revista
MUNDOJAVA®

Artigo publicado
na edição 19

setembro/outubro de 2006

Ivar Jacobson, Scott Ambler, Martin Fowler, Craig Larman. Entre autores famosos existem visões diferentes a respeito do maior benefício do uso da UML, porém, nenhum deles cita a documentação do sistema como uma característica importante! Neste artigo, contaremos a história de um projeto, aplicando a UML de uma maneira simples e concisa. Vamos partir dos requisitos, levantando casos de uso e, então, analisaremos o problema utilizando a UML 2.0 como nossa aliada.

UML não é documentação!

É comum analistas questionarem sobre a melhor maneira de usar a UML para documentar um sistema. Gestores de fábricas de software procuram treinamentos dizendo: - "Queremos documentar o sistema usando UML". Essa necessidade de documentar projetos de software tem aumentado, principalmente, com a febre do CMM. Infelizmente, usar a UML só para documentar o sistema é extremamente desaconselhável! Se você ler a obra de metodologistas famosos, entre eles, existem visões diferentes a respeito do maior benefício do uso da UML, porém, nenhum deles cita que a documentação é um benefício importante. Alguns chegam a dizer que a UML não serve para documentar!

A UML 2.0, na sua melhor forma, é uma ferramenta de análise. A análise é traduzir requisitos em componentes do software, descobrindo informações. Isso também é chamado de design ou modelagem. Neste texto, estaremos referenciando essa tarefa simplesmente como análise.

Para demonstrar essa aplicação da UML, será desenvolvido um estudo de caso. Partiremos da necessidade e, aplicaremos a UML 2.0 para investigar e aprofundar o conhecimento a respeito do problema, assim chegaremos à solução.

Neste ensaio prático, serão quebradas algumas idéias erradas. Um projeto de software não deve ter ambigüidades e burocracias desnecessárias. Devemos sempre respeitar o dinheiro investido nos projetos que participamos, entregando o software funcionando e documentado na medida certa. Requisitos, modelo e código possuem responsabilidades distintas e complementares, cada um tem o seu propósito. Todos eles juntos são o projeto, não importando quais deles são a "documentação". Temos que acabar com as ambigüidades e também com os documentos que tentam controlar essas ambigüidades. A UML aplicada corretamente, como ferramenta de análise, para descobrir informações, tem a sua utilidade.

Primeiramente, vamos explicar o estudo de caso.

Rodrigo Yoshima
(rodrigoy@aspercom.com.br)
É técnico em Processamento de Dados pela UNESP, bacharel em Administração de Empresas pelo Mackenzie-SP e certificado em UML 2.0 pela OMG. Possui 12 anos de experiência em desenvolvimento de software. Trabalha há 7 anos em fábricas de software e projetos de missão crítica para grandes indústrias, bancos e hospitais. É instrutor da ASPERCOM e líder técnico de projetos estratégicos da Procwork.

Estudo de Caso: Oficinas Hot Motors

Já que vamos demonstrar o uso prático da UML 2.0, desenvolveremos um estudo de caso um pouco mais elaborado. Não será o projeto de caixa eletrônico, que aparece em todos os exemplos de UML! Vamos simular um cliente que chegou para a fábrica de software e solicitou um sistema para cadastro de clientes e controle sobre o processo interno da empresa. Tente compreendê-lo.

O problema

Em todo desenvolvimento de software, é importante ter uma visão geral do problema, antes mesmo de investigar o que o sistema deve fazer. Verifique o problema relatado por nosso cliente, a Hot Motors:

A Hot Motors é uma rede de lojas especializada em tuning (customização) de veículos. Atualmente, conta com sete lojas espalhadas em grandes centros urbanos do Brasil. O mercado de customização de veículos está em plena expansão no país e atualmente a rede fatura R$ 12 milhões anuais.

O PROBLEMA

Atualmente, não possui um cadastro dos clientes, seus veículos e acessórios instalados. Em primeiro lugar, sem esse cadastro, tem-se dificuldades quando os clientes retornam às lojas para adicionar mais itens customizados nos seus veículos. O cliente é muito fiel à marca e este cadastro é importante. Com o cadastro de clientes, veículos e acessórios instalados, podería-se saber antecipadamente a configuração do veículo, agilizando o atendimento. Um cadastro de clientes também permitiria campanhas de marketing e promoções futuramente.

Cada oficina possui três setores: pintura, mecânica e elétrica. A pintura, além de pintar, instala pára-choques, aerofólios e outros itens no interior do veículo. A mecânica trabalha com a performance do motor, como a instalação de um Injetor de Nitro. O setor elétrico instala aparelhagem de áudio, vídeo e iluminação. O problema nesses três setores é o fluxo do atendimento. Ele deve ocorrer na seguinte seqüência: pintura, mecânica e elétrica. É comum o cliente solicitar a remoção, ou substituição, de algum acessório que ele já possui instalado no veículo. Esse serviço também é cobrado.

A única parte informatizada na loja é o estoque. Cada loja possui um computador com acesso à internet, onde o vendedor controla o estoque usando o sistema StockOn. Atualmente, tudo o que foi instalado no veículo é anotado em papel. Ao final do atendimento, o vendedor digita essas informações no StockOn e cobra o cliente. Aqui, se tem mais problemas: não na garantia que os instaladores realmente anotaram tudo. Pode acontecer de algum acessório não ser instalado por várias razões. Isso causa discrepâncias no estoque ou, pior ainda, o cliente pode ter algum acessório instalado e sair sem pagar por ele. A razão desses problemas é que determinados acessórios, obrigatoriamente necessitam de outros acessórios para a instalação. Exemplo: um "Injetor de Nitro" requer o acessório "Cilindro" e "Conjunto de Mangueiras". Acessórios também possuem itens opcionais. Eles devem ser sugeridos ao cliente. Exemplo: sugerir um "Intercooler" quando um "Turbo" é solicitado.

Uma boa solução seria um sistema de cadastro de clientes, veículos e acessórios instalados, juntamente com um controle de atendimento integrado com o StockOn. Por razões de infra-estrutura da loja, só os vendedores poderiam acessar esse sistema. O controle do atendimento dentro da oficina (pintura, mecânica e elétrica) continuaria no papel, porém, com uma guia de atendimento emitida pelo sistema garantindo que a cobrança e baixa do estoque sejam corretas.

O desafio de negócio que o sistema deve resolver é esse. Em resumo: o cliente chega à loja e um atendimento é aberto com as instalações e remoções a serem feitas. Uma guia em papel é emitida para acompanhar o carro dentro da oficina, e cada item do atendimento é marcado quando instalado. Finalmente, tudo é instalado ou removido, a guia volta para o vendedor e o atendimento é finalizado. Só os itens instalados ou removidos são cobrados.

É importantíssimo que todo o time do projeto saiba exatamente o problema que o sistema vai resolver. O entendimento do propósito do sistema ajuda a sanar muitas dúvidas. O problema descrito no texto nos dá uma idéia de onde o software vai atuar, mas ainda é muito vago o que ele tem que fazer. Precisamos dos requisitos do software mais detalhados.

Os requisitos

Existem inúmeras maneiras de capturar requisitos de um software. Para citar algumas:

Quadro 1. Técnicas de captura de requisitos.

Técnica	Processo "origem"
User Stories	XP (Extreme Programming)
Features	FDD (Feature Driven Development)
Use Cases (Casos de Uso)	OOSE, atualmente Unified Process

Como este artigo é sobre UML, os requisitos serão detalhados com casos de uso, assim poderemos explicar um pouco a respeito do dia-

grama, mas qualquer uma dessas técnicas poderia ser utilizada. Tudo depende do engajamento do usuário no projeto e a configuração do seu processo de desenvolvimento como um todo. O Caso de Uso é uma maneira de demonstrar o funcionamento do sistema pela visão do usuário. Podemos chamar o Caso de Uso de OBJETIVO, pois ele retrata um objetivo do Ator. Uma explicação mais detalhada dessa técnica é assunto para outro artigo. Olhando para o problema e discutindo com os usuários, chegamos aos Casos de Uso conforme figura 1.

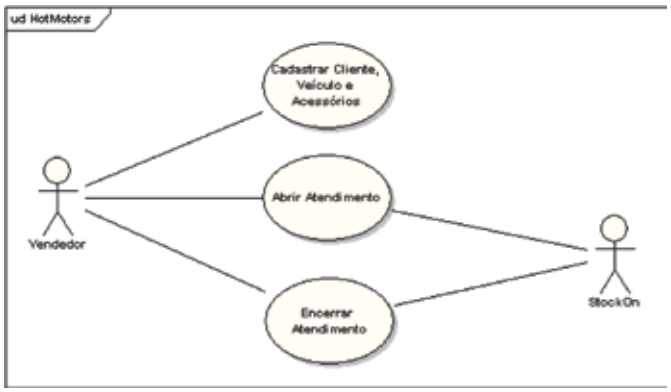

Figura 1. Diagrama de Casos de Uso.

O Diagrama de Casos de Uso é um resumo do sistema. Você pode utilizá-lo para fazer um levantamento inicial dos OBJETIVOS e ATORES, e depois escrever os textos das narrativas. Ou você pode escrever as narrativas e depois fazer o diagrama. De qualquer forma, para a técnica de Casos de Uso, o texto (a narrativa) é muito mais importante que o diagrama. O diagrama também destaca integrações com sistemas externos, e em quais Casos de Uso ocorre essa integração. O StockOn é o sistema de Estoque citado no problema. O diagrama de Casos de Uso é simples mesmo para sistemas grandes. Basicamente, ele mostra ATORES, OBJETIVOS E INTEGRAÇÕES COM SISTEMAS EXTERNOS. Se o seu sistema é simples e o diagrama de Casos de Uso está parecendo um plástico-bolha (cheio de elipses), é bem capaz que tenha alguma coisa errada. Você pode também estabelecer relacionamentos <<include>> e <<extend>> entre Casos de Uso, mas esses conceitos são intrinsecamente ligados à técnica de Casos de Uso (não vou entrar em detalhes aqui, fica para o próximo artigo). Para nosso ensaio, neste artigo, estes Casos de Uso são suficientes. Note que esses **não são todos os requisitos** do sistema, certamente existem muitos outros. Analisamos somente estes três inicialmente, pois esses remediam diretamente as piores dores da Hot Motors. Isso é, são os requisitos que solucionam os maiores problemas (cadastro de clientes e fluxo dentro da loja).

Olhando para o Caso de Uso "Cadastrar Cliente, Veículo e Acessórios", dá para imaginar que seria um cadastro relativamente simples, então, não seria necessária uma narrativa passo-a-passo detalhada. Neste momento, este Caso de Uso teria somente uma descrição:

Caso de Uso: Cadastrar Cliente, Veículos e Acessórios

Descrição geral: este Caso de Uso é responsável por incluir e alterar um Cliente, os Veículos deste Cliente e os Acessórios desses Veículos. Um veículo só pode estar associado a um Cliente e a busca pelo cliente deve ser por nome ou CPF.

O Caso de Uso não precisa necessariamente ter uma narrativa passo-a-passo detalhada. Em cadastros ou funcionalidades simples, uma descrição geral é suficiente para deixar claras as responsabilidades alocadas ao Caso de Uso.

Vamos descrever agora uma narrativa passo-a-passo de um importante Caso de Uso: Abrir Atendimento.

Caso de Uso: Abrir Atendimento

- O Ator informa que deseja abrir um Atendimento, informando a placa do Veículo.
- O Sistema exibe as informações do Veículo, dos Acessórios desse Veículo e do Cliente [A1].
- [P1] O Ator verifica os dados e informa critérios de busca por Acessórios para instalação [A2].
- O Sistema lista os Acessórios que satisfazem a busca.
- O Ator seleciona um Acessório.
- O Sistema confirma a disponibilidade de estoque, exibe os Acessórios obrigatórios para a instalação, sugere os opcionais e informa o preço [A3, A4].• O Ator seleciona os opcionais.
- O Sistema informa o preço atualizado [A3].
- O Ator finaliza as inclusões.
- O Sistema inclui o Acessório, seus obrigatórios e opcionais escolhidos para instalação no Atendimento e atualiza o seu valor total. Volta ao passo [P1] por quantas vezes o ator desejar.
- O Ator confirma os dados do atendimento [A5].
- O Sistema reserva os Acessórios utilizados no estoque e imprime a Guia de Atendimento com os Acessórios por ordem de setor. Os Acessórios a serem removidos aparecem antes dos Acessórios a instalar. O Caso de Uso encerra.

Fluxo Alternativo A1 – Veículo não-existente

- O Sistema informa que o Veículo informado não existe. Volta ao ínicio do Caso de Uso.

Fluxo Alternativo A2 – Removendo Acessório do Veículo

- O Ator informa um Acessório do Veículo para remoção.
- O Sistema inclui o Acessório na lista de remoções do Atendimento e atualiza o valor total. Volta ao passo [P1].

Fluxo Alternativo A3 – Acessório indisponível no estoque

- O Sistema informa que o Acessório (ou algum de seus obrigatórios) está indisponível no estoque. Volta ao passo anterior.

Fluxo Alternativo A4 – Veículo já possui este Acessório instalado

- O Sistema informa que o Veículo já possui este Acessório instalado. Volta ao passo anterior.

Fluxo Alternativo A5 – Removendo Acessório do Atendimento

- O Ator informa um Acessório incluído por engano no Atendimento.
- O Sistema remove o Acessório do Atendimento, atualiza o valor total do Atendimento e volta ao passo [P1].

Dúvidas na técnica de Casos de Uso são muito comuns. A maioria dos analistas erra na escrita da narrativa passo-a-passo. Muitos de vocês podem estar olhando para esse Caso de Uso e podem estar falando que esse texto está simples demais. A narrativa não está simples por se tratar de um sistema fictício. Seria desta forma mesmo se este projeto fosse real. Uma boa narrativa de Caso de Uso deve ser clara e focada nas necessidades do Ator. Outro detalhe importante: nas respostas do Sistema, você deve escrever somente O QUE o Sistema responde e não COMO ele faz para obter essa resposta. O funcionamento interno do sistema será analisado mais tarde. Ainda estamos nos requisitos! O Caso de Uso mapeia o problema e não a solução.

Vamos descrever o último Caso de Uso importante para a arquitetura da aplicação: Encerrar Atendimento. Com esses três Casos de Uso, já podemos iniciar o design da aplicação com uma substancial segurança à respeito das necessidades mais importantes da Hot Motors.

Caso de Uso: Encerrar Atendimento

Ator: Vendedor

- O Ator informa que deseja encerrar um Atendimento informando o número.
- O Sistema exibe os dados do Veículo, do Cliente e uma lista dos Acessórios do Atendimento.
- O Ator informa os Acessórios que realmente foram instalados ou removidos usando a Guia de Atendimento.
- O Sistema efetua a baixa dos Acessórios instalados no Estoque, retira os Acessórios não-instalados da reserva de Estoque, atualiza a configuração do Veículo e atualiza o valor total do Atendimento.
- O Ator confirma o encerramento do Atendimento.
- O Sistema encerra o Atendimento e o Caso de Uso termina.

Mais um Caso de Uso descrito e mais uma vez está demonstrado como Casos de Uso são simples. Eles só explicam como o Ator usa o Sistema. Qual a opinião de vocês à respeito desses requisitos? Vocês imaginam que é um exemplo muito grande para um artigo de revista? Se o nosso foco aqui fosse demonstrar a codificação do sistema pode ser que este seja um exemplo elaborado demais, mas para a ótica da análise, esse sistema não é tão grande assim. Logicamente, se você colocar esse desenvolvimento dentro de um processo burocrático e pesado, tudo pode ficar bem complicado. Gostaria que vocês fizessem um levantamento do esforço em horas desse sistema junto aos seus gestores e coordenadores para analisarmos a estimativa deles.

Estes Casos de Uso já estão muito bons para avançarmos no projeto. Vamos complementar um pouco a visão do problema, entendendo como sistema se encaixa no processo da loja.

O processo da Hot Motors (workflow)

Uma das maiores críticas que os analistas da análise estruturada fazem aos Casos de Uso é que eles não demonstram processo (workflow). E realmente não demonstram! Essa não é a função dos Casos de Uso. Os Casos de Uso são focados em objetivos dos Atores. As elipses retratam o que o usuário poderá fazer com o sistema. Para analisar o processo da Hot Motors, precisamos de outro diagrama da UML 2.0: o Diagrama de Atividades. Veja figura 2.

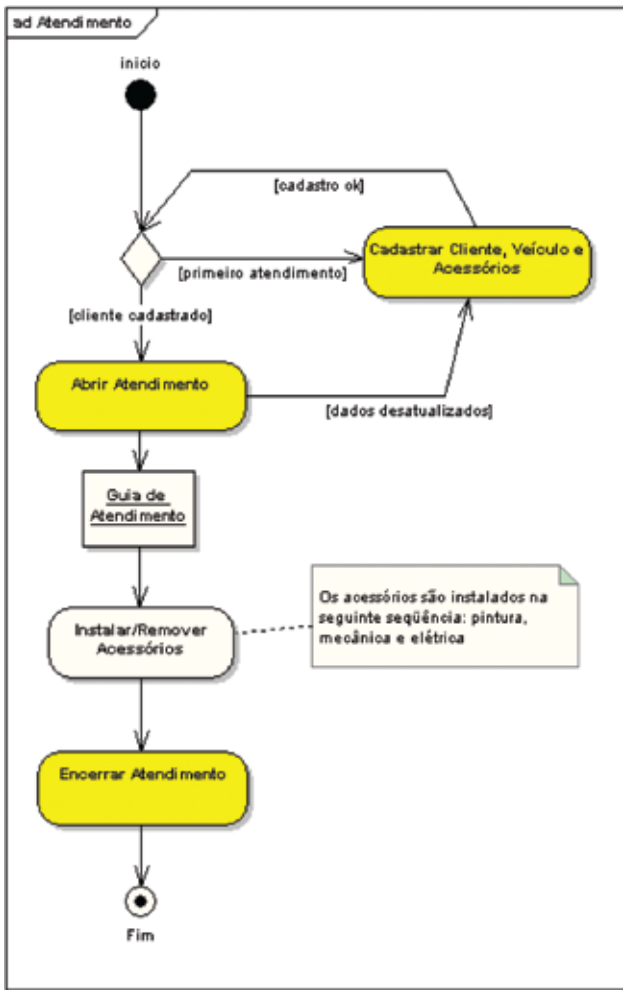

Figura 2. Diagrama de Atividades demonstrando o Atendimento da Hot Motors.

Neste diagrama, temos tarefas que serão desempenhadas no sistema (destacadas em amarelo), e tarefas que estão fora do escopo do sistema (todas as outras). Alguns analistas de negócio, lendo este artigo, podem estar falando: - "Você deveria detalhar melhor esta atividade Instalar/Remover Acessórios", mas não estamos fazendo análise de negócio. O que acontece dentro da oficina não interessa para o sistema. Um Comment (notinha) da maneira como está no diagrama é suficiente para demonstrar que o fluxo segue a seqüência: pintura, mecânica e elétrica, sem precisar detalhar as tarefas desses setores. Quando analisamos um sistema, precisamos saber o propósito daquilo que estamos fazendo. Precisamos deixar as coisas o mais simples possível.

Em primeiro lugar, este digrama se tornou necessário para entender melhor onde o sistema se encaixa dentro do processo da loja, e se esse "encaixe" está coerente. Essa necessidade surgiu porque existe um documento em papel emitido pelo sistema: a Guia de Atendimento. Ela dará suporte a atividades que estão fora do escopo do sistema (o trabalho dos pintores, mecânicos e eletricistas). Essa guia fornecerá todas as informações necessárias para encerrar o atendimento, mas não é preciso analisar como os mecânicos trabalham para visualizar isso. Em segundo lugar, este diagrama confirma como os dois Casos

de Uso "Cadastrar Cliente, Veículo e Acessórios" e "Abrir Atendimento" atenderão aos seguintes cenários:

1. o Cliente chega na loja pela primeira vez e não possui qualquer cadastro;
2. um Cliente cadastrado chega na loja, mas com um Veículo não-cadastrado;
3. um Cliente cadastrado chega na loja, mas o Veículo está desatualizado no sistema.

O modelo de Casos de Uso não demonstra as necessidades do fluxo de trabalho da loja. Conforme citado, Casos de Uso não modelam processos e relacionamentos <<include>> e <<extend>> não resolvem isso! Após entender como os Casos de Uso se encaixam no processo interno da loja, com este diagrama, conclui-se que o Ator Vendedor vai ter que navegar entre as telas dos dois Casos de Uso para resolver os três cenários acima, e nenhum outro Caso de Uso é necessário. Com o processo e objetivos resolvidos, abro um parêntese importante neste parágrafo: se você fabrica software espero que esteja aplicando, ou tentando aplicar, o desenvolvimento iterativo. Resumindo: reduza o risco do projeto. Entregue o software ao usuário em pequenos pedaços incrementais (iterações). Por que isso reduz o risco? Principalmente porque o usuário consegue ver a cara da aplicação poucos dias depois do início do projeto. Ele pode rapidamente avaliar se o que está sendo desenvolvido atende às suas necessidades. Aplicamos isso no levantamento de requisitos: somente os Casos de Uso mais importantes foram levantados.

Agora, vamos fazer um exercício para escolher qual Caso de Uso seria interessante entregar primeiro para a Hot Motors. Precisamos um para analisar, codificar, testar e entregar, assim a Hot Motors avaliará se estamos avançando na direção correta. Qual deles você escolheria? É uma boa prática do desenvolvimento de software iterativo que você ataque os maiores riscos primeiro (logicamente respeitando a dependência entre os requisitos). Fazendo uma analogia, no jogo de xadrez, você deve se esforçar mais para abater a rainha do seu adversário, não os peões. A rainha apresenta um risco maior, você cuida dos peões depois. Se no seu projeto, a equipe está gastando muita energia em matar os peões existe um potencial enorme de falha. Desses três Casos de Uso, o "Abrir Atendimento" seria indicado para uma primeira iteração. Neste Caso de Uso, estão os requisitos mais complexos: Instalação/Remoção de Acessórios e a integração com o StockOn. Sobre o StockOn, vamos simular uma situação bem corriqueira nos projetos: a Hot Motors ainda não liberou qualquer documentação e não sabemos se a integração será via Banco de Dados, API ou Web Services (você já deve ter passado por isso). Isso não deve impedir o início da análise, pois há como abstrair essa integração de maneira muito fácil no modelo. Isso permitirá que essa integração ocorra mais tarde sem invalidar o diagrama. Não é a falta dessa informação que vai paralisar o nosso projeto (mesmo assim, com certeza o coordenador vai guardar essa carta na manga para justificar atrasos).

Escolhido o primeiro Caso de Uso, iniciam-se as tarefas de análise para traduzir esses requisitos em componentes de software. Para isso, vamos usar a UML como nossa aliada e não como instrumento para fomentar a burocracia e a complexidade. Explicando melhor, o intuito deste artigo é deixar claro que:

> Você modela o que precisa ser modelado

Vamos falar mais sobre isso. Guarde esta frase: a UML não é um processo, a UML não diz que ordem você deve seguir para desenhar os diagramas. É comum escutar pessoas questionando quais diagramas são obrigatórios e quais são opcionais. Bem, todos os diagramas são opcionais se você partir do princípio de usar a ferramenta certa para o problema certo. Se o problema não existe, você não precisa da ferramenta! Processos de desenvolvimento de software de várias empresas possuem regras do tipo: "Todo Caso de Uso deve ter um Diagrama de Atividades demonstrando os fluxos possíveis". Será que o Caso de Uso "Encerrar Atendimento" necessita de um Diagrama de Atividades para que seu fluxo se torne claro? Lógico que não! Ele nem possui fluxos alternativos. Imagine que você é um marceneiro e seu chefe exija que para bater um prego, além do martelo, você DEVE também usar a serra e o pincel. Faz sentido? Essas regras ocorrem principalmente quando o processo diz que só um documento prova que a tarefa foi feita. LEMBRE-SE: você modela o que precisa ser modelado.

Com essa regra em mente, vamos descobrir as classes do Domain Model. As classes do Domain Model são abstrações do mundo real da Hot Motors para um mundo virtual, que simulará essa realidade dentro do sistema. O Domain Model em Java foi muito bem explicado no artigo "OO com Padrões de Negócio" de Phillip Calçado "Shoes" da MundoJava número 17, aliás, muitos dos patterns explicados pelo Shoes serão aplicados na Hot Motors.

Encontrando classes e atributos

Uma boa dica para encontrar as classes do Domain Model é observar os substantivos dos requisitos ou o vocabulário do usuário. Vejam alguns desses substantivos: Cliente, Veículo, Acessório, Atendimento. Podem existir muitos outros, mas podemos descobri-los mais tarde. Vamos iniciar um diagrama para investigar o relacionamento entre essas classes. Mas antes, imagino que você já tenha um modelo mental dessas classes enquanto foi lendo os requisitos. Desenhe esse modelo mental para que você possa comparar com a solução proposta. Veja figura 3.

Espero que o seu modelo mental esteja bem próximo disso, apesar de existir inúmeras maneiras de resolver este problema (algumas bem esquisitas). O Diagrama de Classes é o mais popular da UML. Muitos profissionais só conhecem este diagrama e colocam nos seus currículos que conhecem UML, mas a UML possui muitos outros diagramas importantes.

É difícil demonstrar escrevendo como se chega à solução da figura 3. Geralmente são feitos rascunhos, experimentações e as Classes saem "sangrando com tantas flechadas". Mas é isso que é analisar com a UML: olhar os requisitos e desenhar a melhor maneira de atendê-los, pesquisando possibilidades e descobrindo fraquezas no modelo. Muitos de vocês podem estar dizendo que dá para partir do modelo mental diretamente para o código. Sim, é possível, mas o modelo UML é uma perspectiva mais ampla e amigável, consigo ver várias classes ao mesmo tempo. É mais difícil ter uma visão assim no código: o código é texto, você olha uma classe por vez. É necessário navegar entre várias classes para se ter idéia do todo.

Vou relatar como este diagrama da figura 3 foi usado para descobrir informações importantes, que inclusive não constam nos requisitos e vamos precisar revisá-los. Quando estava pensando no Atendimento, logo estabeleci a associação C (não tinha desenhado a associação B

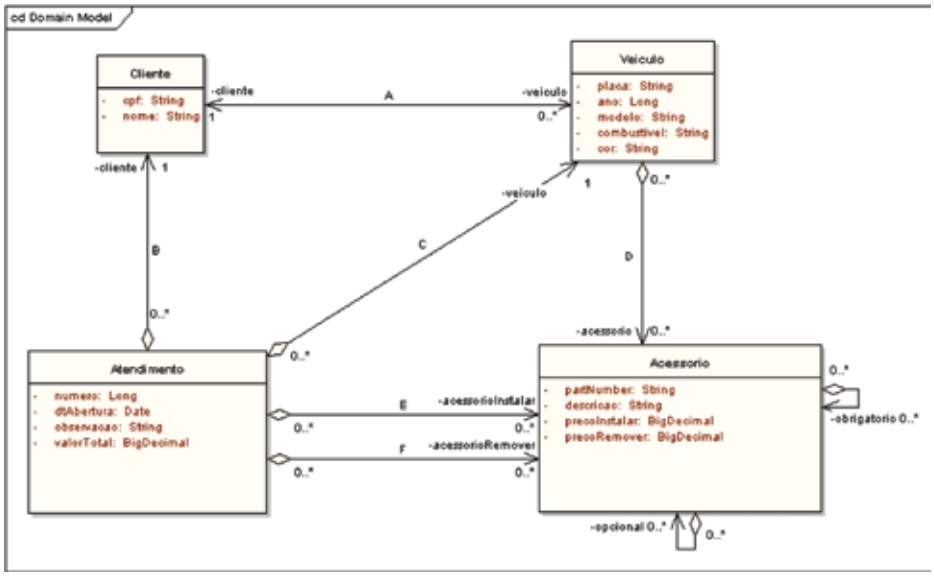

Figura 3. Diagrama de Classes de Domínio.

ainda). De fato, o Atendimento é feito em cima do Veículo, o Veículo é ligado ao Cliente através da associação A (que é bidirecional), dessa forma, imaginei que o Atendimento não necessitaria associação direta com o Cliente. Mas aí veio um pensamento a respeito do mundo real: "Será que é correto afirmar que um veículo sempre vai ser do mesmo dono em todos os Atendimentos? Não poderia ocorrer do João ter um Supra, instalar um Nitro, vender o carro para o José e o José querer instalar um DVD?". No mundo real, isso é perfeitamente possível, mas no nosso mundo virtual, esse cenário não havia passado pela nossa cabeça até agora. Foi bom descobrir isso em tempo de análise e foi o modelo visual que me ajudou a vislumbrar este problema facilmente. O primeiro passo para remediar isso foi o estabelecimento da associação B, assim, um Atendimento é feito em um Veículo para o Cliente que o possui naquele momento.

Agora, como mapear essa descoberta nos requisitos? O Diagrama de Atividades (figura 2) diz que, se existe algum problema de cadastro, o fluxo vai para a atividade "Cadastrar Cliente, Veículo e Acessórios". Este Caso de Uso deve resolver essa transferência entre proprietários. Como ele será desenvolvido nas próximas iterações, não precisamos perder tempo pensando como isso será resolvido. Isso não impacta a iteração atual. Essa independência é conseqüência de Casos de Usos bem coesos e direcionados ao objetivo do Ator. Simplesmente, ao final do Caso de Uso "Cadastrar Cliente, Veículo e Acessórios", escrevemos:

> Este Caso de Uso deverá resolver a situação do João comprar um Supra, instalar um Nitro, vender o carro para o José e ele querer instalar um DVD. Isso é, um Veículo pode ser transferido de um Cliente para outro.

Postergar coisas são comuns no desenvolvimento iterativo. Nesse ponto, o foco deve ser o Caso de Uso "Abrir Atendimento", certo? Então, veja que as classes ainda não possuem todos os atributos. Você encontra atributos das seguintes maneiras:

1. no Caso de Uso (alguns dados que constam na narrativa);
2. ao analisar a interação entre objetos que resolverá o Caso de Uso (faremos isso mais adiante);
3. conversando com os usuários.

Uma ferramenta rica para encontrar atributos é um protótipo da tela. Atributos que são só informativos geralmente não constam no Caso de Uso. As melhores ferramentas para desenhar e validar uma tela com o usuário é o lápis (ou caneta, giz de cera, o que for). Vamos fazer um rascunho da tela de abertura de atendimento. Veja figura 4.

Figura 4. Rascunho da tela de abertura de Atendimento.

Este rascunho foi elaborado junto com a Hot Motors. Ele também foi usado para descobrir informações. Logicamente, este rascunho não é UML 2.0. A UML não é uma boa ferramenta para modelar telas, apesar de ser possível fazer isso. Algumas telas Swing complexas podem ser modeladas usando um Diagrama de Comunicação para mapear eventos.

Para protótipos de sistemas Web, você também pode usar HTML simples. Em Aplicações Desktop, você pode fazer uma casca vazia em Swing ou SWT. Avalie bem o quanto você vai investir no protótipo. Para iterações iniciais é mais importante validar conceitos do que deixar o software visualmente bonito. O protótipo nesse momento aprofunda o nosso conhecimento do problema, e prova aquilo que já compreendemos.

Um rascunho da tela torna o Caso de Uso mais palpável e, também, nos ajuda a achar atributos novos: o Veículo possui fabricante, marca, modelo, ano de fabricação, tipo de combustível. Um acessório também possui fabricante e a sua chave é partNumber. Vamos atualizar essas classes, conforme diagrama de classes das figuras 5 e 6.

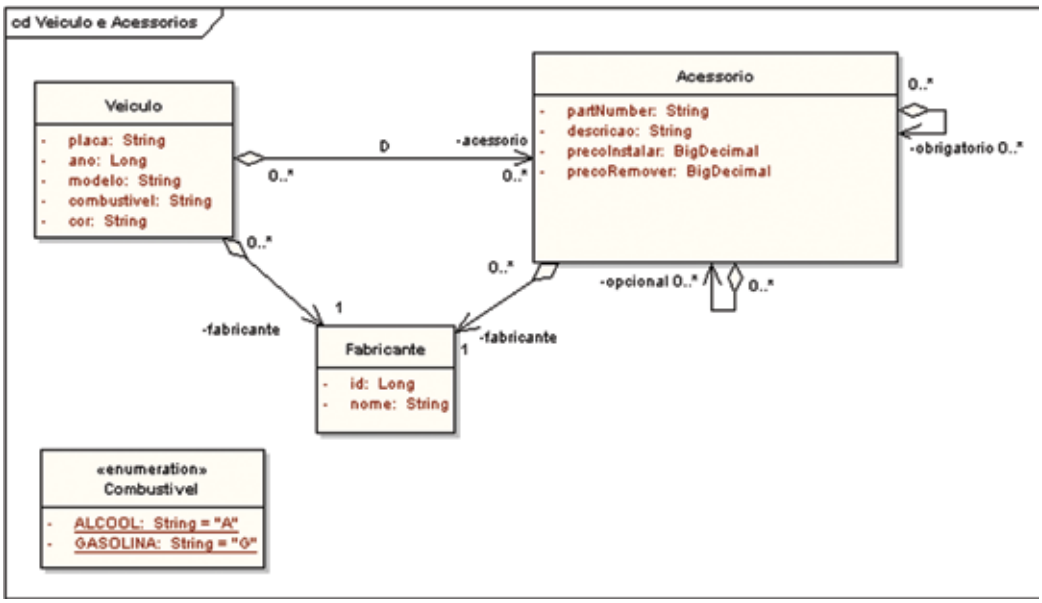

Figura 5. Novos atributos e associações de Veiculo e Acessório.

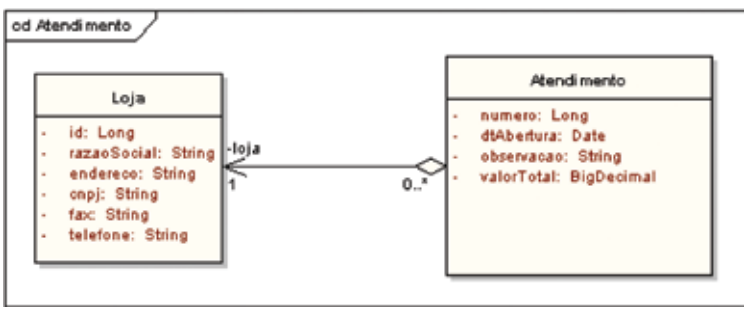

Figura 6. Novos atributos e associações do Atendimento.

Descobrindo operações e comportamentos

Os Diagramas de Classe elucidam o aspecto estático ou estrutural do sistema, mas atributos e associações sozinhos não traduzem todo o funcionamento do software. As classes devem colaborar entre si para resolver os requisitos.

Os próximos diagramas são usados para descobrir como as classes se comportam e quais são as suas responsabilidades através de operações. Dá para imaginar que a classe Atendimento terá uma operação "encerrar()", mas colocar essa operação diretamente na classe na figura 6 é pura especulação e foge do propósito da iteração atual. Sem uma visão de como essa classe interage com outras, não sabemos se essa operação "encerrar()" é realmente necessária e nem quais parâmetros ela precisa. Para descobrir os comportamentos das classes, a UML 2.0 define os diagramas de interação (não confunda com iteração). Temos o Diagrama de Comunicação, que substituiu o Diagrama de Colaboração da UML 1.X. O Diagrama de Visão Geral da Interação, que não existia na UML 1.X, é uma ótima ferramenta para modelar interações mais longas. Por fim, temos o famoso Diagrama de Seqüência, o mais utilizado para este propósito de descobrir operações. É nesse ponto que a técnica de Casos de Uso começa a fazer mais sentido. Comentei que o Caso de Uso mapeia o problema. O Caso de Uso só mostra O QUE o sistema responde e não COMO ele obtém a resposta. Os requisitos nos dizem o que o sistema tem que fazer, e não como ele faz. O "como ele faz" é analisado melhor com Diagramas de Interação. O Caso de Uso demonstra como o sistema funciona pela visão do Ator (lado de fora). Um Diagrama de Seqüência pode ser utilizado para descobrir como as Classes conversam entre si, para realizar o objetivo do Ator (lado de dentro). Veja figura 7.

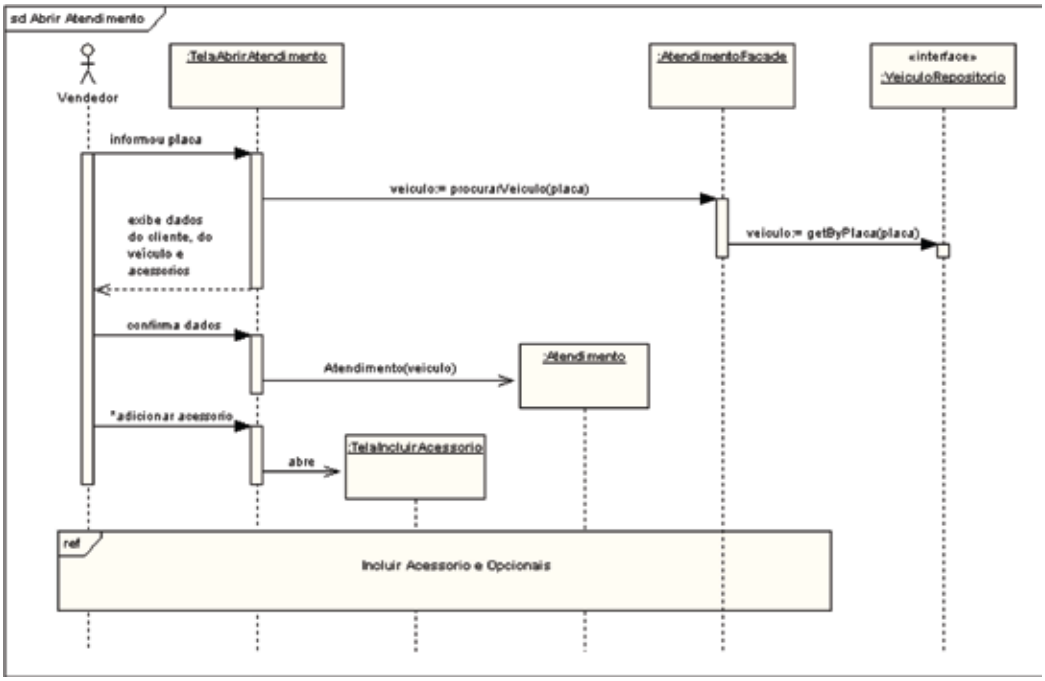

Figura 7. Diagrama de seqüência: Abrir Atendimento.

Como puderam ver, usamos Domain Patterns (vide artigo citado, MundoJava nº 17). A "TelaAbrirAtendimento" é uma representação da tela demonstrada no rascunho da figura 4. Ela recebe as ações do Ator Vendedor. A primeira seta já deixa claro que, para a interação começar, o Vendedor precisa saber uma placa. O Vendedor não terá problemas para ter essa informação, pois o atendimento inicia com o carro na frente dele na loja. A tela recebe essa placa e chama uma operação no façade que procurará o veículo. O façade delega essa chamada para o repositório de Veículos (abstração de uma coleção com todos os veículos do sistema). A interação avança respondendo COMO cada solicitação do Ator é resolvida DENTRO do sistema. Este é o funcionamento interno, que não é responsabilidade do Caso de Uso.

Sobre a arquitetura, modela-se num nível de abstração em concordância com os arquitetos e codificadores do projeto. Isso garante que o modelo seja implementável em código (necessitando de alguns ajustes, principalmente nas primeiras iterações). Os frameworks atuais (EJB 3.0, Spring, Hibernate) permitem que a análise seja feita basicamente se focando no Domain Model com classes POJOS. Padrões como Dependency Injection, Data Mapper, Lazy Load, etc. integram o Domain Model com a infra-estrutura da aplicação. Resumindo: arquiteturas fortes permitem um modelo mais abstrato e a análise é mais rápida.

O Diagrama de Seqüência é uma excelente ferramenta de análise. Com uma narrativa passo-a-passo, ou um protótipo, o Diagrama de Seqüência é ótimo para descobrir qual componente é responsável por cada ação, dando um panorama geral do que está acontecendo.

Mais uma vez, é difícil demonstrar como o analista modela essa interação, mas não demorou mais que 5 minutos. A cada solicitação do Ator, você vai navegando dentro do sistema, esticando setas e atribuindo responsabilidades. Depois da primeira seta "informa placa", o raciocínio é o seguinte: - "O que a tela tem que fazer agora? Ah, procurar o Veículo. Quem procura o Veículo? Alguma coisa na camada de negócios que estaria atrás do façade. Um repositório de Veículos? Sim, vou criá-lo. Já tenho o Veículo. E agora?"

Cada seta atirada define um emissor e um receptor da mensagem. Isso torna claro algumas dependências, mas o maior benefício é o foco em descobrir somente as verdadeiras operações necessárias para atender aos requisitos. Após a análise com o diagrama da figura 7, as classes que só possuíam atributos, possuem operações. Veja figura 8.

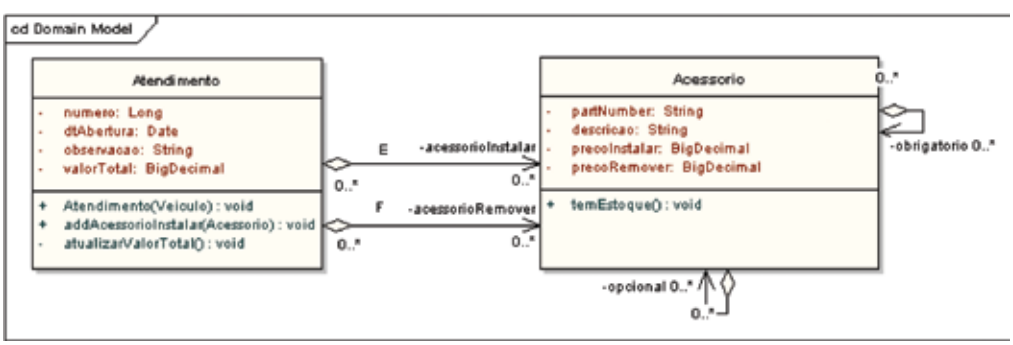

Figura 8. Diagrama de classes atualizado.

Você notou que existe um quadro no diagrama da figura 7 chamado "Incluir Acessorio e Opcionais"? Este é um recurso da UML 2.0 chamado ocorrência de interação. Ele serve para dividir uma interação grande em pedaços menores e reutilizáveis. Veja o que tem dentro deste quadro na figura 9.

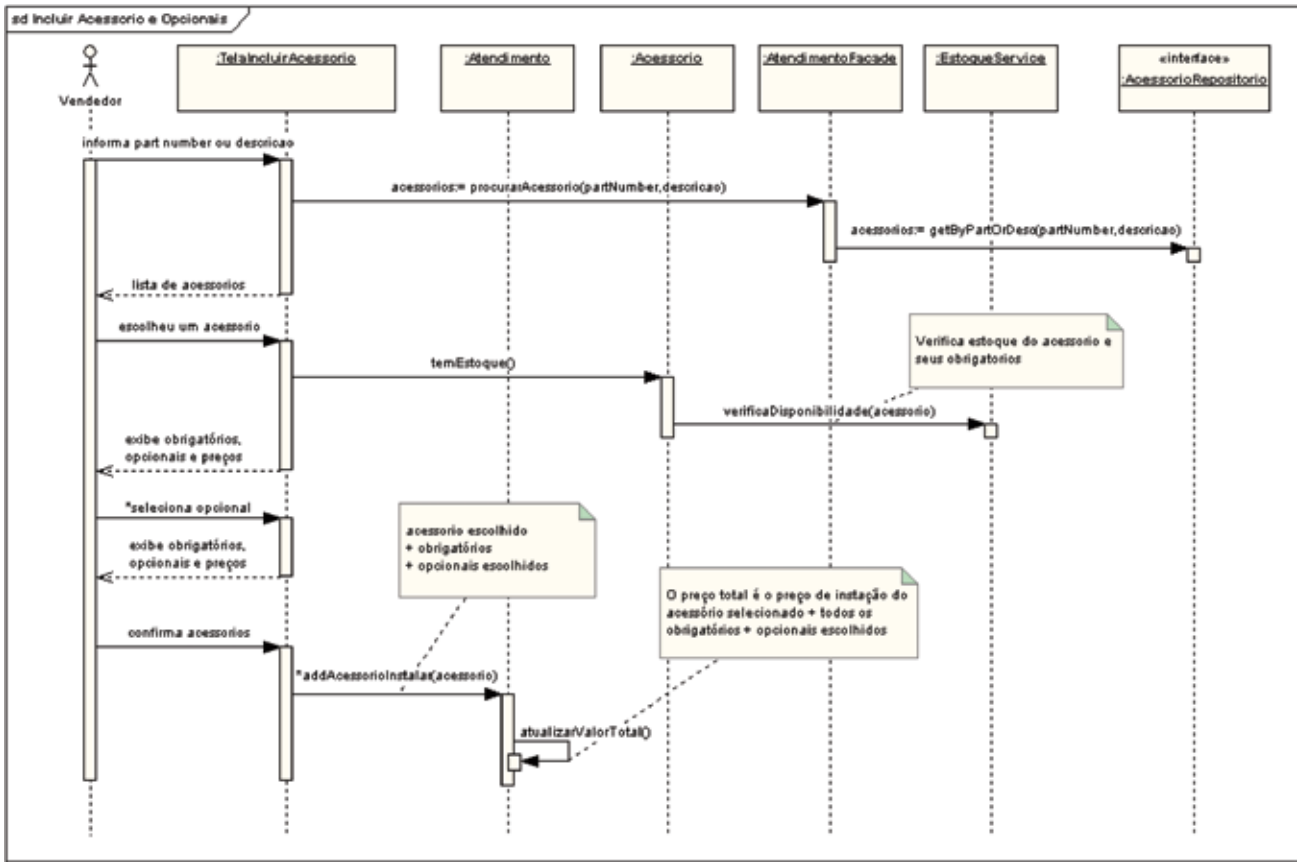

Figura 9. Diagrama de seqüência: inserir Acessório e seus Opcionais.

Obs.: os diagramas de seqüência das figuras 7 e 9 são partes do fluxo básico do Caso de Uso. Infelizmente, não poderemos realizar o Caso de Uso todo aqui, mas as outras interações seguem o mesmo padrão (os diagramas completos podem ser obtidos no site da ASPERCOM, veja a referência no fim do artigo).

Essa interação ocorre na tela para incluir acessórios no atendimento. Não temos o protótipo dessa tela e não precisamos dele, o Caso de Uso deixa claro o que o usuário precisa. Note que o Caso de Uso nem menciona que ela existe. Mais uma vez, na figura 9, parte-se do comando do ator e aloca-se responsáveis, descobrindo novas operações. Aqui, aparece o serviço responsável pelo controle de estoque, o StockOn, mas no modelo ficará somente essa abstração "EstoqueService". Essa interface responderá quais serviços o StockOn deverá atender futuramente. A implementação dessa interface resolverá a integração, quando tivermos a documentação faltante do StockOn.

Os diagramas de seqüência das figuras 7 e 9 são partes do fluxo básico do Caso de Uso. Infelizmente, não poderei realizar o Caso de Uso todo aqui, mas as outras interações seguem o mesmo padrão (os diagramas completos podem ser obtidos no site da revista). Vejam que muitos comments (notas) podem ser usados para explicar o modelo. Alguns deles poderiam até se traduzir em setas, mas entre um diagrama rigorosamente correto e um simples, opte por um diagrama simples. Os detalhes não estão no modelo.

O objetivo do uso dos diagramas de interação é modelar a conversa entre vários objetos, podendo inclusive constar o Ator. Basicamente, são as operações públicas que são encontradas. Esse diagrama não é uma boa ferramenta para demonstrar o detalhe dessas operações. Muitas vezes, analistas esticam uma seta para um receptor, e a partir daí, passam a "programar no modelo", destacando tudo que o receptor faz dentro da operação. Isso é uma péssima idéia. O código é melhor para mostrar esses detalhes e não precisamos dessa ambigüidade. O modelo UML possui as "operações" da classe e não "métodos". As operações do modelo UML são abstrações de como a Classe se comporta. Operações são estímulos que a Classe vai atender, mas o detalhe da operação está dentro do código, no MÉTODO. O método é o programa que implementa a operação. A UML não possui diagramas para demonstrar a implementação de métodos.

Alguns analistas defendem que o Diagrama de Seqüências deve ser um espelho daquilo que está nos métodos, para tudo ficar "bem documentado", mas não consigo imaginar o que o projeto ganha com isso. Essa prática gera modelos pesados, complexos e engessa completamente o trabalho do analista. E pode ter certeza que esse "espelho" é completamente distorcido. As IDEs atuais facilitam bastante a escrita do código: temos autocompletar, checagem de sintaxe automática, controle de

exceções. Coisas que o modelo não tem. Enfim, é mais fácil entender o que ocorre dentro da operação diretamente no código. Para quê ter essa ambigüidade? Cuidado com esses "espelhos" na sua documentação. Eles são um risco real de tornar seu processo pesado e difícil de manter.

E a documentação?

A preocupação excessiva com a documentação do código é herança das experiências que tivemos nos sistemas procedurais: - "E quando o sistema estiver em manutenção?", perguntávamos. Bem, o código procedural é longo, linear, pouco coeso e complexo. Ele não consegue se explicar sozinho. Isso não acontece nas linguagens orientadas a objeto como Java. Num sistema orientado a objetos bem coeso e bem fundamentado, a própria estrutura e semântica da classe são auto-explicativas e necessitam pouca documentação. É completamente errada a idéia de que o modelo UML serve para documentar o código. O modelo UML documenta como componentes interagem para atender aos requisitos. O código é responsável por implementar os detalhes de como esses componentes interagem. Se a semântica da própria classe não for suficiente, o que documenta o código são os comentários dentro dos programas, o Javadoc como exemplo. A UML não é a documentação do código.

Costumo dizer: use a UML como ferramenta de análise e ganhe a documentação. Por quê? Vamos dar um exemplo: escrevi este artigo como uma experiência em tempo real: defini requisitos, fui modelando e contanto a história. Usei a UML 2.0 para analisar o problema. Apliquei cada diagrama quando senti necessidade. Desenhei todos os elementos com o objetivo de descobrir informações. Cada seta tinha um propósito bem definido. Fiz muitos diagramas, mas em nenhum momento senti que estava documentando o sistema. Eu senti que estava produzindo o sistema. No entanto, os diagramas estão aí. Eles são rastros do meu trabalho de análise e suprem as necessidades de documentação. Eles podem ser utilizados para o trabalho do time de arquitetura e também para quem for codificar e testar o software (podendo ser eu ou algum de vocês, a definição de papéis aqui é irrelevante). Esses documentos, frutos do trabalho de análise, já são suficientes para avançar com o projeto. Vejam que temos ambigüidade quase zero. Cada artefato tem o seu propósito bem definido. Não precisa contar a mesma história em cada documento diferente. Quando for definir seus artefatos, pense da mesma maneira que você faz com o código: alta coesão e baixo acoplamento.

Continuando o projeto

Com relação à codificação, outro aspecto importante é que a UML é uma ferramenta independente de plataforma ou linguagem. Se a arquitetura suporta esse nível de abstração do modelo, posso codificar o sistema da Hot Motors em Java, C#, Ruby, etc. As informações descobertas na análise não se perdem. Essa característica é importante em fábricas de software onde o time de análise e design trabalha para projetos de várias tecnologias. Os próximos passos depois da análise concluída são: gerar código, completar os programas, integrar o que for necessário e testar o software. A geração de código é um instrumento para ganhar agilidade. O modelo já tem pacotes, classes, atributos e operações definidas, o codificador não precisa redigitar tudo isso. Mas a geração de código UML é só para essa facilidade e pode variar de ferramenta para ferramenta. Dá para gerar muitas outras coisas se você tiver uma ferramenta MDA, mas antes que isso gere discussões, falaremos sobre MDA em outro artigo também.

Uma das grandes críticas sobre o uso da UML nos projetos é a questão do modelo ficar defasado com relação ao código. Realmente isso acontece, e pode piorar muito se o seu modelo for pesado, muito próximo do código, ou dependente da arquitetura. Os mecanismos que sincronizam o código com o modelo enxergam melhor a estrutura estática das classes. Com isso, é provável que no processo de sincronização, seus diagramas de seqüência, como exemplo, fiquem desatualizados. A técnica passada aqui no artigo demonstra uma maneira de minimizar bastante esse risco. Focando o modelo basicamente na camada de negócio do sistema (Domain Model), deixamos os diagramas mais isolados das complexidades arquiteturais e das especificidades da camada de apresentação. Desse modo, as grandes alterações do modelo seriam motivadas por alterações de negócio ou de requisitos. Não teriam alterações bruscas efetuadas diretamente no código que impactariam tanto o modelo. Se quiser eliminar completamente o risco do modelo ficar desatualizado, jogue fora os diagramas (recomendação do Martin Fowler). Eles já cumpriram o papel como ferramenta de análise.

Conclusão

Você explora todo o potencial da UML 2.0 utilizando-a como uma ferramenta de análise. A UML também é uma ótima ferramenta de comunicação. Um time que "fala" UML dissemina informações mais rapidamente e com menores falhas de entendimento. É difícil analisar numa mesa de reunião três páginas de código. Um diagrama torna isso mais fácil.

Existem outros diagramas na UML 2.0 e todos eles oferecem algumas facilidades que não existiam na UML 1.X. A UML 2.0 é muito grande, mas os diagramas citados neste artigo (Casos de Uso, Atividades, Classes e Seqüência) são os mais utilizados e se aplicam a muitos projetos. Quando pensei no estudo de caso da Hot Motors, imaginei que o Diagrama de Máquina de Estados também seria necessário, mas não foi. Este diagrama explora os estados possíveis de uma Classe (ou qualquer outra coisa), e como ocorrem as transições entre esses estados. A classe Atendimento possui dois estados "Aberto" e "Encerrado", isso é bem claro. A transição entre esses estados é simples, então, não precisamos deste diagrama.

A análise e design é um dos trabalhos mais criativos no desenvolvimento de software. Nesse trabalho, os analistas e designers são os responsáveis, mas todo o time está envolvido, dando sugestões e criticando modelos inflexíveis. Esse trabalho de criação deve ser ágil. Preencher documentos burocráticos e ambíguos conspira contra a criatividade e a motivação da sua equipe, principalmente, se faltar um senso de propósito do artefato no projeto• MJ

O estudo de caso HotMotors analisado parcialmente, devido a restrições de espaço, neste artigo pode ser complementado no site www.aspercom.com.br/hotmotors.

Referências

- *COCKBURN, ALISTAIR (2000). Writting Effective Use Cases, Addison Wesley.*
- *EVANS, ERIC (2004). Domain-Driven Design*
- *PENDER, TOM (2004). UML - A Bíblia, Campus.*
- *OMG (2004). UML 2.0 Final Adopted Specification, www.omg.org/uml.*
- *FOWLER, MARTIN (2002). Patterns of Enterprise Application Architecture, Prentice Hall.*
- *VÁRIOS (2001). Home page. Manifesto for Agile Software Development, www.agilemanifesto.org.*
- *www.aspercom.com.br – Curso On-line Grátis sobre Casos de Uso*