Domain-Driven Design Referência

Sumário de Padrões e Definições

Eric Evans
Domain Language, Inc.

Tradução de Ricardo Pereira Dias

(http://www.ricardopdias.com.br)

Conteúdo

* Novo termo introduzido deste o livro de 2004.

Agradecimentos

Já se passaram dez anos desde a publicação do meu livro, Domain-Driven Design, Tackling Complexity in the Heart of Software (*Design Orientado por Domínio, Atacando as Complexidades no Coração do Software*) ou "O Grande Livro Azul", como algumas pessoas começaram a chamá-lo. Nessa década, os fundamentos discutidos no livro não mudaram muito, mas muita coisa mudou sobre como construímos software. O DDD permaneceu relevante porque pessoas inteligentes e inovadoras balançaram as coisas repetidamente. Eu quero agradecer a essas pessoas.

Deixe-me começar com Greg Young, Udi Dahan e as pessoas inspiradas por eles, pelo CQRS e pelo Event Sourcing. Estas são agora opções bastante mainstream para a arquitetura de um sistema DDD. Este foi o primeiro grande sucesso da visão estreita da arquitetura herdada da virada do século.

Desde então, tem havido várias tecnologias e estruturas interessantes que tinham como objetivo tornar o DDD mais concreto na implementação (entre outros objetivos de seus projetistas), com vários graus de sucesso. Estes incluem Qi4J, Naked Objects, Roo e outros. Esses experimentos têm grande valor, mesmo quando não recebem ampla adoção.

Também quero agradecer às pessoas e comunidades que revolucionaram nosso ecossistema técnico nos últimos anos, de forma a tornar o DDD muito mais divertido e prático. A maioria dessas pessoas tem pouco interesse em DDD, mas seu trabalho nos beneficiou tremendamente. Estou particularmente pensando na liberdade que o NoSQL está trazendo para nós, na redução de ruído-sintático de novas linguagens de programação (algumas funcionais) e no impulso incessante em direção a estruturas técnicas mais leves e a bibliotecas desacopladas e não intrusivas. A tecnologia de uma década atrás era complicada e pesada, tornando o DDD ainda mais difícil. Há novas tecnologias ruins também, é claro, mas a tendência é boa. Por isso, agradeço de forma especial a todos aqueles que contribuíram para essa tendência, embora você nunca tenha ouvido falar de DDD.

Em seguida, quero agradecer aqueles que escreveram livros sobre o DDD. O primeiro livro sobre DDD depois do meu foi de Jimmy Nilsson. Com um livro, você tem um livro. Com dois, você tem um tópico. Em seguida, a InfoQ publicou o *"DDD Quickly"*, que, devido à sua brevidade, sua disponibilidade como download gratuito e o alcance do InfoQ, deu a muitas pessoas o primeiro gosto do tema. Os anos se passaram, e havia muitos artigos de blog valiosos e outros escritos curtos. Havia também livros especializados como o *"DDD with Naked Objects"*. E eu particularmente quero agradecer o indispensável Martin Fowler, que ajudou a comunicar claramente os conceitos de DDD, além de fornecer a documentação definitiva de padrões emergentes. No ano passado, Vaughn Vernon publicou o livro mais ambicioso desde o

meu, "*Implementing Domain-Driven Design*" (que alguns chamam de "O Grande Livro Vermellho").

Sinto uma espécie de desespero com a inevitabilidade de deixar de fora muitas pessoas que fizeram contribuições significativas, e lamento sinceramente por isso. Deixe-me pelo menos dar um caloroso obrigado às pessoas que empurraram DDD para a visão pública e para aqueles que empurraram DDD para cantos tranquilos das organizações. Demora milhares de campionatos para uma filosofia de software ter algum impacto.

Embora esta seja a primeira edição impressa da Referência DDD, a primeira forma na verdade antecede a publicação do meu livro de 2004. Seguindo o conselho de Ralph Johnson, extraí os breves resumos de cada padrão e os usei em workshops, com cada padrão sendo lido em voz alta pelos participantes, seguido de discussão. Eu usei esses documentos em aulas de treinamento por vários anos.

Então, alguns anos depois de meu livro ter sido publicado, Ward Cunningham, como parte de seu trabalho em um repositório de padrões, propôs a alguns autores que colocássemos pequenos resumos de nossos padrões no Creative Commons. Martin Fowler e eu, com o acordo da Pearson Education, nossa editora, fizemos exatamente isso, o que abriu possibilidades para trabalhos derivados, como este.

Obrigado a todos.

Eric Evans, Junho de 2014

Sobre a Tradução

Um assunto da importância do Domain-Driven Design e com tamanha complexidade de termos não é tão fácil de se estudar diretamente em inglês. Na época, a escassês de publicações e o preço do livro "*Domain-Driven Design, Tackling Complexity in the Heart of Software*" não me permitia comprar a obra em português.

Em 2015, Eric Evans, o autor do DDD, publicou um resumo chamado "*DDD Reference - Definitions and Pattern Summaries*" sob a licensa Creative Commons Attribution 4, que permite o compartilhamento e adaptação do conteúdo.

Nos últimos anos, a fim de assimilar os conceitos contidos no DDD, fui fazendo traduções esporádicas do livreto na medida que ia aprofundando no assunto. Conforme a tradução foi ganhando forma, decidi concluí-la para ajudar outras pessoas.

Espero que esta tradução possa ser de muita utilidade e contribua para um aumento do desejo de conhecimento daqueles que decidirem iniciar suas jornadas nesta preciosa descoberta. Caso queira acessar o site do Evans em http://domainlanguage.com/ddd/reference, você poderá obter o livreto original em inglês.

Definições

Domínio

Uma esfera de conhecimento, influência ou atividade. A área de assunto para a qual o usuário aplica um programa é o domínio do software.

Modelo

Um sistema de abstrações que descreve aspectos selecionados de um domínio e pode ser usado para resolver problemas relacionados a esse domínio.

Linguagem Onipresente

Uma linguagem estruturada em torno do modelo do domínio e usada por todos os membros da equipe dentro de um contexto delimitado para conectar todas as atividades da equipe ao software.

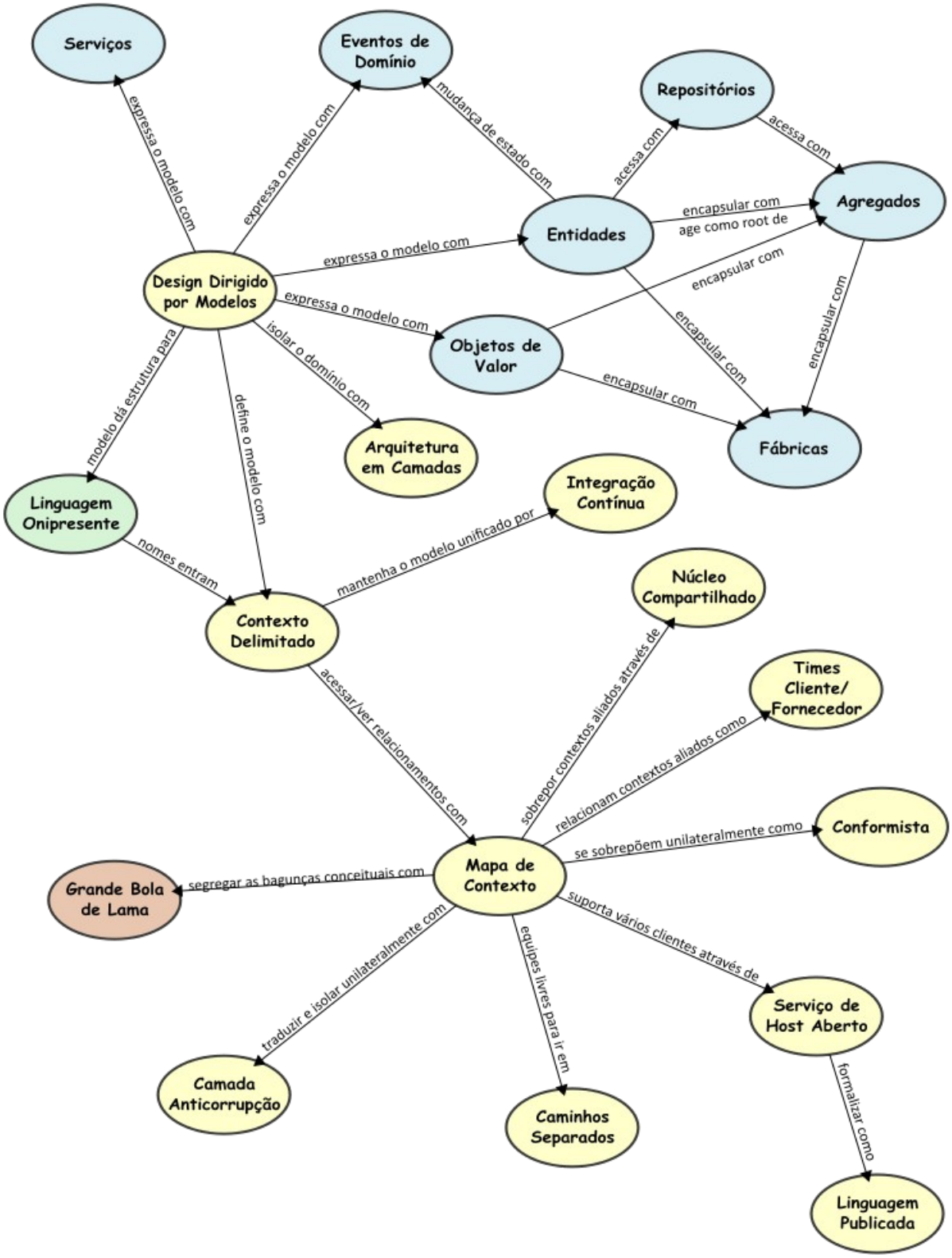
Contexto

A configuração na qual aparece uma palavra ou declaração que determina seu significado. Declarações sobre um modelo só podem ser entendidas em um contexto.

Contexto Delimitado

Uma descrição de um limite (normalmente um subsistema ou o trabalho de uma equipe específica) dentro do qual um modelo específico é definido e aplicável.

Visão Geral da Linguagem de Padrões



1. Colocando o modelo em ação

O *Domain-Driven Design* é uma abordagem para o desenvolvimento de softwares complexos onde buscamos:

1. Nos concentrar no domínio principal;
2. Explorar modelos em uma colaboração criativa entre os profissionais de domínio e os profissionais de software;
3. Falar numa *Linguagem Onipresente* dentro de um contexto limitado explicitamente.

Este resumo de três pontos do *DDD* depende da definição dos termos definidos neste livreto.

Muitos projetos fazem o trabalho de modelagem e, no final, não conseguem obter muitos benefícios reais. Os padrões de *DDD* destilam práticas bem-sucedidas de projetos em que benefícios dramáticos são provenientes da modelagem. Juntos, eles estabelecem uma abordagem bastante diferente para modelagem e desenvolvimento de software que vai de detalhes finos a visão de alto nível. Convenções de modelagem rigorosas devem ser equilibradas com a livre exploração de modelos em colaboração com pessoas não técnicas. Para alcançar sucesso, as táticas e as estratégias devem ser combinadas, por esse motivo, o *DDD* aborda um design tático e estratégico.

Contexto Delimitado

Vários modelos estão em jogo em qualquer projeto grande. Eles surgem por muitas razões. Dois subsistemas geralmente atendem comunidades de usuários muito diferentes, com tarefas diferentes, nos quais diferentes modelos podem ser úteis. Equipes que trabalham independentemente podem resolver o mesmo problema de maneiras diferentes por falta de comunicação. O conjunto de ferramentas também pode ser diferente, o que significa que o código do programa não pode ser compartilhado.

Vários modelos são inevitáveis, mas quando o código baseado em modelos distintos é combinado, o software se torna problemático, pouco confiável e difícil de entender. A comunicação entre os membros da equipe fica confusa. Muitas vezes não está claro em que contexto um modelo não deve ser aplicado.

Expressões de modelo, como qualquer outra frase, só têm significado no contexto.

Assim sendo:

Defina explicitamente o contexto no qual um modelo se aplica. Explicitamente estabeleça limites em termos de organização da equipe, uso em partes específicas do aplicativo e manifestações físicas, como bases de código e esquemas de banco de dados. Aplique a *Integração Contínua* para manter os conceitos e os termos do modelo estritamente consistentes dentro desses limites, mas não se distraia ou confunda com problemas externos. Padronize um único processo de desenvolvimento dentro do contexto, que não precisa ser usado em outro lugar.

Linguagem Onipresente

> Pois primeiro você escreve uma frase,
> Depois corta daqui e corta dali;
> Mistura e separa os pedacinhos
> Sem saber onde vão cair:
> No fim, para a ordem da frase
> Não se está nem aí.
>
> —Lewis Carroll, “Poeta Fit, Non Nascitur”

Para criar um design flexível e rico em conhecimento, é necessário que a equipe tenha uma linguagem versátil e compartilhada, além de uma experimentação ativa de linguagem que raramente acontece em projetos de software.

Dentro de um único *Contexto Delimitado*, a linguagem pode ser fraturada de maneiras que prejudicam os esforços ao aplicar uma modelagem sofisticada. Se o modelo for usado apenas para desenhar diagramas *UML* para os membros técnicos da equipe, isso não estará contribuindo para a colaboração criativa no coração do *DDD*.

Os especialistas de domínio usam seu jargão enquanto os membros da equipe técnica têm seu próprio idioma estabelecido para discutir o domínio em termos de design. A terminologia das discussões do dia-a-dia é desconectada da terminologia embutida no código (que em última análise, é o produto mais importante de um projeto de software). Além disso, a mesma pessoa usa linguagens diferentes na fala e na escrita, de modo que as expressões mais incisivas do domínio geralmente emergem de uma forma transitória que nunca é capturada no código ou mesmo na escrita.

A tradução dificulta a comunicação e faz com que o conhecimento seja anêmico.

No entanto, nenhum desses dialetos pode ser uma linguagem comum, porque nenhum atende a todas as necessidades.

Especialistas em domínio devem se opor a termos que sejam inadequados para as estruturas ou inadequados para transmitir a compreensão do domínio; os desenvolvedores devem observar a ambigüidade ou inconsistência que atrapalhará o design.

Brinque com o modelo enquanto fala sobre o sistema. Descreva os cenários em voz alta usando os elementos e as interações do modelo, combinando conceitos de maneiras

permitidas pelo modelo. Encontre formas mais fáceis de falar o que você precisa dizer e, em seguida, leve essas novas ideias de volta aos diagramas e ao código.

Com uma *Linguagem Onipresente*, o modelo não é apenas um artefato de design. Ele se torna parte integrante de tudo que os desenvolvedores e especialistas em domínio fazem juntos.

Assim sendo:

Use o modelo como a espinha dorsal de uma linguagem. Comprometa o time a exercitar essa linguagem implacavelmente em toda a comunicação dentro da equipe e no código. Dentro de um *Contexto Delimitado*, use a mesma linguagem em diagramas, escrita e especialmente na fala.

Reconheça que uma mudança na linguagem é uma mudança no modelo.

Esgote as dificuldades experimentando expressões alternativas, que refletem modelos alternativos. Em seguida, refatore o código, renomeando classes, métodos e módulos para se adequar ao novo modelo. Resolva a confusão sobre os termos na conversa, de forma que chegue a uma concordância com o significado das palavras comuns.

Integração Contínua

Uma vez definido um contexto delimitado, devemos mantê-lo sadio.

Quando várias pessoas estão trabalhando no mesmo *Contexto Delimitado*, há uma forte tendência do modelo se fragmentar. Quanto maior a equipe, maior o problema, mas apenas três ou quatro pessoas podem encontrar sérios problemas. No entanto, dividir o sistema em contextos cada vez menores acaba perdendo um valioso nível de integração e coerência.

Assim sendo:

Estabeleça um processo para mesclar todo o código e outros artefatos de implementação com frequência, usando testes automatizados para sinalizar rapidamente a fragmentação. Exerça incansavelmente a *Linguagem Onipresente* para elaborar uma visão compartilhada do modelo à medida que os conceitos evoluam na cabeça das diferentes pessoas.

Design Dirigido por Modelos

Relacionar firmemente o código a um modelo subjacente fornece o significado do código e torna o modelo relevante.

Se o design, ou alguma parte central dele, não for mapeada pelo modelo do domínio, esse modelo é de pouco valor, e a exatidão do software é suspeita. Ao mesmo tempo, mapeamentos complexos entre modelos e funções de design são difíceis de entender e, na prática, impossíveis de manter à medida que o design muda. Uma divisão mortal se abre entre a análise e o projeto, de modo que a percepção obtida em cada uma dessas atividades não se sustente na outra.

Extraia do modelo a terminologia usada no projeto e a atribuição básica de responsabilidades. O código se torna uma expressão do modelo, portanto, uma alteração no código deve ser uma alteração no modelo. Seu efeito deve repercutir obrigatoriamente pelo restante das atividades do projeto.

Para vincular a implementação servilmente a um modelo, geralmente são necessárias linguagens e ferramentas de desenvolvimento de software que suportem um paradigma de modelagem, como a programação orientada a objetos.

Assim sendo:

Projete uma parte do sistema de software para refletir o modelo do domínio de uma maneira muito literal, para que o mapeamento seja óbvio. Revisite o modelo e modifique-o para ser implementado de forma mais natural no software, mesmo que você procure fazê-lo refletir uma visão mais profunda do domínio. Exija um modelo único que sirva bem a ambos os propósitos, além de apoiar uma *Linguagem Onipresente* fluente.

Modeladores Envolvidos

Se as pessoas que escrevem o código não se sentirem responsáveis pelo modelo, ou não entenderem como fazer o modelo funcionar para um aplicativo, o modelo não terá nada a ver com o software. Se os desenvolvedores não perceberem que a alteração do código altera também o modelo, a refatoração deles enfraquecerá o modelo, em vez de fortalecê-lo.

Em contrapartida, quando um modelador é separado do processo de implementação, ele ou ela nunca adquire (ou perde rapidamente) a sensação sobre as restrições de implementação. A restrição básica do *Design Dirigido por Modelos* (de que o modelo suporta uma implementação efetiva e abstrai as principais informações sobre o domínio) será aos poucos perdida e os modelos resultantes não serão viáveis.

Por fim, o conhecimento e as habilidades dos designers experientes não serão transferidos para outros desenvolvedores se a divisão do trabalho impedir o tipo de colaboração que transmite as sutilezas da codificação de um *Design Digirigo por Modelos*.

Assim sendo:

Qualquer pessoa técnica contribuindo para o modelo deve passar algum tempo em contato com o código, independentemente do papel principal que ele ou ela desempenhe no projeto. Qualquer responsável por alterar o código deve aprender a expressar o modelo por meio do código. Todo desenvolvedor deve estar envolvido em algum nível de discussão sobre o modelo e ter contato com *Especialistas do Domínio*. Aqueles que contribuem de maneiras diferentes devem conscientemente envolver aqueles que tem contato com o código em uma troca dinâmica de ideias-modelo através da *Linguagem Onipresente*.

Refatorando para uma Visão Mais Profunda

Usar um conjunto comprovado de *Blocos de Construção* básicos junto com uma linguagem consistente traz alguma sanidade para o esforço de desenvolvimento. Isso provoca o desafio de realmente encontrar um modelo afiado, que capture as preocupações sutis dos *Especialistas do Domínio* e possa gerar um design prático. Um modelo que se desprende do superficial e capta o essencial é um modelo profundo. Isso deve tornar o software mais sintonizado com a maneira como os *Especialistas do Domínio* pensam e mais responsivo às necessidades do usuário.

Tradicionalmente, a refatoração é descrita como transformações de código com motivações técnicas. A refatoração também pode ser motivada por uma visão do domínio e um refinamento correspondente do modelo ou sua expressão no código.

Modelos de domínio sofisticados raramente se tornam úteis, exceto quando desenvolvidos por meio de um processo iterativo de refatoração, incluindo o envolvimento próximo dos *Especialistas de Domínio* com desenvolvedores interessados em aprender sobre o domínio.

2. Blocos de Construção de um Design Orientado por Modelos

Esses padrões lançam práticas recomendadas de design orientado a objetos à luz do *DDD*. Eles orientam as decisões para esclarecer o modelo e manter o modelo e a implementação alinhados entre si, cada um reforçando a eficácia do outro. A elaboração cuidadosa dos detalhes dos elementos do modelo individual fornece aos desenvolvedores uma plataforma estável a partir da qual é possível explorar modelos e mantê-los em estreita correspondência com a implementação.

Arquitetura em Camadas

Em um programa orientado a objetos, a interface do usuário, o banco de dados e outros códigos de suporte geralmente são gravados diretamente nos objetos de negócios. As outras lógicas do negócio são embitidas no comportamento dos widgets da interface do usuário e nos scripts de banco de dados. Isso acontece porque é a maneira mais fácil de fazer as coisas funcionarem a curto prazo.

Quando o código relacionado ao domínio é distribuído por uma porção tão grande de outros códigos, torna-se extremamente difícil distinguí-los e raciocinar. Alterações superficiais na interface do usuário podem realmente alterar a lógica de negócios. Para alterar uma regra do negócio, pode ser necessário um rastreamento meticuloso do código da interface do usuário, do código do banco de dados ou de outros elementos do programa. A implementação de objetos coerentes dirigidos por modelos torna-se impraticável. Os testes automatizados são desajeitados. Com todas as tecnologias e lógicas envolvidas em cada atividade, um programa deve ser mantido bastante simples, caso contrário, ele se torna impossível de entender.

Assim sendo:

Isole a expressão do modelo do domínio e a lógica de negócios e elimine qualquer dependência na infraestrutura, na interface do usuário ou mesmo na lógica do aplicativo que não seja lógica de negócios. Particione um programa complexo em camadas. Desenvolva um design dentro de cada camada que seja coeso e que dependa apenas das camadas abaixo. Siga os padrões de arquitetura estabelecidos para fornecer um acoplamento flexível às camadas acima. Concentre todo o código relacionado ao modelo do domínio em uma camada e isole-o do código da interface do usuário, do aplicativo e da infraestrutura. Os objetos de domínio, livres da responsabilidade de se exibir, de se armazenar, de gerenciar tarefas do aplicativo, e assim por diante, podem se concentrar em expressar o modelo do domínio. Isso permite que um modelo evolua para se tornar rico e limpo o suficiente para capturar o conhecimento essencial do negócio e colocá-lo para funcionar.

O principal objetivo aqui é o isolamento. Padrões relacionados, como “Arquitetura Hexagonal”, podem funcionar tão bem ou melhor na medida que as expressões do modelo do domínio sejam facilitadas evitando dependências e referências a outras preocupações do sistema.

Entidades

Muitos objetos representam um segmento de continuidade e identidade, passando por um ciclo de vida, embora seus atributos possam mudar.

Alguns objetos não são definidos principalmente por seus atributos. Eles representam uma linha de identidade que atravessa o tempo e, muitas vezes, representações distintas. Às vezes, um objeto como esse pode ser combinado com outros objetos, mesmo que os atributos sejam diferentes. Um objeto deve ser diferenciado de outros objetos, embora eles possam ter os mesmos atributos. Um erro de identidade pode levar à corrupção de dados.

Assim sendo:

Quando um objeto precisa ser distinguido por sua identidade, em vez de seus atributos, faça que isso seja essencial para sua definição do modelo. Mantenha a definição de classe simples e focada na continuidade do ciclo de vida e na identidade.

Defina um meio de distinguir cada objeto, independentemente de sua forma ou histórico. Fique atento aos requisitos que exigem combinação de objetos por atributos. Defina uma operação que garantidamente gere um resultado único para cada objeto, possivelmente anexando um símbolo que seja garantidamente único. Este meio de identificação pode vir de fora, ou pode ser um identificador arbitrário criado pelo e para o sistema, mas deve corresponder às distinções de identidade no modelo.

O modelo deve definir o que significa ser a mesma coisa.

(também conhecido como “Objetos de Referência”)

Objetos de Valor

Existem objetos que descrevem ou calculam alguma característica de uma coisa.

Muitos objetos não possuem nenhuma identidade conceitual.

O rastreamento da identidade nas *Entidades* é essencial, mas anexar uma identidade a outros objetos pode prejudicar o desempenho do sistema, adicionando trabalhos analíticos e atrapalhando o modelo, fazendo com que todos os objetos pareçam iguais. O design de software é uma batalha constante contra a complexidade. Precisamos fazer distinções para que o controle especial só seja aplicado quando necessário.

No entanto, se pensarmos nessa categoria de objeto apenas como ausência de identidade, não teremos acrescentado muito à nossa caixa de ferramentas ou ao nosso vocabulário. Na verdade, esses objetos possuem características próprias e significados próprios para o modelo. Esses são os objetos que descrevem as coisas.

Assim sendo:

Quando você só se importa com os atributos de um elemento do modelo, classifique-o como um *Objeto de Valor*. Faça com que ele expresse o significado dos atributos que ele transmite e dê a ele uma funcionalidade relacionada. Trate o *Objeto de Valor* como imutável. Transforme em Funções todas as operações "Sem efeitos colaterais" que não dependam de nenhum estado mutável. Não dê ao *Objeto de Valor* nenhuma identidade e evite as complexidades de design necessárias para manter *Entidades*.

Eventos de Domínio*

Algo aconteceu que especialistas de domínio se preocupam.

Uma *Entidade* é responsável por rastrear seu estado e as regras que regulam seu ciclo de vida. Mas, se você precisa conhecer as causas reais das mudanças de estado, isso geralmente não é explícito, e pode ser difícil explicar como o sistema ficou da forma que está. As trilhas de auditoria podem permitir o rastreamento, mas geralmente não são adequadas para serem usadas na lógica do próprio programa. Os históricos de mudança de *Entidades* podem permitir o acesso a estados anteriores, mas ignora o significado dessas mudanças, de modo que qualquer manipulação das informações é procedimental e, muitas vezes, enviada para fora da camada de domínio.

Um conjunto distinto, embora relacionado, de problemas surge em sistemas distribuídos. O estado de um sistema distribuído não pode ser mantido completamente consistente em todos os momentos. Mantemos os *Agregados* internamente consistentes em todos os momentos, enquanto fazemos outras alterações de forma assíncrona. À medida que as alterações se propagam pelos nós de uma rede, pode ser difícil resolver várias atualizações que vêm fora de ordem ou de fontes distintas.

Assim sendo:

Modele informações sobre a atividade no domínio como uma série de eventos discretos. Represente cada evento como um objeto de domínio. Estes são distintos dos eventos do sistema que refletem a atividade dentro do próprio software, embora muitas vezes um evento do sistema esteja associado a um *Evento de Domínio*, como parte de uma resposta ao *Evento de Domínio* ou como uma maneira de transportar informações sobre o *Evento de Domínio* para o sistema .

Um *Evento de Domínio* é uma parte completa do modelo do domínio, uma representação de algo que aconteceu no domínio. Ignore a atividade de domínio irrelevante ao tornar explícitos os eventos que os *Especialistas do Domínio* desejam rastrear ou receber notificações ou que estão associados a alterações de estado nos outros objetos de modelo.

Em um sistema distribuído, o estado de uma *Entidade* pode ser inferido a partir dos *Eventos de Domínio* atualmente conhecidos para um determinado nó, permitindo um modelo coerente na ausência de informações completas sobre o sistema como um todo.

Eventos de Domínio são ordinariamente imutáveis, já que são um registro de algo no passado. Além de uma descrição do evento, um *Evento de Domínio* geralmente contém um registro de data e hora para o momento em que o evento ocorreu e a identidade das *Entidades* envolvidas no evento. Além disso, um *Evento de Domínio* geralmente tem um registro de data e hora

separado, indicando quando o evento foi inserido no sistema e a identidade da pessoa que o inseriu. Quando útil, uma identidade para o *Evento de Domínio* pode ser baseada em algum conjunto dessas propriedades. Assim, por exemplo, se duas instâncias do mesmo evento chegarem a um nó, elas poderão ser reconhecidas como iguais.

Serviços

Às vezes, a situação simplesmente não se trata de uma coisa.

Alguns conceitos do domínio não são naturais para serem modelados na forma de objetos. Forçar a funcionalidade do domínio necessária para que ela seja a responsabilidade de uma *Entidade* ou *Objeto de Valor* distorce a definição de um objeto baseado em modelos ou adiciona objetos artificiais sem sentido.

Assim sendo:

Quando um processo ou transformação significativa no domínio não é uma responsabilidade natural de uma *Entidade* ou *Objeto de Valor*, adicione uma operação no modelo como uma interface autônoma declarada como *Serviço*. Defina um contrato de serviço, um conjunto de asserções sobre interações com o *Serviço*. (Veja "asserções") Torne essas asserções participantes da *Linguagem Onipresente* de um *Contexto Delimitado* específico. Dê um nome ao *Serviço*, que também se torne parte da *Linguagem Onipresente*.

Módulos

Todos usam *Módulos*, mas poucas pessoas os tratam como uma parte do modelo, já completamente desenvolvidas. O código é dividido em vários tipos de categorias, desde aspectos da arquitetura técnica até as atribuições de trabalho dos desenvolvedores. Mesmo desenvolvedores que refatoram muito tendem a se contentar com *Módulos* concebidos no início do projeto.

Explicações de acoplamento e coesão tendem a fazer com que eles pareçam uma medida técnica, a ser julgada mecanicamente com base nas distribuições das associações e das interações. No entanto, não se trata apenas de um código sendo dividido em *Módulos*, mas também de conceitos. Há um limite para quantas coisas uma pessoa pode pensar ao mesmo tempo (daí, um baixo acoplamento). Fragmentos incoerentes de ideias são tão difíceis de entender quanto uma sopa de ideias não diferenciadas (daí, uma alta coesão).

Assim sendo:

Escolha *Módulos* que contem a história do sistema e contenham um conjunto coeso de conceitos. Dê os nomes para os *Módulos* de forma que façam parte da *Linguagem Onipresente*. Os *Módulos* fazem parte do modelo e seus nomes devem refletir a visão do domínio.

Isso geralmente gera um baixo acoplamento entre os *Módulos*, mas não procure uma maneira de alterar o modelo para desvendar os conceitos ou um conceito negligenciado que pode ser a base de um módulo que reuniria elementos de maneira significativa. Busque baixo acoplamento no sentido de conceitos que possam ser entendidos e fundamentados de forma independente. Refine o modelo até que ele seja particionado de acordo com os conceitos de domínio de alto nível e o código correspondente também seja dissociado.

(também conhecido como “Pacotes”)

Agregados

É difícil garantir a consistência das alterações feitas em objetos em um modelo com associações complexas. Objetos devem manter seu próprio estado interno consistente, mas podem ser surpreendidos por mudanças em outros objetos que sejam conceitualmente partes integrantes. Esquemas moderados de bloqueio de banco de dados fazem com que vários usuários interfiram desnecessariamente entre si e possam inutilizar um sistema. Problemas semelhantes surgem ao distribuir objetos entre vários servidores ou ao projetar transações assíncronas.

Assim sendo:

Agrupe as *Entidades* e os *Objetos de Valor* em *Agregados* e defina os limites em torno de cada um. Escolha uma *Entidade* para ser a raiz de cada *Agregado* e permita que objetos externos mantenham referências apenas à raiz. Referências transitórias a membros internos podem ser transmitidas para uso dentro de uma única operação. Defina propriedades e invariantes para o agregado como um todo e atribua responsabilidade de imposição à raiz ou a algum mecanismo de estrutura designado.

Use os mesmos limites a *Agregados* para gerenciar transações e distribuição.

Dentro do limite de um *Agregado*, aplique regras de consistência de forma síncrona. Fora dos limites, processe as atualizações de forma assíncrona.

Mantenha um *Agregado* em um servidor. Permita que diferentes *Agregados* sejam distribuídos entre nós.

Quando essas decisões de design não estão sendo bem orientadas pelos limites dos *Agregados*, reconsidere o modelo. O cenário de domínio sugere uma nova visão importante? Tais mudanças geralmente melhoram a expressividade e a flexibilidade do modelo, bem como resolvem os problemas de transação e distribuição.

Repositórios

Consultar o acesso a Agregados expressos na Linguagem Onipresente.

A proliferação do uso de associações cruzadas servem apenas para mostrar que coisas atrapalham o modelo. Em modelos maduros, as consultas geralmente expressam conceitos de domínio. No entanto, as consultas podem causar problemas.

A complexidade técnica de se aplicar rapidamente a maioria dos acessos à infra-estrutura de banco de dados enfraquece o código do *Cliente*, o que leva os desenvolvedores a simplificarem a camada de domínio e, consequentemente, tornarem o modelo irrelevante.

Usar um framework para fazer consultas pode encapsular a maior parte dessa complexidade técnica, permitindo que os desenvolvedores obtenham os dados exatos de que precisam do banco de dados, de maneira mais automatizada ou declarativa, mas isso só resolve parte do problema.

Consultas irrestritas ao banco podem extrair campos específicos de objetos, violar o encapsulamento ou instanciar alguns objetos específicos do interior de um *Agregado*, ocultando a raiz agregada e impossibilitando que esses objetos imponham as regras do modelo do domínio. A lógica de domínio é movida para as consultas e para os códigos da camada de aplicativo, tornando as *Entidades* e os *Objetos de Valor* meros contêineres de dados.

Assim sendo:

Para cada tipo de *Agregado* que precise de acesso global, crie um objeto que forneça a ilusão de uma coleção em memória com todos os objetos daquele tipo. Crie o acesso através de uma interface global bem conhecida. Ofereça métodos para adicionar e remover objetos, que encapsulem a inserção ou remoção real de informações do armazenamento de dados. Forneça métodos que selecionem objetos com base em critérios significativos para os *Especialistas de Domínio*. Retorne objetos totalmente instanciados ou coleções de objetos cujos valores de atributo atendem aos critérios, encapsulando assim o armazenamento real e a tecnologia de consulta, ou retorne proxies que ofereçam a ilusão de *Agregados* totalmente instanciados através de Lazy Load (carregamento sob demanda). Forneça *Repositórios* somente para raízes de *Agregados* que realmente precisem de acesso direto. Mantenha a lógica do aplicativo focada no modelo, delegando todo o acesso e armazenamento de objetos aos *Repositórios*.

Fábricas

Quando a criação de um todo, agregado internamente consistente ou grande objeto de valor, torna-se complicada ou revela muito da estrutura interna, as fábricas fornecem o encapsulamento.

A criação de um objeto pode ser uma operação importante por si só, mas operações complexas de construção não se encaixam com a responsabilidade dos objetos criados. A combinação de tais responsabilidades pode produzir designs desajeitados e difíceis de entender. Fazer a construção direta para o *Cliente* confunde o design do *Cliente*, viola o encapsulamento do objeto construído ou do *Agregado* e provoca um acoplamento excessivo do *Cliente* com a implementação do objeto criado.

Assim sendo:

Mude a responsabilidade pela criação de instâncias de objetos complexos e *Agregados* para um objeto separado, que pode não ter nenhuma responsabilidade no modelo do domínio, mas ainda faça parte do design do domínio. Ofereça uma interface que encapsule toda a construção complexa e que não exija que o *Cliente* faça referência às classes concretas dos objetos que estão sendo instanciados. Crie *Agregados* inteiros como se fosse uma única unidade, impondo suas invariantes. Crie *Objetos de Valor* complexos como se fossem uma única unidade, possivelmente após invocar os elementos com um construtor.

3. Design Flexível

Ter um projeto acelerado à medida que o desenvolvimento avança (em vez de ser sobrecarregado por seu próprio legado) exige um design com o qual seja prazeroso trabalhar, convidando à mudança. Um *Design Flexível*.

O *Design Flexível* é o resultado da modelagem profunda.

Os desenvolvedores desempenham duas funções, onde cada uma delas deve ser atendida pelo design. É possível que a mesma pessoa desempenhe ambas as funções (até mesmo alternar daque pra lá em minutos), mas, no entanto, a relação dessas funções com o código é diferente.

Uma das funções é ocupada pelo desenvolvedor do *Cliente*, que tece os objetos de domínio no código do aplicativo ou em outro código de outra camada de domínio, utilizando os recursos do design. Um design flexível revela um profundo modelo oculto que torna claro o seu potencial. O desenvolvedor do *Cliente* pode flexivelmente usar um conjunto mínimo de conceitos fracamente acoplados para expressar uma variedade de cenários no domínio. Os elementos do design se encaixam de maneira natural com um resultado previsível, claramente caracterizado e robusto.

Igualmente importante, o design deve atender o *desenvolvedor que trabalha para alterá-lo*. Para estar aberto à mudança, um design deve ser fácil de entender, revelando o mesmo modelo oculto no qual o desenvolvedor do *Cliente* está se baseando. Ele deve seguir os contornos de um modelo profundo do domínio de forma que a maioria das alterações “dobre” o design em pontos flexíveis. Os efeitos de seu código devem ser transparentes e óbvios para que as consequências de uma mudança seram fáceis de antecipar.

- Tornar o comportamento óbvio;
- Reduzir o custo da mudança;
- Criar desenvolvedores de software para trabalhar com o código.

Interfaces Reveladoras de Intenção

Se um desenvolvedor precisar considerar a implementação de um componente para usá-lo, o valor do encapsulamento é perdido. Se alguém diferente do desenvolvedor original tiver que deduzir o propósito de um objeto ou operação com base em sua implementação, esse novo desenvolvedor poderá inserir um propósito que a operação ou classe realiza apenas por acaso. Se essa não era a intenção, o código poderá funcionar no momento, mas a base conceitual do design terá sido corrompida e os dois desenvolvedores trabalharão com objetivos opostos.

Assim sendo:

Dê nome às classes e às operações para descrever seu efeito e propósito, sem fazer referência a como eles fazem o que prometem. Isso alivia o desenvolvedor do *Cliente* da necessidade de entender as partes internas. Esses nomes devem estar em conformidade com a *Linguagem Onipresente* para que os membros da equipe possam deduzir rapidamente o seu significado. Escreva um teste para um comportamento antes de criá-lo, para forçar você mesmo a raciocinar de acordo o modo do desenvolvedor do *Cliente*.

Funções livres de efeitos colaterais

Interações de várias regras ou composições de cálculo se tornam extremamente difíceis de prever. O desenvolvedor que invoca uma operação deve entender a sua implementação e a implementação de todas as suas delegações para antecipar o resultado. A utilidade de qualquer abstração de interfaces é limitada se os desenvolvedores forem forçados a rasgar esta cortina. Sem abstrações seguramente previsíveis, os desenvolvedores devem limitar a explosão combinatória, colocando um teto baixo para a riqueza de comportamentos possíveis de se construir.

Assim sendo:

Coloque o máximo possível da lógica do programa em funções, ou seja, nas operações que retornam resultados sem nenhum efeito colateral observável. Segregue os comandos (métodos que resultam em modificações no estado observável) estritamente em operações bastante simples que não retornam informações de domínio. Controle ainda mais os efeitos colaterais transferindo a lógica complexa para *Objetos de Valor* quando aparecer um conceito adequado à responsabilidade.

Todas as operações de um objeto de valor devem ser funções livres de efeitos colaterais.

Asserções

Quando os efeitos colaterais das operações são definidos apenas implicitamente pela sua implementação, os projetos com muita delegação se tornam um emaranhado de causa e efeito. A única maneira de entender o programa é rastreando sua execução por meio de ramificações dos caminhos. O valor do encapsulamento é perdido. A necessidade de rastrear a execução concreta acaba com a abstração.

Assim sendo:

Declare as pós-condições de operações e invariantes das classes e dos *Agregados*. Se as *Asserções* não puderem ser codificadas diretamente na sua linguagem de programação, escreva testes de unidade automatizados para eles. Escreva-os na documentação ou nos diagramas onde isso estiver de acordo com o estilo do processo de desenvolvimento do projeto.

Procure modelos com conjuntos coerentes de conceitos, que levem um desenvolvedor a deduzir as *Asserções* pretendidas, acelerando a curva de aprendizado e reduzindo o risco de códigos contraditórios.

As *Asserções* definem os contratos de *Serviços* e os modificadores de entidades.

As *Asserções* definem as invariantes nos *Agregados*.

Classes Autônomas

Mesmo dentro de um *Módulo*, a dificuldade de interpretar um design aumenta virtiginosamente à medida que as dependências são adicionadas. Isso aumenta a sobrecarga mental, limitando a complexidade de design que um desenvolvedor pode controlar. Conceitos implícitos contribuem para essa sobrecarga até mais que as referências explícitas.

Um baixo acoplamento é fundamental para o design de objetos. Quando possível, vá até o fim. Elimine todos os outros conceitos do cenário. Depois disso, a classe estará completamente independente e poderá ser estudada e entendida isoladamente. Cada uma dessas classes independentes facilita significativamente o fardo de se compreender um módulo.

Fechamento de Operações

A maioria dos objetos interessantes acabam realizando tarefas que não podem ser caracterizadas apenas por primitivos.

Assim sendo:

Quando possível, defina uma operação cujo tipo de retorno seja do mesmo tipo de seu(s) argumento(s). Se o implementador possuir um estado que é usado na computação, ele é efetivamente um argumento da operação e, por isso, o(s) argumento(s) e o valor de retorno devem ser do mesmo tipo que o implementador. Uma operação como essa é fechada sob um conjunto de instâncias daquele tipo. Uma operação fechada fornece uma interface avançada sem introduzir qualquer dependência em outros conceitos.

Esse padrão é aplicado com mais frequência às operações de um *Objeto de Valor*. Como o ciclo de vida de uma *Entidade* tem importância no domínio, você não pode criar uma nova para responder a uma solicitação. Existem operações que são fechadas sob um tipo de *Entidade*. Você poderia solicitar a um objeto ***Funcionário*** que ele fornecesse o seu supervisor e obter outro ***Funcionário***. Mas, de um modo geral, as *Entidades* não são o tipo de conceito provável de ser o resultado de uma computação. Então, geralmente, essa é uma oportunidade a ser procurada nos *Objetos de Valor*.

Às vezes você fica a meio caminho desse padrão. O argumento corresponde ao implementador, mas o tipo de retorno é diferente ou o tipo de retorno corresponde ao receptor e o argumento é diferente. Essas operações não são fechadas, mas dão algumas das vantagens do *Fechamento de Operações*, poupando a mente.

Design Declarativo

Não pode haver garantias reais em um software procedural. Para citar apenas uma maneira de evitar *Asserções*, o código pode ter efeitos colaterais adicionais que não foram especificamente excluídos. Não importa o quanto nosso design seja orientado pelo modelo, ainda acabamos escrevendo procedimentos para produzir o efeito das interações conceituais. E passamos muito do nosso tempo escrevendo código clichê que realmente não adiciona nenhum significado ou comportamento. As *Interfaces Reveladoras de Intenção* e os outros padrões deste capítulo ajudam, mas nunca podem dar rigor formal aos programas orientados a objetos convencionais.

Estas são algumas das motivações por trás do *Design Declarativo*. Esse termo significa muitas coisas para muitas pessoas, mas geralmente indica uma maneira de escrever um programa, ou parte de um programa, como um tipo de especificação executável. Uma descrição muito precisa das propriedades realmente controla o software. Em suas várias formas, isso poderia ser feito através de um mecanismo de reflexão ou em tempo de compilação através da geração de código (produzindo código convencional automaticamente, baseado na declaração). Essa abordagem permite que outro desenvolvedor assuma a declaração pelo valor nominal. É uma garantia absoluta.

Muitas abordagens declarativas podem ser corrompidas se os desenvolvedores as ignorarem intencionalmente ou não. Isso é provável quando o sistema é difícil de usar ou excessivamente restritivo. Todos devem seguir as regras do framework para obter os benefícios de um programa declarativo.

Um estilo declarativo de design

Uma vez que seu design tenha *Interfaces Reveladoras de Intenção*, *Funções livres de efeitos colaterais* e *Asserções*, você está se aproximando do território declarativo. Muitos dos benefícios do *Design Declarativo* são obtidos a partir do momento que você tiver elementos combináveis que comuniquem seu significado e tenham efeitos característicos ou óbvios, ou nenhum efeito observável. Um design flexível pode possibilitar ao código do *Cliente* utilizar um estilo declarativo de design. Para ilustrar, a próxima seção reunirá alguns dos padrões deste capítulo para tornar a especificação mais flexível e declarativa.

Baseando-se em Formalismos Estabelecidos

Criar uma estrutura conceitual rígida a partir do zero é algo que você não pode fazer todos os dias. Às vezes, você descobre e refina uma dessas estruturas ao longo da vida de um projeto. Mas, muitas vezes, é possível usar e adaptar sistemas conceituais, já há muito tempo estabelecidos em seu domínio ou outros, alguns dos quais foram refinados e destilados durante séculos. Muitas aplicações de negócios envolvem contabilidade, por exemplo. A contabilidade define um conjunto bem desenvolvido de *Entidades* e regras que oferecem uma fácil adaptação a um modelo profundo e a um design flexível.

Existem muitas dessas estruturas conceituais formalizadas como essa, mas a minha favorita é a matemática. É surpreendente como pode ser útil resolver uma questão através da aritmética básica. Muitos domínios incluem matemática em algum lugar. Procure por ela. Tire-a do buraco. A matemática especializada é limpa, combinável por regras claras e as pessoas acham fácil entendê-la.

Um exemplo do mundo real de "ações matemáticas", foi discutido no capítulo 8 do livro, *Domain-Driven Design*.

Contornos Conceituais

Às vezes as pessoas cortam as funcionalidades em pedacinhos finos para permitir combinações flexíveis. Às vezes, contam-nas em pedaços maiores para encapsular a complexidade. Às vezes, procuram uma granularidade consistente, fazendo todas as classes e operações segundo uma escala semelhante. Essas são simplificações exageradas que não funcionam bem como regra geral. Mas são motivadas por problemas básicos.

Quando elementos de um modelo ou design são incorporados em uma ideia monolítica, suas funcionalidades são duplicadas. A interface externa não diz tudo com que um *Cliente* pode se preocupar. Seu significado é difícil de entender, porque conceitos diferentes são misturados.

Por outro lado, dividir classes e métodos pode complicar desnecessariamente o *Cliente*, forçando os objetos do *Cliente* a entender como as pequenas peças se encaixam. Pior ainda, um conceito pode ser perdido completamente. Metade de um átomo de urânio não é urânio. E, obviamente, não é apenas o tamanho do grânulo que conta, mas por onde ele percorre.

Assim sendo:

Decomponha elementos de design (operações, interfaces, classes e *Agregados*) em unidades coesas, levando em consideração sua intuição sobre as divisões importantes existentes no domínio. Observe o eixo de mudanças e estabilidade através de sucessivas refatorações e procure encontrar os *Contornos Conceituais* ocultos que explicam esses padrões de divisão. Alinhe o modelo com os aspectos consistentes do domínio que o tornam uma área viável do conhecimento.

Um design flexível baseado em um modelo profundo produz um conjunto simples de interfaces que se combinam logicamente para fazer declarações sensatas na *Linguagem Onipresente* e sem o fardo de distração e manutenção de opções irrelevantes.

4. Mapeamento de Contexto para Design Estratégico

Contexto Delimitado

Uma descrição de um limite (normalmente um subsistema ou o trabalho de uma equipe específica) dentro do qual um modelo específico é definido e aplicável.

Upstream-Downstream

Uma relação entre dois grupos em que as ações do grupo “upstream” afetam o sucesso do projeto do grupo “downstream”, mas as ações do downstream não afetam significativamente os projetos do upstream. Por exemplo, se duas cidades estão ao longo do mesmo rio, a poluição da cidade mais alta (upstream) afeta principalmente a cidade mais baixa (downstream). A equipe upstream pode ter sucesso independentemente do futuro da equipe downstream.

Mutuamente Dependente

Uma situação na qual dois projetos de desenvolvimento, em contextos separados, devem ser entregues para que o software seja considerado um sucesso. Quando dois sistemas dependem de informações ou de funcionalidades um do outro (algo que geralmente tentamos evitar) naturalmente vemos os projetos que os constroem como interdependentes. No entanto, há também projetos mutuamente dependentes em que as dependências do sistema executam apenas uma direção. Se o sistema dependido tiver pouco valor sem o sistema dependente ou sem uma integração com ele (talvez porque esse seja o único local em que ele é usado), a falha na entrega do sistema dependente seria uma falha de ambos os projetos.

Livre

Um contexto de desenvolvimento de software no qual a direção, o sucesso ou o fracasso do desenvolvimento ocorrido em outros contextos tem pouco efeito na entrega.

Mapa de Contexto

Para traçar uma estratégia, precisamos de uma visão realista e em larga escala do desenvolvimento do modelo, estendendo-se por todo o nosso projeto e outros com os quais nos integramos.

Um *Contexto Delimitado* individual pode possuir alguns problemas por causa da ausência de uma visão global. O contexto de outros modelos pode ainda ser vago e em andamento.

As pessoas das outras equipes não estarão muito conscientes dos limites dos contextos e, sem saber, farão alterações que ofuscarão os limites ou complicarão a interligação entre contextos. Quando for necessário fazer ligações entre contextos diferentes, essas ligações tendem a escoar umas para dentro das outras.

Mesmo quando os limites são claros, os relacionamentos com outros contextos impõem restrições à natureza do modelo ou ao ritmo viável de mudança. Essas restrições se manifestam principalmente através de canais não técnicos que às vezes são difíceis de relacionar com as decisões de design que as estão afetando.

Assim sendo:

Identifique cada modelo que está em jogo no projeto e defina seu *Contexto Delimitado*. Isso inclui os modelos implícitos de subsistemas não orientados a objetos. Dê um nome a cada *Contexto Delimitado* e faça com que os nomes passem a fazer parte da *Linguagem Onipresente*.

Descreva os pontos de contato entre os modelos, delineando a tradução explícita para qualquer comunicado, destacando qualquer compartilhamento, mecanismos de isolamento e níveis de influência.

Mapeie o terreno existente. Faça as transformações mais tarde.

Este mapa pode ser uma base para uma estratégia de design realista.

A caracterização dos relacionamentos vai se tornar mais concreta nas páginas seguintes, com um conjunto de padrões comuns de relações entre *Contextos Delimitados*.

Parceria *

Quando as equipes em dois contextos tiverem sucesso ou falharem juntas, frequentemente surge um relacionamento cooperativo.

A coordenação deficiente de subsistemas *Mutuamente Dependentes* em contextos separados leva à falha de entrega para ambos os projetos. Uma característica fundamental que falte em um sistema pode fazer que o outro sistema não seja entregue. Interfaces que não correspondem às expectativas dos desenvolvedores do outro subsistema podem causar falha na integração. Uma interface mutuamente acordada pode se tornar tão difícil de usar que atrasa o desenvolvimento do sistema do *Cliente*, ou tão difícil de implementar que atrasa o desenvolvimento do subsistema do *Fornecedor*. Falhas levam abaixo os dois projetos.

Assim sendo:

Onde a falha de desenvolvimento em qualquer um dos dois contextos resultaria em falha de entrega para ambos, crie uma parceria entre as equipes responsáveis pelos dois contextos. Estabeleça um processo de planejamento coordenado de desenvolvimento e gestão conjunta da integração.

As equipes devem cooperar na evolução de suas interfaces para acomodar as necessidades de desenvolvimento de ambos os sistemas. Recursos interdependentes devem ser agendados para que sejam concluídos no mesmo lançamento.

Não é necessário, na maioria das vezes, que os desenvolvedores entendam o modelo do outro subsistema em detalhes, mas devem coordenar o planejamento do projeto. Quando o desenvolvimento em um contexto atingir obstáculos, deve-se exigir uma análise conjunta do problema, para encontrar uma solução de projeto eficiente que não comprometa excessivamente o contexto.

Além disso, é necessário um processo claro para governar a integração. Por exemplo, um conjunto de testes especial pode ser definido para provar que a interface atende às expectativas do sistema do *Cliente*, que pode ser executado como parte da integração contínua no sistema *Fornecedor*.

Núcleo Compartilhado

Compartilhar parte do modelo e código associado é uma interdependência muito íntima, que pode alavancar o trabalho de design ou prejudicá-lo.

Quando a integração funcional é limitada, a sobrecarga de *Integração Contínua*, em um contexto grande, pode ser muito alta. Isso pode ser verdade principalmente quando as equipes não possuem a técnica ou organização política para manter a integração contínua, ou quando uma única equipe é simplesmente grande demais, e, por isso, difícil de manejar. Portanto, *Contextos Delimitados* separados podem ser definidos e várias equipes podem ser formadas.

Uma vez separadas, as equipes descoordenadas que trabalham em aplicativos intimamente relacionados podem, por um tempo, avançar rapidamente, mas o que uma produz pode não se encaixar perfeitamente com o que a outra faz. Elas podem acabar gastando mais em camadas de tradução e ajustes, do que se tivessem colocado a Integração Contínua em prática desde o início, duplicando esforços e perdendo benefícios de uma *Linguagem Onipresente* comum.

Assim sendo:

Determine, com um limite explícito, algum subconjunto do modelo do domínio com o qual as duas equipes concordem em compartilhar. Mantenha este *Núcleo* pequeno.

Obviamente, isso inclui, junto com esse subconjunto do modelo, o subconjunto de códigos ou do design do banco de dados associado àquela parte do modelo. Esse compartilhamento explícito tem um status especial e não deve ser alterado sem que a outra equipe seja consultada.

Defina um processo de integração contínua que mantenha o modelo do *Núcleo* firme e alinhe a *Linguagem Onipresente* das equipes. Integre um sistema funcional frequentemente, embora com menos frequência do que o ritmo de integração contínua dentro das equipes.

Desenvolvimento de Cliente/Fornecedor

Quando duas equipes estão em um relacionamento acima / abaixo, onde a equipe acima pode ter sucesso independentemente do futuro da equipe abaixo, as necessidades da equipe abaixo passam a ser abordadas de várias maneiras, com uma ampla gama de consequências.

Uma equipe abaixo pode ficar desamparada, à mercê das prioridades iniciais. Enquanto isso, a equipe acima pode ser inibida, preocupada com a quebra de sistemas da equipe abaixo. Os problemas da equipe abaixo não são aprimorados por causa de procedimentos complicados de solicitação de mudança com processos de aprovação complexos. E o desenvolvimento livre da equipe mais acima pode ser atrapalhado se a equipe mais abaixo tiver poder de vetar as alterações.

Assim sendo:

Estabeleça uma relação clara de cliente/fornecedor entre as duas equipes, o que significa estebelecer as prioridades da equipe de baixo no planejamento da equipe de cima. Negocie e oriente as tarefas para os requisitos da equipe de cima, para que todos entendam o compromisso e o cronograma.

Em sessões de planejamento, faça com que a equipe abaixo assuma o papel do *Cliente* com relação à equipe mais acima. Testes de aceitação automatizada desenvolvidos em conjunto podem validar a interface esperada da equipe acima. Adicionando esses testes ao conjunto de testes da equipe acima, para ser executado como parte de sua integração contínua, a equipe abaixo estará apta para fazer alterações sem medo de efeitos colaterais.

Conformista

Quando duas equipes de desenvolvimento têm uma relação *acima / abaixo* na qual a equipe acima não tem nenhuma motivação para atender às necessidades da equipe abaixo, esta fica desamparada. O altruísmo pode motivar os desenvolvedores acima a fazer promessas, que provavelmente não serão cumpridas. A crença nessas boas intenções leva a equipe mais abaixo a fazer planos com base em recursos que nunca estarão disponíveis. O projeto mais abaixo será adiado até que a equipe aprenda a viver com o que lhe é dado. Uma interface feita de acordo com as necessidades da equipe mais abaixo não é possível.

Assim sendo:

Elimine a complexidade da tradução entre *Contextos Delimitados* aderindo deligentemente ao modelo da equipe mais acima. Embora isso prejudique o estilo dos designers mais abaixo e provavelmente não produza o modelo ideal para o aplicativo, a escolha pela *Conformidade* simplifica muito a integração. Além disso, você vai compartilhar uma *Linguagem Onipresente* com sua equipe fornecedora. O fornecedor está no banco do motorista, por isso é bom facilitar a comunicação com ele. O altruísmo pode ser suficiente para que ele compartilhe informações com você.

Camada Anticorrupção

Camadas de tradução podem ser simples, até mesmo elegantes, quando ligam Contextos Delimitados bem projetados com equipes cooperativas. Mas quando o controle ou a comunicação não é adequada para obter uma relação de Núcleo Compartilhado, Parceria ou Cliente/Fornecedor, a tradução se torna mais complexa. A camada de tradução assume um tom mais defensivo.

Uma interface grande com um sistema fornecedor pode acabar sobrecarregando a intenção do modelo do cliente, fazendo com que ele seja modificado para se assemelhar ao modelo do outro sistema de maneira *ad hoc*, ou seja, para um fim específico. Os modelos de sistemas legados geralmente são fracos (se não *Grandes Bolas de Lama*), e até mesmo a exceção que é claramente projetada pode não atender às necessidades do projeto atual, tornando impraticável a conformidade com o modelo fornecedor. No entanto, a integração pode ser muito valiosa ou até mesmo necessária para o projeto cliente.

Assim sendo:

Como um cliente mais abaixo, crie uma camada de isolamento para fornecer ao sistema as funcionalidades do sistema mais acima em termos de seu próprio modelo do domínio. Esta camada conversa com o outro sistema através de sua interface existente, exigindo pouca ou nenhuma modificação no outro sistema. Internamente, a camada traduz em ambas as direções conforme necessário entre os dois modelos.

Serviço de Host Aberto

Normalmente, para cada Contexto Delimitado, você vai definir uma camada de tradução para cada componente fora do contexto com o qual você precisa integrar. Onde a integração é feita uma única vez, essa abordagem de inserir uma camada de tradução para cada sistema externo evita a corrupção dos modelos com um custo mínimo. Mas ao perceber que o seu subsistema está com demanda alta, talvez você precise de uma abordagem mais flexível.

Quando um subsistema precisa ser integrado com muitos outros, personalizar um tradutor para cada um pode sobrecarregar a equipe. Há cada vez mais coisas para serem mantidas e cada vez mais coisas para se preocupar quando mudanças forem feitas.

Assim sendo:

Defina um protocolo que dê acesso ao seu subsistema como um conjunto de *Serviços*. Abra o protocolo para que todos os que precisem se integrar a você possam usá-lo. Aprimore e expanda o protocolo para lidar com novos requisitos de integração, exceto quando uma única equipe tiver necessidades específicas. Em seguida, use um tradutor único para incrementar o protocolo para aquele caso especial de forma que o protocolo compartilhado possa permanecer simples e coerente.

Isso coloca o provedor do *Serviço* na posição mais acima. Cada *Cliente* estará na posição mais abaixo e, normalmente, alguns deles serão *Conformistas* e alguns construirão *Camadas Anticorrupção*. Um contexto com um *Serviço de Host Aberto* pode ter qualquer tipo de relacionamento com contextos diferentes de seus *Clientes*.

Linguagem Publicada

A tradução entre os modelos de dois Contextos Delimitados requer uma linguagem comum.

A tradução direta entre os modelos de domínio existentes pode não ser uma boa solução. Esses modelos podem ser estremamente complexos ou mal fatorados. Eles provavelmente não são documentados. Se um for usado como a linguagem de intercâmbio de dados, ela fica essencialmente engessado e não pode responder a novas necessidades de desenvolvimento.

Assim sendo:

Use uma linguagem compartilhada bem documentada que possa expressar as informações de domínio necessárias como um meio comunicação em comum, traduzindo a partir daquela linguagem, ou para ela, conforme necessário.

Muitas indústrias estabelecem *Linguagens Publicadas* na forma de padrões de intercâmbio de dados. As equipes de projeto também desenvolvem suas próprias para uso dentro de sua organização.

A *Linguagem Publicada* é frequentemente combinada com o *Serviço de Host Aberto*.

Caminhos Separados

Devemos nos concentrar nos requisitos, sem piedade. Dois conjuntos de funcionalidades sem nenhuma relação significativa podem ser completamente separados uns dos outros.

A integração é sempre cara e, às vezes, os benefícios são pequenos.

Assim sendo:

Declare um *Contexto Delimitado* que não tenha nenhuma conexão com os outros, permitindo que os desenvolvedores encontrem soluções simples e especializadas dentro deste pequeno escopo.

Grande Bola de Lama *

À medida que pesquisamos sistemas de software existentes, tentando entender como modelos distintos estão sendo aplicados dentro de limites definidos, encontramos partes de sistemas, geralmente grandes, onde os modelos são misturados e os limites são inconsistentes.

É fácil ficar atolado em uma tentativa de descrever os limites do contexto de modelos em sistemas onde simplesmente não há limites.

Limites de contexto bem definidos surgem apenas como resultado de escolhas intelectuais e forças sociais (mesmo que as pessoas que criaram os sistemas nem sempre tenham sido conscientemente responsáveis por essas causas na época). Quando esses fatores estão ausentes ou desaparecem, múltiplos sistemas conceituais se misturam, tornando definições e regras ambíguas ou contraditórias. Os sistemas são feitos para funcionar por lógica contingente, à medida que os recursos vão sendo adicionados. Dependências cruzam o software. Causa e efeito tornam-se cada vez mais difíceis de rastrear. Eventualmente, o software se engessa numa *Grande Bola de Lama*.

A *Grande Bola de Lama* é realmente bastante prática para algumas situações (como descrito no artigo original de Foote e Yoder), mas ela previne quase completamente a sutileza e a precisão necessárias para modelos úteis.

Assim sendo:

Desenhe um limite em torno de toda a bagunça e chame-o de *Grande Bola de Lama*. Não tente aplicar modelagem sofisticada nesse contexto. Esteja alerta para a tendência de tais sistemas se espalharem em outros contextos.

(leia http://www.laputan.org/mud/mud.html. Brian Foote e Joseph Yoder)

5. Destilação para Design Estratégico

Como concentrar-se em seu problema principal e evitar afundar em um mar de questões secundárias?

A *Destilação* é o processo de separar os componentes de uma mistura para extrair a essência em uma forma que a torne mais valiosa e útil. O modelo é uma destilação do conhecimento. A cada refatoração em direção a uma visão mais profunda, abstraímos alguns aspectos cruciais do conhecimento e das prioridades do domínio. Agora, dando um passo atrás para termos uma visão estratégica, este capítulo analisa maneiras de distinguir amplas seções do modelo e destilar o modelo do domínio como um todo.

Domínio Principal

Em um sistema grande, há tantos componentes envolvidos, todos complicados e absolutamente necessários para o sucesso, que a essência do modelo do domínio, o ativo comercial real, pode ser obscurecido e negligenciado.

A dura realidade é que nem todas as partes do design vão ser refinadas igualmente. Prioridades devem ser definidas. Para que o modelo do domínio se torne um patrimônio, o núcleo fundamental do modelo precisa ser elegante e totalmente aperfeiçoado para criar funcionalidades para o aplicativo. Mas os raros desenvolvedores altamente qualificados tendem a transferir suas atenções para a infra-estrutura técnica ou problemas do domínio perfeitamente definíveis que podem ser compreendidos sem um conhecimento especializado do domínio.

Assim sendo:

Reduza o seu modelo. Defina um *Domínio Principal* e forneça meios de diferenciá-lo facilmente com relação à massa de modelos e códigos de suporte. Reduza a sobrecarga sobre os conceitos mais valiosos e especializados. Crie um *Núcleo* pequeno.

Recrute e use as pessoas mais talentosas para trabalharem com o *Domínio Principal*. Dedique seus esforços no *Núcleo* para encontrar um modelo profundo e desenvolver um design flexível o suficiente para satisfazer a visão do sistema.

Justifique o investimento em qualquer outra parte pela forma como ele suporta o *Núcleo* destilado.

Subdomínios Genéricos

Algumas partes do modelo acrescentam complexidade sem capturar ou comunicar um conhecimento especializado. Qualquer coisa estranha torna mais difícil entender o *Domínio Principal*. O modelo fica engessado com princípios gerais que todos conhecem ou detalhes que pertencem a especialidades que não são o seu foco principal, mas desempenham um papel de apoio. No entanto, por mais genéricos que sejam, esses outros elementos são essenciais para o funcionamento do sistema e para a expressão completa do modelo.

Assim sendo:

Identifique os subdomínios coesos que não sejam a motivação do seu projeto. Fatore modelos genéricos desses subdomínios e coloque-os em *Módulos* separados. Não deixe nenhum rastro de suas especialidades neles.

Depois de separados, dê menos prioridade à continuidade do seu desenvolvimento do que ao *Domínio Principal* e evite designar seus principais desenvolvedores a essas tarefas (porque eles vão obter muito pouco conhecimento sobre o domínio a partir delas). Considere também soluções prontas ou modelos publicados para esses *Subdomínios Genéricos*.

Declaração da Visão de Domínio

No início de um projeto, o modelo geralmente nem mesmo existe, mas a necessidade de concentrar seu desenvolvimento já está lá. Em estágios posteriores do desenvolvimento, há uma necessidade de uma explicação do valor do sistema que não requer nenhum estudo aprofundado do modelo. Além disso, os aspectos fundamentais do modelo do domínio podem abranger vários *Contextos Delimitados*, mas, por definição, esses modelos distintos não podem ser estruturados de froma que mostrem seu foco comum.

Assim sendo:

Escreva uma breve descrição (cerca de uma página) do *Domínio Principal* e o valor que ele trará, a "proposição de valor". Ignore os aspectos que não distinguem esse modelo do domínio dos outros. Mostre como o modelo do domínio funciona e equilibra os interesses diversos. Mantenha-o pequeno. Escreva esta declaração logo de início revise-a à medida que você passa a adquirir novas visões.

Núcleo Destacado

Uma Declaração da Visão do Domínio identifica o Domínio Principal em termos gerais, mas deixa a identificação dos elementos específicos do modelo do Núcleo a cargo da interpretação individual. A menos que exista um nível de comunicação excepcionalmente alto na equipe, a Declaração da Visão do Domínio sozinha terá pouco impacto

Embora os membros das equipes possam saber amplamente o que constitui o *Domínio Principal*, pessoas diferentes não vão assimilar exatamente os mesmos elementos e, até a mesma pessoa não conseguirá manter consistência de um dia para o outro. O trabalho mental de filtrar constantemente o modelo para identificar as partes principais absorve a concentração que, do contrário, seria mais bem usada no raciocínio sobre o design, e requer amplo conhecimento do modelo. É preciso facilitar a visualização do *Domínio Principal*.

Alterações estruturais significativas feitas no código são a maneira ideal de identificar o *Domínio Principal*, mas nem sempre são práticas a curto prazo. De fato, essas grandes mudanças nos códigos são difíceis de serem realizadas sem se ter exatamente aquela visão que a equipe não tem.

Assim sendo (como uma forma de *Núcleo Destacado*):

Escreva um documento bem sucinto (três a sete páginas esparsas) que descreva o *Domínio Principal* e as principais interações entre os elementos do *Núcleo*.

e / ou (como outra forma de *Núcleo Destacado*):

Sinalize cada elemento do *Domínio Principal* dentro de um recipiente principal do modelo, sem tentar elucidar sua função. Faça-o de forma que um desenvolvedor, sem muito esforço, possa saber o que está dentro e fora do *Núcleo*.

Se o documento de destilação descreve os pontos essenciais do *Domínio Principal*, ele serve como um indicador prático da importância de uma alteração no modelo. Quando uma alteração feita no modelo ou no código afeta o documento de destilação, é necessário consultar outros membros da equipe. Quando a operação é feita, ela exige notificação imediata a todos os membros da equipe e a divulgação de uma nova versão do documento. Alterações feitas fora do *Núcleo* ou em detalhes não incluídos no documento de destilação podem ser integradas sem consulta ou notificação e serão encontradas por outros membros no decorrer de seu trabalho. Assim, os desenvolvedores têm a autonomia total sugerida pelo processo Ágil.

Embora a Declaração da Visão do Domínio e o Núcleo Destacado sirvam para informar e orientar, eles, na verdade, não modificam o modelo ou código em si. O particionamento de Subdomínios Genéricos elimina fisicamente alguns problemas que desviam a atenção. A seguir,

veremos outras maneiras de alterar estruturalmente o modelo e o design em si para tornar o Domínio Principal mais visível e gerenciável...

Mecanismos Coesos

Às vezes, as omputações atingem um nível de complexidade que começa a sobrecarregar o design. O "quê" conceitual é dominado pelo "como" mecanicista. Uma grande quantidade de métodos que fornece algoritmos para resolver o problema obscurece os métodos que expressam o problema.

Assim sendo:

Divida um *Mecanismo Coeso* em uma estrutura separada e mais leve. Observe especialmente formalismos ou categorias bem documentadas de algoritmos. Exponha os recursos da estrutura com uma *Interface Reveladora de Intenções*. Agora, os outros elementos do domínio podem se concentrar em expressar o problema ("o quê"), delegando as complicações da solução ("como") para a estrutura.

A fatoração de *Subdomínios Genéricos* reduz a desordem, e *Mecanismos Coesos* servem para encapsular operações complexas. Assim, é deixado para trás um modelo mais enfocado, com menos distrações que não agregam nenhum valor específico à maneira como os usuários conduzem suas atividades. Mas é improvável que você encontre bons abrigos para tudo que faça parte do modelo do domínio e que não seja o *Núcleo*. O *Núcleo Segregado* assume uma abordagem direta para demarcar estruturalmente o *Domínio Principal*...

Núcleo Segregado

Os elementos do modelo podem servir parcialmente ao *Domínio Principal* e desempenhar parcialmente funções de apoio. Os elementos do *Núcleo* podem estar fortemente acoplados aos elementos genéricos. A coesão conceitual do *Núcleo* pode não ser forte ou visível. Toda essa confusão e esse emaranhado sufocam o *Núcleo*. Os designers não conseguem ver claramente os relacionamentos mais importantes, levando a um design fraco.

Assim sendo:

Refatore o modelo para separar os conceitos do *Núcleo* de suas funções de apoio (incluindo as mal definidas) e fortalecer a coesão do *Núcleo* reduzindo, ao mesmo tempo, seu acoplamento com outros códigos. Fatore todos os elementos genéricos ou de apoio em outros objetos e coloque-os em outros pacotes, mesmo que isso signifique refatorar o modelo de forma a separar elementos altamente acoplados.

Núcleo Abstrato

Até mesmo o modelo do *Núcleo Principal* geralmente possui tantos detalhes que a comunicação do cenário geral pode se tornar difícil.

Quando há muita interação entre subdomínios em *Módulos* separados, muitas referências terão que ser criadas entre os *Módulos*, o que vai contra grande parte do valor da divisão em partições, ou a interação terá que ser feita indiretamente, o que torna o modelo obscuro.

Assim sendo:

Identifique os conceitos mais fundamentais no modelo e fatore-os em classes distintas, classes abstratas ou interfaces. Projete esse modelo abstrato para que ele expresse grande parte das interações os entre componentes significativos. Coloque esse modelo abstrato e geral em seu próprio *Módulo*, enquanto as classes de implementação especializadas e detalhadas são mantidas em seus próprios *Módulos* definidos pelo subdomínio.

6. Estrutura em Larga Escala para Design Estratégico

Em um sistema grande, sem nenhum princípio abrangente que permita que os elementos sejam interpretados em termos de suas funções em padrões que englobem todo o design, os desenvolvedores não conseguem ver e discernir muito bem a situação. Precisamos ser capazes de entender a função de uma determinada parte do todo, sem nos aprofundar nos detalhes do todo.

Uma "estrutura em larga escala" é uma linguagem que permite discutir e entender o sistema em linhas gerais. Um conjunto de conceitos ou regras avançados, ou ambos, estabelece um padrão de design para todo o sistema. Esse princípio de organização pode orientar o design e ajudar no seu entendimento. Ele ajuda a coordenar trabalhos independentes, porque existe um conceito compartilhado do cenário geral: como as funções de várias partes dão forma ao todo.

Assim sendo:

Elabore um padrão de regras ou funções e relacionamentos que abranjam todo o sistema e que permitam algum entendimento do lugar ocupado por cada parte no todo - mesmo que não se tenha o conhecimento detalhado da responsabilidade das partes.

Ordem de Evolução

A liberdade exagerada no design produz sistemas que, como um todo, ninguém consegue entender e se tornam muito difíceis de manter. Porém, as arquiteturas podem atravancar um projeto com considerações iniciais de design e roubar grande parte do poder dado aos desenvolvedores/designers em determinadas partes do aplicativo. Em pouco tempo, os desenvolvedores vão reduzir o aplicativo para se ajustar à estrutura, ou vão subvertê-lo e ficar sem nenhuma estrutura, trazendo de volta os problemas de um desenvolvimento descoordenado.

Assim sendo:

Deixe que essa estrutura conceitual em larga escala evoluia com o aplicativo, possivelmente se transformando em um tipo completamente diferente de estrutura ao longo do caminho. Não restrinja exageradamente as decisões detalhadas de design e de modelo que devem ser tomadas com base em um conhecimento detalhado.

Uma estrutura em larga escala deve ser aplicada quando pode ser encontrada uma estrutura que esclareça bastante o sistema sem forçar restrições que não sejam naturais ao desenvolvimento do modelo. Como uma estrutura mal formada é pior do que nenhuma, é melhor não optar pela abrangência, mas, sim, encontrar um conjunto mínimo que resolva os problemas que surgiram. Quando menos, melhor.

O que se segue é um conjunto de quatro padrões particulares de estrutura em larga escala que emergem em alguns projetos e são representativos desse tipo de padrão.

Metáfora de Sistema

O raciocínio metafórico se espalha rapidamente no desenvolvimento de softwares, principalmente com modelos. Mas a prática de “metáforas” adotata pelo Extreme Programming passou a significar uma forma específica de se usar uma metáfora para dar ordem ao desenvolvimento de todo um sistema.

Os designs dos softwares tendem a ser bastante abstratos e difíceis de entender. Desenvolvedores e usuários precisam de formas tangíveis para entender o sistema e compartilhar uma visão do sistema como um todo.

Assim sendo:

Quando surge uma analogia concreta para o sistema que capta a imaginação dos membros da equipe e parece condizir o reciocínio em uma direção adequada, adote-a como uma estrutura de larga escala. Organize o design em torno dessa metáfora e absorva-a na *Linguagem Onipresente*. A *Metáfora de Sistema* deve facilitar a comunicação sobre o sistema e orientar o seu desenvolvimento. Isso aumenta a consistência em diferentes partes do sistema, possivelmente até entre diferentes *Contextos Delimitados*. Porém, como todas as metáforas são inexatas, reexamine continuamente a metáfora para verificar sua abrangência exagerada ou inaptidão e esteja pronto para abandoná-la se ela começar a atrapalhar.

Camadas de Responsabilidade

No design dirigido por objetos, cada objeto recebe conjuntos restritos de responsabilidades relacionadas. O design dirigido por responsabilidades também se aplica a escalas maiores.

Quando cada objeto possui responsabilidades definidas manualmente, não há nenhuma diretriz, nenhuma uniformidade e nenhuma capacidade de controlar conjuntamente grandes partes do domínio. Para dar coerência a um modelo grande, é útil impor alguma estruturação na designação dessas responsabilidades.

Assim sendo:

Observe as dependências conceituais do seu modelo e as várias velocidades e fontes de mudança das diferentes partes do seu domínio. Se você identificar estratos naturais no domínio, distribua-os como grandes responsabilidades abstratas. Essas responsabilidades devem contar uma história sobre o propósito e o design avançados do seu sistema. Refatore o modelo para que as responsabilidades de cada objeto do dominio, *Agregado* e *Módulo* se encaixem perfeitamente dentro da responsabilidade de uma camada.

Nível de Conhecimento

Um grupo de objetos que descreve como outro grupo de objetos deve se comportar.

Em um aplicativo no qual as funções e as relações entre as *Entidades* variam em diferentes situações, a complexidade pode explodir. Nem modelos completamente gerais nem modelos altamente personalizados atendem às necessidades dos usuários. Os objetos acabam fazendo referências a outros tipos para abranger uma variedade de casos ou acabam tendo atributos que são usados de maneira diferente em situações diferentes. Classes que tenham os mesmos dados e comportamentos podem se multiplicar só para atender a diferentes regras de montagem.

Assim sendo:

Crie um conjunto distinto de objetos que possa ser usado para descrever e restringir a estrutura e o comportamento do modelo básico. Mantenha essas preocupações separadas na forma de dois "níveis", um bastante concreto, o outro refletindo regras e o conhecimento que um usuário ou superusuário consiga de personalizar.

(leia Fowler, M. 1997. Analysis Patterns: Reusable Object Models, Addison-Wesley.)

Estrutura de Componentes Plugáveis

As oportunidades surgem em um modelo bastante maduro, profundo e destilado. Uma *Estrutura de Componentes Plugáveis* geralmente só entra em ação depois que alguns aplicativos já foram implementados no mesmo domínio.

Quando uma variedade de aplicativos precisam operar em conjunto, todos baseados nas mesmas abstrações, mas projetados de forma independente, as traduções entre vários *Contextos Delimitados* limitam a integração. Um *Núcleo Compartilhado* não é viável para equipes que não trabalham intimamente juntas. A duplicação e fragmentação aumentam os custos do desenvolvimento e da instalação, e a interoperabilidade se torna muito difícil.

Assim sendo:

Destile um *Núcleo Abstrato* de interfaces e interações e crie uma estrutura que permita que diversas implementações dessas interfaces sejam substituídas livremente. Da mesma forma, permita que qualquer aplicativo use esses componentes, desde que funcione estritamente através das interfaces do *Núcleo Abstrato*.