprendemos até agora. Em vez de tentar manter um grande modelo que vai desmoronar mais tarde, devemos conscientemente dividi-lo em vários modelos. Vários modelos bem integrado pode evoluir de forma independente, enquanto eles obedecem o contrato eles são obrigados a. Cada modelo deve ter uma borda bem delimitada, e as relações entre os modelos devem ser definidos com precisão.

Vamos apresentar um conjunto de técnicas utilizadas para manter a integridade do modelo. O seguinte desenho apresenta estas técnicas e a relação entre eles.



Contexto limitada

Cada modelo tem um contexto. Quando lidamos com um único modelo, o contexto é implícito. Nós não precisamos defini-lo. Quando criamos uma aplicação que é suposto para interagir com outros softwares, por exemplo, um aplicativo de legado, é claro que a nova aplicação tem seu próprio modelo e contexto, e eles são separados do modelo de legado e de seu contexto. Elas não podem ser combinados, misturados, ou confuso. Mas quando trabalhamos em um aplicativo corporativo grande, precisamos definir o contexto para cada modelo que criamos.

Vários modelos estão em jogo em qualquer grande projeto. No entanto, quando o código baseado em modelos distintos é combinado, o software torna-se de buggy, não confiável, e difícil de entender. A comunicação entre os membros da equipe se torna confuso. Nem sempre é claro em que contexto de um modelo não deve ser aplicado.

Não existe uma fórmula para dividir um modelo grande em partes menores. Tente colocar em um modelo dos elementos que estão relacionados, e que formam um conceito natural. *Um modelo deve ser pequeno o suficiente para ser atribuído a uma equipe.* cooperação da equipe e comunicação é mais fluido e completa, o que ajuda os desenvolvedores que trabalham no mesmo modelo. O contexto de um modelo é o conjunto de condições que precisam ser aplicadas para se certificar de que os termos utilizados no modelo tem um significado específico.

A idéia principal é definir o escopo de um modelo, para elaborar os limites do seu contexto, em seguida, fazer o máximo possível para manter o modelo unificado. É difícil manter um modelo puro quando se estende por todo o projeto da empresa, mas é muito mais fácil quando se está limitado a uma área especificada. Explicitamente definir o contexto em que a modelo se aplica. Explicitamente definir limites em termos de organização da equipe, o uso dentro de partes específicas do aplicativo, e as manifestações físicas, tais como bases de código e esquemas de banco de dados. Manter o modelo estritamente consistente dentro destes limites, mas não ser distraído ou confuso com questões fora.

A Bounded Contexto não é um módulo. Um delimitada Contexto fornece o quadro lógico dentro dos quais passa a modelo. Os módulos são utilizados para organizar os elementos de um modelo, de modo delimitada Contexto engloba o módulo.

Quando diferentes equipas têm de trabalhar no mesmo modelo, devemos ser muito cuidadosos para não pisar uns dos outros dedos. Nós temos que estar constantemente ciente de que muda para o modelo pode interromper a funcionalidade existente. Ao usar vários modelos, todos podem trabalhar livremente em seu próprio pedaço. Nós todos sabemos os limites do nosso modelo, e permanecer dentro das fronteiras. Nós apenas temos que ter certeza de que manter o modelo puro, consistente e unificada. Cada modelo pode suportar refatoração muito mais fácil, sem repercussões sobre outros modelos. O desenho pode ser refinados e destilados, a fim de alcançar o máximo de pureza.

Há um preço a pagar por ter vários modelos. Precisamos definir as fronteiras e as relações entre os diferentes modelos. Isso requer esforço extra de trabalho e design, e haverá talvez alguma tradução entre diferentes modelos. Nós não seremos capazes de transferir objetos entre diferentes modelos, e nós não pode invocar o comportamento livremente como se não houvesse limite. Mas esta não é uma tarefa muito difícil, e os benefícios são vale a pena o problema.

Por exemplo, queremos criar um aplicativo de e-commerce usado para coisas vender na Internet. Esta aplicação permite que os clientes para se inscrever, e nós recolher os seus dados pessoais, incluindo números de cartão de crédito. Os dados são mantidas em um banco de dados relacional. Os clientes estão autorizados a entrar, navegar no site à procura de mercadorias e encomendas. A aplicação terá de publicar um evento sempre que uma ordem foi colocada, porque alguém terá de enviar o item solicitado. Também queremos construir uma interface de relatórios usados para criar relatórios, para que possamos monitorar o status dos produtos disponíveis, o que os clientes estão interessados em comprar, o que eles não gostam, etc. No início, começamos com um modelo que cobre todo o domínio do comércio eletrônico. Somos tentados a fazê-lo, porque afinal de contas temos sido solicitado para criar uma grande aplicação.

a tarefa em mãos mais cuidadosamente, descobrimos que o aplicativo de e-shop não está realmente relacionado com o relatório. Eles têm preocupações diferentes, eles operam com conceitos diferentes, e podem até mesmo precisar usar diferentes tecnologias. A única coisa realmente comum é que os dados do cliente ea mercadoria é mantido no banco de dados e aplicações acessá-lo.

A abordagem recomendada é criar um modelo separado para cada um dos domínios, um para o e-commerce, e um para a geração de relatórios. Ambos podem evoluir livremente, sem muita preocupação com o outro, e até mesmo se tornar aplicativos separados. Pode ser o caso que o aplicativo de relatórios precisa de alguns dados específicos que o aplicativo de e-commerce deve armazenar no banco de dados, mas caso contrário eles podem crescer de maneira independente.

Um sistema de mensagens é necessário para informar o pessoal do armazém sobre as encomendas, para que eles possam enviar a mercadoria comprada. O pessoal de correio irá utilizar uma aplicação que lhes dá informação detalhada sobre o item comprado, a quantidade, o endereço do cliente e os requisitos de entrega. Não há necessidade de ter a tampa modelo de e-shop ambos os domínios de actividade. É muito mais simples para o aplicativo de e-shop para enviar Valor objetos que contêm informações de compra para o armazém usando mensagens assíncronas. Definitivamente, existem dois modelos que podem ser desenvolvidas separadamente, e só precisamos ter certeza de que a interface entre eles funciona bem.

Integração contínua

Uma vez que um Contexto delimitada foi definida, devemos mantê-lo soar. Quando um número de pessoas que estão trabalhando no mesmo contexto limitada, há uma forte tendência para o modelo a se fragmentar. Quanto maior a equipe, maior o problema, mas tão poucos como três ou quatro pessoas podem encontrar problemas graves.

No entanto, quebrar o sistema em contextos cada vez menores, eventualmente, perde um nível importante de integração e coerência.

Mesmo quando uma equipe trabalha em um Contexto Bounded há espaço para erro. Precisamos comunicar dentro da equipe para garantir que todos nós entender o papel desempenhado por cada elemento no modelo. Se alguém não entender as relações entre objetos, eles podem modificar o código de tal forma que vem em contradição com a intenção original. É fácil cometer esse erro quando não manter 100% foco na pureza do modelo. Um membro da equipe pode adicionar código que duplica o código existente sem o saber, ou eles podem adicionar código duplicado em vez de alterar o código atual, com medo de quebrar a funcionalidade existente.

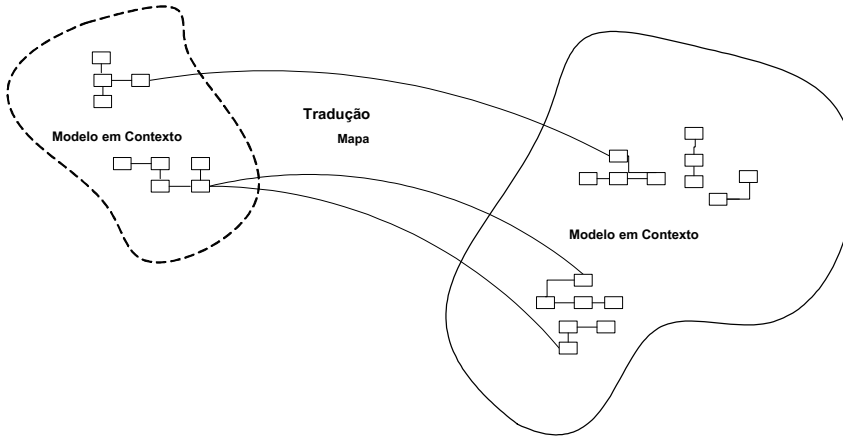
Um modelo não está totalmente definido desde o início. Ele é criado, em seguida, ele evolui continuamente com base em uma nova visão no domínio e feedback do processo de desenvolvimento. Isso significa que novos conceitos podem entrar no modelo, e novos elementos são adicionados ao código. Todos estes necessitam devem ser integrados em um modelo unificado, e implementado em conformidade no código. É por isso que Integração Contínua é um processo necessário dentro de um contexto limitada. Precisamos de um processo de integração para se certificar de que todos os novos elementos que são adicionados se encaixam harmoniosamente o resto do modelo, e são implementados corretamente no código. Precisamos ter um procedimento usado para mesclar o código. Quanto mais cedo nos fundimos o código melhor. Para uma única equipe pequena, fusões diárias são recomendadas. Nós também precisa ter um processo de construção no lugar. O código mesclado precisa ser construído automaticamente para que ele possa ser testado. Outro requisito necessário é a realização de testes automatizados. Se a equipe tem uma ferramenta de teste, e criou um conjunto de testes, o teste pode ser executado em cima de cada construção, e quaisquer erros são sinalizados. O código pode ser facilmente alterado para corrigir os erros relatados, porque eles são capturados cedo, eo, construção e processo de teste de mesclagem é iniciado novamente.

Integração Contínua é baseada na integração de conceitos no modelo, em seguida, encontrar o seu caminho para a implementação onde é testada. Qualquer inconsistência do modelo podem ser vistos na

implementação. Integração Contínua aplica-se a um contexto de limitada, ele não é usado para lidar com as relações entre vizinhos contextos.

contexto Mapa

Um aplicativo corporativo tem vários modelos e cada modelo tem seu próprio contexto de limitada. É aconselhável usar o contexto como base para a organização da equipe. As pessoas na mesma equipe pode se comunicar mais facilmente, e eles podem fazer um trabalho melhor integração do modelo e da implementação. Enquanto cada equipe trabalha em seu modelo, é bom para que todos possam ter uma idéia do quadro geral. Um Contexto Mapa é um documento que descreve os diferentes Bounded contextos e as relações entre eles. Um Contexto Mapa pode ser um diagrama como a mostrada abaixo, ou pode ser qualquer documento escrito. O nível de detalhe podem variar. O que é importante é que todos os que trabalham sobre as ações do projeto e entende-lo.



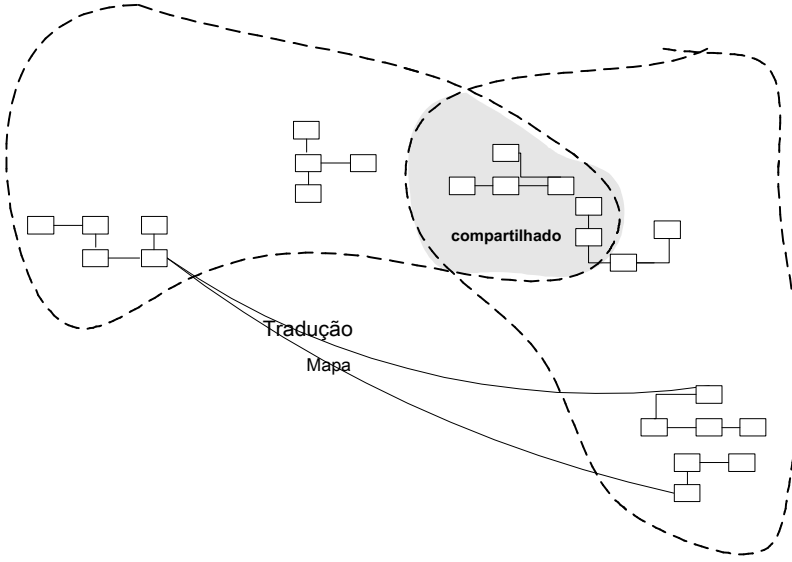
Não é o suficiente para ter modelos unificados separadas. Eles têm que ser integrado, pois a funcionalidade de cada modelo é apenas uma parte de todo o sistema. No final, as peças têm que ser montados

juntos, e todo o sistema deve funcionar corretamente. Se os contextos não são claramente definidos, é possível que eles se sobrepõem uns aos outros. Se as relações entre os contextos não são descritas, há uma chance de eles não irão funcionar quando o sistema é integrado.

Cada Contexto Limitada deve ter um nome que deve fazer parte do Ubiquitous Language. Que ajuda a comunicação da equipe muito ao falar sobre todo o sistema. Todos devem saber os limites de cada contexto e o mapeamento entre contextos e código. Uma prática comum consiste em definir os contextos, em seguida, criar módulos para cada contexto, e utilizar uma convenção de nomenclatura para indicar o contexto cada módulo pertence.

Nas páginas seguintes falamos sobre a interação entre diferentes contextos. Nós apresentamos uma série de padrões que podem ser usadas para criar Contexto Mapas onde contextos têm papéis claros e suas relações são apontadas. O Kernel Shared e cliente-fornecedor são padrões com um alto grau de interação entre contextos. Separate Ways é um padrão usado quando queremos os contextos de ser altamente independente e evoluir separadamente. Há mais dois padrões que lidam com a interação entre um sistema e um sistema legado ou um externo, e eles estão abertos Serviços de host e Layers anticorrupção.

Kernel Shared



quando funcional integração é limitada, a sobrecarga de Integração Contínua pode ser considerado muito alto. Isso pode ser especialmente verdadeiro quando as equipes não têm a habilidade ou a organização política para manter a integração contínua, ou quando uma única equipe é simplesmente demasiado grande e pesado. Contextos delimitadas tão separadas pode ser definido e várias equipes formadas.

equipes descoordenados trabalhando em aplicações estreitamente relacionados pode ir correndo para a frente por um tempo, mas o que eles produzem podem não se encaixam. Eles podem acabar gastando mais em camadas de tradução e adaptação do que gastaria em Integração Contínua, em primeiro lugar, entretanto duplicação de esforços e perder os benefícios de um comum Ubiquitous Language.

Portanto, designar um subconjunto do modelo de domínio que as duas equipes concordam em compartilhar. É claro que isso inclui, junto com este subconjunto do modelo, o subconjunto de código ou de design de banco de dados associado a essa parte do modelo. Este material explicitamente compartilhada tem estatuto especial, e não deve ser alterado sem consulta com a outra equipe.

Integrar um sistema funcional com frequência, mas um pouco menos frequentemente do que o ritmo de integração contínua dentro das equipes. Durante essas integrações, executar os testes de ambas as equipes.

O objetivo do Kernel Shared é reduzir a duplicação, mas ainda mantém dois contextos distintos. Desenvolvimento em um Kernel Shared precisa de muito cuidado. Ambas as equipes podem modificar o código do kernel, e eles têm de integrar as alterações. Se as equipes usam cópias separadas do código do kernel, eles têm que mesclar o código mais rapidamente possível, pelo menos semanalmente. Um conjunto de testes deve ser no lugar, então cada mudança feita para o kernel a ser testado imediatamente. Qualquer alteração do kernel devem ser comunicados a outra equipe, e as equipes devem ser informados, sensibilizando-os para a nova funcionalidade.

Fornecedor do cliente

Há momentos em que dois subsistemas têm uma relação especial: um depende muito do outro. Os contextos em que existem esses dois subsistemas são diferentes, e o resultado do processamento de um sistema é alimentado para o outro. Eles não têm um Kernel Shared, porque não pode ser conceitualmente correto para ter um, ou até pode não ser tecnicamente possível para os dois subsistemas de compartilhar código comum. Os dois subsistemas estão em um relacionamento cliente-fornecedor.

Vamos voltar a um exemplo anterior. Nós conversamos antes sobre os modelos envolvidos em um aplicativo de e-commerce, que inclui relatórios e mensagens. Já disse que é muito melhor para criar modelos separados para todos esses contextos, porque um único modelo seria um gargalo constante e fonte de discórdia no processo de desenvolvimento. Assumindo que estamos de acordo para ter modelos separados, quais devem ser as relações entre o subsistema de compras na web e um relato? faz o Kernel Shared não parece ser a escolha certa. O subsistema provavelmente irá utilizar diferentes tecnologias a serem implementadas. Um é

uma experiência de navegação pura, enquanto o outro poderia ser uma aplicação gráfica rica. Mesmo se o aplicativo de relatório é feito através de uma interface web, os principais conceitos dos respectivos modelos são diferentes. Pode haver alguma sobreposição, mas não o suficiente para justificar uma Kernel Shared. Então, nós optar por ir a um caminho diferente. Por outro lado, o subsistema-shopping e não depende em absoluto, de um relatório. Os usuários do aplicativo-shopping e são clientes web que navegam para as encomendas de mercadorias e lugar. Todos os clientes, mercadorias e encomendas dados são colocados em um banco de dados. E é isso. O aplicativo de compras e não está realmente interessado no que acontece com o respectivo dados. Enquanto isso, o aplicativo de relatório está muito interessado em e precisa os dados salvos pelo aplicativo-shopping e. Ele também precisa de alguma informação extra para realizar os serviços de relatórios que fornece. Os clientes podem colocar alguma mercadoria no cesto, e depois soltá-lo antes do check out. Os clientes podem visitar alguns links mais do que outros. Este tipo de informação não tem qualquer significado para o aplicativo de compras e, mas pode significar muito para o relatório. Depois disso, o subsistema de fornecedor tem de implementar algumas especificações que são necessários para o subsistema de cliente. Esta é uma conexão entre os dois subsistemas. o subsistema de fornecedor tem de implementar algumas especificações que são necessários para o subsistema de cliente. Esta é uma conexão entre os dois subsistemas. o subsistema de fornecedor tem de implementar algumas especificações que são necessários para o subsistema de cliente. Esta é uma conexão entre os dois subsistemas.

Outro requisito está relacionado com o banco de dados usado, mais exatamente seu esquema. Ambas as aplicações fará uso do mesmo banco de dados. Se o subsistema-shopping e foi o único a acessar o banco de dados, o esquema de banco de dados pode ser alterado a qualquer momento para refletir suas necessidades. Mas o subsistema de comunicação precisa acessar o banco de dados também, então ele precisa de alguma estabilidade do seu esquema. É impossível imaginar que o esquema do banco não vai mudar durante todo o processo de desenvolvimento. Isso não vai representar um problema para o aplicativo de compras e, mas certamente vai ser um problema para o relatório. As duas equipes terão de comunicar, provavelmente eles vão ter que trabalhar na base de dados juntos, e decidir quando a mudança é para ser realizado. Este funcionará como uma limitação para o subsistema de comunicação, porque essa equipe iria preferir fazer rapidamente a mudança e seguir em frente com o desenvolvimento, em vez de esperar no app-compras e. Se o time-shopping e tem direitos de veto, podem impor limites ao

mudanças a serem feitas ao banco de dados, prejudicando a atividade da equipe de reportagem. Se o time-shopping e pode agir independentemente, eles vão quebrar os acordos mais cedo ou mais tarde, e implementar algumas mudanças que a equipe de reportagem não está preparado para. Esse padrão funciona bem quando as equipes estão sob a mesma gestão. Isso facilita o processo de decisão, e cria harmonia.

Quando somos confrontados com o cenário tal, devemos começar a agir. A equipe de reportagem deve desempenhar o papel de cliente, enquanto a equipe eShopping deve desempenhar o papel de fornecedor. As duas equipes devem se reunir regularmente ou a pedido, e conversar como um cliente faz com o seu fornecedor. A equipe de cliente deve apresentar as suas exigências, enquanto a equipe de fornecedor deve fazer os planos nesse sentido. Enquanto todos os requisitos da equipe o cliente terá que ser cumprida no final, o calendário para fazer isso é decidido pela equipe fornecedor. Se alguns requisitos são considerados realmente importante, que deve ser implementado mais cedo, enquanto outros requisitos pode ser adiada. A equipe do cliente também vai precisar de entrada e conhecimento para ser compartilhado pela equipe de fornecedor. Este processo flui de uma maneira, mas é necessário em alguns casos.

A interface entre os dois subsistemas precisa ser definido com precisão. Um conjunto de testes de conformidade deve ser criado e usado para teste a qualquer momento se os requisitos de interface são respeitados. A equipe de fornecedor será capaz de trabalhar mais sem reservas em seu design, porque a rede segura dos testes de interface alertas privada los sempre é um problema.

Estabelecer uma relação cliente / fornecedor clara entre as duas equipes. Em sessões de planejamento, fazer parte da equipe do cliente desempenham um papel de cliente para a equipe de fornecedor. Negociar e tarefas orçamentais para as necessidades do cliente para que todos compreendam o compromisso e cronograma.

Conjuntamente desenvolver testes de aceitação automatizados que irá validar a interface esperada. Adicionar esses testes para suite de testes da equipe fornecedor, para ser executado como parte de sua integração contínua. Este teste irá liberar a equipe de fornecedor a mudanças make sem medo de efeitos colaterais para a aplicação da equipe do cliente.

Conformista

A relação cliente-fornecedor é viável quando ambas as equipes estão interessados no relacionamento. O cliente é muito dependente do fornecedor, enquanto o fornecedor não é. Se não houver uma gestão para fazer este trabalho, o fornecedor pagará a atenção necessária e vai ouvir os pedidos do cliente. Se a gestão não decidiu claramente como as coisas devem estar entre as duas equipes, ou se houver má gestão ou falta dela, o fornecedor será lentamente mais preocupado com o seu modelo e design, e menos interessado em ajudar o cliente. Eles têm os seus próprios prazos, afinal. Mesmo que eles são boas pessoas, dispostas a ajudar a outra equipe, a pressão de tempo terá uma palavra a dizer, e a equipe do cliente sofrerá. Isso também acontece quando as equipes pertencem a diferentes empresas. A comunicação é difícil, e empresa do fornecedor pode não estar interessado em investir muito nessa relação. Eles vão quer fornecer ajuda esporádica, ou simplesmente se recusar a cooperar em tudo. O resultado é que a equipe do cliente é por conta própria, tentando fazer o seu melhor com o modelo eo design.

Quando duas equipes de desenvolvimento têm uma relação cliente-fornecedor em que a equipe fornecedor não tem nenhuma motivação para prever as necessidades da equipe cliente, a equipe do cliente é impotente. Altruísmo pode motivar desenvolvedores fornecedor para fazer promessas, mas eles não são susceptíveis de ser cumprida. A crença na essas boas intenções lidera a equipe do cliente para fazer planos com base em características que nunca serão disponível. O projeto do cliente será adiada até que a equipe em última análise, aprende a viver com o que é dado. Uma interface sob medida para as necessidades da equipe cliente não está nos cartões.

A equipe cliente tem algumas opções. A mais óbvia é a de separar do fornecedor e para ser completamente por conta própria. Vamos olhar para isso mais tarde no padrão caminhos separados. Às vezes, os benefícios proporcionados pelo subsistema de fornecedor não valem a pena. Talvez seja mais simples para criar um separado

modelo, e design sem ter que dar um pensamento para o modelo do fornecedor. Mas isso nem sempre é o caso.

Às vezes, há algum valor em modelo do fornecedor, e uma conexão tem que ser mantida. Mas porque a equipe fornecedor não ajudar a equipe do cliente, este último tem de tomar algumas medidas para se proteger de mudanças do modelo realizadas pelo ex-equipe. Eles terão que aplicar uma camada de conversão, que liga os dois contextos. Também é possível que o modelo da equipe fornecedor poderia ser mal concebido tornando sua utilização estranho. O contexto cliente ainda pode fazer uso dele, mas ele deve se proteger usando uma camada Anticorrupção que discutiremos mais tarde.

Se o cliente tem que usar o modelo da equipe fornecedor, e se isso for bem feito, pode ser hora de conformidade. A equipe do cliente pode aderir a modelo da equipe fornecedor, conformando-se inteiramente a ele. Isto é muito parecido com o Kernel Shared, mas há uma diferença importante. A equipe do cliente não pode fazer alterações no kernel. Eles só pode usá-lo como parte de seu modelo, e eles podem construir sobre o código existente fornecido. Há muitas ocasiões em que tal solução é viável. Quando alguém fornece um componente rico, e fornece uma interface para isso, podemos construir nosso modelo incluindo a respectiva componente como seria nossa. Se o componente tem uma interface pequena, pode ser melhor do que simplesmente criar um adaptador para ele, e traduzir entre o nosso modelo eo modelo do componente. Este seria isolar o nosso modelo,

anticorrupção Camada

Que muitas vezes encontramos circunstâncias quando nós criamos uma aplicação que tem de interagir com o software legado ou um aplicativo separado. Este é outro desafio para o modelador de domínio. Muitas aplicações legadas não foram construídos usando técnicas de modelagem de domínio, e seu modelo é confusa,

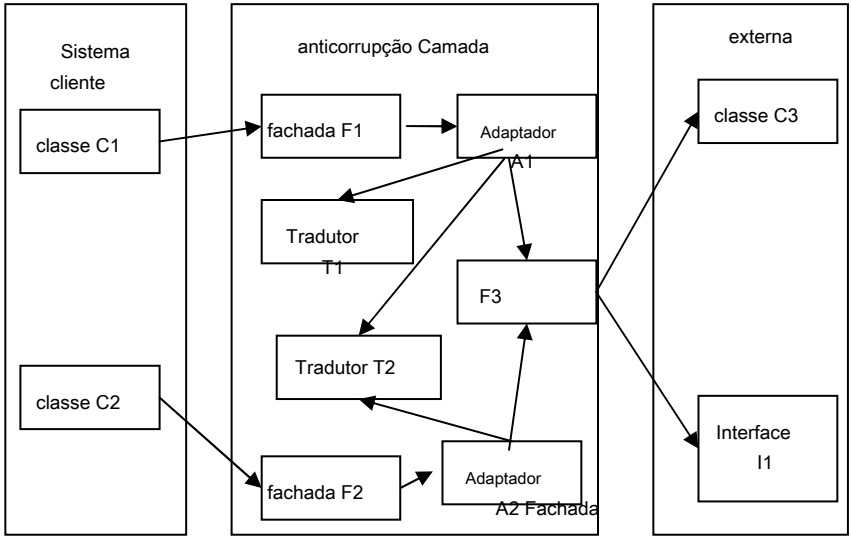
emaranhado difícil de entender e difícil de trabalhar. Mesmo se ele foi bem feito, o modelo de aplicativo legado não é de muita utilidade para nós, porque o nosso modelo é provável que seja bastante diferente. No entanto, tem de haver um nível de integração entre o nosso modelo eo legado, porque é um dos requisitos para usar o aplicativo antigo.

Há maneiras diferentes para o nosso sistema de cliente para interagir com um externo. Uma delas é através de conexões de rede. Ambos os aplicativos precisam usar os mesmos protocolos de comunicação de rede, e o cliente precisa aderir à interface usada pelo sistema externo. Outro método de interacção representa a base de dados. O sistema externo trabalha com dados armazenados em um banco de dados. O sistema cliente deve acessar o mesmo banco de dados. Em ambos os casos, estamos lidando com dados primitivos sendo transferidos entre os sistemas. Enquanto isso parece ser bastante simples, a verdade é que os dados primitivo não contém qualquer informação sobre os modelos. Não podemos levar dados de um banco de dados e tratar tudo como dados primitivos. Existe uma grande quantidade de semântica escondidos atrás de dados. Uma base de dados relacional contém dados primitivos relacionados com outros dados primitivos, criando uma rede de relações. A semântica de dados é muito importante e precisa ser considerado. O aplicativo cliente não pode acessar o banco de dados e escrever para ela sem entender o significado dos dados utilizados. Vemos que partes do modelo externo são refletidas no banco de dados, e fazer o seu caminho em nosso modelo.

Há o risco do modelo externo para alterar o modelo cliente se permitirmos que isso aconteça. Não podemos ignorar a interação com o modelo externo, mas devemos ter cuidado para isolar o nosso próprio modelo a partir dele. Devemos construir uma camada Anticorrupção que fica entre o nosso modelo de cliente e a externa. Do ponto de vista do nosso modelo, a camada Anticorrupção é uma parte natural do modelo; ele não se parece com algo estranho. Ele opera com os conceitos e ações familiares ao nosso modelo. Mas a camada de palestras anticorrupção para o modelo externo, usando a linguagem externa e não a um cliente. Esta camada funciona como um tradutor bidirecional entre dois domínios e idiomas. O melhor

realização é que o modelo cliente permanece pura e consistente sem ser contaminado pelo outro externo.

Como devemos implementar a camada Anticorrupção? Uma solução muito boa é ver a camada como um serviço a partir do modelo cliente. É muito simples de usar um serviço porque ele abstrai a outro sistema, e vamos enfrentá-lo em nossos próprios termos. O serviço vai fazer a tradução necessária, para que os nossos restos modelo isolado. Em relação à implementação real, o serviço será feito como uma fachada. (Veja Design Pattern por Gamma et al., 1995) Além disso, a camada Anticorrupção provavelmente irá precisar de um adaptador. O adaptador permite converter a interface de uma classe para aquele compreendido pelo cliente. No nosso caso, o adaptador não necessariamente envolver uma classe, porque o seu trabalho é traduzir entre dois sistemas.



A camada de Anticorrupção pode conter mais de um serviço. Para cada serviço existe uma fachada correspondente e para cada Fachada podemos adicionar um adaptador. Não devemos usar um único adaptador para todos os serviços, porque nós desordenar-lo com funcionalidade mista.

Nós ainda temos que adicionar mais um componente. O adaptador cuida de embrulho-se o comportamento do sistema externo. Também precisamos objeto e dados de conversão. Isso é feito usando um tradutor.

Este pode ser um objeto muito simples, com pouca funcionalidade, servindo a necessidade básica de tradução de dados. Se o sistema externo tem uma interface complexa, pode ser melhor adicionar uma fachada adicional entre os adaptadores e essa interface. Isto vai simplificar o protocolo do adaptador, e separá-lo do outro sistema.

Caminhos separados

Até agora, temos tentado encontrar maneiras de integrar subsistemas, torná-los trabalhar juntos, e fazê-lo de tal maneira que iria manter o modelo eo som design. Isso requer esforço e compromisso. Equipes que trabalham nos respectivos subsistemas precisa gastar um tempo considerável para resolver as relações entre os subsistemas. Eles podem precisar de fazer fusão constante de seu código e realizar testes para ter qualquer coisa certeza que eles não têm quebrado. Às vezes, um dos a equipe precisa gastar um tempo considerável apenas para implementar alguns requisitos que são necessários para o outro time. Há também compromete a ser feita. É uma coisa para desenvolver de forma independente, para escolher os conceitos e associações livremente, e outra coisa é ter certeza de que seus ataques modelo no quadro de um outro sistema. Podemos precisar de alterar o modelo só para fazê-lo funcionar com o outro subsistema. Ou nós pode precisar de introduzir camadas especiais que realizam traduções entre os dois subsistemas. Há momentos em que temos que fazer isso, mas há momentos em que podemos percorrer um caminho diferente. Precisamos avaliar atentamente os benefícios da integração e usá-lo apenas se houver valor real em fazê-lo. Se chegar à conclusão de que a integração é mais problema do que vale a pena, então devemos ir os caminhos separados.

Os caminhos separados endereços padrão o caso quando um aplicativo corporativo pode ser composta de várias aplicações menores que pouco ou nada têm em comum a partir de uma perspectiva de modelagem. Não há um único conjunto de requisitos, e

a partir da perspectiva do utilizador esta é uma aplicação, mas a partir de um ponto de vista modelação e desenho pode feito usando modelos separados com implementações distintas. Devemos olhar para os requisitos e ver se eles podem ser divididos em dois ou mais conjuntos que não têm muito em comum. Se isso pode ser feito, então nós podemos criar separado Bounded Contextos e fazer a modelagem de forma independente. Isto tem a vantagem de ter a liberdade de escolher as tecnologias utilizadas para

implementação. o

aplicações estamos criando podem compartilhar uma GUI fina comum que funciona como um portal com links ou botões usados para acessar cada aplicativo. Isso é uma integração menor que tem a ver com a organização das aplicações, ao invés do modelo por trás deles.

Antes de ir em Separate Ways precisamos ter certeza de que não vai voltar a um sistema integrado. Modelos desenvolvidos de forma independente são muito difíceis de integrar. Eles têm tão pouco em comum que é apenas não vale a pena fazê-lo.

Open Service Anfitrião

Quando tentamos integrar dois subsistemas, que geralmente criam uma camada de tradução entre eles. Esta camada age como um amortecedor entre o subsistema de cliente eo subsistema externo queremos integrar. Esta camada pode ser um consistente, dependendo da complexidade dos relacionamentos e como o subsistema externo foi projetado. Se o subsistema externo acaba por ser não utilizado por um subsistema de cliente, mas por vários outros, precisamos criar camadas de tradução para todos eles. Todas essas camadas irá repetir a mesma tarefa de tradução, e conterá um código semelhante.

Quando um subsistema tem de ser integrado com muitos outros, personalizando um tradutor para cada um pode atolar a equipe. Há mais e mais para manter, e cada vez mais com que se preocupar quando são feitas alterações.

A solução é ver o subsistema externo como um prestador de serviços. Se nós podemos envolver um conjunto de serviços em torno dele, então todos os outros subsistemas irá aceder a estes serviços, e não vamos precisar de qualquer camada de tradução. A dificuldade é que cada subsistema pode precisar interagir de uma maneira específica com o subsistema externo, e para criar um conjunto coerente de serviços pode ser problemático.

Definir um protocolo que dá acesso ao seu subsistema como um conjunto de serviços. Abra o protocolo para que todos os que precisam integrar com você pode usá-lo. Melhorar e expandir o protocolo para lidar com novas exigências de integração, exceto quando uma única equipe tem necessidades idiossincráticas. Em seguida, use um one-off tradutor para aumentar o protocolo para esse caso especial para que o protocolo compartilhada pode permanecer simples e coerente.

Destilação

A destilação é o processo de separação das substâncias que compõem uma mistura. O objectivo da destilação é a extrair uma determinada substância a partir da mistura. Durante o processo de destilação, alguns subprodutos podem ser obtidos, e eles também podem ser de interesse.

Um grande domínio tem um modelo grande, mesmo depois temos refinado-lo e criou muitas abstrações. Ele pode permanecer grande mesmo depois de muitas refatorações. Em situações como esta, pode ser hora para uma destilação. A idéia é definir um domínio do núcleo que representa a essência do domínio. Os subprodutos do processo de destilação será subdomínios genérica que compreendem as outras partes do domínio.

Na concepção de um sistema de grande porte, há tantos componentes que contribuem, todos complicado e todos absolutamente necessárias para o sucesso, que a essência do modelo de domínio, o ativo de negócios real, pode ser obscurecida e negligenciada.

Ao trabalhar com um modelo grande, devemos tentar separar os conceitos essenciais de genéricos. No início, deu o exemplo de um sistema de monitoramento de tráfego aéreo. Nós disse que um plano de vôo contém o projetado Rota do avião deve seguir. A rota parece ser um conceito sempre presente neste sistema. Na verdade, este conceito é um genérico, e não um fator essencial. O conceito de rota é usada em muitos domínios, e um modelo genérico pode ser projetado para descrevê-lo. A essência do monitoramento de tráfego aéreo está em outro lugar. O sistema de monitoramento sabe a rota que o avião deveria seguir, mas também recebe entrada de uma rede de radares de rastreamento do avião no ar. Isto mostra dados do caminho real seguido pelo plano, e é geralmente diferente do previsto. O sistema terá de calcular a trajetória do avião com base em seus parâmetros de vôo atuais, características de avião e tempo. A trajetória é um caminho dimensional quatro que descreve completamente a rota que o avião vai viajar no tempo. A trajetória pode ser computado para o próximo par de minutos, para os próximos dezenas de minutos ou para o próximo par de horas. Cada um desses cálculos ajudar o processo de tomada de decisão. Todo o propósito de calcular a trajetória do avião é para ver se há alguma chance para esse caminho do avião para cruzar outro. Nas imediações dos aeroportos, durante a descolagem ea aterragem, muitos aviões estão circulando no ar ou fazer manobras. Se um avião se afasta de sua rota planejada, há uma possibilidade elevada de um acidente de avião a ocorrer. O sistema de monitoramento de tráfego aéreo irá calcular as trajetórias de aviões, e emitirá um alerta se houver uma possibilidade de um cruzamento. Os controladores de tráfego aéreo terá de tomar decisões rápidas, dirigindo os aviões, a fim de evitar a colisão. Quando os planos estão mais afastados, as trajectórias são calculados por longos períodos de tempo, e não há mais tempo para a reacção. O módulo que sintetiza a trajetória de avião a partir dos dados disponíveis é o coração do sistema de negócios aqui. Isto deve ser marcado como o domínio do núcleo. O modelo de roteamento é mais de um domínio genérico. as trajectórias são calculados por longos períodos de tempo, e não há mais tempo para a reacção. O módulo que sintetiza a trajetória de avião a partir dos dados disponíveis é o coração do sistema de negócios aqui. Isto deve ser marcado como o domínio do núcleo. O modelo de roteamento é mais de um domínio genérico. as trajectórias são calculados por longos períodos de tempo, e não há mais tempo para a reacção. O módulo que sintetiza a trajetória de avião a partir dos dados disponíveis é o coração do sistema de negócios aqui. Isto deve ser marcado como o domínio do núcleo. O modelo de roteamento é mais de um domínio genérico.

O Núcleo de domínio de um sistema depende de como olhamos para o sistema. Um sistema de roteamento simples vai ver a rota e suas dependências como central para o desenho. O monitoramento de tráfego aéreo

sistema considerará a rota como um subdomínio genérico. O Núcleo de domínio de uma aplicação pode tornar-se um subdomínio genérico de outro. É importante identificar corretamente o Core, e determinar as relações que tem com outras partes do modelo.

Ferva o modelo para baixo. Encontre o domínio do núcleo e fornecer um meio de distinguir-se facilmente a partir da massa de modelo e código de suporte. Enfatizar os conceitos mais valiosos e especializados. Faça o Core pequena.

Aplicar o seu talento superior para o domínio do núcleo, e recrutar em conformidade. Passe o esforço no núcleo de encontrar um modelo de profundidade e desenvolver um projeto-suficiente flexível para cumprir a visão do sistema. Justificar o investimento em qualquer outra parte pela forma como ele suporta o Core destilada.

É importante atribuir os melhores desenvolvedores para a tarefa de implementar o domínio do núcleo. Os desenvolvedores geralmente tendem a como tecnologias, para aprender a melhor e mais recente da linguagem, sendo dirigido mais para a infra-estrutura, em vez de lógica de negócios. A lógica de negócios de um domínio parece ser chato para eles, e de pouca recompensa. Afinal, qual é o ponto em detalhes sobre trajetórias de avião aprendizagem? Quando o projeto é feito, todo esse conhecimento se torna uma coisa do passado com muito pouco benefício. Mas a lógica de negócios do domínio é o coração do domínio. Erros na concepção e implementação do núcleo pode levar a todo o abandono do projeto. Se a lógica do núcleo de negócios não fazer o seu trabalho, todos os sinos e assobios tecnológicos equivale a nada.

Um domínio do núcleo não é geralmente criado em uma etapa final. Há um processo de refinamento e refatorações sucessivas são necessários antes que o núcleo emerge mais claramente. Precisamos impor o Core como peça central do projeto, e delimitar suas fronteiras. Nós também precisamos repensar os outros elementos do modelo em relação com o novo Core. Eles podem precisar de ser reformulado também, algumas funcionalidades podem precisar de ser mudado.

Algumas partes do modelo de complexidade add sem capturar ou comunicar conhecimento especializado. Qualquer coisa estranha faz com que o domínio do núcleo mais difícil de discernir e compreender. O modelo entope com os princípios gerais todos conhecem ou detalhes que pertencem a especialidades que não são seu foco principal, mas desempenham um papel de apoio. Ainda assim, no entanto genérico, esses outros elementos são essenciais para o funcionamento do sistema e a expressão completa do modelo.

Identificar subdomínios coesivos que não são a motivação para o seu projeto. Fator fora modelos genéricos desses subdomínios e colocá-los em módulos separados. Não deixar nenhum rastro de suas especialidades neles.

Depois de terem sido separados, dar o seu desenvolvimento contínuo prioridade menor do que o domínio do núcleo, e evitar a atribuição de seus desenvolvedores do núcleo para as tarefas (porque eles vão ganhar pouco conhecimento de domínio a partir deles). Considere também off-theshelf soluções ou modelos publicados para

estes genérico

Subdomínios.

Cada domínio utiliza conceitos que são utilizados por outros domínios. Dinheiro e seus conceitos relacionados, como moeda ea taxa de câmbio pode ser incluído em sistemas diferentes. Traçando um outro conceito amplamente utilizado, que é muito complexo em si mesmo, mas ele pode ser usado em muitas aplicações.

Existem maneiras diferentes de implementar um subdomínio genérico:

1. **Off-the-shelf Solution.** Este tem a vantagem de
   tendo toda a solução já feito por outra pessoa. Há ainda uma curva de aprendizagem associada com ele, e uma tal solução apresenta alguns dependências. Se o código é buggy, você tem que esperar para ser corrigido. Você também precisa usar certos compiladores e versões de bibliotecas. A integração não é tão facilmente realizado em comparação com um sistema in-house.

2. **Terceirização.** A concepção e implementação é dado a
   outra equipe, provavelmente de outra empresa. Isto permite-lhe concentrar-se no domínio do núcleo, e tira o fardo

de outro domínio para lidar com eles. Há ainda o inconveniente de integrar o código terceirizado. A interface usada para se comunicar com as necessidades de subdomínio a ser definida e comunicada à outra equipe.

3. **Existente Model.** Uma solução prática é usar um já criado modelo. Existem alguns livros que publicaram padrões de análise, e eles podem ser usados como inspiração para os nossos subdomínios. Pode não ser possível copiar o literam padrões anúncio, mas muitos deles podem ser usados com pequenas mudanças.

4. **In-House Implantação.** Esta solução tem a vantagem de alcançar o melhor nível de integração. Ele faz um esforço extra média, incluindo a carga de manutenção.

Versão online grátis.

Apoiar este trabalho, comprar a cópia impressa:
http://www.infoq.com/minibooks/domain-drivendesign-quickly

6

DDD importa hoje: Uma entrevista com Eric Evans

Eu nfoQ.com entrevistas Domain Driven Projeto fundador Eric Evans colocar Domain Driven Design, em um contexto moderno:

Por que é DDD tão importante hoje como sempre?

Fundamentalmente, DDD é o princípio que deve estar centrada sobre as questões profundas do domínio nossos usuários estão envolvidos em, que a melhor parte da nossa mente deve ser dedicado à compreensão desse domínio, e colaborando com especialistas nesse domínio lutar-lo em uma forma conceitual que podemos usar para construir poderoso, software flexível.

Este é um princípio que não vai sair de moda. Aplica-se sempre que estamos operando em um domínio intricado complexo.

The long-term trend is toward applying software to more and more complex problems deeper and deeper into the heart of these businesses. It seems to me this trend was interrupted for a few years, as the web burst upon us. Attention was diverted away from rich logic and deep solutions, because there was so much value in just getting data onto the web, along with very simple behavior. There was a lot of that to do, and just doing simple things on the web was difficult for a while, so that absorbed all the development effort.

Mas agora que nível básico de uso da web tem sido amplamente assimilada, e os projetos estão começando a ficar mais ambicioso novamente sobre a lógica de negócios.

Muito recentemente, plataformas de desenvolvimento web começaram a amadurecer o suficiente para tornar o desenvolvimento web bastante produtivo para DDD, e há uma série de tendências positivas. Por exemplo, SOA, quando é bem utilizado, fornece-nos uma forma muito útil de isolar o domínio.

Enquanto isso, processos ágeis tiveram influência suficiente para que a maioria dos projetos têm agora pelo menos uma intenção de iteração, trabalhando em estreita colaboração com parceiros de negócios, aplicando integração contínua, e trabalhar em um ambiente de alta comunicação.

Então DDD parece ser cada vez mais importante para o futuro previsível, e algumas fundações parecem ser estabelecidas.

plataformas de tecnologia (Java, .NET, Ruby, outros) são em constante evolução. Como é Domain Driven projeto se encaixa?

Na verdade, novas tecnologias e processos devem ser julgados em se eles apoiar as equipes se concentrem no seu domínio, em vez de distraí-los a partir dele. DDD não é específico para uma plataforma de tecnologia, mas algumas plataformas dar formas mais expressivas da criação de lógica de negócios, e algumas plataformas têm a desordem menos distração. No que diz respeito à tarde, os últimos anos indicam uma direção esperançoso, particularmente após os terríveis final de 1990.

Java tem sido a escolha padrão dos últimos anos, e quanto a expressividade, é típico das linguagens orientadas a objeto. Como para distrair a desordem, o idioma base não é muito ruim. Ele tem coleta de lixo, o que, na prática, acaba por ser essencial. (Em contraste com C ++, que apenas exigiu muita atenção a detalhes de baixo nível.) A sintaxe Java tem alguma desordem, mas Plain Old Java Objects (POJOs) ainda podem ser lidos. E alguns dos Java 5 inovações sintaxe ajudar a legibilidade.

Mas de volta quando os quadros J2EE primeiro saiu, totalmente enterrado que expressividade básica sob montanhas de código do framework. Após as primeiras convenções (como EJB casa,

get / set assessores prefixados para todas as variáveis, etc.) produziu objetos terríveis. As ferramentas foram tão complicado que absorveu toda a capacidade das equipes de desenvolvimento apenas para fazê-lo funcionar. E foi tão difícil mudar objetos, uma vez que a enorme confusão de código gerado e XML tinha sido cuspido, que as pessoas simplesmente não alterá-los muito. Esta foi uma plataforma que fez domínio eficaz modelagem quase impossível.

Combine isso com o imperativo de produzir Web UIs mediada por http e html (que não foram projetados para esse fim) utilizando ferramentas bastante primitivo, de primeira geração. Durante esse período, criar e manter uma UI decente tornou-se tão difícil que pouca atenção foi deixado para o projeto de funcionalidade interna complexa. Ironicamente, no mesmo momento em que a tecnologia objeto assumiu, modelagem e design sofisticado levou um golpe pesado.

A situação foi semelhante na plataforma .Net, com algumas questões a ser tratada um pouco melhor, e outros um pouco piores.

Esse foi um período desanimador, mas as tendências transformaram nos últimos quatro anos. Primeiro, olhando para Java, houve uma confluência de uma nova sofisticação na comunidade sobre como usar estruturas seletivamente, e uma mistura variada de novas estruturas (principalmente open-source) que são incrementalmente melhorando. Frameworks como Hibernate e trabalhos específicos punho Spring que J2EE tentaram endereço, mas de uma forma muito mais leve. Abordagens como AJAX que tentar resolver o problema UI, de uma forma menos trabalhosa. E projetos são muito mais espertos agora sobre escolher e escolher os elementos do J2EE que lhes dão valor e misturando em alguns desses elementos mais novos. O termo foi cunhado POJO durante esta época.

O resultado é uma diminuição gradual, mas perceptível no esforço técnico dos projectos, e uma nítida melhoria no isolamento da lógica de negócios do resto do sistema de modo que ele pode ser escrito em termos de POJOs. Esta não produz automaticamente um Domain-Driven Design, mas torna-se uma oportunidade realista.

Esse é o mundo Java. Então você tem os recém-chegados como Ruby. Ruby tem uma sintaxe muito expressivo, e, a este nível básico deve ser uma muito boa idioma para DDD (embora eu não tenha ouvido falar de muita utilidade real nisso nesses tipos de aplicações ainda). Rails tem gerado muita emoção porque finalmente parece fazer criação de Web UIs tão fácil como UIs estavam de volta no início de 1990, antes da Web. Agora, essa capacidade tem principalmente sido aplicada para a construção de alguns do grande número de aplicações Web que não têm muita riqueza de domínio por trás deles, uma vez que mesmo estes têm sido dolorosamente difícil no passado. Mas minha esperança é que, como a parte de implementação UI do problema é reduzida, que as pessoas vão ver isso como uma oportunidade para se concentrar mais de sua atenção no domínio. Se o uso de Ruby já começa a ir nessa direção, Eu acho que poderia proporcionar uma excelente plataforma para DDD. (Algumas infra-estrutura peças provavelmente teria de ser preenchido.)

Mais para fora na ponta são os esforços na área de linguagens específicas de domínio (DSLs), que eu há muito tempo acreditava poderia ser o próximo grande passo para DDD. Até à data, ainda não temos uma ferramenta que realmente nos dá o que precisamos. Mas as pessoas estão experimentando mais do que nunca nesta área, e isso me deixa esperançoso.

Agora, tanto quanto eu posso dizer, a maioria das pessoas que tentam aplicar DDD estão trabalhando em Java ou .Net, com alguns em Smalltalk. Por isso, é a tendência positiva no mundo Java que está a ter o efeito imediato.

O que está acontecendo na comunidade DDD desde que você tenha escrito o seu livro?

Uma coisa que me excita é quando as pessoas tomam os princípios que eu falei no meu livro e usá-los de maneiras que eu nunca esperava. Um exemplo é o uso do design estratégico da Statoil, a companhia petrolífera nacional norueguês. Os arquitetos não escreveu uma experiência relatório sobre isto. (Você pode ler isto em http://domaindrivendesign.org/articles/.)

Entre outras coisas, eles levaram mapeamento contexto e aplicou a avaliação de software off-the-shelf em construção vs. decisões de compra.

Como um exemplo bem diferente, alguns de nós têm vindo a explorar algumas questões através do desenvolvimento de uma biblioteca de código Java de alguns objetos de domínio fundamentais necessários para muitos projetos. As pessoas podem verificar isso em:

http://timeandmoney.domainlanguage.com

Nós temos vindo a explorar, por exemplo, o quão longe podemos empurrar a idéia de uma linguagem fluente, de domínio específico, enquanto ainda implementar objetos em Java.

Um pouco está acontecendo lá fora. Eu sempre aprecio quando as pessoas em contato comigo para me dizer sobre o que estão fazendo.

Você tem algum conselho para quem está tentando aprender DDD hoje?

Leia o meu livro! ;-) Além disso, tente usar timeandmoney em seu projeto. Um dos nossos objetivos originais era fornecer um bom exemplo de que as pessoas podem aprender a partir de usá-lo.

Um aspecto a ter em mente é que DDD é em grande parte algo equipes fazem, assim você pode ter que ser um evangelista. Realisticamente, você pode querer ir busca de um projeto onde eles estão fazendo um esforço para fazer isso.

Tenha em mente algumas das armadilhas de modelagem de domínio:

1) Fique hands-on. Modeladores precisa código.

2) Concentre-se em cenários concretos. O pensamento abstrato tem que ser ancorado em casos concretos.

3) Não tente aplicar DDD para tudo. Desenhe um mapa de contexto e decidir sobre onde você vai fazer um esforço para DDD e onde você não vai. E então não se preocupe com isso fora desses limites.

4) experimentar muito e esperar para fazer um monte de erros. A modelagem é um processo criativo.

Sobre Eric Evans

Eric Evans é o autor de "Domain-Driven Design: Roubada Complexidade em Software", Addison-Wesley 2004.

Desde o início de 1990, ele trabalhou em muitos projetos em desenvolvimento de grandes sistemas de negócios com objetos com muitas abordagens diferentes e muitos resultados diferentes. O livro é uma síntese dessa experiência. Apresenta um sistema de técnicas de modelagem e design que equipes bem sucedidas têm usado para sistemas de software complexos alinhar-se com as necessidades do negócio e para manter os projetos ágeis como sistemas de crescer grandes.

Eric agora lidera "Language Domain", um grupo de consultoria que treinadores e equipes trens aplicação Domain-Driven Design, ajudando-os a fazer o seu trabalho de desenvolvimento mais produtivo e mais valioso para os seus negócios.

Anúncio em nome de Eric Evans

Nossos serviços

Nós ajudamos projetos de software ambiciosos perceber o potencial do Domain-Driven Design e processos ágeis.

Para fazer a modelagem de domínio e projetar muito trabalho para um projeto requer que o alto-nível e projeto detalhado vêm juntos. É por isso que nós oferecemos uma combinação de serviços que podem realmente ter um processo de design orientado a domínio do chão.

Os nossos cursos de formação e hands-on mentores reforçar as competências básicas da equipe na modelação e a implantação de uma eficaz implementação. Nossos treinadores concentrar o esforço da equipe e resolver as falhas de processo que ficam no caminho de projetar o sistema mais significativo para o negócio. Nossos consultores de design estratégico assumir esses problemas que afetam a trajetória de todo o projecto, facilitando o desenvolvimento de uma grande figura que apóia o desenvolvimento e novilhos o projeto para os objetivos da organização.

Iniciar com a avaliação

A avaliação nós fornecemos vai lhe dar perspectiva, bem como recomendações concretas. Vamos esclarecer onde você está agora, onde você quer ir, e começar a desenhar um roteiro de como chegar a esse objetivo.

Para programar uma avaliação

Para as taxas, horários e mais informações, ligue para 415-401-7020 ou escreva para info @ domainlanguage.com.

www.domainlanguage.com