Naked Objects

http://www.nakedobjects.org/

por Richard Pawson and Robert Matthews
tradução e revisão técnica: Osvaldo Kotaro Takai e João Eduardo Ferreira (takai,jef)@ime.usp.br

Introdução

Existem várias pessoas que acreditam que estão projetando e desenvolvendo orientado a objetos, mas na verdade não estão! Isto porque elas ignoram o princípio mais fundamental da orientação a objetos. Nós descrevemos este princípio como 'completeza comportamental': *um objeto deve modelar completamente o comportamento das coisas que ele representa*.

Ao invés disso, essas pessoas continuam a projetar sistemas de negócio que separam o processo dos dados, embora revestidos com linguagens e tecnologias orientadas a objetos.

Por que esta distinção é importante? Porque a completeza comportamental é a chave para obter o principal benefício da orientação a objetos. Tal benefício pode ser resumido na habilidade de lidar com as inesperadas mudanças de requisitos.

Acreditamos que, mesmo que existam objetos com completeza comportamental, haverá forças de atração tentando separar dados de processos. Nós identificamos cinco dessas forças:

- Orientação do processo de negócio.
- Interfaces de usuário otimizadas a tarefas.
- Métodos orientados a use-case.
- Padrão *Model-View-Controller*.
- Desenvolvimento de software baseado em componentes.

O problema é que além dessas cinco forças serem consideradas as melhores práticas, elas também têm o status de 'vaca sagrada' entre a comunidade de desenvolvimento de software. Nós não estamos dizendo que essas forças sejam ruins, estamos apenas lembrando que elas atrapalham a realização de bons projetos orientados a objetos.

O Naked Objects[1] é um framework escrito em Java, projetado especificamente para encorajar a criação de sistemas de negócio a partir de objetos de negócio comportamentalmente completos. De fato, com o framework Naked Objects você não tem alternativa a não ser a de fazer com que os seus objetos de negócio sejam comportamentalmente completos. A razão é que o framework expõe o núcleo dos seus objetos de negócio, tais como Clientes, Produtos e Pedidos, diretamente ao usuário. Todas as ações consistem de chamar métodos diretamente desses objetos de negócio, ou algumas vezes das classes desses objetos. Não existem scripts ou controladores, muito menos caixas de diálogo entre o usuário e os naked objects[2]. Os naked objects são projetados para funcionar com o framework, e então são expostos diretamente aos usuários.

Sistemas construídos usando o framework Naked Object são ágeis. Isso significa que eles podem acomodar mudanças de requisitos de negócio, porque os objetos comportamentalmente completos entalham a funcionalidade do sistema como sendo uma parte natural do próprio objeto. Além disso, a falta de construtores adicionais ao redor dos naked objects indica que eles não terão muita coisa que possa ser mudada.

Os sistemas resultantes são também mais participativos da perspectiva do usuário. Por permitir que usuários acessem diretamente os naked objects, ao invés de restringir a sua interação através de tarefas prescritas, demonstra que são muito intuitivos e claramente cooperativos. Isso é assegurado pelo uso extensivo de gestosm, arrastar e soltar que o framework fornece automaticamente. O resultado final é que sistemas de negócio projetados dessa maneira se parecem mais como uma planilha eletrônica ou um programa de desenho que um sistema transacional convencional: eles tratam o usuário como um solucionador de problemas, não meramente como um seguidor de um processo. Isso pode trazer significativos benefícios não apenas em termos de motivação do usuário, mas no aprimoramento dos serviços aos clientes e nas tomadas de decisões operacionais.

Muito mais significativo que o aprimoramento dos sistemas resultantes é impacto no processo de desenvolvimento. Os naked objects fornecem uma linguagem comum entre o desenvolvedor e o usuário, a qual melhora significativamente a comunicação durante o processo de exploração de requisitos. O desenvolvedor não precisa escrever uma única linha de código para desenvolver a interface do usuário, pois ela pode ser autogerada a partir das definições dos objetos de negócio. De fato, toda a noção convencional de interface de usuário simplesmente desaparece. Isso facilita a prototipação rápida. Mas diferente de outras formas de prototipação rápida, você não faz meramente a prototipação da interface do usuário – você está simultaneamente prototipando o núcleo do modelo de objetos.

O objetivo deste livro é introduzir os conceitos de projetar sistemas de negócio a partir dos naked objects e habilitá-lo para que você tenha condições de construir

[1] O termo será utilizado originalmente em inglês, pois ainda não encontramos uma tradução adequada. A tradução literal para objetos pelados não representa seu verdadeiro siginificado.

[2] Sempre que o termo 'Naked Objects' iniciar com letras maiúsculas, estaremos nos referindo ao framework Java. Quando começar com letras minúsculas, estaremos nos referindo a objetos de negócio.

tais sistemas usando o framework Naked Objects. O livro irá atrair principalmente dois tipos de leitores: modeladores de objetos que têm ao menos algum conhecimento de programação Java e desenvolvedores de banco de dados que possuem algum conhecimento de modelagem de objetos.

Sem dúvida uma das mensagens deste livro é que as noções de modelagem de objetos de negócio e programação orientada a objetos estão muito mais próximas do que convencionalmente se apresenta.

Estrutura do Livro

A Seção 1 deste livro fornece um breve histórico da orientação a objetos e uma análise das práticas que tendem a forçar a separação do processo dos dados nos projetos de sistemas de negócio. A seção é finalizada com um conjunto de novos princípios de projeto que podemos chamá-lo de 'manifesto do naked objects'. Você não precisa ler esta história nem a análise a fim de tirar proveito do Naked Objects – você pode ir diretamente para as seções práticas se preferir – mas você irá descobrir que este conhecimento pode ampliar seu entendimento sobre a natureza da orientação a objetos e a razão pelas quais as correntes práticas se desviaram da intenção original.

A Seção 1 apresenta um **Estudo de Caso** sobre o Departamento de Assuntos Sociais e da Família do governo irlandês (*Department of Social and Family Affairs* – DSFA). Essa foi à primeira oportunidade de aplicar os princípios de projeto para um sistema de negócio de missão crítica, e foi um grande sucesso. O DSFA usou uma versão inicial de nosso framework para prototipar seu sistema, mas eles nos encarregaram de ditar a arquitetura para implementar o projeto resultante. Isso foi uma das coisas que nos estimularam a re-desenvolver o Naked Objects a fim de adequá-lo ao desenvolvimento completo do sistema.

A Seção 2 faz uma revisão geral do framework Naked Objects e descreve os principais benefícios que você pode obter ao utilizá-lo, tanto em termos de quantidade dos sistemas resultantes, quanto em termos de aprimoramentos no processo de projeto e desenvolvimento. Esta seção finaliza com um conjunto de críticas mais freqüentes que temos recebido durante os últimos dois anos.

Esta seção também apresenta um segundo **Estudo de Caso** que mostra o primeiro protótipo de um sistema de reservas para Serviços de Carros Executivos. Este caso fornece uma explicação visual da íntima correspondência entre visão do usuário dos naked objects e a estrutura fundamental da aplicação escrita em Java. Se você quiser uma rápida introdução sobre o que o framework faz, este estudo de caso é um bom ponto de partida.

A Seção 3 fornece uma introdução à programação com o framework Naked Objects. Esta seção é escrita para programadores Java. Quando possível, as técnicas de programação são ilustradas usando o sistema descrito no caso de estudo anterior, bem como o código completo fornecido junto com o framework Naked Objects.

Nesta seção temos o próximo **Estudo de Caso** que descreve a experiência da *Safeway Stores*, a quarta maior rede de supermercados da Inglaterra. Inicialmente, a Safeway começou explorando Naked Objects com a intenção de reforçar as habilidades de modelagem de objetos de seus desenvolvedores Java, e desde então fizeram uso do framework tanto para desenvolver protótipos quanto para desenvolver sistemas de fato. O caso descreve como a Safeway interligou o Naked Objects com o EJB – *Enterprise Java Beans* e implantou esta combinação com o nível de desempenho parecido com os sistemas de processamento de transações baseados em computadores 'de grande porte' (mainframes).

A Seção 4 examina o processo de projeto e desenvolvimento para um projeto Naked Objects. Recomendamos que você gerencie o projeto em três fases: exploração, especificação e liberação. As duas últimas podem ser gerenciadas dentro do contexto de um método existente, seja o Processo Unificado ou *Extreme Programming*.

Depois disso, segue-se um **Estudo de Caso** descrevendo algumas experiências dessa fase de exploração conduzida num grande banco financeiro. Neste particular caso examinamos como os naked objects se comportam num domínio que tradicionalmente é dominado por uma visão de negócio orientado a processos.

A Seção 5 faz um breve exame sobre o que o futuro reserva ao Naked Objects. O framework está sendo constantemente aprimorado e recursos novos estão sendo adicionados, dirigidos pelas necessidades da comunidade de usuários e desenvolvidos pelos interessados em códigos abertos. Nós explicamos onde procurar por mais informações, como se envolver nesse processo de desenvolvimento e as várias outras maneiras de estender o framework.

O **Estudo de Caso** final relaciona-se com um conglomerado norueguês, Norsk Hydro, que usou o framework Naked Objects para construir um protótipo altamente gráfico, no qual todas as ações do usuário são executadas sobre objetos dentro de uma representação de mapa da rede elétrica da Europa. Este caso envolveu a adição de um outro tipo de visualização ao framework que, quando finalizado, a toda a aplicação de negócio pôde ser escrita sem uma única linha de código relativa à interface de usuário.

No final do livro existem vários **Apêndices** úteis. O primeiro introduz todo o processo de instalação do framework e construção de uma aplicação muito simples, passo a passo.

Você pode obter o framework Naked Objects e sua documentação a partir do nosso site: http://www.nakedobjects.org. O framework Naked Objects está evoluindo rapidamente. Nós restringimos o conteúdo deste livro aos aspectos do framework as quais acreditamos estarem razoavelmente estáveis. Quaisquer atualizações necessárias serão fornecidas em nosso website.

Como um projeto *Open Source,* esperamos que você não apenas faça uso do framework e das técnicas associadas, mas que pense em contribuir de alguma maneira para avançar o desenvolvimento. Por enquanto: divirta-se!

Agradecimentos

As idéias agora embutidas em Naked Objects baseiam-se nos resultados de dez anos de pesquisa sobre técnicas de orientação a objetos, arquiteturas para agilizar negócios e 'sistemas expressivos', conduzidos por Richard, e financiado pelo CSC's Research Services. Nós agradecemos a generosidade ao permitir que os frutos desta pesquisa sejam publicados, bem como a contribuição realizada pelos colegas da CSC de Richard durante as várias discussões e debates. Três anos atrás, Richard se associou com o Robert Matthews, amigo de longa data e antigo colega, com quem começou a projetar um framework Java para incorporar os princípios de projeto descobertos por esta pesquisa.

Embora alguns tenham elogiado o Naked Objects como uma nova e importante idéia, nós preferimos vê-lo como uma tentativa de retomar uma velha idéia. O que levou ao Naked Objects foi a nossa crença de que os aspectos mais poderosos do conceito original da orientação a objetos, agora com quase quarenta anos de idade, não são entendidos por muitas pessoas que afirmam praticar OO. Nós tentamos resumir as percepções de alguns dos pioneiros em projetos de software orientados a objetos na primeira seção do livro, mas lamentamos que esta estória não esteja tão detalhada. Nós agradecemos profundamente estes pensadores. Algumas dessas pessoas cujos trabalhos admiramos há vários anos – Alan Kay, Rebecca Wirfs-Brock, Oliver Sims, Dave Thomas – nos deram coragem neste projeto, nos ajudaram a fazer com que as nossas idéias fossem amplamente conhecidas e fizeram importantes críticas nos rascunhos iniciais. Várias outras pessoas deram contribuições relevantes nos últimos manuscritos, entre elas James Cooper, Ian Graham, Kevlin Henney, Alan Griffiths, Dan Haywood, e Andrew Broughton. A rigorosa demanda do orientador de doutorado de Richard, Vincent Wade em Trinity College Dublin, trouxeram, indiretamente, vários aprimoramentos neste livro.

Igualmente importante foi a contribuição daqueles que já colocaram nossas idéias em prática. O Departamento de Assuntos Sociais e da Família da Irlanda (*Department of Social and Family Affairs* – DSFA), foi a primeira organização que levou a sério as idéias de Richard, e encontra-se agora em sua segunda grande fase de desenvolvimento, usando os mesmos princípios de projeto que o Naked Objects, embora com suas próprias tecnologias. A equipe da DSFA – Philip, Niall, Joan, JB, Helen, Peg e muitos outros – têm feito um bom trabalho nesses últimos três anos. Seus constantes desafios significam que os princípios de projeto estão verdadeiramente atravessando um teste de fogo!

Em Norsk Hydro, Ragnar Blekeli viu o potencial dos Naked Objects e ficou atento aguardando até o momento em que ele pôde realizar seu primeiro projeto exploratório dentro de sua organização. As pessoas em Safeway trabalharam muito para tornar o Naked Objects uma ferramenta viável para o desenvolvimento

de sistemas de negócio reais. Rick Smith foi um campeão incansável para o nosso trabalho. Dave Slaughter, um desenvolvedor muito dinâmico, tornou-se o principal parceiro de Robert nos assuntos de infra-estrutura do framework. Ele ligou o Naked Objects com EJB, e implementou sobre um servidor mainframe, acabando com as predições de que os naked objects não atenderiam aos requisitos de desempenho empresarial. Muitas outras pessoas aplicaram nossos conceitos, os testaram e fizeram várias sugestões de aprimoramentos. Agradecimentos especiais para Suki, Alison, Pam, Ian e Chris.

Antes deste livro havíamos feito muito pouco esforço para tornar público o Naked Objects. Apesar disso, a comunidade de desenvolvedores cresceu constantemente. Robert gostaria de agradecer a aqueles desenvolvedores que estiveram envolvidos nos testes do framework ou que fizeram contribuições e sugestões específicas, especialmente a Frank Harper, Paul Hammant, Sylvian Liege, Paul M Bethe, Mark Crocker, Bjarte Walaker, Lindsay Laird e Per Lundholm.

O livro é fruto de um esforço de equipe. Agradecemos a Karen Mosman por patrocinar nossa causa com a Wiley, e a Robert Hambrook por concordar em fazer mudanças na tecnologia de produção apesar do curto cronograma. E agradecemos também por nos deixar trabalhar com a nossa editora favorita, Anne Pappenheim, e com o projetista, Ian Head, os quais foram responsáveis pelo logo do Naked Objects usados na capa.

Finalmente, agradecemos às nossas respectivas famílias por aceitar as longas horas e frustrações que qualquer livro parece provocar, e por pacientemente explicar aos amigos que nosso projeto ‘naked objects’ não era o que eles pensavam que fosse. ;-)

Richard Pawson e Robert Matthews, Agosto de 2002.

Informação da Versão

Esta é a primeira impressão da primeira edição do livro.

Para quaisquer correções desta impressão ou atualizações subseqüentes veja http://www.nakedobjects.org.

Este livro é compatível com a versão 1 do framework Naked Objects, que por sua vez é compatível com Java versão 1.1.7a ou posterior. (O desenvolvimento atual do Naked Object está sendo realizado com Java 2 SDK versão 1.3.1).

Uma observação crítica da orientação a objetos

'Nova Iorque, 5 de fevereiro de 2002… A ACM – *Association for Computing Machinery* – apresentou o prêmio A. M. Turing, considerado o 'Prêmio Nobel da Computação', para Ole-Johan Dahl e Kristen Nygaard da Noruega pela invenção da programação orientada a objetos, o modelo de programação mais amplamente usado nos dias de hoje. Seus trabalhos foram fundamentais para a mudança em como os sistemas de software são projetados e programados, resultando em aplicações reusáveis, confiáveis e escaláveis; e simplificou o processo de escrever códigos e ainda facilitou a programação de software'

www.acm.org/announcements/turing_2001.html

Agora é oficial: a orientação a objetos ganhou. Acabou a discussão, é hora de festejar e aprimorar!

Pedimos licença para discordar!!!

Mais e mais sistemas de negócio são projetados usando métodos orientados a objetos e escritos em linguagens de programação orientadas a objetos tais como Java. Suas interfaces são invariavelmente projetadas com ferramentas baseadas em objetos tais como Visual Basic. E exceto por alguns sistemas mainframes, eles são construídos no topo de infra-estruturas de objetos distribuídas tais como COM+, EJB ou CORBA. Assim, como podemos discutir o sucesso do paradigma da orientação a objetos?

Podemos discutir devido às práticas atuais quase sempre não demonstrarem compromisso com a verdadeira essência da orientação a objetos. Definimos essa essência como 'completeza comportamental'. Para entendermos este conceito e seu significado, é importante fazer um breve relato sobre a história da orientação a objetos.

Uma breve história de objetos

O conceito de software orientado a objetos tem quase quarenta anos de idade. Durante esse período, as idéias evoluíram consideravelmente, e sua evolução pode ser grosso modo, dividida em quatro fases:

- Simula e o nascimento da programação orientada a objetos.
- Smalltalk e a interface de usuário orientada a objetos.
- O surgimento dos métodos orientados a objetos.
- Infra-estruturas de objetos distribuídos.

Simula e o nascimento da programação orientada a objetos

A idéia de software orientado a objetos surgiu na Noruega em meados de 1960, com Simula, uma extensão da linguagem de programação Algol. Simula foi projetada para facilitar a escrita de programas que simulassem fenômenos tais como processos industriais, problemas de engenharia ou doenças epidêmicas [Dahl 1966].

Antes, todas as linguagens e técnicas de programação separavam explicitamente um sistema em processos e dados. Sua suposição foi a de que um sistema de computador aplicava repetidas vezes o mesmo processo em diferentes dados.

A simulação questionou esta suposição. Algumas vezes os dados são fixos e o programador manipula as características funcionais do sistema até que os critérios de saída requeridos sejam encontrados. Por exemplo, os dados podem representar a aspereza de uma típica estrada e o programador pode alterar o projeto de um sistema de suspensão de caminhão até que a qualidade desejada durante o percurso seja obtida.

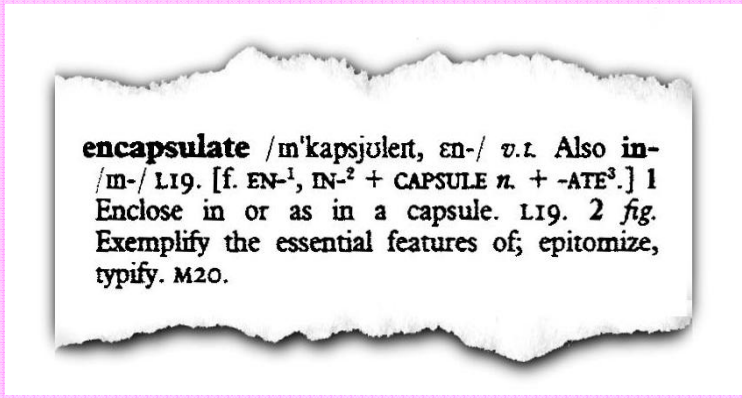
Algumas vezes é até difícil separar dados de funcionalidades: quando você adiciona um outro eixo ao seu caminhão simulado, por exemplo, você está alterando os dados (o número de rodas) ou a funcionalidade (a maneira como o caminhão se move na estrada num percurso)?

Os inventores de Simula tinham a idéia de construir sistemas de ‘objetos’, onde cada um representa algum elemento dentro do domínio de simulação. Uma simulação normalmente envolve classes de objetos – uma classe Roda, uma classe Mola, uma classe Chassis, entre outras. Cada classe forma um modelo a partir do qual instancias individuais são criadas quando necessários para a simulação.

Cada objeto de software não apenas conhecia as propriedades da entidade do mundo real que eles representavam, mas também sabia como modelar o comportamento dessas entidades. Assim, cada objeto Roda sabia não só as dimensões e peso de uma roda, mas sabiam também como girar, saltar, modelar o atrito e transmitir as forças para o objeto Eixo. Esses comportamentos podiam operar continuamente, ou eles podiam ser chamados especificamente enviando uma mensagem ao objeto.

Podemos chamar esse princípio de ‘completeza comportamental’. Isso não significa que o objeto deva implementar todos os comportamentos possíveis que um dia pudéssemos precisar. Significa que todos os comportamentos associados a um objeto, que são necessários à aplicação sendo desenvolvida, deveriam pertencer a esse objeto e não implementados em um outro lugar dentro do sistema.

A palavra ‘encapsulamento’ é normalmente usada neste contexto. A palavra tem dois significados em Inglês. O primeiro tem a ver com existência fechada, como uma cápsula medicinal. Este é o significado que muitas pessoas usam no contexto de orientação a objetos: um objeto é fechado por uma interface de mensagem, com uma implementação interna escondida. Este é uma propriedade importante de objetos, mas não é a única. As idéias de operação caixa-preta e ‘ocultamento de informações’ são comuns em muitas formas de desenvolvimento de sistemas baseados em componentes. O segundo significado de encapsulamento é que alguma coisa exemplifica as características essenciais de outra coisa, como em ‘este documento encapsula nossa estratégia de marketing’. Este segundo significado – que corresponde ao princípio de completeza comportamental – é muito mais importante no contexto de modelagem orientada a objetos.



encapsulate /ɪn'kapsjʊleɪt, ɛn-/ *v.t.* Also **in-** /ɪn-/ L19. [f. EN-[1], IN-[2] + CAPSULE *n.* + -ATE[3].] 1 Enclose in or as in a capsule. L19. 2 *fig.* Exemplify the essential features of; epitomize, typify. M20.

O Pequeno Dicionário Oxford revela os dois significados da palavra ‘encapsulamento’. No contexto de orientação a objetos, muitas pessoas assumem o primeiro significado, mas o segundo significado é a mais importante.

O valor da completeza comportamental é que qualquer mudança requerida na aplicação seja mapeada de forma simples para mudanças no código do programa. Por exemplo, adicionar uma válvula entre dois canos num modelo Simula que representa uma refinaria de óleo simplesmente envolveria criar uma nova instância da classe Válvula, configurar seus parâmetros de operação e ligar os objetos Cano de maneira apropriada. O novo objeto válvula trouxe com ele a habilidade de abrir e fechar, alterando o fluxo de óleo apropriadamente, bem como o modelo de impacto sobre o custo da construção. Se a mesma refinaria fosse modelada usando uma linguagem de programação convencional, vários outros comportamentos associados à válvula estariam, provavelmente, distribuídos dentro do programa e assim difícil de encontrar e mudar.

Smalltalk e a interface de usuário orientada a objetos

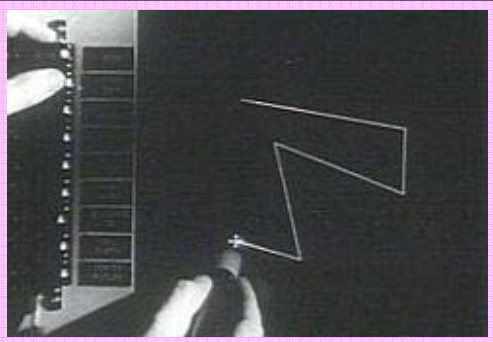
Embora o trabalho dos Noruegueses tenha continuado por mais alguns anos, por volta de 1970, uma nova linha de pensamento orientada a objetos emergiu do novo Centro de Pesquisas em Palo Alto da Xerox – Parc. Alan Kay, que liderou o Grupo de Pesquisas de Aprendizagem da Parc foi seduzido pela orientação a

objetos por várias razões. A primeira tinha a ver com a escalabilidade. Naquele tempo, muitas pessoas estavam preocupadas com a escalabilidade do software, mas o que muitos deles entendiam por escalabilidade era aperfeiçoar um ou dois níveis de complexidade.

Mas em 1965, Gordon Moore, que mais tarde co-fundou a Intel, escreveu na revista Electronics que o número de transistores num circuito integrado iria continuar a duplicar a cada ano ao menos durante os próximos 10 anos. A tendência atual está próxima de duplicar a cada dois anos e isso continua até os dias de hoje. Kay foi um dos poucos pesquisadores a perceber as implicações dessa novidade com maior seriedade, conhecida como a lei de Moore. Ele se interessou em como a complexidade de software poderia ser aperfeiçoada por um fator de um bilhão para tomar vantagens desse hardware. A concepção de Kay do futuro da computação – dos computadores do tamanho de um notebook com conexões sem fio dentro de uma rede gigante de informações – parecia pura ficção científica no início dos anos 90.

Fazendo uma analogia com a microbiologia, Kay disse que a única maneira de aperfeiçoar a complexidade do software por um fator de um bilhão era se o software pudesse ser homogêneo e similar em todos os sentidos: o bloco construtor mais elementar do software foi, com efeito, finalizado por computadores em miniatura – em outras palavras, ‘objetos’.

A primeira aplicação real desta potencial complexidade foi na interface de usuário. Interface gráfica não era uma idéia nova: Ivan Sutherland tinha demonstrado muitas idéias sobre saídas gráficas e manipulação direta das entradas com seu sistema Sketchpad em 1963 [Sutherland1963], mas suas idéias não eram fáceis de generalizar para outras aplicações. No início dos anos 70, a redução nos custos de processamento tornou possível a criação de efeitos similares em software puro usando display de matriz de bits. A equipe da Parc usou as técnicas da programação orientada a objetos para gerenciar centenas de objetos gráficos sobre um display de matriz de bits simultaneamente, cada um monitorando e reagindo às mudanças em outros objetos a eles associados, ou aos eventos iniciados pelo usuário tais como movimentos do mouse. Embora existissem centenas e não bilhões de objetos, esta já estava além da complexidade que poderia ser alcançada usando abordagens de programação convencional.



Sketchpad de Ivan Sutherland, desenvolvido em 1963, introduziu idéias de saída gráfica e manipulação direta das entradas. De certa forma, o Sketchpad antecipou as idéias de interfaces de usuário orientadas a objetos, mas era difícil de generalizá-la.

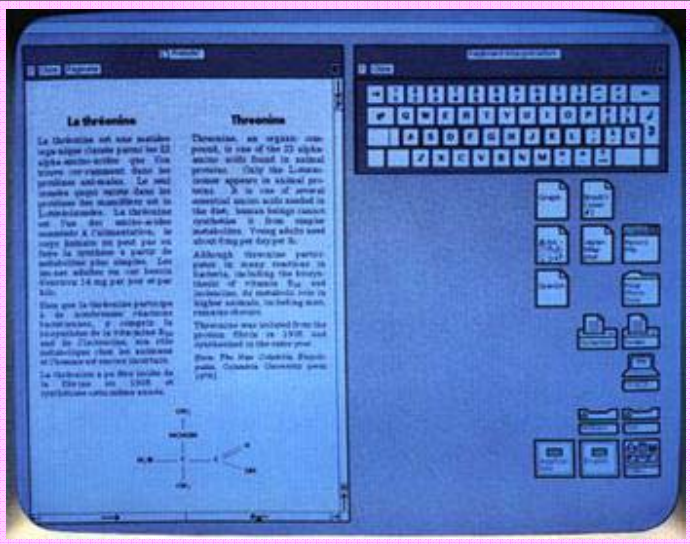
Kay também observou que o conceito de objetos tinha enorme potencial como uma ferramenta cognitiva: havia uma boa correspondência com a maneira de pensar das pessoas sobre o mundo [Kay1990]. Ele notou que considerando um substantivo isolado faz aparecer uma imagem concreta na mente das pessoas (pense numa maçã), um verbo isolado normalmente não tem esse efeito (tente visualizar 'correr'). Isso porque os verbos são efetivamente propriedades dos substantivos: os garotos correm, o cachorro corre, a água corre, o trem corre. Isso despertou o princípio da orientação a objeto conhecido como 'polimorfismo': você pode assumir o mesmo comando (verbo) para diferentes objetos, mas é deixado para o objeto decidir como executar este verbo.

Um dos produtos desta maneira de pensar foi a linguagem Smalltlak [Kay1996]. Embora o crescimento subseqüente tenha sido uma linguagem de programação profissional, a idéia original do Smalltalk era de uma linguagem que crianças poderiam usá-la para instruir um computador a realizar tarefas simples e então construir tarefas mais complexas.

Um outro resultado foi o poder que temos hoje de descrevermos uma interface de usuário orientada a objetos ou IUOO. Infelizmente, o termo se diluiu com o passar dos anos. Por exemplo, uma das referências mais populares de projeto de IUOO [Collins1995] utiliza o exemplo de uma simples calculadora, contrastando com o velho estilo de uma versão de linha de comando com o acessório agora familiar fornecida tanto pelas interfaces do Windows quanto do Mac. A calculadora possui uma interface gráfica (uma figura na tela), reforça a metáfora do usuário (parecer como uma calculadora do mundo real), e usa a manipulação direta [Schneiderman1982] (você aponta e clica sobre os botões). Todos esses conceitos foram associados a IUOO, mas elas são meramente produtos. A essência de uma IUOO é que o usuário pode referenciar objetos individuais, e identificar e ativar comportamentos de objetos diretamente a partir dessas referências. Uma implementação efetiva disso representa os objetos como ícones, e o comportamento como ações sobre um menu pop-up. No entanto, é perfeitamente possível ter uma interface de usuário orientada a objetos onde a interação do usuário é via uma linha de comando, desde que a forma desses

comandos seja objeto-ação (isto é, substantivo-verbo). De fato, alguns dos primeiros exemplos do uso do Smalltalk para propósito de ensino usam esta abordagem.

Estes conceitos foram inicialmente implementados em Alto, uma máquina experimental que se ficaria conhecida como o primeiro computador pessoal. A primeira realização comercial foi no Workstation ‘Star’ 8010 da Xerox, que chegou com um conjunto fixo de aplicações para processamento de texto, desenho e outras funções relacionadas a documento. O Alto e Star inspiraram diretamente na criação do Apple Lisa e Macintosh [Levy1994], e subseqüentemente a Microsoft Windows e Office, e uma grande gama de outros sistemas.



O workstation ‘Star’ 8010 da Xerox foi a primeira realização comercial das idéias de interfaces de usuários orientadas a objetos desenvolvidas pela Parc da Xerox no início dos anos de 1970.

Ainda que as idéias de uma metáfora desktop, ícones e sobreposição de janelas estejam impregnados agora, muitas das principais idéias foram ignoradas, distorcidas ou diluídas a ponto de perderem seu valor. Muitas aplicações agora escritas para executar dentro de ambientes GUI modernos mostram pouca evidência dos pensamentos da orientação a objetos. Considere dois exemplos.

Os primeiros ícones foram concebidos na Parc para representar os substantivos instanciáveis, apesar disso, os ícones atualmente são mais utilizados nas barras de ferramentas para representar ações ou verbos. Isso não é uma boa idéia, isso distancia os desenvolvedores dos conceitos mais importantes de ícones como substantivos, onde o ícone indica as propriedades e comportamentos que podem ser ativados sob o item em questão.

Segundo, na maioria dos casos existe uma barra de menu estático no topo da janela ao invés de menus pop-ups, os quais significam que verbos e substantivos

estão fisicamente separados. Tanto no sistema operacional Windows quanto o Mac agora possuem menus pop-up. Entretanto curiosamente, eles são chamados de 'short-cuts', indicando que o principal local da ação está em outro lugar.

Algumas pessoas respondem que nos processadores de textos, por exemplo, várias ações se aplicam ao documento como um todo ao invés de se aplicarem aos objetos individuais. Isso implica que o projetista deve fornecer ou uma maneira fácil de selecionar todo o documento, ou uma representação icônica direta de todo o documento com seu próprio menu pop-up.

Poucas aplicações reconheceram os conceitos originais de orientação a objetos e continuaram com eles, mas quase em sua maioria estavam preocupados com projetos de interfaces gráficas, desktop publishing e outras atividades relacionadas a documentos. Nem os pesquisadores originais da Parc, nem seus sucessores imediatos na Apple, mostraram interesse no mundo dos sistemas transacionais de negócio que são responsáveis, atualmente, pela maior parte dos usuários de computador por todo o mundo. Em vez disso, técnicas orientadas a objetos desenvolveram sua própria maneira de desenvolver sistemas de negócio tradicionais na forma de métodos de análise e/ou projeto.

O surgimento dos métodos orientados a objetos

O primeiro método que incluiu princípios de orientação a objetos e pôde ser aplicado nos sistemas de negócio tradicionais surgiu no final dos anos de 1980. Eles tinham características comuns, mas também tinham diferenças sutis, cada um com vantagens ou aplicabilidades específicas.

Muitos desses métodos aplicaram conceitos orientados a objetos a partir de práticas existentes. Por exemplo, a Análise de Sistemas Orientada a Objetos de Shlaer e Mellor [Shlaer1988] e Técnicas de Modelagem de Objetos de Rumbaugh [Rumbaugh1991] ambos desenvolveram-se das técnicas de modelagem de dados. Para colocar de forma positiva, esses métodos atendiam as noções incipientes de análise e projeto orientados a objetos com técnicas comprovadas de engenharia de software. Mas, por outro lado, pode-se argumentar que eles carregaram muitas bagagens que sobrecarregaram a abordagem orientada a objetos.

Apesar das afirmações de que esses métodos satisfaziam plenamente ao desenvolvimento de sistemas de negócio tradicionais, muitos deles foram originalmente projetados para atender às necessidades de sistemas de engenharia de grande escala, tais como sistemas de defesa (no caso do Projeto Orientado a Objetos de Boch [Booch1986]) ou de centrais telefônicas (no caso da Engenharia de Software Orientada a Objetos de Jacobson [Jacobson1992]). Algumas das características da engenharia de sistemas são consideravelmente mais exigentes do que os sistemas de negócio, em assuntos específicos relacionados ao desempenho em tempo-real, confiabilidade e segurança. No entanto, em outros aspectos, sistemas de engenharia são mais fáceis de projetar do que sistemas de negócio. Os requisitos de software em sistemas de engenharia, em muitos casos, serão especificados por engenheiros de produtos

ao invés de serem projetados por clientes; e embora os requisitos possam mudar com a evolução geral do projeto, não é tão difícil de se obter um razoável primeiro corte na especificação de requisitos. Para aqueles que têm a responsabilidade de obter requisitos para sistemas de negócio, isso parece um luxo.

Na década seguinte, esses métodos começaram a convergir. A UML (*Unified Modeling Languages*) emergiu como um padrão para representar projetos orientados a objetos de forma gráfica, e três dos metodologistas pioneiros (Booch, Jacobson e Rumbaugh) colaboraram para especificar o Processo de Desenvolvimento de Software Unificado [Rumbaugh1999].

Uma abordagem que ainda é o único da espécie é a noção de RDD (Responsability-Driven Design) de Wirfs-Brock [Wirfs-Brock1989]. Ainda não seja fundamentalmente incompatível com os outros metodologistas, RDD empregou maior ênfase na noção de responsabilidades de objetos. O método ensina que os objetos devem ser concebidos somente em termos de responsabilidades que eles devem preencher. Elas podem ser, em geral, dividida em coisas que o objeto é responsável por conhecer, e coisas que o objeto é responsável por fazer, com maior ênfase quando possível na última. Em outras palabvras, os 'o quês' são especificados em termos de o que o objeto deve conhecer a partir da perspectiva externa, não no que pode ou não armazenar internamente. O uso dos cartões CRC (*Class-Responsibility-Collaboration*) [Beck1989] é uma técnica útil para registrar tais responsabilidades durante os estágios iniciais de análise e/ou projeto. O valor de ambas as abordagens de RDD e a técnica de peso leve dos cartões CRC é que eles encorajam a noção de completeza comportamental. Como veremos brevemente, isto não é muito verdadeiro em outros métodos.

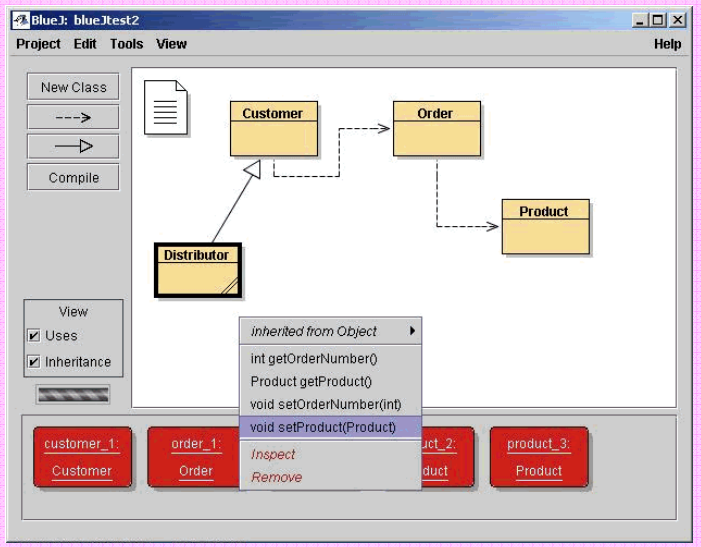
No início dos anos de 1990 existiam vários sistemas de negócio de larga escala projetados com sucesso usando abordagens orientadas a objetos. Mas a demanda da orientação a objetos em termos de gerenciamento de memória e poder de processamento relativo às capacidades de hardware e sistemas operacionais da era impôs impedimentos técnicos substanciais.

Infra-estruturas de objetos distribuídos

Durante os anos de 1990, a ênfase trocou as tecnologias de infra-estrutura necessária para apoiar sistemas de negócio orientados a objetos no ambiente empresarial, incluindo versões comerciais da linguagem Smalltalk, banco de dados orientados a objetos, e vários tipos de middleware para sustentar objetos distribuídos. A formação do Grupo de Gerenciamento de Objeto comanda a especificação do CORBA (*Common Object Request Broker Architecture*) e uma série de outros padrões públicos, que eventualmente serão implementados em dúzias de produtos. As duas tecnologias proprietárias, DCOM/COM+ da Microsoft e EJB/J2EE da Sun, também desenvolveram substanciais bases de instalação durante os anos de 1990. Todas estas três tecnologias forneceram a infra-estrutura de serviços necessários para implementar sistemas orientados a objetos em escala empresarial, incluindo persistência, comunicação distribuída, segurança e autorização, controle de versão e monitoramento de transações.

Todas as três tecnologias fizeram pesado uso de princípios orientados a objetos tais como encapsulamento (no primeiro dos dois significados descritos anteriormente), passagem de mensagens, polimorfismo e em alguns casos até herança. A pesar disso, elas não foram primariamente concebidas com a intenção de apoiar o grande compromisso dos objetos comportamentalmente completos. Suas principais intenções eram de habilitar sistemas distribuídos, implementar arquiteturas em camadas e alcançar independência de plataforma.

A segunda metade dos anos de 1990 apresentou mais alguns novos desenvolvimentos que mais diretamente apoiaram no projeto de objetos comportamentalmente completos. Um foi a linguagem de programação Java, que se tornou muito popular. Java tornou a programação orientada a objetos consideravelmente fácil de implementar, devido ao seu apoio ao gerenciamento de memória e *garbage collection*. Mas a razão da popularidade da linguagem Java provavelmente foi mais pela portabilidade entre múltiplas plataformas, sua característica de segurança e sua disponibilidade para Internet do que suas características orientadas a objetos. Muitos programadores Java possuem atualmente um pobre entendimento das técnicas orientadas a objetos, e isso não ajuda o fato de que a linguagem Java é freqüentemente ensinada da mesma maneira como as linguagens procedimentais. Um livro texto da linguagem Java [Dietel1999] freqüentemente usado em cursos universitários, não introduz os conceitos de classes até a página 326! O projeto BlueJ é uma tentativa digna de reverter essa situação.



O BlueJ é uma ferramenta open-souce para Java que é especificamente projetada para ajudar pessoas a aprender Java como uma linguagem de programação orientada a objetos. Muitos livros textos de Java e ferramentas de de desenvolvimento tendem a inconscientemente reforçar o paradigma da linguagem tradicional procedimental. No BlueJ o usuário pode inspecionar diretamente as instâncias de objetos bem como as classes. A ferramenta também elimina a necessidade do programador escrever um método principal – uma característica de Java que pode encorajar o pensamento procedimental. BluJ é um excelente ambiente para se aprender Java, e é conceitualmente compatível com os pensamentos de Naked Objects.

O estado da arte

Onde nós estamos, no início dos anos do século vinte? Os obstáculos técnicos de projetar e implementar sistemas de negócio orientados a objetos foram eliminados, e a terminologia orientada a objetos penetrou, de certa forma, na maioria das organizações de desenvolvimento de sistemas. Mesmo assim, o os sistemas de negócio tradicionais continuam a separar processos dos dados. Eles podem ser rotulados como 'objetos de processos' e 'objetos de dados' mas a separação apesar de tudo é real. E a separação é contrária ao princípio mais importante da orientação a objetos: completeza comportamental.

Algumas pessoas vêem a idéia de objetos comportamentalmente completos como algo ideal e impossível, simplesmente não realizáveis em qualquer projeto prático de sistemas. Alguns dizem que a separação de processos e dados em algum nível é necessária e desejável para sistemas de negócio: com técnicas orientadas a objetos, esta separação meramente ocorre nas abstrações de alto nível do que em projetos clássicos de sistemas. Outros dizem que separação de processos e dados não tem importância: desde que uma organização obtenha algum benefício a partir da aplicação dos princípios orientados a objetos, dizem eles, não importa como, ou qual profundidade, eles se aplicam. Este ponto de vista é um exemplo

do argumento relativista de que não existe a tal coisa de bons projetos OO, algo mais do que um bom Inglês, ou uma boa música – o que funciona é o que conta.

Atualmente, existe um crescente consenso sobre o que constitui um bom projeto e implementação OO, refletida em vários livros respeitáveis sobre padrões de projeto, de heurísticas e de técnicas OO. Veja, por exemplo, Gamma et al [Gamma1995], Fowler [Fowler2000], Meyer [Meyer1998], Hunt e Thomas [Hunt2000], e Riel [Riel1996]. Embora a frase 'completeza comportamental' não seja um termo comum, várias heurísticas apontam para esta idéia. Riel, por exemplo, identifica tais heurísticas:

- Mantenha dados e comportamentos num único lugar.
- Distribua a inteligência do sistema horizontalmente, tão uniforme quanto possível, isto é, as classes do nível superior num projeto devem compartilhar o trabalho uniformemente.
- Não crie classes/objetos "deus" no seu sistema. Seja extremamente desconfiado de uma classe que cujo nome contenha Dirigir, Gerenciar, Sistema ou Subsistemas.

Tente provar formalmente que um conjunto de heurísticas de projeto é melhor do que outros raramente efetivos em qualquer domínio. E que para projetos de sistemas de negócio, dificilmente existem oportunidades de desenvolver o mesmo sistema de duas maneiras diferentes e compará-las.

Alguns experimentos limitados têm produzido resultados que apóiam a completeza comportamental. Por exemplo, um dos poucos exemplos documentados do mesmo sistema projetado usando duas abordagens distintas [Sharble1993] quando subseqüentemente analisados usando métricas aceitas tais como tráfico de mensagens [Wirfs-Brock1994] indicou que a abordagem onde as entidades de negócio foi mais comportamentalmente completa tinha menor acoplamento entre os objetos e deviam dessa forma, serem mais fáceis de estender e modificar.

Um outro estudo comparou dois projetos, um que nitidamente caracterizava as 'classes deus' [Deligiannis2002] e mediu o esforço requerido (pelos programados não familiarizados com este ou aquele projeto) para introduzir as mesmas modificações: novamente, o estilo mais comportamentalmente completo de projeto ganhou.

Devemos exigir que esses experimentos limitados formem uma prova conclusiva. De qualquer forma, o nosso objetivo aqui é provar as vantagens e desvantagens da completeza comportamental para aqueles que são céticos. Particularmente, nós queremos ajudar a grande comunidade de desenvolvedores que se identificaram fortemente com as metas de projetar sistemas a partir de objetos comportamentalmente completos – fato que os atraiu para a orientação a objetos em primeiro lugar – mas que disse nunca ter funcionado por completo. Eles dizem que projetos de sistemas de negócio parecem degenerar em processos e dados com quase a mesma inevitabilidade que o leite fica coalhado com soro, ou o molho para saladas em óleo e vinagre, e eles não podem explicar porque isso acontece.

Cinco práticas que separam processos e dados

A separação continuada de processos e procedimentos pode estar relacionada principalmente a inércia: isto é, com o como as pessoas aprenderem a projetar sistemas e que encontram dificuldades para pensar de outra maneira. No entanto, essa inércia individual é normalmente reforçada por um número de práticas organizacionais específicas que tendem a forçar a separação de processos e dados mesmo quando o projetista de software queira adotar uma abordagem mais pura da orientação a objetos. Nós identificamos cinco dessas práticas:

- Orientação a processos de negócio.
- Interfaces de usuário otimizadas a tarefas.
- Métodos orientados a use-cases.
- O padrão Model-View-Controller.
- Desenvolvimento de software baseado em componentes.

Pode se dizer que essa lista é controvertida! Vários, se não todos esses fenômenos possuem o status de vaca sagrada dentro da comunidade de desenvolvimento de sistemas. Nenhum deles pode ser descartado simplesmente como um hábito ruim. Todas elas são práticas conscientes de que ou claramente gera benefícios ou foi projetado para reduzir um risco conhecido no processo de desenvolvimento. Nós não estamos sugerindo que tais práticas sejam ‘ruins’; e sim que elas tem um efeito colateral de desencorajar o projeto de objetos comportamentalmente completos.

Porém, qualquer prática alternativa que proponha ir contra esta separação não pode desperdiçar os benefícios da abordagem titular, nem introduzir de certa forma, problemas em locais onde elas foram projetadas para superar.

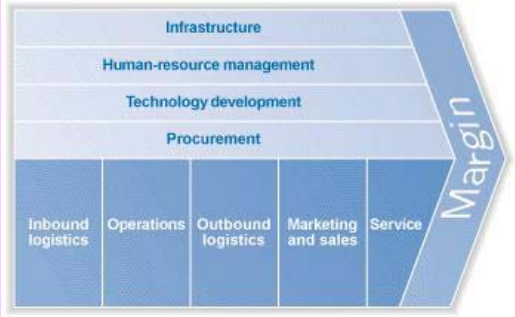
Orientação de processos de negócio

Antes de 1990, o termo ‘processo’ raramente se aplicava para negócios exceto naqueles que estavam interessados na manufatura de processos contínuos tais como processos petrolíferos e químicos. A idéia de modelar todas as atividades de negócio em termos de processos tornou se popular no início dos anos de 1990 com a idéia da reengenharia de processos de negócio [Hammer1993]. Depois de sair da moda no final da década de 1990, o pensamento de processos de negócio experimentou sua segunda onda, abastecido em parte pela emergência de uma nova geração de modelagem e ferramentas de gerenciamento de processos de negócio. Veja, por exemplo, http://www.bpmi.org.

A orientação a processos envolve seriamente duas idéias. A primeira é a de que você deve focar, organizar e alcançar os resultados definidos externamente (tais como preencher um pedido) ao invés de atividades puramente definidas internamente. Isto é uma contribuição útil. A segunda idéia é que processos podem e devem ser reduzidos para um processo determinista que transforma entradas em saídas.

O problema com essa noção de orientação a processo, como John Seeley Brown coloca, é que ela tende a se transformar num 'monoteístico' [Brown2000]. Como mais de um entrevistado nos diz, 'se em nossa organização não existir um processo, o gerenciamento não pode realizado'. Tal visão é central e danosa. Muitas coisas que um negócio faz simplesmente não se encaixam, de forma alguma, nesta forma de modelo de processo: existe um monte de atividades onde não é possível identificar entradas e saídas discretas, sem falar nos passos seqüenciais. Mesmo dentro de domínios que pode legitimamente ser descrito como processos, muitos das maiorias das atividades importantes, incluindo muitas atividades de gerenciamento e muitas formas de serviços de clientes, saem fora das definições formais de processo. Brown chegou a dizer que as pessoas mais importantes numa organização são precisamente aquelas que sabem como trabalhar ao redor de processos formais. Antropólogos sociais tiram uma definição útil entre 'processo' e 'prática', um gigantesco assunto que está fora do escopo deste livro.

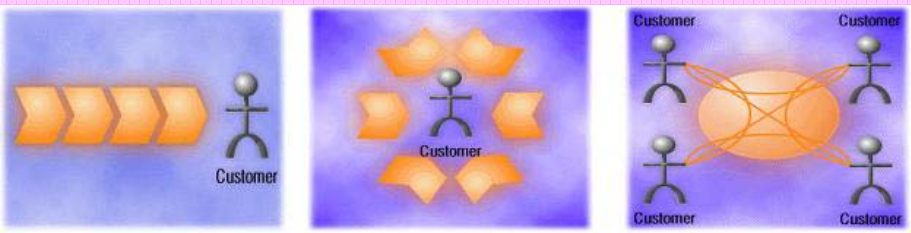
Um dos causadores desta obsessão por processos é a cadeia de valores de 'Porter' [Porter1985]. Michael Porter propôs isso como um modelo universal de negócio: a margem é criada através de um conjunto seqüencial de estágios adicionando valores. É claro como isso se aplica, dizer, General Motors, mas Porter argumenta que o mesmo modelo se aplica a um banco, onde 'a logística de chegada' é a tomada de depósitos, 'logística de saída' é o empréstimo, e por ai vai.



O modelo de negócio de cadeia de valores, como proposto pelo Prof. Michael Porter no início dos anos de 1980, é visto por muitos como um modelo universal de criação de valores de negócio. Este modelo reforça um estilo de orientação a processo de pensamento. No entanto, a validade universal tem sido fortemente desafiada.

Esta idéia tem sido desafiada por Charles Stabell e Oystein Fjeldstadt [Stabell1998] que dizem que a cadeia de valores é atualmente um modelo de negócio muito pobre, e seu uso pode conduzir a falhas perigosas no nível estratégico. Eles sugerem que existem três mecanismos diferentes através dos quais negócios criam valores: a cadeia de valores, loja de valores e rede de valores. Eles também argumentam que a proporção do negócio compatível com mo modelo de cadeia está decaindo rapidamente. Loja de valores (tal como consultorias, construtores e principalmente organizações de saúde) cria valor por aplicar recursos para resolver problemas individuais. Suas atividades raramente seguem uma seqüência linear e são freqüentemente iterativos por natureza. Rede de valores (que inclui muitas companhias de telecomunicações, bancos e seguradoras) cria valor pela venda de clientes para outros clientes. Suas criações de valores mostram efeitos de rede significativos, os quais não são as mesmas coisas como a 'economia de escala' proposta pela cadeia de valores.

O conceito de cadeia de valores enfatiza a adição seqüencial de valores para materiais de entrada. Isso combina bem com o paradigma de software de separar valores adicionando processos a partir de dados inanimados sob os quais eles operam. Se você perguntar aos profissionais de TI para definir o que é um sistema de informação, muitos irão dizer que é um mecanismo de transformar informações de entrada para informações de saída através da aplicação sucessiva de pequenas transformações. Historicamente isso pode ser uma descrição precisa, mas é uma maneira muito pobre de descrever as capacidades modernas de TI, que estão conceitualmente muito próximos a um modelo de rede de valores ou modelo de loja de valores. A metáfora que compara o papel dos sistemas de informação para uma linha de produção somente adiciona problemas.



Stabell e Fjeldstadt argumentam que existem três modelos fundamentais de criação de valores de negócio: a cadeia de valores (esquerda), a loja de valores (centro) e a rede de valores (direita). O modelo da loja de valores, em que recursos são organizados para resolver problemas individuais, e que representa uma proporção crescente do negócio, normalmente sofrem pelo apoio pobre de TI.

Interfaces de usuário otimizadas a tarefas

A mesma linha de pensamento ocorre numa escala muito menor em projetos de interfaces de usuário. Muitas interfaces de usuário são projetadas para implementar um conjunto finito de tarefas programadas. Isso está implícito no método de projeto de interfaces, como também fica explícito no projeto resultante: um menu de tarefas é apresentado ao usuário que o guia através da tarefa escolhida, selecionando as subopções quando requerido. O estilo de interface de usuário é conveniente para desenvolvedores de sistemas porque é fácil realizar o mapeamento para um conjunto de transações definidas, que um após o outro manipula estruturas de dados.

Uma outra idéia subjacente a esta abordagem é a de que as ações programadas é a chave para a otimização. Isso pode ser comparado diretamente as idéias de Frederick Taylor e seus princípios de gerenciamento científico [Taylor1911] (veja o painel).

O legado de Frederick Winslow Taylor

A busca de Frederick Taylor por ‘uma melhor maneira’ de trabalho começou com a própria tecnologia. Na casa de máquinas onde ele trabalhou, inicialmente como um artesão e então como capataz, conduziu milhares de experimentos para encontrar a velocidade ótima de corte para a operação de vários tipos de máquinas. Seus resultados mostraram significativos melhoramentos na média de produtividades comparado aos métodos gerais usados pelos trabalhadores. Seu próximo alvo foi a organização do trabalho. Novamente, usando milhares de experimentos, ele tentou encontrar a resposta ótima para todos os aspectos de organização do trabalho manual: desde o tamanho ótimo de uma pá a freqüência ótima de interrupção de trabalho. Para encorajar os trabalhadores a adotarem suas melhores práticas, ele defendeu a introdução da diferença percentual de peças, onde o percentual adicional de pagamento elevava-se com o seu desempenho. Taylor ainda não ficou satisfeito. Os trabalhadores usavam suas técnicas, mas não consistentemente. A solução foi remover todos os direitos de decisões dos trabalhadores criando roteiros de todas as suas ações. Taylor percebeu que os trabalhadores não desistiriam de sua autodeterminação tão facilmente. Para que eles obedecessem exatamente seus comandos – a seqüência de tarefas como ele havia definido, organizou os locais de trabalho e disse, trabalhem quando ele ordenar, e descansem quando ele ordenar – ele terá que aumentar o percentual geral de pagamento. A quantidade de aumento desse pagamento foi apenas um outro valor a ser determinado cientificamente: girou em torno de 35%. O biógrafo Robert Kanigel chamou isso de ‘ato Faustiano’ (Kanigel 1997). Taylor acreditava que ele estava motivando os trabalhadores, no sentido específico da motivá-los a obter uma vida melhor. Mas no sentido moderno da palavra, ele foi evidentemente um desmotivador. As instruções programadas eram fornecidas em cartões indexados. Podemos apenas especular como ele poderia reagir com as tecnologias de informação modernas, mas nós suspeitamos que ele teria se deleitado pela sua capacidade de estender seus métodos tanto em largura quanto em profundidade. Muitos sistemas de negócio tratam o usuário como um mero seguidor de processos. O sistema controla todo o processo, subcontratando o usuário somente para aquelas subtarefas que ele não está capacitado a realizar autonomamente.

Barbara Garson sugeriu em seu livro, 'The Electronic Sweatshop: How Computers turned the Office of the Future into the Factory of the Past' [Garson1988], que este paradigma não era necessariamente dirigido pela eficiência: ‘Presumo que

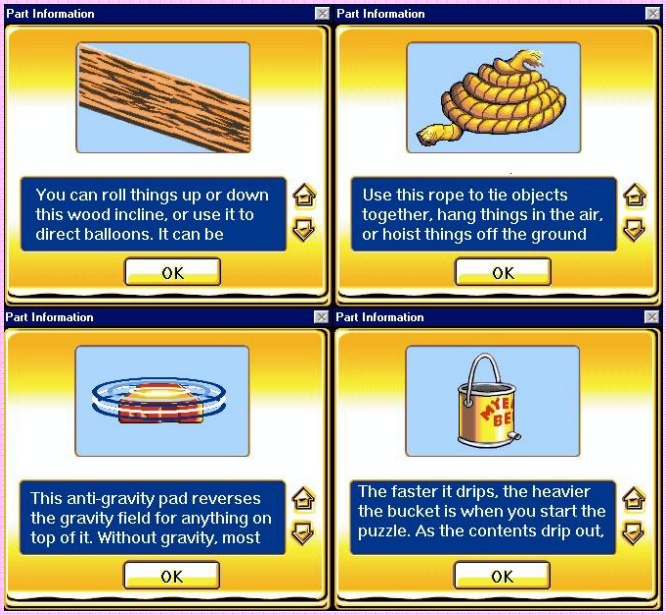
empregados autômatos são para reduzir custos. Sem dúvida a redução de custos é o que normalmente ocorre. Mas eu descobri durante esta pesquisa que nem os projetistas e nem os usuários de tecnologias altamente centralizadoras estavam interessados em conhecer sobre seus custos e benefícios, sua eficiência financeira. A forma específica que a automação estava sendo levada parecia não estar baseada nos desejos racionais por benefícios, muito menos nos prejuízos irracionais contra pessoas'.

A alternativa é projetar sistemas que tratem o usuário como um solucionador de problemas. Muitos negócios já possuem alguns sistemas que são, por natureza, problemas a serem resolvidos. Todos os programas de desenho, do PowerPoint aos sistemas CAD/CAE, são desse tipo, assim como as planilhas eletrônicas. No entanto, na maioria dos negócios, esses sistemas não são considerados 'tradicionais'. Os sistemas tradicionais estão normalmente preocupados com o processamento padronizado das transações de negócio, e são otimizados para um conjunto de tarefas, os quais são quase sempre implementados como processos seqüenciais.

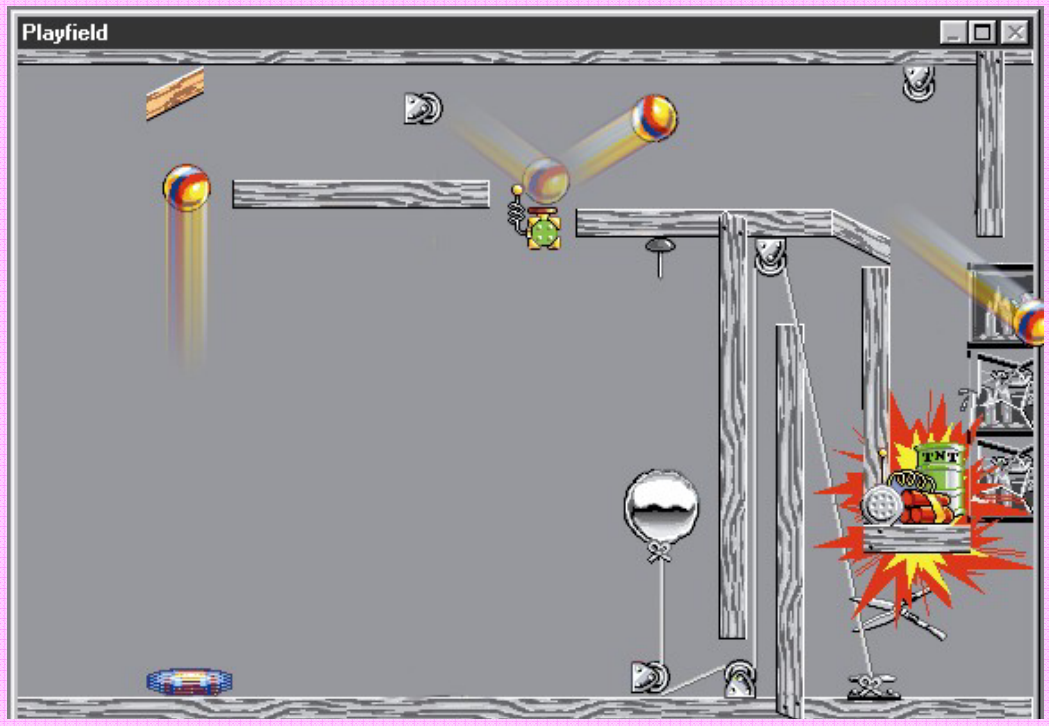
A Incredible Machine (www.sierra.com) é um grande exemplo de um sistema de problema a ser resolvido, e que claramente revela sua estrutura orientada a objetos ao usuário.

É apresentado ao usuário um problema a ser resolvido – neste caso um balão de prata que tem que ser estourado

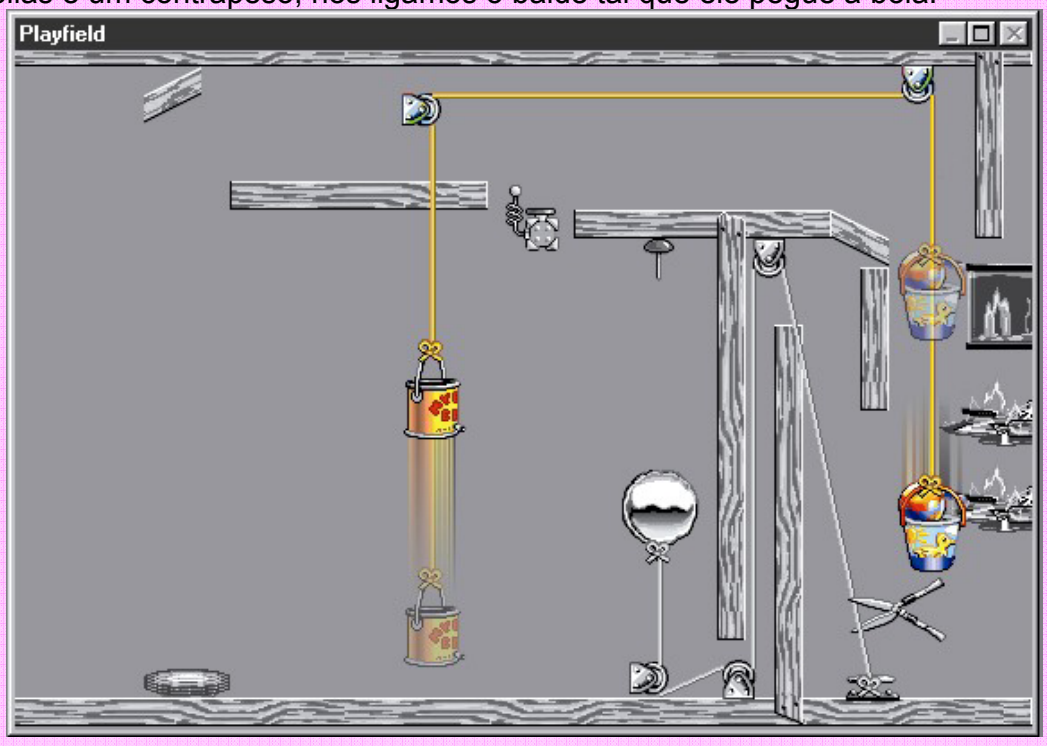
... usando algumas das partes disponíveis (objetos), sendo que cada um pode ser inspecionado para revelar seus propriedades físicas e seus comportamentos:



Inicialmente, nós usamos uma almofada antigravitacional e um bloco de madeira para desviar a bola sobre o êmbolo.



… a qual irá detonar o TNT pelo controle remoto. Usando uma corda e duas polias e um contrapeso, nós ligamos o balde tal que ele pegue a bola:



O contra-peso é um balde furado, e quando uma quantidade suficiente de água vazar, o balde e a bola caem sobre a tesoura, liberando o balão, o qual irá estourar quando encostar no alfinete acima:

Muitas pessoas acham que sistemas de problemas a serem resolvidos e sistemas transacionais refletem duas necessidades muito diferentes dentro do negócio, que não existe necessidade de uní-los, e ainda que fazer tal coisa só iria tirar a otimização do processamento das tarefas padrão que representam a maior parte das atividades de negócio. Nós sugerimos que existe uma necessidade muito real de trazer as duas idéias próximas uma da outra: em outras palavras a de tornar os sistemas transacionais tradicionais tão 'expressivos' quanto um programa de desenho [Pawson1995].

Colocar estas duas idéias juntas não significa apenas colocar interfaces de usuários gráficas em sistemas transacionais tradicionais. De fato, como muitas empresas aéreas descobriram a muito custo, a primeira geração dos sistemas de reservas baseados em GUI foram menos expressivos do que as velhas interfaces de linha de comando – os quais podiam ser difíceis de aprender, entretanto permitiam ao usuário um alto grau de controle. Tais experiências podem despertar conclusões erradas de que 'usuários realmente não gostam de GUIs'. O que os usuários realmente não gostam é de GUIs ruins.

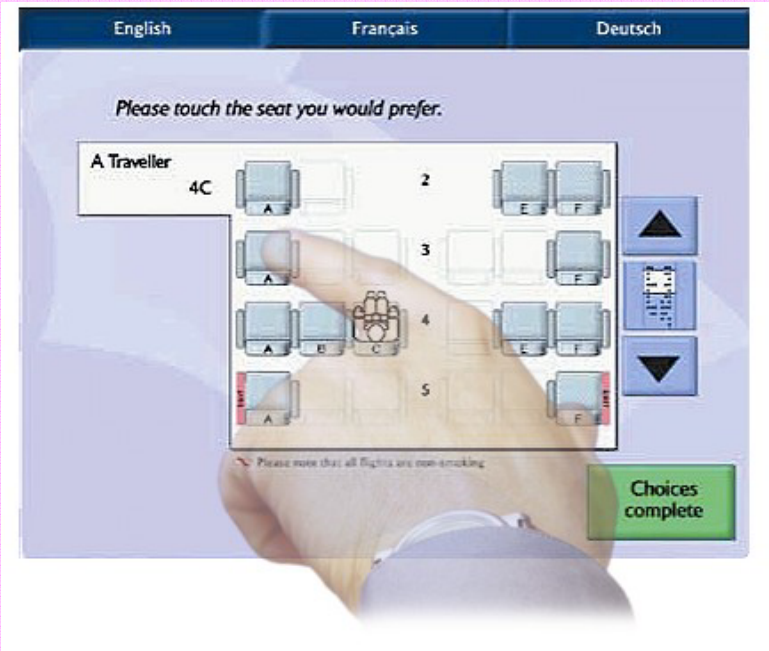
Um outro preconceito é que introduzir mais expressividade num sistema transacional tradicional irá reduzir sua eficiência. Isso pode ser verdade num contexto reduzido de tarefas padrões específicos, mas no sentido amplo a eficiência pode, na verdade, aumentar.

Considere a área de serviços ao cliente. Todos podem relatar suas frustrações quando se relacionou com um representante de serviços ao cliente, talvez num *call center*, onde toda as interações são ou definidas ou restritas pelo roteiro do sistema de computador. Algumas vezes o problema que o cliente quer resolver (usando 'problema' no sentido amplo da palavra) não parece se encaixar num dos roteiros padrões; ou tarefas não podem ser realizadas sem um número de desvios imprevisíveis para completar uma tarefa baseada em passos; ou o cliente preferira dar a informação numa ordem diferente daquele que o computador espera. A frustração aumenta quando o cliente necessita de um resultado intermediário não previsto nas perguntas e respostas pré-configuradas.

- Cliente: Quanto tempo leva para sair de Londres e chegar a Sydney?
- Agente: Em que dia você pretende sair de Londres?
- Cliente: Isso depende de quanto tempo se leva para chegar a Sydney !!!...

Esta abordagem de forçar o roteiro para interações está se tornando muito comum e frustante. No serviço de clientes, uma crescente proporção de problemas 'padrão' estão agora sendo cobertos pelos auto-serviços. Se o cliente quiser apenas solicitar um livro, relatar uma falha na linha telefônica, ou fazer o *check-in* num vôo, então uma interface web, sistema interativo de resposta audível, ou kiosk, respectivamente, pode resolver o problema. Conseqüentemente, a proporção crescente de chamadas ou visitas de um centro de serviços ao cliente está relacionada aos problemas não padrões, ou são das pessoas que simplesmente não desejam trabalhar confinadas e limitadas tal como esta abordagem. E mesmo assim, sistemas de *call center* continuam se esforçando

para aparar segundos na duração média de chamadas tentando ‘otimizar os roteiros’



Sistemas de auto-check-in, como o da British Airways, podem agora cuidar de processos padrões de check-in. Conseqüentemente, estes sistemas disponibilizam aos funcionários um painel de check-in necessários para, adequadamente, solucionar problemas individuais de processos não padronizados.

Métodos orientados a use-cases

O conceito de use-case foi definido por Ivar Jacobson como ‘uma seqüência de transações num sistema cuja tarefa produz um valor mensurável para um particular ator de sistema’ [Jacobson1995]. Uma abordagem de desenvolvimento de sistemas orientada a use-case escreve use-cases que capturam os requisitos de um sistema e, então, procura identificar objetos comuns entre eles. Os métodos orientados a objetos mais populares são orientados a use-cases.

O caso contra use-cases foi bem resumido por Don Firesmith [Firesmith1996]: ‘Use cases não são orientados a objetos. Cada use case captura uma grande abstração funcional que pode causar inúmeros problemas com decomposição funcional que a tecnologia de objetos supostamente deveria evitar. Os Use Cases são criados antes que objetos e classes tenham sido identificados, assim eles ignoram o encapsulamento de atributos e operações em objetos’. Ele continua a dizendo que uma abordagem orientada a use-case resulta num ‘protótipo da arquitetura de subsistema um objeto de controle funcional isolado representando a lógica de um use-case individual e vários objetos de entidade estúpidos

controlados pelo objeto controlador. Tais arquiteturas exibem, normalmente, encapsulamento pobre, excessivo acoplamento, e uma distribuição inadequada da inteligência de aplicação entre as classes'.

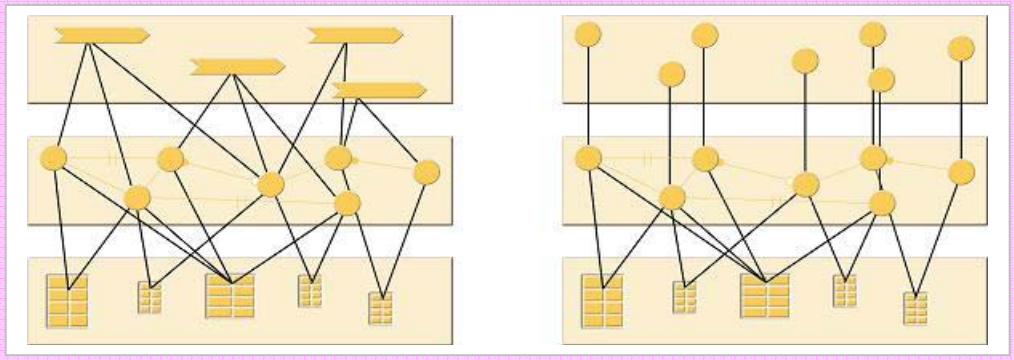
Jacobson também observou que use-cases podem ser utilizados para um outro propósito: testar o sistema resultante. 'Os use-cases constituem numa excelente ferramenta para testes de integração desde que eles explicitamente interconectam várias classes e blocos. Quando todos os use-cases tiverem sido testados (em vários níveis) o sistema estará testado em sua totalidade' [Jacobson1992]. Este conceito não tem impacto negativo na qualidade da modelagem de objeto, e consideramos ser uma boa prática.

Nós sugerimos que use-cases são muito poderosos quando eles são escritos em termos de operações sobre objetos que já tenham sido identificados e especificados, e são usados para testar este modelo de objetos. Por outro lado, use-cases são muito perigosos quando são escritos antes do modelo de objetos e usados para identificar objetos e suas responsabilidades compartilhadas – que é precisamente o que a abordagem orientada a use-cases defende.

Então, surge a seguinte questão: como os objetos de negócio são identificados? A resposta é através de conversas diretas e não-estruturadas entre usuários e desenvolvedores. A idéia da interação direta ou do 'cliente *on-site*' é defendida por todos os métodos 'ágeis' modernos, apesar de não especificamente permitir o que foi sugerido aqui. Existem grandes evidências que bons modeladores de objetos, dado um contexto como este, estejam aptos a identificar os objetos diretamente sem a necessidade de quaisquer outros artefatos formais [Rosson1989].

A crítica sobre esta abordagem é que ela depende de modeladores de objetos especialistas. Por contraste, métodos de desenvolvimento de sistemas são projetados para evitar a necessidade de tais especialistas. De fato, existe um tipo de ciclo vicioso: métodos prescritivos reduzem tanto a necessidade, quanto a chance de projetar com intuição.

Nós sugerimos que nossa abordagem não necessita de um especialista super-homem de modelagem de objetos. Tudo que essa abordagem realmente precisa é um meio correto de capturar um modelo de objetos emergente na forma de um protótipo de trabalho que tanto o usuário quanto os desenvolvedores possam entender e contribuir. Isso não significa desenvolver um protótipo convencional que capture os requisitos das tarefas do usuário em termos de formulários e menus, mas um protótipo com uma interface de usuário orientada a objetos (IUOO), onde o que o usuário vê na tela confirma o relacionamento direto que existe entre os objetos de negócio fundamentais – em termos não apenas de atributos e associações, mas também de comportamento. Trabalhos anteriores relacionados a este conceito incluem IBM's Common User Access [IBM1991] e método OVID [Roberts1998], e trabalhos de Oliver Sims sobre Newi e a versão da Business Object Facility 'Lite' [Sims1994] [Eeles1998].



(Esquerda) Com a abordagem orientada a use-case, existe uma forte tendência dos use-cases no final, serem implementados como construtores de software explícitos (primeira camada) que mascara os objetos (camada intermediária) do usuário. O resultado é que as responsabilidades naturais dos objetos de negócio chegam usurpadas na camada da interface de usuário. (Direita) Por adotar uma interface de usuário orientada a objetos (IUOO) é possível preservar uma correspondência 1:1 entre construtores da interface de usuário e o modelo de objetos de negócio subjacente. Isso ajuda a manter o conceito de objetos de negócio comportamentalmente completo.

O padrão Model-View-Controller

Como discutido no ponto anterior, use cases buscam traduzir tudo tão facilmente em objetos controladores que deixam os objetos de entidade estúpidos. Este efeito é reforçado por um padrão arquitetural comum que explicitamente separa três papéis: os objetos de negócio fundamentais que correspondem às entidades de negócio, objetos que fornecem a visão do modelo ao usuário e objetos que controlam a interação entre o usuário e o modelo. Uma versão deste padrão é o *Model-View-Controller*, ou MVC [Krasner1988]; um outro (como usado no *Unified Process*) é *Entity-Boundary-Controller* [Jacobson1999]. Existem distinções sutis entre esses dois padrões, mas eles são, de modo geral, similares. Nós iremos usar o MVC como um exemplo.

A motivação para usar o MVC é a separação de conceitos. O argumento é que dada alguma classe de objeto de negócio fundamental, tais objetos podem ser visualizados de várias maneiras diferentes: sobre diferentes plataformas, em diferentes contextos e usando diferentes representações visuais. O conhecimento combinado de todas essas visões diferentes, bem como o conhecimento de como afetá-los dentro de objetos de negócio faz com que os objetos fiquem inchados e pesados devido à duplicação de funcionalidades entre objetos. Usando MVC, os objetos do Modelo não têm conhecimento dessas diferentes visões. Objetos de Visão dedicados especificam o que aparece em cada visão, e de que forma, e têm o *know-how* de criar a apresentação visual. Objetos Controladores fornecem a cola entre os dois: povoando as visões com atributos a partir dos objetos de

entidade de negócio, e chamando métodos desses objetos em resposta a eventos do usuário.

Esse é um pensamento legítimo, mas tem alguns efeitos colaterais negativos. Embora não tenha sido a intenção original da abordagem MVC, os objetos Controladores tendem a ser uma representação explícita das tarefas de negócio – especialmente se a abordagem de projeto for orientada a use-case, mas isso ocorre também em outros casos. Quando isso acontece, os objetos Controladores deixam de exercer o papel limitado de ser apenas uma 'cola' técnica entre interfaces de usuários e objetos de negócio, e passam a assumir o papel de roteiro de tarefas, incorporando não apenas a seqüência de atividades otimizadas, como também regras de negócio – com isso usurpando as responsabilidades que deveriam ser dos objetos de negócio fundamentais. Por outro lado, não se pode dizer que os objetos de Visão contenham a lógica de negócio no sentido de algoritmos, eles podem no final, apesar de tudo, incorporar uma forma de conhecimento específico de negócio apresentado na seleção e layout de campos de uma tarefa particular, e (algumas vezes) uma pequena lógica de negócio tal como manter o movimento total dos dados informados.

O resultado final é que o conhecimento específico de negócio é espalhado através dos domínios do modelo, visão e controlador. Qualquer mudança no modelo de objetos fundamentais irá potencialmente exigir mudanças num grande conjunto de objetos Visão e Controlador [Holub1999]. Naturalmente este problema não está restrito ao projeto orientado a objetos – ele se aplica a muitas formas de arquiteturas multicamadas. Além disso, não existe nada no MVC que force esta tendência, mas a prática sugere que devemos contestá-lo veementemente quando procuramos por objetos comportamentalmente completos.

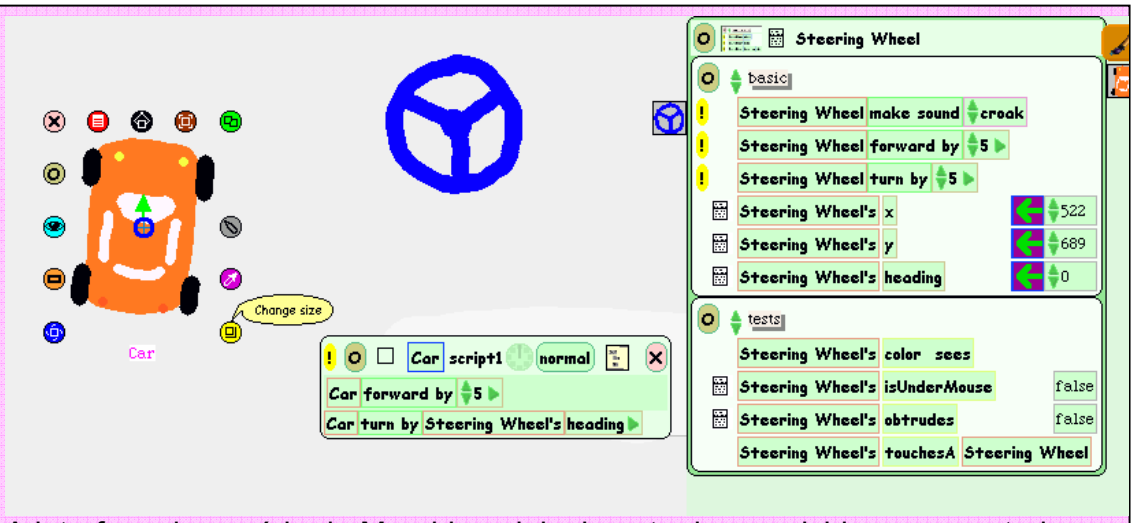
Como no caso de outras forças, qualquer proposta alternativa deve evitar cair no problema para o qual o MVC foi projetado para evitar: ela deve ainda facilitar a portabilidade de uma aplicação entre múltiplas plataformas técnicas, até entre múltiplos estilos de interação, sem necessitar que o modelo de negócio seja editado. Ao mesmo tempo, ela deve acomodar a necessidade de representar múltiplas visões do modelo sob a mesma plataforma, onde genuinamente exista a necessidade. O uso da palavra 'genuinamente' é uma referência para uma pequena, mas crescente, crença de que a ênfase na personalização do usuário tornou-se excessiva nos recentes anos: isso consome vastos recursos de desenvolvimento com benefícios limitados. Como Raskin (2000) indica, qual ponto de venda investiria pesadamente no projeto de padrões de interfaces de usuário para maximizar a compreensibilidade e minimizar o estresse, só para descobrir que o usuário, ou o agente do usuário, irá querer posteriormente, personalizar tais benefícios novamente?

Em vez disso, forneça um mecanismo de visualização genérico, incorporando os papéis dos objetos de Visão e Controlador. Isso significa escrever um mecanismo de visualização para cada plataforma de cliente solicitada (por exemplo: Windows, Linux, browser web ou Palm Pilot). Mas uma vez que um mecanismo de

visualização genérica exista para a plataforma alvo, tudo que o desenvolvedor precisa é escrever os objetos do Modelo de negócio. O mecanismo de visualização genérica traduz, automaticamente, os objetos do Modelo de negócio, incluindo os comportamentos disponíveis, dentro de uma representação do usuário. Os objetos do Modelo e suas associações podem ser apresentados, por exemplo, como ícones com métodos ou comportamentos disponibilizados em opções sobre um menu pop-up. Essa abordagem não viola a essência do MVC, mas é uma re-interpretação radical de como aplicá-la. Uma maneira de enxergar isso é que ela gera os objetos de Visão e Controlador considerando o Modelo e vice-versa.

A idéia de autogerar a interface do usuário a partir do modelo subjacente não é nova. O conceito existia em muitas linguagens proprietárias de quarta geração e nos geradores de aplicações, e re-emergiu em várias iniciativas baseadas em XML tais como o Xforms da W3C. No entanto, poucas dessas abordagens são orientadas a objetos: a interface do usuário é normalmente uma representação explícita da estrutura de dados e módulos ou processos funcionais. Eles continuam a encorajar a separação dos processos e dados.

Talvez a abordagem mais próxima de nossa solução, e certamente um dos maiores desafios para a hegemonia do MVC, é a interface de usuário do Morphic [Maloney1995]. Usando Morphic, objetos de aplicação podem herdar a habilidade de se exibirem e de serem manipulados pelo usuário. Morphic foi originalmente desenvolvido como parte da linguagem Self [Smith1995], e subseqüentemente prosseguiu com a linguagem Squeak [Ingalls1997] – veja a figura. Squeak dá ao usuário um sentido muito forte de interação direta com os objetos próprios da aplicação. O usuário pode selecionar qualquer objeto e invocar diretamente a exibição associada bem como específica de negócio – até quando os objetos são movidos na tela. Portanto, Squeak confunde a linha entre programação e uso do sistema. No entanto, embora Squeak claramente tenha potencial de propósito geral (em palavras da equipe central do Squeak, Squeak é o que 'Smalltalk pretendia ser'), a maioria da ênfase atual tem sido em aplicações educacionais, gráficos animados, e ferramentas de autoria multimídia. Ela ainda será aplicada para projeto de sistemas de negócio transacionais.



A interface de usuário do Morphic, originalmente desenvolvido como parted a linguagem Self e agora adotado pelo Squeak (desenho) é um dos mais compreensíveis desafios do paradigma *Model-View-Controller*. Todos os objetos do Morphic herdam a habilidade de se exibirem e tornar seus comportamentos visíveis e acessíveis. Como ilustra a figura, quando o usuário clica sobre um objeto (no caso um carro vermelho), ele exibe um 'círculo' de pequenos ícones ao redor do carro, o qual fornece funções de manipulação padrão. Veja www.squeak.org e www.squeakland.org para maiores detalhes deste excitante projeto *open-source*.

Desenvolvimento de sistemas baseado em componentes

Nós não somos contrários à idéia de desenvolvimento de sistemas baseados em componentes, mas estamos preocupados com a maneira em que esta idéia consegue confundir o projeto orientado a objetos. Certamente eles possuem elementos comuns, como quaisquer duas idéias possuem elementos comuns no desenvolvimento de software. Mas o projeto orientado a objetos e desenvolvimento de sistemas baseado em componentes são conceitos um tanto diferente.

Desenvolvimento baseado em componentes está principalmente interessado em fornecer uma abordagem *plug-and-play* para o desenvolvimento de sistemas. O *plug-and-play* dá a você, em teoria, maior flexibilidade no fornecimento de seus sistemas: você estará apto a comprar componentes no mercado, copiá-los de uma biblioteca pública, ou reusar componentes que você tenha escrito internamente tendo em mente várias aplicações. Tal flexibilidade pode potencialmente economizar gastos diretos, reduzir o esforço de desenvolvimento, elevar a qualidade e promover a padronização.

A modelagem orientada a objetos não está, ou não deveria estar preocupada com *plug-and-play*. Ela está preocupada em adequar a estrutura do software para a estrutura do domínio de negócio do mundo real, seja periodicamente em resposta

às mudanças nos requisitos de negócio, ou dinamicamente em resposta a um problema específico.

O modelo de componentes tem tido muito sucesso nos serviços técnicos – no sentido de que você pode agora mudar seu banco de dados sem necessariamente ter que mudar outras camadas na arquitetura. No contexto de negócio existe atualmente um grau muito grande de compatibilidade entre pacotes de aplicação: você pode escolher seu pacote de planejamento de manufatura independentemente de seu sistema de pedido de peças entre outros. Mas tentar reduzir a granularidade dos componentes de negócio não tem no final aparência de objetos – ao menos no sentido de entidades instanciáveis comportamentalmente completos – mas a aparência de sub-rotinas – blocos de código que pode transformar uma entrada numa saída. Esta forma de componente combina bem com a idéia de modelagem de processos de negócio, completando o círculo.

Existem algumas tentativas de combinar o conceito de objetos de negócio com montagem de software *plug-and-play* (notavelmente pelo Oliver Sims e colegas [Eeles1998] [Herzum2000]). No entanto, combinar dois paradigmas em um é um risco: o paradigma de maior aceitação (montagem de componentes) é o que provavelmente irá dominar o de menor aceitação (objetos comportamentalmente completos). Nossa visão é que é melhor deixar esses dois conceitos separados. Aplique o conceito de montagem de componentes na sua infra-estrutura técnica, e continue com a modelagem pura de objetos na camada de negócio.

Pode-se argumentar que uma das razões que levou várias organizações para o lado de componentes no debate 'objetos versus componentes' é que elas desenvolveram um medo quase patológico de fazer seus próprios projetos e desenvolvimentos. Elas foram atormentadas pela lembrança da paralisia de análise, desenvolvimento interminável e de sistemas finalmente liberados cujas especificações falharam ao atender às necessidades reais de negócio. Nós esperamos demonstrar que isso não tem que ser assim. O desenvolvimento de seus próprios modelos de objetos de negócio não precisa ser uma experiência tão dolorosa.

Definindo uma nova abordagem

Como estamos encorajando o projeto de sistemas de negócio a partir de objetos comportamentalmente completos, precisamos superar um número de forças que desencorajam as pessoas de modelar objetos apropriadamente, e/ou que separam processos e dados mesmo no tão chamado projeto orientado a objetos. Superar essas forças irá requerer novas ferramentas e técnicas, e devem demonstrar que elas não vão nos expor a riscos ou problemas para as quais foram projetadas para superar. Especialmente:

- Ao invés de imaginar, apenas, o papel dos sistemas de negócio como um meio de executar um processo determinístico, que transforma informações de entrada para informações de saída através de uma seqüência de passos

que adicionam valor precisamos encontrar metáforas alternativas. Uma dessas metáforas é a da loja de valores, onde o usuário constrói uma solução para um problema específico. (é muito fácil adicionar roteiros/ scripts otimizados para um modelo de cadeia de valores).

- Ao invés de perseguir a eficiência ótima na execução de cada um dos conjuntos finitos de tarefas roteirizadas, projete uma forma de interação com o usuário que maximize a efetividade geral do usuário em satisfazer sua gama de responsabilidades. Isso significa dar aos usuários maior controle, por exemplo, na ordem em que as capacidades são chamadas a fim de alcançar uma meta. Devemos também projetar sistemas que permitam aos usuários tornarem-se mais experientes conforme eles aprendem, ao invés de restringir todos para o menor denominador comum.
- Ao invés de capturar os requisitos de um sistema como um conjunto de use-cases e então usá-los para identificar os objetos compartilhados e suas responsabilidades, procure identificar os objetos e suas responsabilidades diretamente, em conversas com usuários. Precisamos também de algum meio de capturar o modelo de objetos emergente de maneira que os usuários possam, de forma concreta, se identificar com esse meio e obter benefícios.
- Ao invés de permitir que a lógica de negócio fique esparramada através dos objetos Modelo, Visão e Controlador, encontre uma maneira de tornar os papéis de Visão e Controlador genéricos, tal que o desenvolvedor escreva apenas os objetos do Modelo e que todas as interações do usuário sejam derivadas desses objetos automaticamente.
- Ao invés de permitir que a arquitetura de nossos sistemas seja dominada pelas idéias de poder comprar peça por peça da arquitetura de diferentes fornecedores, reconheça que a verdadeira vida do sistema precisa do modelo de objetos de negócio homogêneo que não pode ser comprado, tem que ser projetado internamente para refletir as verdadeiras necessidades de negócio.

Estudo de caso: Processando benefícios do governo

Em 1999 nós tivemos a oportunidade de projetar um conjunto de objetos comportamentalmente completos os quais modelaram o negócio de uma grande organização, e criou um sistema que atenderam aos requisitos do usuário simplesmente expondo esses objetos de negócio diretamente ao usuário.

O Departamento de Assuntos Sociais e da Família (*Department of Social and Family Affairs* – DSFA) é a administração da previdência social da Irlanda. Anterior a 1998 ele era conhecido como Departamento de Bem-Estar Social. O Departamento dependia pesadamente das tecnologias de informação para realizar suas tarefas. Ele tinha alguns PCs 2000, mas seus principais programas de processamento de transação eram todos baseados em mainframe, e acessados via algum terminal. Esses sistemas estão tecnologicamente desatualizados e sua manutenção é incrivelmente cara. Por exemplo, eles precisam re-programar manualmente todas as vezes que o governo altera as regras ou taxas dos benefícios. Atualmente, existe um sistema separado para cada grande tipo de benefício – Pensão para Crianças, Deficiência, Desempregado, entre outros. Embora exista um Sistema de Registro Central para armzenamento de dados debenefeciários, existe muito pouco compartilhamento tanto de informação quanto de funcionalidade do que poderia existir. Por exemplo, muitos sistemas têm seu próprio mecanismo separado para gerar pagamentos.

A demanda por maior agilidade organizacional dentro do Departamento cresceu em poucos anos, e isso foi traduzido como uma demanda por maior agilidade dentro dos sistemas de informação. Desenvolvimentos tecnológicos tais como a Internet e smartcards forneceram, potencialmente, significativos benefícios tanto para o DSFA quanto para os seus beneficiários, incluindo a facilidade de acesso, riqueza de informação e economia de custos. O governo também estava pressionado por maior agilidade, tanto na habilidade de introduzir novas formas de apoio bem como aprimorar os serviços existentes disponibilizados aos seus beneficiários. Existem várias iniciativas e-government na Irlanda, das quais a mais significativa é o programa REACH, que irá fornecer um ‘e-broker’ comum para acessar informações sobre serviços oferecidos pelas múltiplas agências do governo, um meio central de identificação e autenticação, e um repositório de dados pessoais que dá ao cliente maior controle sobre sua própria privacidade.

Em resposta a essas várias demandas, em 1999 o DSFA concebeu um novo Modelo de Liberação de Serviços (*Service Delivery Model* - SDM) que enfatiza o comércio eletrônico, agilidade e de resposta rápida aos beneficiários. O SDM destacou as necessidades para uma arquitetura completa para os principais sistemas. Ele não apenas atende às necessidades específicas do SDM, mas deve também ser mais adaptável para o futuro, até para as mudanças inesperadas de negócio.

Inicialmente, foi decidida que a nova arquitetura deveria ser multicamada, e orientada a objetos. O departamento de SI da DSFA tinha, anteriormente, experimentado as técnicas de orientação a objetos por uns três ou quatro anos, focalizando sobre a idéia de que a orientação a objetos poderia melhorar a produtividades de desenvolvimento através do reuso, mas tinha produzido pouco na forma de resultados ou saídas tangíveis. Em retrospectiva, a administração de SI sentia que as iniciativas anteriores tinham sido focadas muito internamente no próprio departamento de SI. Desta vez, a motivação para pensar sobre objetos deveria ser para aprimorar a agilidade do negócio.

Experimento inicial

No início de 1999 a administração de SI tomou conhecimento de um conjunto de idéias emergentes que iriam eventualmente se tornar em Naked Objects. Eles foram atraídos para o conceito devido a vários motivos. Primeiro, o conceito defendia uma versão de orientação a objetos mais pura, baseada nas idéias de completeza comportamental, e a motivação para isso foi claramente a de facilitar a agilidade de negócio ao invés de promover o reuso. Embora a administração não tenha aceitado todos os argumentos neste estágio, estava claro que os objetivos estavam bem alinhados com o que eles queriam atingir. Segundo, a administração de SI estava atraída pela tangibilidade visual da abordagem. Eles sentiam que poderiam facilmente obter dos gerentes não vinculados a Tecnologia de Informação (TI) maior envolvimento com as decisões de projeto cruciais.

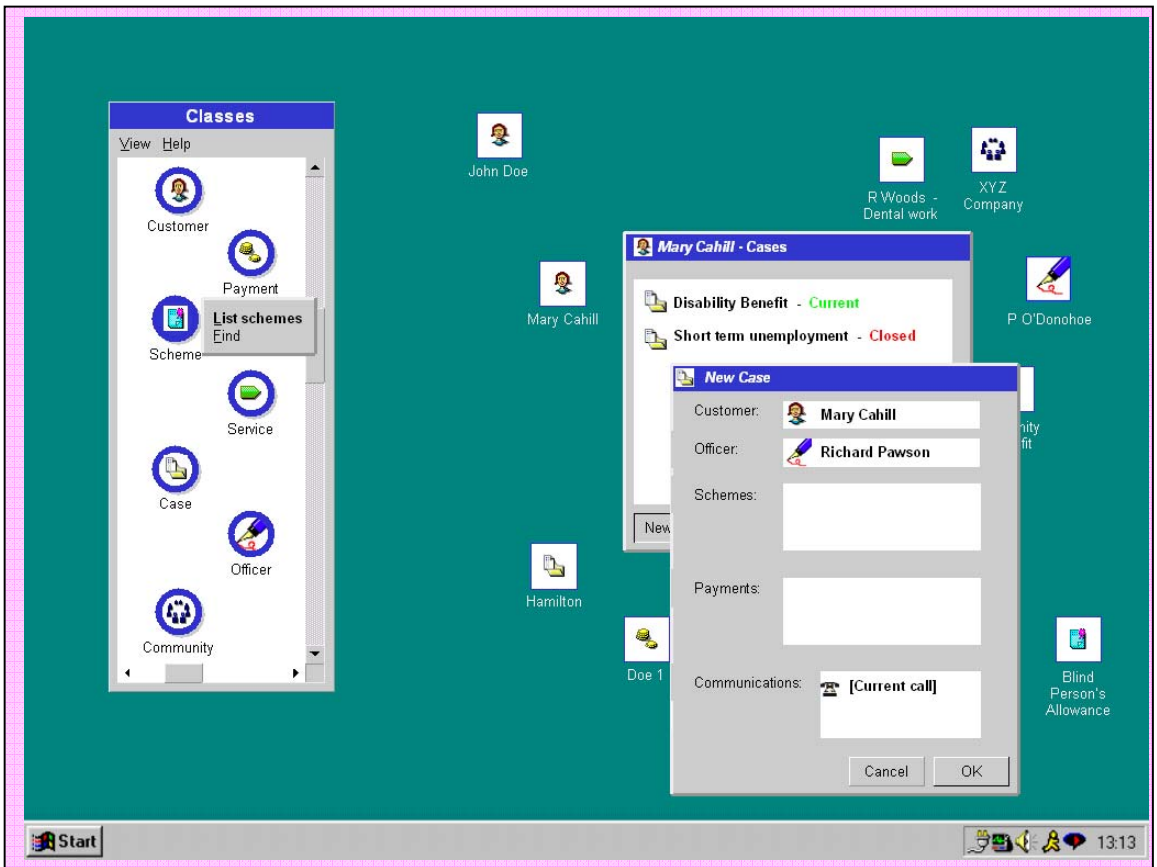
No início de 1999, a administração de SI iniciou alguns workshops educacionais sobre conceitos orientados a objetos para alguns de seus gerentes seniores, tanto de TI quando de negócio, com a ajuda de um dos autores (Richard Pawson). Dado que o framework Naked Objects não existia naquele momento, e não existia nenhum exemplo de sistemas de negócio transacionais construídos nessa maneira, o jogo The Incredible Machine foi usado como uma metáfora durante o workshop. O jogo *The Incredible Machine*, apresenta ao usuário uma série de desafios físicos, e permite construir uma simulação de uma complexa máquina, porém irreal, no estilo do artista Heath Robinson ou Rube Goldberg para resolvê-los. Da mesma forma que demonstra claramente a noção de um sistema para solução de problemas, *The Incredible Machine* também demonstra muito claramente que é orientado a objetos da perspectiva do usuário: elementos que o usuário arrasta com o *mouse* a partir de peças dentro da área de trabalho não são apenas representações visuais, mas trazem consigo uma simulação completa do comportamento físico dessas peças.

Um dos workshops, um exercício de um dia envolvendo seis gerentes, tentou identificar os objetos de negócio fundamentais que poderiam potencialmente modelar o negócio do DSFA. Os participantes foram convidados a sugerir diretamente as categorias de objetos: nenhuma tentativa anterior foi feita para capturar requisitos ou especificar use-cases. No final da manhã uma lista de aproximadamente vinte candidatos tinha sido produzida. Durante à tarde, esta lista foi reduzida devido à identificação de candidatos em duplicidade, declaração de

certos candidatos como subclasses de outras, e eliminação de candidatos que, após discussões posteriores, foram colocados fora do conjunto de objetos de negócio fundamentais, pois podiam ser mais bem representados como simples atributos ou métodos de outros objetos. O resultado foi uma lista de sete classes fundamentais: Beneficiário, Caso, Esquema (significando um Esquema de Benefícios), Pagamento, Serviço (um benefício não-monetário tal como um passe de ônibus), Comunicação e Funcionário.

Depois, foi solicitado ao grupo que imaginasse como os usuários deveriam interagir com cada objeto, fazendo questões como: Como este objeto irá aparecer na tela? Para onde o usuário poderá querer movê-lo? Quais ações o usuário requisitará que estejam disponíveis pressionando o botão direito do mouse sobre o objeto? O resultado foi uma lista de responsabilidades ‘saber o que’ e ‘saber como’, com ênfase sobre o último para evitar a redução a objetos meramente a conjuntos de dados complexos. O resultado foi uma definição muito crua de um conjunto de objetos comportamentalmente ricos.

Esse rascunho das definições de responsabilidades de objetos foi traduzido para um modelo visual provisório de um sistema que poderia ser usado por um ‘funcionário que tem poder de decisão’. O modelo teve realmente uma série de telas feito à mão mantidos com slides em PowePoint, mas conseguiu-se criar uma demonstração bem prática com efeitos realísticos, com a impressão de ícones sendo arrastados na tela e menus *pop-ups* surgindo ao pressionar o botão direito do mouse.



Este modelo do sistema DSFA foi criado em PowerPoint no modo *full-scren*. Usando cuidadosamente uma demonstração roteirizada, o modelo criado deu a ilusão de estar respondendo aos comandos do mouse, aparecendo menus e novas janelas.

O modelo foi apresentado a um grupo de gerentes seniores, os quais não estiveram envolvidos no workshop de modelagem. Algumas reações resumem o seu impacto:

- 'Eu percebo que todas as pessoas da organização, até mesmo o próprio ministro, podem usar o sistema de forma idêntica'. Isso não significa que todos os usuários devam realizar as mesmas operações, ou que de fato tenha o mesmo nível de autorização. De preferência, todas as coisas que a organização faz poderiam ser representadas como ações sobre alguns objetos chaves. Da mesma forma, uma interface consistente pode ajudar a quebrar algumas barreiras interdepartamentais, bem como facilitar a adaptação dos indivíduos em diferentes áreas de responsabilidade.
- 'Esta interface pode não ser muito boa para tarefas com grande volume de entrada de dados'. Existiram alguns debates sobre isso, até que alguém disse que o compromisso da DSFA com o acesso eletrônico (via web, smartcards e call center telefônico), acrescida por uma abordagem mais

integrada para seus sistemas, indicavam que muitos trabalhos de entrada de dados iriam desaparecer de qualquer jeito.

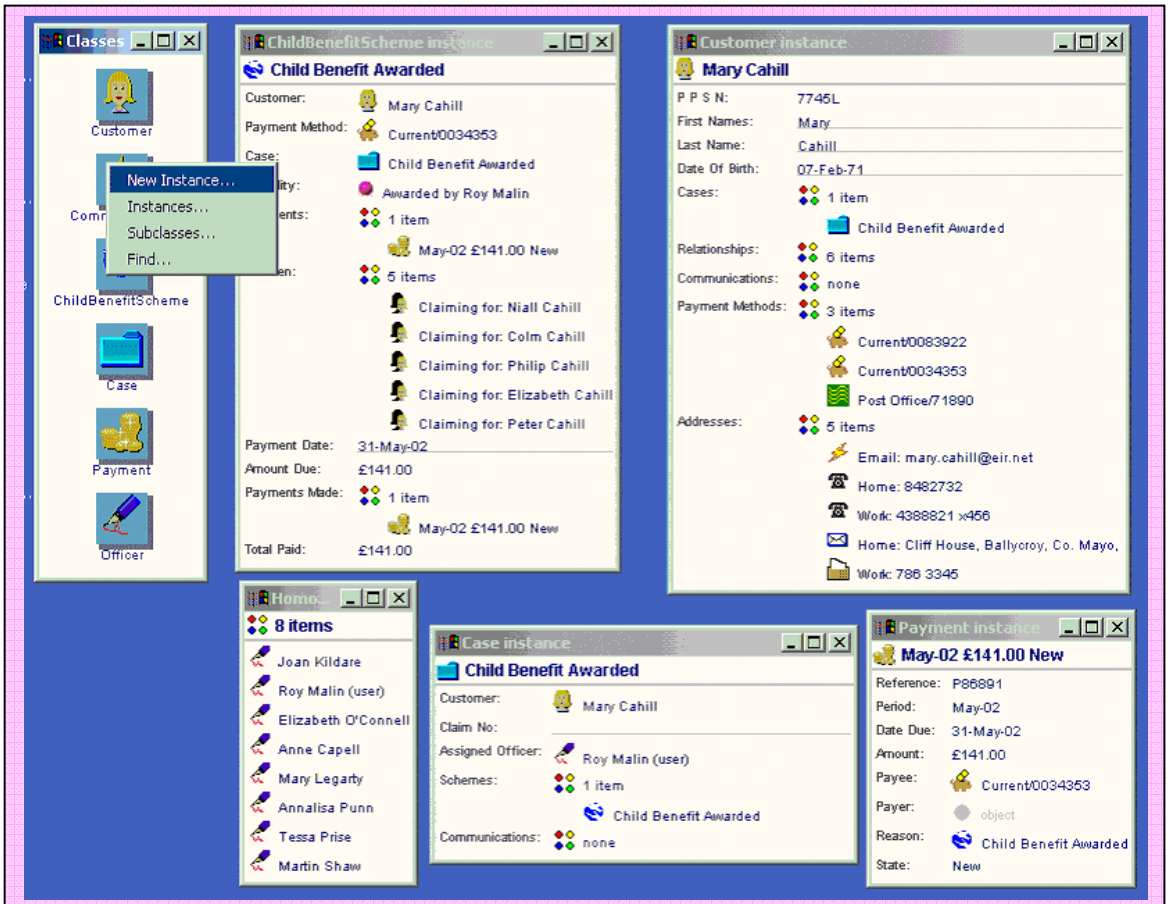
- 'Este sistema reforça as mensagens que nós estivemos enviando para os funcionários sobre mudar o estilo de trabalho'. O DSFA está comprometido em mudar a abordagem convencional de linha de montagem do processamento de solicitações, onde cada pessoa realiza um pequeno passo no processo, em direção a um modelo onde muitos funcionários podem manipular uma solicitação completa, e funcionários treinados apropriadamente podem no futuro manipular todos os benefícios para um beneficiário. Os gerentes, na demonstração, sentiram que mesmo este simples modelo poderia ajudar a transmitir aos usuários a mensagem que eles são solucionadores de problemas, não seguidores de processo. Esse foi o contraste marcante comparada à abordagem proposta por alguns fornecedores, que enfatizavam o uso de tecnologias baseadas em regras ou 'agentes de software inteligentes' para automatizar tantas tomadas de decisões quanto possíveis. Ao invés disso, o modelo baseado em objetos sugeriu um ambiente onde a habilidade natural de resolver problemas do usuário fosse altamente privilegiada. Nesse sentido, o projeto de sistemas pode ser visto como uma ajuda para facilitar fundamentalmente na mudança cultural.

Além destas reações positivas dos representantes do usuário, a administração de SI também ficou impressionada com a velocidade deste exercício comparada a tentativas anteriores na modelagem de todo o departamento, como objetos, dados ou processos. Eles ficaram muito familiarizados com a frase 'paralisia de análise'. Embora o DSFA não estivesse pronto para se comprometer com esta nova abordagem para completa implementação neste estágio, combinou-se que o conceito deveria ser explorado em maior profundidade.

O sistema de benefícios para menores

No início de 2000 a situação do negócio de tomada de decisão tornou-se mais urgente. O sistema de Administração de Benefícios para Menores precisou ser trocado com alguma urgência. O Governo tinha indicado as possíveis mudanças futuras para Benefícios para Menores que o sistema existente não podia ser modificado para atender. O Benefício para Menores é uma dos planos mais simples administrados pelo DSFA e é de escala significativamente pequena: o sistema existente tinha apenas 50 usuários. Apesar disso, ele tinha muito em comum com outros planos. Pareceu ser uma oportunidade ideal para começar a implantação de uma nova abordagem.

Numa série de workshops envolvendo tanto gerentes seniores quanto representantes de usuários, as responsabilidades traçadas no modelo de objetos original foram agora trabalhadas a partir da perspectiva particular da Administração de Benefícios para Menores. Essas idéias foram imediatamente prototipadas usando uma versão muito prematura de uma ferramenta de prototipação projetada pelos autores.



O protótipo do sistema de Benefícios para Menores foi criado usando uma versão muito prematura do framework que iria eventualmente se tornar no Naked Objects, apesar deste nome não ter sido usado naquele momento. Ele não tinha mecanismos de persistência mas tinha funcionalidades suficientes para simular alguns cenários de negócio operacional reais.

Durante os workshops, as responsabilidades de objetos foram refinadas e novas responsabilidades identificadas. Novas subclasses e objetos secundários ou 'agregados' foram também adicionados. E o modelo todo foi em sua forma bruta testada frente a alguns cenários de negócio. Foram apresentados algumas telas deste protótipo e um exemplo de uma aplicação muito simples de Benefícios para Menores Além disso, o modelo foi em sua forma bruta, testada para uma variedade de cenários de como o negócio do DSFA poderia mudar no futuro. O extraordinário foi o quão bem o modelo inicial do workshop de um dia ficou diante dessas subseqüentes demandas e testes.

Fornecedores de tecnologia

Nesse meio tempo, um outro grupo ficou procurando traduzir estes conceitos numa infra-estrutura funcional que atenda os requisitos técnicos da DSFA. Como

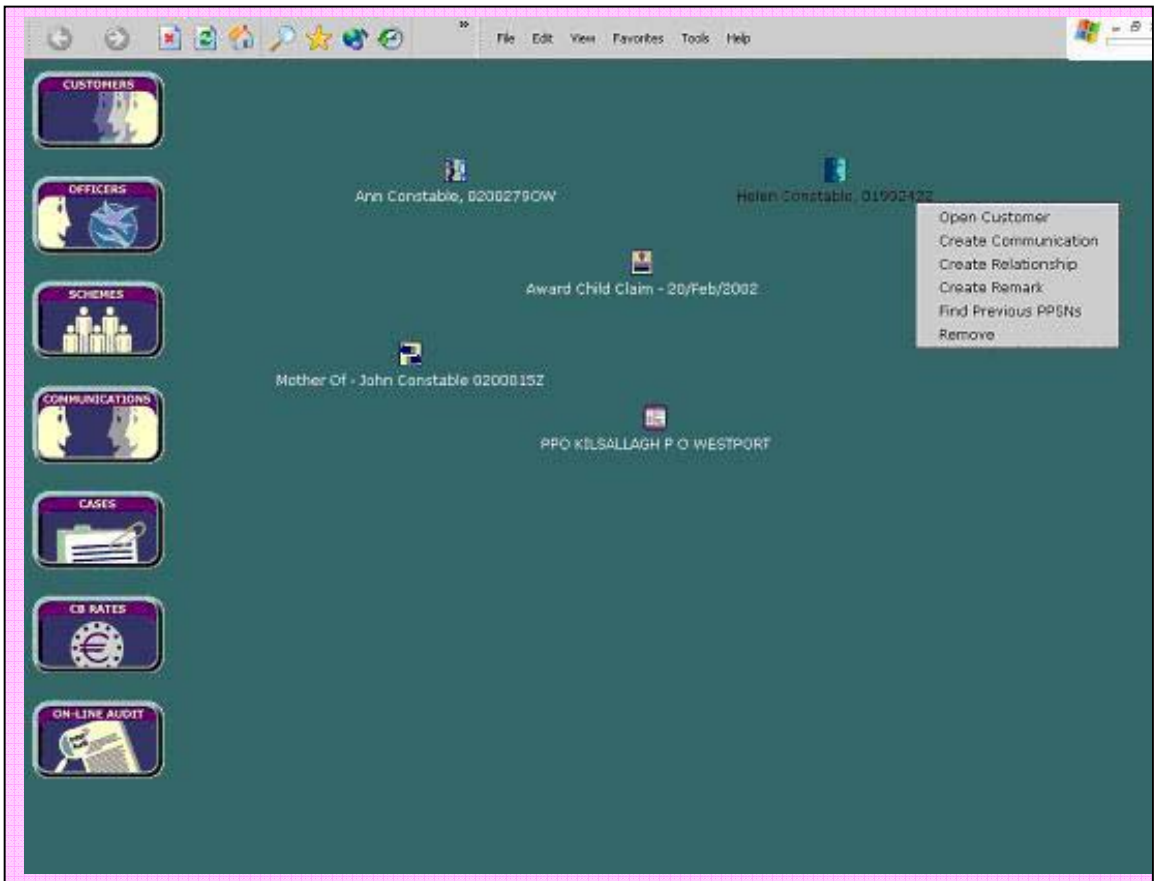
numa agência de setor público, todas as aquisições estão submetidas às regras de licitação da União Européia. De acordo com isso, o DSFA publicou um RFT (*Request For Tender* – Pedido de Licitação) para 'fornecedores de tecnologia' que deveriam implementar todas as camadas principais de uma arquitetura empresarial escalável. Previa-se a inclusão de escalonadores de mensagens, monitoramento de transações e persistência de dados usando um banco de dados relacional. Estes elementos de software deveriamrefletir os conceitos de projeto do protótipo. O Departamento contatou três desses fornecedores, cada um utilizando diferentes tecnologias: EJB, COM+, e um pacote orientado a objetos proprietário.

Implementação da fase I

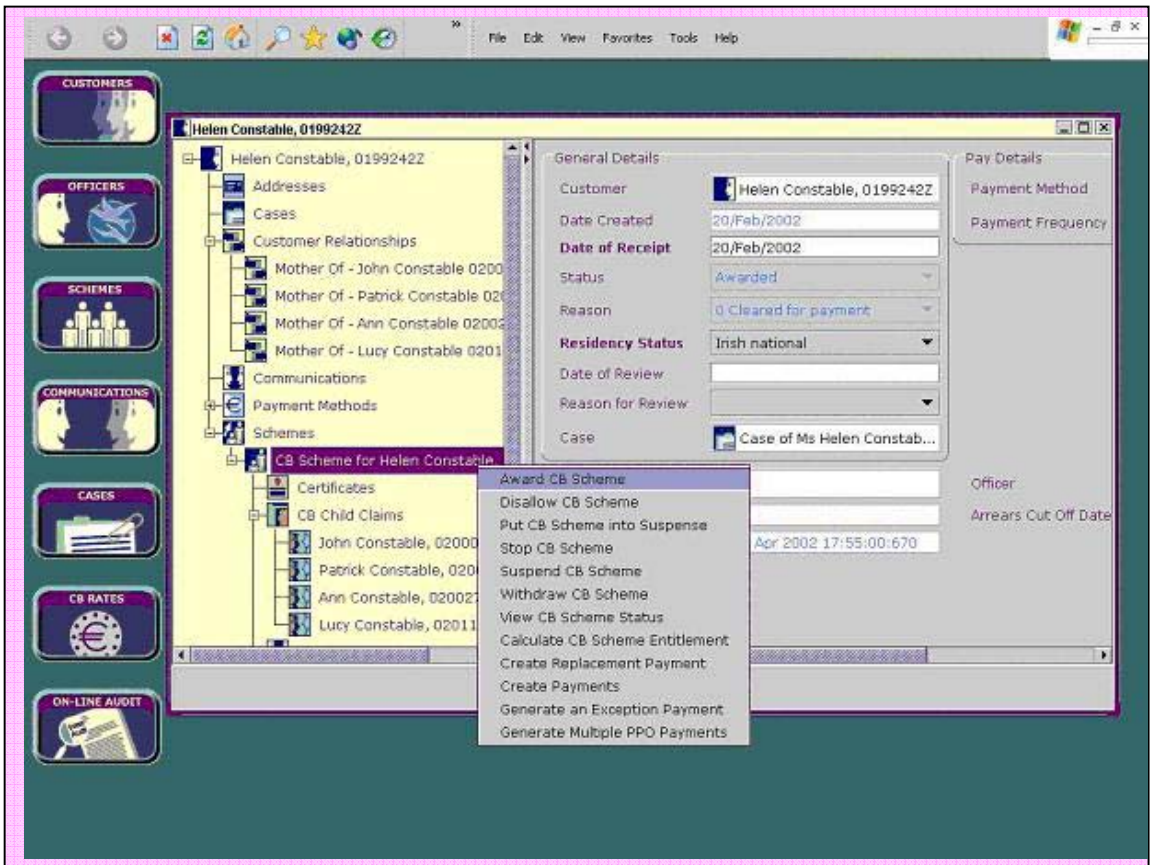
Após avaliar os fornecedores, o DSFA publicou um novo Pedido de Licitação para o projeto e implementação de uma 'Arquitetura de Objetos Expressivos' de larga escala e a funcionalidade de negócio necessária para trocar o sistema de Administração de Benefícios para Menores. Cabe observar que o Pedido de Licitação não forneceu uma especificação funcional, mas forneceu uma descrição das responsabilidades de alto-nível requerido dos objetos de negócio fundamentais, reproduzido nesta seção. O contratado deveria se responsabilizar por uma especificação de requisitos detalhados no início do projeto, usando essas descrições de objetos. Considerou-se que tais informações eram suficientes para que os contratados estabelecessem um preço fixo.

O contrato foi fechado em fevereiro de 2001 com o DMR Consulting, conhecido agora como Fujitsu Consulting – com a intenção original de que o novo sistema deveria entrar em operação em março de 2002. Seguindo alguns assuntos de relações comerciais para satisfazer as mudanças de negócio sendo introduzidas, a transferência foi postergada até o verão de 2002.

A interface de usuário implementada no sistema de Benefícios para Menores não é idêntica ao protótipo apresentado anteriormente, mas claramente adere aos mesmos princípios: objetos de negócio apresentados diretamente através da interface de usuário, tanto como classes e instâncias, e todas as ações de negócio são iniciadas ou via menus pop-up de objetos ou via drag-and-drop. Além disso, toda a interface de usuário é autogerada, dinamicamente, a partir dos objetos de negócio fundamentais. As telas do sistema implementado são apresentadas abaixo.



Esta tela foi capturada do sistema de Benefícios para Menores. No lado esquerdo da tela estão os ícones representando as classes de objetos de negócio; em outros locais do desktop estão os ícones representando instâncias individuais. Foi pressionado o botão direito do mouse sobre um dos ícones de cliente revelando o menu pop-up dos métodos de instância disponíveis para o usuário.



A visão aberta de um objeto cliente revela tanto atributos de dados simples quanto os vários objetos associados num painel de navegação no lado esquerdo. O menu pop-up para o esquema CB (Child Benefit – Benefício para Menores) mostra que todos os comportamentos relevantes podem ser chamados deste contexto sem abrir necessariamente uma visão desse objeto.

Indo para a fase II

No início de 2002, quando uma quantidade substancial do desenvolvimento da Fase I estava completada, mas antes de ser implantada, o DSFA iniciou o plano da Fase II, de trocar o sistema atual pelo esquema de pensão estabelecido. O objetivo era o de usar a Arquitetura de Objetos Expressivos construída na Fase I, entretanto com um escopo maior tanto em escala quanto em escopo.

Bem no início do projeto original em 1999, Richard Pawson aconselhou o DSFA que a reutilização não deveria ser tratada como um objetivo. Nós recomendamos que o objetivo deveria ser a agilidade: agilidade estratégica de negócio, agilidade operacional de negócio e agilidade do processo de desenvolvimento. Se essas metas fossem perseguidas, e a modelagem de objetos for bem realizada, então a reutilização seria uma conseqüência positiva. De fato isso foi provado. A

modelagem preliminar de objetos da Fase II, conduzida no início de 2002, apresentou níveis de reutilização da Fase I que assustaram até mesmo os mais entusiasmados dos modeladores de objetos. A mensagem: reutilizar é bom como resultado, mas ruim como um objetivo.

O Modelo de Objetos de Negócio do DSFA

O Modelo de Objetos de Negócio do DSFA identificou seis principais classes de objetos: Beneficiário, Esquema, Comunicação, Funcionário, Pagamento e Caso. As principais classes de objetos tiraram proveito de vários outros objetos, cujos papéis podem ser considerados secundários, mas que, apesar disso, podem ser manipulados como objetos pelo usuário. Em outras palavras, os objetos secundários são normalmente 'agregados' dos objetos principais. Objetos são definidos em termos de um conjunto de responsabilidades que eles devem cumprir. As responsabilidades 'saber o que' indicam os atributos e associações com outros objetos que um usuário pode esperar ver e/ou editar quando esse objeto é apresentado na tela. As responsabilidades 'saber como' resumem o comportamento do objeto. Essas responsabilidades 'saber como' normalmente são traduzidas em métodos de objetos, embora o mapeamento não seja necessariamente um-para-um.

Nós listamos as responsabilidades 'saber o que' e 'saber como' para cada um das seis principais classes de objetos de negócio abaixo. Nós agradecemos o DSFA por permitir que essas classes fossem apresentadas.

As responsabilidades de objetos secundários foram definidas de forma similar, mas não foram incluídas aqui por razões de espaço. Nós também editamos certos detalhes que não eram importantes para o entendimento do modelo de objetos.

Algumas dessas responsabilidades não foram consideradas seriamente até a Fase II do projeto, a qual foi proposta quando este livro estava sendo impresso.

Nós incluímos este modelo como um bom exemplo de como realizar projetos de sistemas em termos de um conjunto pequeno de objetos de negócio comportamentalmente completos. Não causa problemas o fato de que este particular modelo esteja disponível fora do DSFA, ainda que existam certos padrões dentro do modelo que possam ser de ampla aplicação.

Objeto beneficiário

Um beneficiário é alguém que tem um acordo com o Estado e que possui um PPSN (*Personal Public Service Number* - Número de Serviço Público Pessoal). A intenção do objeto Beneficiário é fornecer um único ponto de acesso a toda e qualquer informação relacionada ao beneficiário que possa ser importante em mais de um contexto, por exemplo, para mais de um pagamento.

A maior parte dos principais dados do objeto Beneficiário, e de alguns novos objetos secundários tal como Histórico de Contribuições, é manipulada no banco

de dados CRS (*Central Records* – Registro Central) do Departamento. Porém, com o advento da iniciativa REACH (o framework para o E-government da Irlanda), o objeto Beneficiário deve também ser visto como uma interface para o 'agente de serviços públicos' que forma a mais importante tela do REACH. Várias responsabilidades listadas abaixo foram explicitamente planejadas para o agente de serviços públicos.

Isso enfatiza a importância da especificação das responsabilidades do objeto Beneficiário e, portanto, da interface com outros objetos, principalmente em termos de responsabilidades 'saber como', ao invés de 'saber o que'. Com o passar do tempo, várias dessas responsabilidades 'saber como' serão delegadas para o agente de serviços públicos, ou a outros sistemas, e a intenção do projeto é tornar esta transferência invisível para o resto do sistema que estiver conectado através de interfaces com o objeto Beneficiário. Supomos que mais cedo ou mais tarde, apenas as responsabilidades dos objetos Beneficiário, realizadas dentro dos próprios sistemas do DSFA, sejam conhecidas de outros objetos específicos do DSFA. Em médio prazo, o objeto Beneficiário terá que realizar a maioria dessas responsabilidades diretamente.

Responsabilidades 'saber o que'

- `Casos` nos quais o beneficiário é citado.
- `Relacionamentos` com outros beneficiários, incluindo 'mãe de', 'esposa/esposo de', 'nomeado por', 'guardião legal de'.
- `Comunicações` de e para o cliente (que são tratados dento de um Caso específico, por exemplo, aviso de mudança de endereço).
- `Métodos de Pagamento` – métodos pelos quais pagamentos podem ser realizados ao beneficiário, por exemplo: depósito em conta bancária, serviço especial de correio.
- `Endereço` de comunicação.
- `Esquemas` nos quais o beneficiário é citado.
- `Restituição de Pagamento` – objeto pertencente ao beneficiário.
- Se um beneficiário é um funcionário do DSFA ou outro servidor civil – neste caso, o objeto somente pode ser acessado pela unidade especial de funcionários.
- `Histórico de Contribuições` – resumo anual detalhado das contribuições de seguro social dos beneficiários.
- `Meios de Avaliação`.

Responsabilidades 'saber como'

- **Encontrar e Recuperar**. Este método permite ao usuário encontrar uma instância existente de beneficiário usando vários critérios disponíveis de busca. É implementado por pacote de recursos de busca especializado baseado no OpenVMS e disponibilizado através do Objeto Cliente.

- **Comunicar**. O objeto beneficiário fornece a habilidade de Comunicar com o beneficiário, usando algum meio ou endereço especificado (atualmente, correio, e-mail e telefone). Este método cria um novo objeto de Comunicação que cuida da transmissão e arquivamento da Comunicação. A Comunicação pode ser em irlandês se o beneficiário desejar, de modo que o objeto beneficiário saiba a linguagem preferida do beneficiário.
- **Autenticar**. Esta responsabilidade pretende verificar que o beneficiário, com o qual você está lidando (por exemplo: por telefone, face a face, sob a web, ou por e-mail) é a pessoa que diz ser. Esta facilidade pode usar entradas diretas do beneficiário na forma de um password, PIN, PKI, smart card, assinatura eletrônica, ou biométrica. Em algum ponto no futuro isso pode evoluir para a publicação e manutenção de Cartões de Serviços Públicos. A autenticação será provavelmente implementada em conjunto com a iniciativa REACH.
- **Criar uma declaração**. O beneficiário deve estar apto a listar todos os Pagamentos, incluindo pagamentos adicionais, realizados num período específico.
- **Atualizar**. Um funcionário deve estar apto a solicitar, através do objeto beneficiário, que o beneficiário verifique suas informações e atualize-as quando apropriado (sujeito às regras individuais de validação de campos). Ela irá, no futuro, iniciar e processar dados de outras aplicações.
- **Registrar eventos da vida**. Devem existir métodos específicos para lidar com os principais eventos da vida, tais como aniversário, casamento, aposentadoria e morte.
- **Reagir aos eventos da vida**. Como por exemplo, notificar o beneficiário da necessidade de solicitar o direito de bem estar social três meses antes de completar 66 anos de idade.

Objeto esquema

Um objeto Esquema é responsável pela administração de um particular benefício ou conjunto de benefícios. O Esquema é uma interface. Cada esquema de benefício que o DSFA administra será representado por um objeto Esquema que implementa esta interface.

Existe de modo geral, duas diferentes formas de Esquema: Esquema de composição e Esquema de componente. O Esquema de componente modela benefícios individuais. Esquemas de composição são repositórios que manipulam um ou mais Esquemas de componentes. Assim, uma instância de um Esquema de Componente somente pode existir no contexto de um Esquema de Composição. Esquemas individuais irão variar no tipo de apoio que ele fornece ao Funcionário que manipula a solicitação. Em nível mais simplificado, a instância de Esquema meramente fornece um lugar conveniente para registrar fatos e decisões tomadas. Num nível mais sofisticado, o Esquema pode implementar algumas formas de máquinas de inferência e/ou uma calculadora como uma planilha eletrônica. No entanto, a filosofia fundamental do projeto é que o sistema forneça uma bancada

para elevar as habilidades, e aumentar a produtividade dos Funcionários – não tente automatizar um processo que necessariamente envolve julgamento e ponderação.

Um Esquema de composição irá sempre existir dentro de um Caso, criando um novo se necessário, a fim de ser processado pelo Funcionário. No entanto, uma vez que o Esquema está 'Em Pagamento', então o Caso normalmente não irá representar um papel ativo. O sistema em batch irá interagir diretamente com os Esquemas. Certos Esquemas precisarão de habilidade para executar por si só a revisão após um certo período, ou sob certos eventos – via o mecanismo do Caso.

Qualquer Esquema (composição ou componente) deve implementar as seguintes responsabilidades genéricas, mais algumas responsabilidades adicionais específicas para as suas próprias necessidades.

Responsabilidades 'saber o que'

- O `Beneficiario` que está solicitando o benefício e quaisquer outros beneficiários citados na solicitação. (Esquemas de Componente não precisam de uma forma; desde que eles podem ser obtidos do Esquema de composição do qual fazem parte, mas eles podem precisar, por exemplo, de um objeto beneficiário representando o Menor ou um Adulto Qualificado).
- O `método de Pagamento` que pelo qual o Beneficiário deseja ser pago, incluindo pagamentos nominais. (Os Esquemas de Componentes irão ter por definição o Método de Pagamento especificado no seu Esquema de Composição pai, mas isso pode ser sobrescrito se, por exemplo, o beneficiário desejar que diferentes componentes sejam pagos em diferentes parcelas ou diferentes contas. Note que para o Esquema Livre, o Método de Pagamento especifica o Provedor de Serviços e sabe como lidar com este Provedor de Serviços).
- Esquemas de Composição manipulados dentro deste Esquema (se ele for de composição).
- Datas Iniciais e Finais.
- Status.
- Quaisquer outras informações específicas deste Esquema (ou compartilhados pelos seus Esquemas de componentes) que não podem ser obtidos do objeto Beneficiario relevante.
- O `Caso` dentro do qual o Esquema é atualmente manipulado.
- Todos os `pagamentos` realizados de acordo com este Esquema.
- `Autorização` para as várias decisões realizadas pelo funcionário, incluindo elegibilidade e data de revisão.

Responsabilidades de 'saber como'

- **Solicitar informações necessárias**. Alinhado com o Modelo de Liberação de Serviços, esta capacidade pode gerar um formulário personalizado (papel ou eletrônico) sobre o qual o beneficiario pode confirmar detalhes

relevantes existentes e fornecer quaisquer informações omitidas. Para alguns esquemas, esta solicitação pode ter vindo de uma outra agência (por exemplo, uma escola ou doutor). A solicitação deveria normalmente gerar um objeto de Comunicação padronizado, preenchendo os campos quando apropriado. Onde as informações estiverem omitidas, devem ser manipuladas com o objeto Beneficiário, então a responsabilidade de solicitar a informação necessária é delegada ao objeto Beneficiário.

- **Registrar decisão de elegibilidade**. É equivalente à decisão de elegibilidade do beneficiário para o Esquema. Isso envolve criar um Certificado que represente a decisão legal do Funcionário de Decisão. A solicitação não pode ser prosseguir até que este estágio tenha passado. Se a solicitação não for aprovada, então o objeto Esquema continua a existir, mas o Caso que o contém pode ser fechado. A decisão de uma solicitação pode gerar automaticamente uma nota de aviso, usando a capacidade de comunicação do Beneficiário. (Nota: isso será normalmente necessário para que o Funcionário formalmente decida a elegibilidade para o Esquema de composição e por cada um dos Esquemas de componente que ele contém).
- **Calcular direitos** para algum período específico. Esta responsabilidade é realizada pela referência a um particular Esquema sendo processado. Isso implica que o Esquema deve conhecer as taxas e regras dos anos anteriores e não apenas do corrente ano. Ele deve conhecer também a freqüência de pagamentos as quais se aplicam a um particular esquema. Quando novas taxas e regras entram em validade, elas serão adicionadas na definição do Esquema. Se as novas regras e taxas seguirem a mesma estrutura como das anteriores, então pode se pensar em apenas adicionar uma linha numa tabela. Se elas introduzirem novas estruturas então as modificações serão mais complexas. Note, no entanto, que todas as mudanças nas taxas e regras do esquema estão contidas num particular Esquema – elas não irão se espalhar nos objetos Beneficiário ou Pagamento. A responsabilidade do cálculo dos direitos é usada como uma antecipação para gerar um pagamento para aquele período, mas pode também ser usada apenas para informar ao Beneficiário de quanto ele irá receber.
- **Calcular a data de início da solicitação**. Quando existir atrasos na submissão de uma solicitação, são permitidos alguns direitos retroativos. Existe um conjunto de regras para calcular a data de início dos retroativos.
- **Gerar novos pagamentos** para um certo período. Dependendo do Método de Pagamento selecionado, esta capacidade será normalmente executada por um processo externo em batch executado com alguma freqüência, por exemplo, semanas, quinzenal ou mensal. Aplicações de regras de taxação (com referência ao status da taxação do Beneficiário) pode ser parte embutida desta responsabilidade.
- **Gerar um cronograma de pagamentos**. Este método será usado quando o método preferido de pagamento é o depósito bancário (isto é, é necessário gerar o documento de pagamento em intervalos regulares – digamos, todo o dia 6 do mês ou todo ano – ou pagamentos especiais tal

como um Bônus de Natal. A freqüência da geração de documentos e o número de documentos para o depósito bancário irão variar de esquema para esquema). Ele pode ser usado durante a transição, por compatibilidade com os sistemas existentes.

- **Gerar pagamento da diferença** para um certo período. Este método irá executar o Gerar Novo Pagamento, mas não irá incidir qualquer desconto frente a algum pagamento existente para o Esquema no mesmo período. Isso será usado quando, por exemplo, as circunstâncias do beneficiário mudarem após uma programação de pagamentos futuros ter sido gerada (por exemplo, nasceu uma nova criança durante o ano). Esta responsabilidade pode também produzir um Pagamento negativo (por exemplo, quando é descoberto um pagamento excessivo). Nota: Isto serve apenas para fazer correções nos pagamentos futuros, e dentro de um único esquema. Em geral, a recuperação do pagamento excessivo é tratada por um Esquema de Recuperação de Pagamentos Excessivos dedicado.
- **Parcelamento dos pagamentos** de acordo com os requisitos individuais e legais. Isso pode ser realizado gerando-se pagamentos separados com base numa divisão percentual de acordo com os Beneficiários envolvidos.
- **Corrigir uma sobreposição**. Isso significa gerar um objeto Sobreposição que irá, efetivamente, transferir pagamentos espúrios realizados sob este Esquema para um outro Esquema.
- **Guiar o usuário**. Isso significa permitir que o usuário consulte a legislação relevante, ou mais provavelmente o guia do funcionário, relevantes para o processamento de solicitações. Isso pode ser implementado como um help sensível ao contexto de negócio.

As seguintes responsabilidades se aplicam apenas aos Esquemas de Composição:

- **Adicionar Esquemas de componentes**. Isso inclui reforçar as regras com as quais tais Esquemas de componentes podem ser adicionados.
- **Gerenciar interdependências** entre Esquemas de componentes. Um exemplo simples disto é o que o Esquema CB precisa para associar um ‘pedido de elegibilidade’ para cada um dos objetos de Solicitação para Menores (componente) – desde que eles não podem calcular por si mesmos.
- **Disseminar métodos** sobre os Esquemas de componentes. O método mais comum é o Gerar Pagamentos regularmente.
- **Agregar Pagamentos** gerados pelos Esquemas de componentes. A maioria das responsabilidades é delegada ao próprio objeto Pagamento. No entanto, o Esquema de composição precisará reforçar certas regras sobre quais Pagamentos podem ou não ser agregados (por exemplo, porque eles são pagos legalmente em diferentes dias da semana).
- **Auto-revisar** com base em certos eventos ou passagem do tempo.

Objetos de comunicação

O objeto de Comunicação modela uma única comunicação, por exemplo, entre um Funcionário e um Beneficiário, ou entre dois Funcionários. O papel do objeto de Comunicação não é apenas de permitir que tais comunicações sejam criadas, mas também de permitir que elas sejam entregues e arquivadas.

As Comunicações podem chegar ou sair. Além disso, o Objeto Comunicação é usado para registrar as observações.

O mecanismo de transmissão para uma Comunicação é alcançado através do objeto Endereço. A mesma interface de usuário é usada independentemente do canal de transmissão escolhido.

Uma observação é uma Comunicação que não tem um destinatário. Ela é normalmente realizada dentro de um objeto Esquema, Caso ou Beneficiário.

Responsabilidades 'saber o que'

- O `Endereço` do destinatário (obtido da lista de Endereços contida pelo objeto destinatário, por exemplo, Beneficiário, Funcionário ou qualquer outro objeto que é comunicável). O usuário pode escolher o endereço particular, mas será default para a primeira entrada na lista.
- O `Endereço` do remetente (obtido do objeto remetente). Ele terá como default a primeira entrada da lista de endereços de remetentes que é do mesmo tipo que o endereço do destinatário (embora todas as comunicações escritas irão listar várias maneiras de responder).
- Assunto. Se a Comunicação foi gerada dentro de um Caso, então esse objeto será registrado como assunto. Isso não só permite que a Comunicação seja entregue no local correto, mas também poderá permitir que alguém responda que está de acordo. Este campo pode também influenciar outros objetos de contextos.
- Data.
- Estado: esboço, enviado, retornado, carta padrão (somente leitura), etc..
- Conteúdo. O texto será manipulado em alguma linguagem de marcação generalizada (por exemplo, HTML).
- `Certificação` (por exemplo, assinatura digital) se requerido.
- Anexos. Numa grande variedade de formatos (por exemplo, Word, Acrobat) que são por si só capazes de ser transformado em várias formas tais como imagem em papel ou eletrônico. Além disso, o conteúdo pode incluir ponteiros para quaisquer objetos expressivos do sistema, embora eles somente sejam usados para que os destinatários que estejam no sistema (por exemplo, correio interno). Anexos podem também incluir registros de voz digitalizados.
- Linguagem de conteúdo. (Inglês ou Irlandês, inicialmente).

Responsabilidades 'saber como'

- **Transmitir**. A execução desta responsabilidade é realizada através do objeto Endereço.
- **Editar**. Permite que conteúdos de texto sejam criados e editados.
- **Responder**.
- **Encaminhar**. Isso é realizado criando uma nova comunicação que possua atualmente algum Assunto.
- **Recuperar** (responsabilidade de classe). Comunicações anteriores serão recuperadas da lista contendo objetos Beneficiário, Funcionário ou Caso.
- Anexar um arquivo como uma imagem para uma Comunicação.
- **Confirmar** sucesso de entrega (também executada em colaboração com Endereço).
- **Assinar**. Criar um Certificado de assinatura digital do remetente. Esta responsabilidade pode anexar uma imagem digitalizada de uma assinatura física, se desejado.
- **Copiar**. Copia toda a Comunicação.
- **Fechar**. Muda uma Comunicação para o padrão de comunicação somente leitura que pode ser copiado.
- **Anexar**. Usado para criar uma carta a partir de parágrafos padrões.

Objeto Funcionário

O objeto funcionário é o único ponto de contato de informação e funcionalidade associada com um indivíduo (um empregado do Departamento ou um associado) que pode usar o sistema de informação. Existe uma instância para cada indivíduo.

Usuários do sistema têm seu próprio objeto Funcionário prontamente acessível, com o qual pode-se realizar login e logoff, e armazenar sua visão no desktop pessoal. Além disso, o objeto Funcionário fornece acesso aos trabalhos atuais.

Um objeto Funcionário pode ser 'virtual', isto é, pode representar papeis e/ou seções de departamentos (por exemplo, a Seção de Registro de Solicitações).

Responsabilidades 'saber o que'

- Relacionamentos com outros `Funcionários`. Inclui supervisores, supervisionados e parceiros.
- `Casos`. Casos que estão atualmente associados ao funcionário.
- `Comunicações` para ou de Funcionário.
- `Endereço` para comunicação.
- Papéis executados.

Responsabilidades 'saber como'

- **Encontrar e Recuperar**. Estas responsabilidades são, genericamente, similares a aqueles especificados para o Beneficiario.
- **Logon e Logoff**.

- **Capturar e restaurar o desktop do usuário**.
- **Casos presentes**. Esta responsabilidade pode mostrar todos os casos atualmente associados ao Funcionário decompostos em várias categorias incluindo o status.
- **Gerenciar entradas e saídas da caixa de correio**.
- **Gerenciar níveis de autorização**. A autorização (para realizar um método específico sobre um objeto específico) será realizada por um amplo sistema servidor de autenticação e autorização. No entanto, o objeto Funcionário será uma interface de usuário principal sobre este servidor (isto é, o meio através do qual os níveis de autorização para papeis específicos e/ou individuais são especificados).
- **Comunicar**. Funciona da mesma maneira que a responsabilidade Comunicar do objeto Beneficiário.
- **Criar um Certificado** para registrar a base da decisão do Funcionário.

Objeto Pagamento

Um objeto Pagamento representa um único pagamento a partir de um pagador (por default, o Departamento) para o beneficiário (normalmente um Beneficiário). Um Pagamento é de várias formas análogo à Comunicação e compartilha algo de sua estrutura. Assim, o papel do Endereço numa Comunicação é trocado por um Método de Pagamento, onde ele pode representar um cheque, transferência eletrônica de valores, transferência eletrônica de informações ou um documento (este último normalmente formando parte de um registro de pagamento).

A quantidade de pagamento será determinada sempre que o objeto Pagamento for criado (por exemplo, um Esquema, ou, em raros casos, um Funcionário autorizado), junto com a data da dívida. O Pagamento pode ser uma quantidade negativa para propósitos de restituição de pagamento em excesso.

Os Pagamentos são geralmente criados no nível mais baixo possível para permitir que sejam postados de forma precisa dentro do sistema de contabilidade financeira. Assim, uma solicitação pode gerar a criação de vários objetos de Pagamento representando diferentes Esquemas de componentes tais como Pensão de Aposentadoria, Pensão Auxiliar, Desconto de Adulto Incapacitado ou Desconto de Dependentes Menores. Pagamentos que tenham sido criados mas ainda não executados e que tenha o mesmo beneficiário e data, podem ser combinados ou unidos com um outro Pagamento com o mesmo período de pagamento para formar uma única transferência de valores.

Responsabilidades 'saber o que'

- `Esquema` que provocou a criação do Pagamento.

- `Método de Pagamento` do Beneficiário. O rótulo descritivo do Método de Pagamento inclui o nome do Beneficiário, e pode fornecer acesso direto ao objeto representando o beneficiário (por exemplo, um Beneficiário ou Agência).
- Identificação do Pagamento. Por exemplo, número do cheque ou número do documento.
- Componente `Pagamento`. Significa que qualquer composição de pagamento conhece quais outros pagamentos do qual ele é feito.
- Quantidade (expresso numa moeda).
- Status (emitido, pago, parado, reconciliado).
- Razões da Parada, se o status indicar que o pagamento foi parado.
- Tipo de Pagamento. Isso indica se o pagamento é um pagamento regular, um pagamento de substituição, pagamento de doação, etc. Este campo é de formato livre cujo conteúdo são normalmente determinados pelo Esquema que cria o Pagamento.
- Período de Pagamento em que está associado.
- Data da dívida. (pode ser que esta seja uma função de período de pagamento, por exemplo, primeira terça-feira no período).

Responsabilidades 'saber como'

- **Unir com um outro pagamento** (sujeito às regras). Normalmente, instâncias de Pagamento são criadas no nível de elementos de Esquema (por exemplo, dependentes menores, desconto de combustível) e então unidas para formar um único pagamento que é transferido ao Beneficiário.
- **Postar** no sistema de contabilidade financeira.
- **Autorizar**. Muitos pagamentos são gerados dentro de Esquemas, que irá cuidar de seus próprios níveis de autorização. No entanto, pode ser apropriado colocar alguns conceitos genéricos adicionais de autorização dentro do objeto de Pagamento (por exemplo, para pagamentos acima de 5.000 Euros).
- Parar o pagamento individual ou todos os registros de documentos de pagamento.
- **Publicar** de maneira apropriada o Método de Pagamento.
- **Deduzir taxa**, se apropriado, pela referência ao status passível de taxação do beneficiário e/ou a razão para o pagamento. (Isso pode ser pensado como um parcelamento do pagamento entre o beneficiário e a autoridade de cobrança de impostos).
- **Avisar**. Gerar um aviso ou pagamento para o beneficiário, gerando um objeto de comunicação apropriado ou gerando uma nota de aviso para inclusão no livro de beneficiários.

Objeto Caso

O Objeto Caso é usado atualmente para manipular instâncias de Esquema e é o mecanismo segundo o qual uma instância de Esquema pode estar ligada a um outro Funcionário. O Caso pode atuar como um detentor para apoiar quaisquer comunicações (incluindo Observações) e pode no futuro possuir imagens de outros documentos associados ao Esquema sendo processado, mas que pode não estar explicitamente organizada dentro do Esquema. No entanto, o trabalho contido num caso não tem que estar relacionado a um Esquema. Ao invés disso pode ser qualquer tipo de trabalho Departamental a partir da correspondência com intuito de investigações.

O Caso fornece certas características similares ao workflow, incluindo a habilidade de transferir o caso para um outro funcionário.

Responsabilidades 'saber o que'

- O `Funcionário` atualmente responsável pelo caso.
- O `Funcionário` para que o Caso estava previamente associado (se houver algum).
- `Esquema` que cria parte do caso (que por sua vez conhece o Beneficiário).
- `Comunicações` relacionadas ao caso.
- Outros documentos relevantes (incluindo, potencialmente imagens) e notas.
- Status atual. Por exemplo: Pendente – beneficiário; Pendente – outros; Em pagamento; Fechado.
- Data de Revisão – a data quando será dada atenção ao caso para revisão. Esta data usualmente será determinada pelo Funcionário.

Responsabilidades 'saber como'

- **Consultar um outro Funcionário**. Esta consulta pode ser temporária (por exemplo, para autorizar prosseguir) ou uma situação permanente. A natureza da consulta deixará isso claro. A consulta pode ser iniciada meramente arrastando o objeto Caso sobre o objeto Funcionário apropriado. Adicionalmente à mudança do Funcionário associado, a consulta irá gerar uma Comunicação padronizada para aparecer no quadro do destinatário.
- **Alerta**. Publicar e assinar é uma capacidade genérica da Arquitetura de Objeto Expressivo. Espera-se que o Caso seja significativo ao usuário desta capacidade – usando um evento especificado sobre um objeto dentro de um Caso que pode mudar o status do caso por si só.
- **Manter Histórico**. Todos os objetos mantêm um histórico completo tanto de acessos quanto de mudanças para propósitos de auditoria. No entanto, para o objeto Caso, a história das mudanças precisa ser mantida na forma que seja facilmente acessível pelo funcionário – por exemplo, enquanto estiver falando com o beneficiário ao telefone.

Introduzindo os 'naked objects'

O projeto do DSFA foi a nossa primeira demonstração de que era possível projetar um sistema de negócio de missão crítica de larga escala a partir de um conjunto de objetos de negócio comportamentalmente completos, os quais foram expostos diretamente ao usuário ao invés de estarem escondidos em uma interface de usuário convencional. Durante o projeto nós nos vimos chamando esses objetos de negócio como 'naked objects' (‘objetos pelados’).

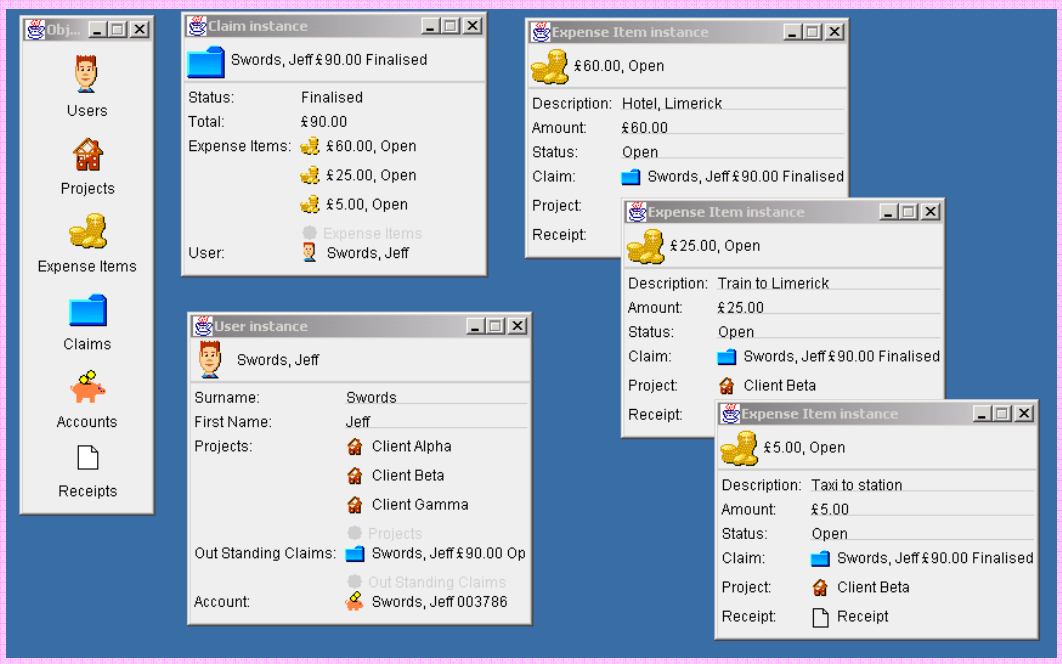
Seguindo este sucesso, procuramos criar um framework de propósito geral para habilitar este conceito e ser mais largamente aplicado. Nós empregamos o framework de prototipação simples, baseado em Java, que nós tínhamos usado para construir o protótipo ilustrado nos casos de estudo, e o reconstruímos a partir do zero para tornar viável como uma ferramenta de implementação. Isso significa atender os assuntos de persistência de objetos, compartilhamento e distribuição.

Este framework é conhecido agora como Naked Objects.

O framework Naked Object

Para desenvolver uma aplicação usando Naked Objects, tudo que o desenvolvedor escreve são os naked objects: os objetos de negócio que modelam o domínio. Cada classe de objetos de negócio (por exemplo, Beneficiário, Produto e Pedido) deve ser descrita como uma classe Java, e devem implementar uma interface chamada NakedObject que é fornecida pelo framework. A maneira mais simples de fazer isso é declarar cada uma de suas classes de negócio como sendo subclasses da classe AbstractNakedObject fornecida com o framework. O programador deve escrever para o objeto de negócio, o código necessário para especificar cada atributo, associação com outros objetos de negócio e comportamentos. O código deve seguir algumas convenções simples, mas em geral os objetos de negócio são codificados da mesma forma que os objetos de negócio comportamentalmente completos deveriam ser escritos para a camada do modelo de negócio de qualquer sistema multicamadas.

Quando o conjunto de classes de negócio é compilado e executado, o ‘mecanismo de visualização’ genérico do framework fornece ao usuário com uma visão dos objetos de negócio na forma como é mostrada na tela abaixo. Instâncias de objeto de negócio individuais apresentam-se como ícones. O estilo do ícone indica a qual classe ele pertence, e um único título, especificado pelo programador e normalmente derivado de um ou mais atributos primitivos do objeto, tais como nome, data e número de referência, distingue esta instância. Quaisquer desses ícones podem receber “duplo-clique” para abrir uma ou mais visões detalhadas dos atributos e associações do objeto com outros objetos, que mostram seus próprios ícones.



Para desenvolver o protótipo neste estágio inicial para um sistema de processamento de despesas usando o framework do Naked Objects, os desenvolvedores têm apenas que escrever o código especificando as classes de objetos de negócio ‘naked’. Eles são mostrados em janelas de classes. Instâncias individuais capturam o ícone de suas classes por “default”. Eles podem ser abertos para revelar seus atributos e associações arrastando-os e soltando-os sobre outros objetos ou campos específicos. O botão direito do mouse é utilizado para revelar o menu pop-up de ações que incluem tanto operações genéricas quanto comportamentos de negócio.

Os benefícios dos naked objects

Pressionando o botão direito do mouse sobre qualquer objeto, este irá produzir um menu pop-up de ações, métodos de instâncias, que usuários podem chamar sobre esse objeto. Pressione o botão direito do mouse sobre um objeto Beneficiário e você poderá estar apto a se ‘comunicar’ com esse beneficiário. Essa comunicação pode ser via um dos endereços de comunicação do beneficiário ou talvez ‘avaliar o valor’ do beneficiário para o negócio baseado nos pedidos passados.

Além disso, nesses métodos de negócio, o menu pop-up pode oferecer aos usuários diferentes maneiras de ver o objeto. Por exemplo, você pode estar apto a ver um objeto Produto como uma imagem fotográfica; uma coleção de objetos similares como uma tabela, ao invés de uma lista, ou um conjunto de valores numéricos como um grafo. E se o objeto possuir coordenadas espaciais associadas (por exemplo, latitude e Longitude) então você pode querer vê-lo sobre

um mapa ou layout gráfico. Essas opções de visualização são criadas automaticamente, baseadas sobre o tipo de objetos: elas não têm que ser escritas pelo desenvolvedor. O framework pode ser estendido para criar novas maneiras de visualizar um objeto, mas isso pode não ser necessário para uma aplicação muito simples.

Assim como as ações de menu pop-up, o comportamento pode também ser iniciado por ações arrastar e soltar do usuário. O usuário pode arrastar um objeto sobre um outro objeto, o qual irá disparar uma operação pré-definida envolvendo os dois objetos; ou arrastar um objeto dentro de um campo específico dentro de um outro objeto, normalmente para especificar uma associação entre os dois. Se o usuário tentar arrastar o tipo de objeto errado de objeto, a zona de soltura ficará vermelha e a soltura não funcionará. Similarmente uma ação de menu talvez fique cinzenta se a ação não for permitida, normalmente por causa do estado corrente do objeto. O conjunto de ações de menus, atributos e associações que o usuário vê pode também ser personalizado de acordo com os papeis que eles fornecem.

Além disso, para as instâncias individuais, e coleções de instâncias, o sistema apresenta ao usuário uma representação direta de suas próprias classes – apresentada nas telas como janelas de Classes. Ai será de onde o usuário irá chamar os comportamentos que não pertencem a uma única instância: assim como criar uma nova instância dessa classe, para encontrar uma instância específica, ou para listar as instâncias dessa classe que atendam a algum critério específico.

Tudo que o usuário faz é uma operação direta sob um dos objetos de negócio, ou sob sua classe. Não existe nenhuma decoração entre o usuário e os naked objects. Não existem menus de nível superior, nenhum script. Não existe nem mesmo quaisquer caixas de diálogo.

O Naked Objects encorajam o projeto de objetos comportamentalmente completos de duas formas. Primeira, positivamente: a representação visual completa dos objetos de negócio ajuda os usuários a se identificarem com o modelo de objetos que conseguem mais facilmente contemplar os objetos comportamentalmente completos. Segunda, negativamente: se não existe outra maneira de construir funcionalidades de negócio dentro do sistema, então os projetistas são forçados a associar todos os comportamentos requeridos com um único objeto de negócio.

O framework Naked Objects encoraja o projeto de objetos de negócio comportamentalmente completos de forma direta. Mas qual é o verdadeiro benefício? A completeza comportamental pode satisfazer alguns objetivos intelectuais, mas nossa abordagem libera algum benefício de negócio concreto?

Nossa experiência até hoje tem demonstrado quatro desses benefícios:

- Os naked objects podem acomodar melhor as futuras mudanças nos requisitos de negócio.

- Os naked objects dão “poder” ao usuário.
- Os naked objects melhoram a comunicação entre desenvolvedores e usuários.
- Os naked objects podem acelerar o processo de desenvolvimento.

Em nossa experiência nós vimos cada um desses benefícios um após o outro. Abaixo apresentamos uma síntese de cada um dos benefícios acima citados.

Os naked objects podem acomodar melhor as futuras mudanças nos requisitos de negócio

Sistemas construídos a partir dos naked objects são mais ágeis, no sentido de que eles podem mais facilmente acomodar as futuras mudanças nos requisitos de negócio – mudanças nas especificações de produto, na estrutura da organização, nas regras internas e regulamentos externos, nos processos de negócio, e nos relacionamentos com fornecedores e beneficiários.

Existem muitas técnicas para construir maior agilidade em sistemas de negócio. Uma abordagem comum é identificar as coisas que são mais prováveis de mudar, separá-las da operação do sistema e representá-las de forma explícita e modificável. Workflow e outras ferramentas de modelagem de processos extraem uma representação do fluxo de trabalho entre pessoas e entre sistemas e a torna modificável. Máquinas de inferência fazem o mesmo, digamos, com as regras de configuração de produto. No entanto, a limitação dessa abordagem é que as mudanças de negócio freqüentemente não se ajustam muito bem com esses pacotes.

Uma abordagem alternativa é dividir todo o sistema em componentes de “granularidade” fina, de muitos tipos diferentes, e então usar uma arquitetura de propósito geral para costurá-los em sistemas usáveis. Isso na essência é a abordagem de serviços web. Apesar dessa técnica oferecer maior flexibilidade que imensos sistemas monolíticos, é uma ilusão assumir que tal granularidade mais fina seja sinônimo de maior agilidade. Uma mudança aparentemente simples de negócio pode facilmente requerer modificações de dezenas de componentes. Se tais componentes não estiverem bem isolados um dos outros, eles podem facilmente gerar um efeito dominó.

Não existe a “tal coisa” como uma representação de propósito geral que pode facilmente acomodar qualquer tipo de mudanças inesperadas. Entretanto boas modelagens de negócio são os que mais aproximam você desse ideal. Isso porque quando a modelagem de objetos é bem feita, as responsabilidades dos objetos não são dirigidas principalmente pelos requisitos específicos de negócio. Ao invés disso, os projetistas procuram identificar as responsabilidades naturais que podem ser antevistas para cada objeto. As implementações iniciais dessas

responsabilidades podem estar otimizadas para as necessidades imediatas, mas a noção mais abstrata terá sido preservada no projeto.

Tome um exemplo simples: é natural quando se especifica um objeto Beneficiário, estabelecer que ele precisa conhecer o endereço postal do beneficiário. Mas uma abordagem dirigida por responsabilidade deveria sugerir que a real responsabilidade de um objeto Beneficiário deveria ser de conhecer como se comunicar com o beneficiário (mesmo se a primeira iteração do sistema cubra apenas a comunicação postal). Se a interface de mensagem do objeto é definida como:

```
comunicarViaOCanalPreferidoDoBeneficiario(esteConteúdo)
```

Nós exageramos no nome do método para frisar o ponto ao invés de:

```
obterEndereçoPostal()
```

Se novas formas de comunicação forem introduzidas, por exemplo: e-mail, fax, mensagem instantânea, apenas o código definindo na classe Beneficiário precisará ser mudada. Todos os outros objetos de negócio que precisarem comunicar com o beneficiário podem continuar a chamar o mesmo método sobre o objeto Beneficiário sem ter que se preocupar em especificar qual das várias opções usar (embora a opção de especificar o canal possa ser adicionada). Como Alec Sharp observa: 'O código procedural obtém informação e então toma as decisões. O código orientado a objetos descreve objetos para fazer algo', [Sharp1997].

Os naked objects dão poder ao usuário(Empowerment)

'Empowerment'[3] é um termo exagerado em objetos de negócio. Com freqüência ele é meramente um eufemismo para 'downsizing', 'de-layering' ou para planos tradicionais de delegação. Mas o fato é que um termo exagerado não nega a importância da idéia.

Andrew Clement argumenta que 'empowerment' toma duas formas distintas: funcional e democrática [Clement1996]. O 'empowerment' funcional é orientado para fornecer 'desempenho voltado para as metas organizacionais que se assumem que sejam comuns a todos os participantes' – por exemplo, quando se dá grande importância a um representante de serviço do beneficiário para resolver um problema do beneficiário no interesse de proteger o beneficiário e a reputação da empresa. O empowerment democrático (nós preferimos chamar isso de empowerment 'intrínseco') preocupa-se em dar ao indivíduo uma 'grande compreensão e senso de seu próprio poder'. Isso é feito no interesse do indivíduo e não é orientado em direção a alcançar qualquer meta externa explícita, embora seja um benefício indireto ao negócio no aprimoramento motivacional e

[3] Decidimos manter o termo orignal em inglês, pois a tradução literal poder de usuário na revela o verdadeiro significado da palavra original.

manutenção da equipe. O empowerment intrínseco é claramente uma noção mais sutil. Muitas iniciativas com intenção de fortalecer o empowerment são, embora retóricas, puramente funcionais. Clement diz que ‘Para que o empowerment ofereça uma promessa autêntica de melhoria na experiência e resultado de trabalho, ele precisa combinar a atenção para os aspectos efetivos de trabalho da abordagem funcional com as aspirações de emancipação da abordagem democrática ’.

Como os naked objects apóiam o empowerment funcional

A principal maneira na qual os naked objects apóiam o empowerment funcional é pela redução da modalidade. Muitos sistemas são fortemente modais, e isso pode normalmente enfurecer usuários. A estória do processador de texto Bravo, contada com freqüência, ilustra uma forma desse problema: se você teclar a palavra ‘EDIT’ enquanto estiver no modo errado, isso poderia ser interpretado como '*select Everything; Delete it, then Insert the letter 'T'* (‘selecione tudo; remova-o, então insira a letra ‘T’)! [Hiltzik1999]. Uma forma mais comum do problema ocorre em muitos sistemas de negócio transacionais onde usuários são forçados a completar um roteiro de tarefa antes que eles possam iniciar um outro. Aplicações discretas são, por si só, formas de modalidade: usuários têm que trocar de aplicação para chamar uma função particular ou obter uma peça de informação.

Além de causar frustração, a modalidade pode estar funcionalmente desautorizando, pois os modos freqüentemente cortam caminho na maneira de pensar do usuário sobre o problema, ou na ordem em que a informação torna-se disponível. Tarefas orientadas a interface de usuário, como discutido na seção 1 são sempre fortemente modais. A modalidade não pode ser eliminada, mas se os projetistas forem conscientes do problema, eles podem torná-la menos obstrutiva.

Uma das maneiras mais efetivas de reduzir a modalidade é adotar a forma de interação ‘substantivo-verbo’. Quase todos os principais sistemas de negócio da atualidade seguem o estilo oposto de interação (‘verbo-substantivo’): o usuário seleciona um verbo ou uma tarefa a partir de um menu inicial (tal como ‘Modificar registro de beneficiário’ ou ‘Registrar nova solicitação’) e então fornece os dados necessários, em resposta à solicitação do sistema. No estilo de interação substantivo-verbo, o usuário seleciona um substantivo e então escolhe um dos verbos disponíveis que deseja aplicar. A metáfora de desktop adota este estilo de interação: o ícone de arquivo representa o substantivo e o menu pop-up que surge a partir do clique direito do mouse, fornece o verbo tais como Abrir, Imprimir e Enviar e-mail. No entanto, muitas aplicações executadas a partir do desktop apresentam compromissos bem menores com a forma substantivo-verbo. Embora a interação substantivo-verbo não seja necessariamente mais eficiente do que verbo-substantivo, existem vários argumentos sugerindo que ele está mais próximo de como as pessoas pensam, especialmente nas atividades de resolver problemas.

O estilo substantivo-verbo de interação ajusta-se bem a interfaces de usuário orientadas a objetos (IUOO). Mas existem poucas ferramentas de apoio a projetos de IUOO (como as diversas ferramentas orientadas a objetos para apoio a projeto de GUIs convencionais). Os naked objects, por definição, fornecem uma IUOO. O framework do Naked Objects permite que você produza esta IUOO automaticamente a partir das definições de objetos de negócio.

O empowerment funcional, ou quando se trata o usuário como solucionador de problemas, não implica na eliminação de todas as restrições. Certas regras necessitam que sejam reforçadas tanto por razões legislativas quanto por razões de negócio. Mas quando objetos de negócio são nakeds (pelados), todas as regras e restrições são encapsuladas em tais objetos ao invés de estarem associadas a roteiros ou procedimentos que se assentam sobre eles. Esta é melhor solução de todas. Ela dá aos usuários maior liberdade de decidir como fazer para cuidar do problema. Ela obriga os projetistas a distinguirem mais claramente entre uma regra absoluta e uma mera convenção procedural. E tendo decidido que algo é uma regra absoluta, embutem-na dentro do objeto de negócio para o qual a regra se aplica, tornando mais difícil que sejam evitadas pelos usuários. O painel ilustra os riscos de usar procedimentos roteirizados para implementar regras de negócio ao invés de encapsulá-las no objeto apropriado.

Como os naked objects apóiam empowerment intrínseco

Brenda Laurel diz que ‘Operar um programa de computador é normalmente uma experiência em segunda pessoa: uma pessoa que faz declarações imperativas ou solicitações para o sistema, e a ação executada pelo sistema, roubando completamente o papel da pessoa como agente’ [Laurel1991]. Com o tempo isso traz um sutil impacto, porém acumulativo, sobre o senso dos usuários de seus próprios valores e motivações. Os naked objects contribuem para o senso dos usuários sobre o empowerment intrínseco, pois procuraram dar ao usuário uma experiência em primeira pessoa.

Tome isso num outro estágio: Hutchins, Hollan e Norman estabeleceram que ‘existem duas principais metáforas para a interação homem-máquina: uma metáfora de conversação e uma metáfora do modelo do mundo. Num sistema construído sobre a metáfora de conversação, a interface é uma linguagem mediadora em que o usuário e o computador têm uma conversação sobre um mundo fictício, mas não representado explicitamente. Neste caso, a interface é um mediador implícito entre o usuário e o mundo sobre o qual as coisas são ditas. Num sistema construído sobre a metáfora do modelo do mundo, a interface por si só é um mundo onde o usuário pode atuar, e que muda de estado em resposta às ações do usuário. O uso apropriado da metáfora do modelo do mundo pode causar no usuário a sensação de que está atuando sobre os objetos do domínio da tarefa’ [Hutchins1986]. Eles descrevem assim os naked objects!

O framework Naked Objects fornece mecanismos que reforçam a metáfora do modelo do mundo e positivamente, dificulta a adoção da abordagem

conversacional. Por exemplo, a razão do extensivo uso do arrastar e soltar no framework Naked Objects não tem nada a ver com eficiência, mas sim porque os gestos físicos ajudam a causar a sensação de engajamento direto e empowerment intrínseco.

Ações físicas ajudam a entender e memorizar: psicólogos de crianças nos dizem que crianças desenvolvem uma forte representação mental das coisas que eles podem fisicamente mover do que com coisas que eles não podem mover – o qual Bruner chamou de uma 'representação ordenada' [Bruner1966]. A idéia de Bruner foi uma das inspirações subjacentes ao trabalho do Grupo de Pesquisa de Aprendizagem da Parc [Kay1990]. Quando crianças se desenvolvem, elas elevam o uso de representações simbólicas e visuais, mas as representações ordenadas ainda possuem um papel importante. Isto pode ser verificado na situação na qual duas pessoas estão viajando de carro para um novo destino: um está dirigindo, o outro está navegando utilizando o mapa, observando os sinais da estrada e indicando direções. Subseqüentemente, ambos indivíduos seguem o mesmo caminho utilizando as suas memórias. O motorista acha essa tarefa muito fácil: o movimento dos músculos desempenha um papel importante na memorização quando empregado em atividades necessárias.

A abordagem do Naked Objects procura evitar mensagens verbais sempre que possível. Ao invés de esperar que usuários cometam um erro para então apresentar uma mensagem de alerta, ele previne que o usuário cometa erros. Por exemplo, arrastando um objeto através de uma série de campos fará com que cada campo fique na cor vermelha ou verde indicando se o objeto pode ou não ser solto sob o campo.

Encontrar formas de mensagens verbais requer algum pensamento criativo. A melhor forma de evitar mensagens ('Está certo disso?') inoportunas é fornecer ao usuário uma função genérica de desfazer operações. Mas existem situações onde isso é muito difícil de implementar – e agora? No mundo físico, se um botão ou chave puder causar sérias conseqüências, então normalmente ele é pintado em vermelho, e se ele puder causar conseqüências muito mais sérias, então provavelmente ele terá uma proteção. Colocar uma proteção num botão ou menu operado por mouse seria fácil: ele teria o mesmo efeito positivo do que a mensagem 'Está certo disso?', mas sem causar a sensação de intrusão, ou padronização. Nós esperamos implementar isso numa futura versão do framework.

Nós não estamos dizendo que a interface de usuário produzida pelo framework Naked Objects seja o supra-sumo de interface amigável, ao menos no sentido convencional da frase. Existem princípios de projeto bem estabelecidos (tais como as 10 regras de ouro de Shneiderman [Shneiderman1998]) e técnicas específicas de projetar interfaces de usuário com maior apelo estético, mais eficiente em termos do número de cliques de mouse ou toques necessários para realizar uma tarefa [Raskin2000], facilidade para o iniciante entender, e menor propensão a erros do usuário. Nós adotamos alguns desses princípios; nós podemos fazer

mais. Mas alguns são incompatíveis com a autogeração de interface de usuário que é parte de nossa abordagem. Por essa razão, alguns receberam fortes críticas do framework Naked Objects.

A pesar disso, usuários continuem a nos dizer que eles realmente gostam dos sistemas que são construídos usando o framework. Nós concluímos que eles gostam é do sentido de empowerment, tanto funcional quanto intrínseco, que os naked objects fornecem, e que os seus valores estão além das noções convencionais de 'amigável' e de otimização.

Interfaces de usuário autogeradas possuem, inevitavelmente, limitações, e algumas aplicações irão requerer uma interface de usuário fácil de usar. No entanto, como os estudos de casos deste livro testemunham, a interface de usuário autogeradas pelo Naked Objects provou ser tanto efetivos quando largamente aplicáveis.

Os naked objects melhoram a comunicação entre desenvolvedores e usuários

O primeiro e mais notável benefício de usar nossa abordagem de projeto e desenvolvimento é que os naked objects facilitam a comunicação entre usuários e desenvolvedores.

Têm se exigido que a orientação a objetos seja a ponte de comunicação (algumas vezes chamado de 'gap semântico') entre usuários e desenvolvedores. Mas na prática isso quer dizer que ligar por uma ponte a lacuna existente entre diferentes estágios do ciclo de desenvolvimento de software, por exemplo, entre a análise e o projeto, e não entre o que o usuário fala e o que o programador codifica. A representação do usuário final dos sistemas desenvolvidos usando técnicas orientadas a objetos é raramente orientada a objetos. Por outro lado, em muitos sistemas que têm interfaces de usuários orientadas a objetos os objetos experimentados pelo usuário não imitam necessariamente os objetos da camada de negócio.

Os naked objects fornecem uma linguagem comum e genuína entre desenvolvedores e usuários finais do sistema, e isso facilita o envolvimento dos usuários no projeto do sistema, que é uma das máximas dos métodos de desenvolvimento mais modernos.

Em todos os projetos onde nós empregamos esta abordagem, os usuários envolvidos no processo de projeto sentiram-se rapidamente confortáveis com a linguagem de objetos. Isso não se limita somente na idéia de objetos de negócio (tais como Beneficiários, Produtos, Pedidos, Pagamentos, entre outros), mas também nos conceitos tais como 'instâncias', 'classes' e 'subclasses', 'método de instância' e 'método de classe', 'atributo' e 'associação'. Após apenas algumas horas de envolvimento num projeto com Naked Objects, os usuários começam a expressar suas idéias e a solicitar novas funcionalidades em termos de objetos:

’Seria possível colocar uma nova ação no objeto Promoção para visualizar a propaganda?’; ‘Eu queria associar um Método de Pagamento individual (objeto) com cada diferente Benefício (objeto), não apenas com o Esquema completo (objeto)’; entre outros. Os naked objects tornam visíveis os objetos de negócio aos usuários, assim, usuários conversam com eles com muita naturalidade.

Em contraste com a situação normal, onde usuários especificam requisitos em termos de alterações de processos: novo item de menu, novos relatórios, alterações no formulário. Alguém (talvez os desenvolvedores, ou talvez um intermediário tal como um analista de sistemas) deve traduzir tais requisitos não apenas dentro das mudanças necessárias para dados e funções apropriadas, mas dentro de todas as telas e controladores de interação de usuário que são afetados.

Os naked objects podem acelerar o processo de desenvolvimento

O fato da interface de usuário ser autogerada a partir das definições do objeto de negócio torna-o possível prototipar muito rapidamente. Ao usar nosso framework, solicitações simples de mudanças no protótipo (tal como um novo atributo, associação, ou um método) podem, normalmente, ser implementadas em tempo real, na frente dos usuários que estão fazendo a sugestão. Nem todos os protótipos podem ou deveriam ser realizados dessa maneira – algumas vezes será necessário codificar. Mas nós verificamos que é possível fazer muitas iterações no tempo em que muitas abordagens permitem apenas uma.

A idéia de ferramentas de prototipação rápidas não é nova, mas muitas dessas ferramentas são meramente ferramentas de desenvolvimento de interface de usuário. Com naked objects você cria o protótipo a partir do modelo de objeto ao mesmo tempo em que a interface de usuário, porque eles são efetivamente a mesma coisa. Durante a prototipação, todos os atributos, associações e métodos tornam-se imediatamente visíveis ao usuário. (O sistema finalizado pode ocultar alguns deles, mas isso ocorre no final). A vantagem disso é que o protótipo por si só documenta o modelo de objetos.

Qualquer coisa que possa eliminar a documentação será bem vinda pelas organizações de desenvolvimento. Uma pessoa de fora observando os principais métodos de desenvolvimento de software em operação poderia ser perdoada por pensar que o seu propósito era gerar documentação. A tarefa de criar, editar, formatar, circular e validar documentos consome grandes quantidades de tempo e esforço. Mas o verdadeiro problema de métodos com forte peso em documentação é a manutenção da consistência entre eles. Os mais recentes métodos ‘ágeis’ normalmente geram menos documentação e poucos tipos de documentos que precisam ser mantidos sincronizados.

Questões mais freqüentes

Nesta seção nós cobrimos algumas das questões que as pessoas normalmente levantam quando são apresentados pela primeira vez aos conceitos de naked objects – mas antes que eles tivessem começado a usar o framework. Muitas questões estão relacionadas como se a abordagem é adequada aos seus tipos de sistemas ou seus ambientes de negócio em geral.

Quais tipos de sistemas de negócio se beneficiam mais quando projetados como naked objects?

Os naked objects funcionam bem sempre que:

- É reconhecido que o sistema de negócio resultante precisa apoiar uma forma de resolver problemas.
- Existir consideráveis incertezas nos requisitos de negócio para o sistema.

Nós podemos argumentar que ambos se aplicam a qualquer projeto de novos sistemas, mas o grau varia.

Sistemas com interface beneficiários são bons candidatos a uma abordagem problema-solução. Dito isso, deve-se reconhecer que o empowering, a natureza expressiva dos sistemas resultantes, assume um grau razoável de familiaridade por parte do usuário. Nós não afirmamos que sistemas naked objects sejam 'intuitivos'. De fato, nós apoiamos a visão de Jeff Raskin de que não existe essa coisa de intuição (pelo menos da maneira como as pessoas pensam), mas apenas familiaridade [Raskin2000]. Se os usuários são representantes de serviços de beneficiário os quais interagem freqüentemente com o sistema, se não continuamente, então eles irão ficar rápidos e suficientemente familiarizados com o sistema para obter vantagens de ter acesso direto aos objetos. Se eles usam o sistema apenas ocasionalmente, a falta de familiaridade pode ser um problema.

Por essa razão, os naked objects quase nunca são apropriados para sistemas que irão ser usados diretamente pelos beneficiários – eles estarão melhores com uma abordagem de roteiro. Existem algumas exceções, tais como o banco on-line com o gerenciamento de finanças pessoais do tipo Quicken, ou mercearias on-line. Ambos envolvem uso freqüente e um tanto intensivo, e é possível ver como os naked objects podem trazer benefícios a ambos.

No entanto, uma vez que você tenha desenvolvido um sistema com os naked objects, não existirá nada que possa impedir você de desenvolver uma interface de usuário roteirizada convencional que chame as capacidades desses mesmos objetos ou de fornecer interação restrita para uma classe específica de usuários, ou ainda para arcar com as limitações de uma interface de browser.

A interface de beneficiário não é apenas uma área que se beneficia de um estilo problema-solução de interação. Ela é também a área de agendamento de recursos, especialmente na interação entre planejamento e operações. As companhias de aviação, por exemplo, têm ferramentas sofisticadas de planejamento e execução de agenda. Mas quando interrupções significativas ocorrem, causadas talvez por uma tempestade, os sistemas fornecem apoio muito limitado para simulação e então execução ao vivo de contornar o problema. Vendas, promoções, comércio, operações de rede, e gerenciamento de riscos são também exemplos de atividades operacionais intensas que demandam uma abordagem problema-solução. Olhe também para qualquer coisa que combine o modelo de loja de valores de negócio: gerenciamento de projeto, resposta de emergência, gerenciamento de campanha, entre outros.

Mesmo onde não exista uma demanda explicita para o estilo problema-solução de interação de usuário, naked objects podem ser uma abordagem efetiva para qualquer projeto de sistemas onde requisitos do sistema são incertos. (Em muitos projetos dessa natureza em que estivemos envolvidos, os responsáveis tiveram eventualmente que reconhecer que uma interface problema-solução era de fato uma das chaves de sucesso).

Nossa experiência sugere que projetos de sistemas de negócio de um modo geral se ajustam em duas categorias: a dominada pela engenharia e a dominada pelos requisitos. (Alguns se ajustam a ambas as categorias; são invariavelmente uns pesadelos, mas felizmente eles são raros). Aplicações dominadas pela engenharia (tais como sistemas autorização de uma transação de cartão de crédito) precisam de um método muito rigoroso. Mas muitas novas aplicações de negócio não fazem fortes exigências sobre a tecnologia. Ao invés disso, o desafio vem das mudanças de requisitos funcionais, e até do propósito do sistema. Os naked objects são particularmente bem satisfatórios para esses problemas.

A expressividade não se contrapõe à eficiência de negócio?

Se você tem uma tarefa absolutamente padronizada, então fornecer ao usuário várias formas de realizar essa tarefa será menos eficiente do que otimizar o sistema para uma melhor solução. Mas a razão de tarefas não-padrão para padrão é crescente, porque problemas que podem ser completamente padronizados estão ou sendo totalmente automatizadas, removendo o ser humano do laço, ou, através do uso de tecnologias auto-serviço, os quais estão sendo delegados aos usuários fora da organização. O exército de secretários que realizam grandes volumes de tarefas de ‘entrada de dados’ estão desaparecendo rapidamente.

Considere o exemplo de check-in num aeroporto. Quiosques de auto-check-in estão se tornando cada vez mais comuns. Passageiros recuperam suas reservas, verificam seus assentos ou mudam-nos para algum assento disponível, então imprimem seus cartões de embarque e as etiquetas auto-adesivas de bagagem quando precisarem. Para todos, exceto para aqueles que possuem necessidades

especiais, o quiosque economiza considerável tempo. Enquanto o uso dessas máquinas cresce, uma crescente proporção de passageiros entra na fila do balcão de check-in tradicional têm problemas não-padrões: excesso de bagagem, uma mudança de rota, ou amigos em diferentes reservas querendo se sentar próximos uns dos outros. Conseqüentemente, os sistemas usados pelo pessoal de check-in precisam ser projetados principalmente para solucionar problemas ao invés de otimizar transações rotineiras que estão progressivamente sendo realizados pelos quiosques de auto-serviços.

No DSFA, quando o sistema implementado foi submetido pela primeira vez aos testes de aceitação do usuário, um dos primeiros usuários relatou que ele podia processar um benefício para menores 'padrão' em apenas seis minutos, comparado aos vinte no velho sistema, e ele podia processar uma 'prorrogação 16+' (permanência de uma criança na educação pré-universitária) em um minuto ao invés de cinco. Esse foi um simples relato, não uma análise abrangente. Mas o que é interessante é que o sistema não foi explicitamente projetado para otimizar a manipulação de casos padrões!

É comum afirmar que os usuários avançados preferem atalhos de teclas a operações dirigidas por mouse. Existe alguma verdade nisso, mas precisa de um exame mais detalhado. Uma das grandes ironias na história da computação é que o inventor do mouse (Doug Englebart) não estava procurando facilitar o uso de computadores para usuários iniciantes, mas para fazer com que os computadores fossem mais expressivos para usuários avançados [Englebart1968]. O mouse foi apenas uma meia invenção. Do outro lado estendendo o teclado estava um periférico de 'corda' com cinco teclas de piano, nas quais o usuário podia pressioná-las simultaneamente em combinações memorizadas para chamar diferentes operações sobre o item apontado pelo mouse. De fato, o mouse (ou dispositivo indicador equivalente) é um mecanismo altamente eficiente por associar relacionamentos entre objetos. Mas os atalhos de teclado podem algumas vezes ser muito mais eficiente do que os menus pop-ups ativados por mouse. Isso é outra coisa que será fornecido numa futura atualização do framework Naked Objects.

Seu pessoal está usando esse estilo de sistema?

Ninguém quer mais responsabilidade ou maior liberdade de expressão. Alguns podem preferir uma posição confortável de não ter que pensar. Mas gerentes de negócio não devem tirar conclusões precipitadas sobre isso. Como um gerente de sistemas de um banco nos disse durante uma entrevista: 'Por anos nós dizemos aos usuários para não fazer qualquer coisa a menos que esteja expresso em termos de sete principais transações que o nosso sistema mainframe de banco pode realizar. Agora nós temos este sistema beneficiário-servidor, todos cantando e dançando com um uma interface de usuário enfeitada e adivinhe o que os usuários estão se queixando que eles não sabem o que fazer com eles. Nós tiramos deles toda a criatividade!'.

O conceito completo de naked objects indubitavelmente incorpora uma visão otimista das capacidades do usuário. Ele não é bom para tudo. Mas nós achamos que a filosofia oposta é terrível de considerar.

No final do trabalho, nós devemos reconhecer que qualquer transição para um estilo de trabalho mais poderoso exige maior educação, e pode ditar um período transitório onde exista maior guia e controle. Como dissemos anteriormente, se você projetar um sistema a partir dos naked objects, então é sempre possível construir uma camada de procedimentos roteirizados no topo desses objetos – de fato eles são muito fáceis de escrever. Mas se você projetar um sistema ao redor dos procedimentos roteirizados, você irá achar muito mais difícil tornar o sistema mais expressivo posteriormente.

Como podemos ainda implementar controles de negócio necessários?

Todos os exemplos contidos neste livro mostram alguma forma de controle de negócio na operação: você não pode criar um Pagamento exceto no contexto de um Esquema de Benefício que tenha sido finalizado por um Funcionário; você não pode violar o tempo mínimo de conexão entre Segmentos de Vôo em qualquer Aeroporto (ao menos não dentro de uma única Reserva); você não pode criar dois diferentes Ajustes de Preços sobre o mesmo Grupo de Produtos, a menos que eles sejam do tipo Acumulativo. Com os naked objects esses controles ou restrições devem ser implementados dentro dos objetos de negócio para os quais eles se aplicam. Por contraste, muitos sistemas de negócio reforçam controles no nível de interações roteirizadas. O reforço no nível de objetos de negócio não dá aos usuários maior liberdade de chamar objetos na ordem que melhor satisfaça o problema que eles estão resolvendo, mas na prática permite um forte nível de controle.

Esta abordagem não será bem-vinda por aqueles que defendem projetos de sistemas ‘baseados em regras’, onde o princípio é extrair todas as regras de negócio numa representação separada tal que as regras possam ser alteradas mais prontamente. Nós não gostamos dessa abordagem por duas razões. A primeira é que você pode terminar com um monte de regras terríveis (até mesmo para verificar a validade de uma data), com o resultado de encontrar todas as regras afetadas por uma mudança de negócio proposta pode verdadeiramente ser difícil. A segunda é que a abordagem baseada em regras parece encorajar aquelas cujas metas são de remover todos os direitos de decisões dos usuários.

Implementar uma regra ou restrição dentro do objeto ou relacionamento para a qual ela se aplica facilita encontrá-las quando precisarem ser alteradas. E se você pratica a disciplina de escrever código que seja fácil de ler ou alterar, então as regras serão fáceis de editar.

Como você contorna a necessidade de caixas de diálogos?

Projetar sistemas sem acessar caixas de diálogos é um tanto desafiador. Até mesmo mais extremo do que não permitir a separação de objetos entidades e controladores. Mas uma vez aceita essas restrições, se conduz a melhores modelos de objetos.

Considere o exemplo de um sistema bancário – um exemplo favorito e freqüente em livros textos de objetos. Uma Conta é claramente um objeto, e é natural pensar na Conta Corrente e Conta Poupança como subclasses especializadas. Uma análise use-case convencional sugere que todas as Contas precisarão de métodos para realizar o depósito, saque, declarações de criação e mudanças de aplicação, entre outros. Mas como o sistema deveria manipular uma transferência de fundos entre contas?

Muitas pessoas argumentam que esta funcionalidade pertence a algum tipo de processo, não a este ou aquele objeto Conta.

Se nós não permitimos a adição de roteiros ou controladores no topo dos objetos entidades, então uma outra opção é implementada TransferirPara()(Conta, ValorMonetário) como um método de ambas as classes Conta. Isso pode então gerar uma caixa de diálogo, o usuário pode especificar a Conta que receberá a transferência e o seu montante. Mas o framework Naked Objects não representam caixas de diálogos. Pode-se transformar um método sem parâmetros em uma ação sobre o menu pop-up para aquele objeto, e executar um método de um parâmetro arrastando um objeto para o outro. Mas ele não pode traduzir um método com múltiplos parâmetros numa ação do usuário.

A solução é implementar Transferência como uma classe de objeto com seus próprios direitos. Seus atributos são duas contas, a quantidade a transferir e a data/hora. Os métodos sobre os dois objetos de conta então se tornam efetivamente, ‘Criar Nova Transferência’, que cria uma nova instância de Transferência, prontamente povoada com a conta ‘De’. Alternativamente, o usuário pode cortar caminho arrastando uma conta sobre a outra, que retorna um novo objeto Transferência, ambos povoados com contas ‘De’ e ‘Para’. Após especificar a quantidade da transferência, o usuário então chama o método ‘Executar’ ou ‘Faça isso’ sobre o objeto Transferência.

Críticos irão dizer que tudo o que temos a fazer é criar um processo de Transferência, ou até uma caixa de diálogo de Transferência, e chamá-lo de um objeto, mas eles estão esquecendo de um ponto importante. Devido a cada transferência estar agora sendo modelada como um objeto instanciado, pode ser que ele seja diretamente referenciado no futuro, para propósitos de auditoria, ou mesmo revisão. De fato, para a nossa aplicação bancária, Saques e Depósitos podem da mesma forma, serem tratados como substantivo-objetos não como verbos. Não seria legítimo referenciar todos esses exemplos como objetos de ‘formulários’, onde os formulários fornecem um registro permanente dessas instâncias de um processo. Se você tratar cada um desses conceitos meramente como um verbo ou processo então você não poderá referenciá-los diretamente,

digamos um particular saque, mas deve reconstruí-lo, conseqüentemente, a partir da trilha de auditoria.

Muitos bons modeladores teriam visto esta solução imediatamente. Para o restante – o Naked Objects não irá apresentar automaticamente a melhor solução e dificultará consideravelmente a implementação de algumas pobres soluções.

Como o Naked Objects atende aos requisitos de múltiplas visões?

Quando se usa o Naked Objects, todas as visões de usuários são geradas automaticamente a partir das definições de objetos de negócio; mas é um requisito comum de um sistema fornecer múltiplas visões do mesmo objeto de negócio. Como o Naked Objects atende a esse requisito?

Você precisa distinguir dois tipos diferentes de visões alternativas. O primeiro pode ser chamado de múltiplas representações: nós queremos ver uma coleção de objetos como uma lista, ou como uma tabela, ou talvez como um grafo. O Mecanismo de Visualização, parte do framework Naked Objects, já fornece várias visões alternativas genéricas de objetos. A visão default é um pequeno ícone. A visão 'aberta' lista os atributos e mostra todos os objetos associados como ícones ativos. Alternativamente, um objeto pode ser retratado como texto plano, ou, se ele implementar o 'hasImage' (tem imagem), ele pode ser apresentado como uma imagem. Coleções de objetos podem ser mostradas como um único ícone ou uma lista. Se os objetos da lista são todos do mesmo tipo, então a coleção pode ser vista como uma tabela, ou um layout de imagens quadriculadas. Se os objetos numa coleção implementar a interface 'mappable' (que pode ser mapeado), então eles podem ser alocados sobre um mapa geográfico ou esquemático bidimensional. Todas essas visões genéricas são automaticamente fornecidas, sem qualquer programação explícita, e usuários podem selecionar qualquer um deles. Esta capacidade é conceitualmente o mesmo que a capacidade do Windows (ou equivalente) para visualizar o conteúdo de um folder como ícones, uma lista sumarizada, thumbnails (foto em forma de slides) ou detalhada. Nós esperamos que uma grande gama de visões genéricas seja adicionada com o tempo – por exemplo, a habilidade de visualizar qualquer vetor de números como um gráfico, como numa aplicação de planilha. Programadores capacitados podem estender o framework para adicionar novos tipos genéricos de visões.

O segundo conceito pode ser chamado de visões seletivas: fornecer ao usuário uma visão parcial de um objeto, mostrando apenas certos atributos, associações e métodos. Isso é necessário porque certos usuários não estão autorizados a ver outras informações, ou simplesmente com a finalidade de reduzir os elementos na tela.

O framework Naked Objects permite que você personalize as visões através do uso de objetos About (sobre) – você pode especificar quais atributos, associações e métodos serão disponibilizados para um dado usuário ou papel. O mesmo mecanismo pode ser usado para controlar a visão, ou comportamento, de um

objeto de acordo com o seu estado (onde este estado é ou representado explicitamente, ou derivado de outros atributos e associações).

Mas, em geral, a idéia de projetar sistemas a partir de naked objects é evitar tanto quanto a contextualização. O objetivo é dar ao usuário uma visão potencialmente rica de objetos quando possível, a menos que a tela fique muito confusa. É este o contraste marcante na abordagem usual de projetar sistemas, a qual dá ao usuário o mínimo necessário de informação para completar uma particular tarefa, limitando a oportunidade dos usuários aprenderem a enxergar todo o contexto.

Um efeito colateral positivo dos naked objects é que eles ajudam a preservar a integridade do modelo do objeto. Por exemplo, um sistema convencionalmente projetado pode exibir num pedido, o nome e o endereço do beneficiário, o qual teria sido extraído do objeto Beneficiário. Enquanto que isso não transgride a regra de encapsulamento no sentido técnico, é certamente danoso para o entendimento do modelo do objeto do usuário. Usando naked objects, o Pedido não exibe o nome e o endereço do beneficiário como campos separados: ele irá exibir um ícone representando o Beneficiário dentro do Pedido. Para verificar, ou alterar o endereço do beneficiário, o usuário pode abrir o objeto beneficiário e fazer a alteração ali mesmo.

Como você evita que objetos compartilhados fiquem 'inchados?

Se a única maneira do usuário chamar uma ação é pela seleção de um método sobre um objeto de negócio fundamental, e se objetos são compartilhados por diferentes usuários ou departamentos, então tais objetos de negócio fundamentais não se tornarão rapidamente inchados com métodos, atributos e associações?

Um objeto com muitos métodos, atributos e associações cheira como um programa monolítico – sugerindo que a agilidade seja reduzida. A resposta, em muitos casos é usar a técnica de agregação: quebre o objeto em componentes naturais, mesmo se esses componentes nunca sejam acessados fora do contexto de seus objetos pais, e agregue tais componentes. Por exemplo, o objeto Beneficiário pode ter componentes que modelam a informação de Identidade, histórico Médico, entre outros. Isso também ajuda no desempenho. Um objeto com muitos atributos leva muito tempo para ser carregado na memória a partir do armazenamento secundário. Usando agregação, o objeto pode escolher carregar apenas ponteiros para seus objetos componentes, e carregar seus atributos somente se um objeto componente estiver sendo visualizado pelo usuário, ou chamado por um método sobre o objeto principal.

O Naked Objects pode funcionar dentro de uma arquitetura 'cliente magro'?

Mais e mais organizações afirmam ter adotado uma política 'cliente magro', mas o significado desse termo varia consideravelmente. Para algumas pessoas ele significa que toda a aplicação deve executar dentro de um browser; mas então

alguns grandes plug-ins são aceitos, enquanto para outros isso não é aceitável; alguns até têm a visão de que o cliente deve executar em puro HTML, sem nenhum código Java. Alternativamente, cliente magro pode não significar operação dentro de um browser, mas dentro de uma outra arquitetura ou restrições de hardware.

O framework pode funcionar dentro de qualquer uma dessas restrições – ainda que isso possa envolver a escrita de um mecanismo alternativo de visualização genérica para a sua arquitetura preferida de cliente. Por exemplo, uma implementação pura em HTML pode retratar cada objeto como uma página web, com métodos de ação exibida como botões. A ação de arrastar ou soltar pode não ser possível, mas apontar e clicar poderia. Múltiplas janelas, incluindo uma visão de árvore de objetos recentemente visitados, poderiam permitir a realização de das mesmas operações. Mas não daria a sensação tão expressiva ao usuário.

No entanto, acreditamos que a idéia de 'cliente magro' (ou, ao menos, em muitas interpretações dessa idéia) precisa ser contestada. Quando proposta inicialmente, a idéia de executar todas as coisas dentro de um browser tem considerável apelo, devido a eliminar a necessidades de manter o software diretamente sobre máquinas clientes. Mas existem agora muitos meios técnicos de alcançar esta meta. Uma outra motivação foi fornecer um ambiente de interface de usuário padronizado, mas, como nós temos visto, existem poucas padronizações no modelo de interação de usuário oferecido pelas diferentes aplicações baseadas em browser. Mais ainda, a interface HTML básica não é uma boa interface para realizar alguns tipos de atividades problema-solução. O browser tinha a intenção de apenas publicar informações. Capacidades transacionais tiveram que gradualmente ser adicionados, mas eles são algo como um "quebra galho" em termos de projeto de interface de usuário. A terceira motivação para a padronização sobre uma interface de browser é para acesso do consumidor. No entanto, os naked objects são melhores adaptados para aplicações de uso freqüente, os quais antes de tudo pensam nos usuários domésticos.

Os Naked Objects permitem que usuários desenvolvam suas próprias aplicações?

Muitas pessoas apontam que enquanto naked objects aprimoram a expressividade do sistema para usuários finais, o framework Naked Objects não faz o mesmo para o desenvolvedor. Por que fazer o desenvolvedor escrever o código num editor de texto, quando muito do desenvolvimento inicial poderia também ser realizado com técnicas tais como arrastar e soltar?

Por muitas razões. A primeira é que nós estávamos ansiosos para não deixar a idéia de desenvolver expressividade confusa para o usuário. Ferramentas tais como Visual Basic já oferecem uma grande expressividade aos desenvolvedores, mas interfaces de usuário que resultam dessas ferramentas, não são normalmente expressivas para o usuário final. Por deliberadamente contrastar o ambiente do desenvolvedor e ambiente do usuário final, nosso objetivo foi manter o

desenvolvedor focado em criar sistemas expressivos para seus usuários, não para eles mesmos.

Segundo, não estava claro que representações gráficas e gestos de arrastar e soltar eram particularmente relevantes aos desenvolvedores. Certamente, eles são relevantes no projeto de layouts de telas, mas a idéia completa de Naked Objects é que você não projeta layouts de tela – eles são autogerados pelo sistema. Representações gráficas podem também ajudar os desenvolvedores a navegar através do código; mas igualmente fazem o estado da arte dos ambientes de desenvolvimento integrados tais como VisualAge, Eclipse ou TogetherJ, os quais podem todos ser usados em conjunto com o Naked Objects para desenvolver aplicações. Tudo isso nos levou a escrever o código funcional numa linguagem de programação tal como Java. Nesse estágio, as representações gráficas adicionam pouco valor – os benefícios dos desenvolvedores na maior parte deste estágio são as ferramentas avançadas de apoio ao código tais como as caixas de teste automatizadas e navegadores de refatoração.

Nós pensamos que o empreendimento de tornar a programação uma operação totalmente gráfica seria desviar dos objetivos. Seria voltado para que usuários finais pudessem programar; mas é isso que nós precisamos? Os naked objects foram concebidos para apoiar a solução de problemas pelos usuários finais, não para a programação pelos usuários finais. Resolver problemas significa encontrar uma solução para um contexto específico único. A programação envolve encontrar uma solução abstrata para uma classe completa de problemas. Existem habilidades diferentes. Uma planilha é uma ferramenta extraordinariamente poderosa para solucionar problemas individuais e é simples de usar, mas escrever um modelo de planilha que seja suficientemente robusta para outros usarem requer um grau elevado e diferenciado de disciplina. É isso o que nós queremos que os usuários finais façam?

Estudo de Casos: Reserva de Viagem

O ECS (*Executive Car Services* – Serviços Carros Executivos) é uma grande empresa de limusine com uma central de chamadas servindo operações em múltiplas cidades. (ECS é uma empresa fictícia, mas este caso é baseado em experiências reais). Embora a empresa goze de excelente reputação no serviço ao cliente, a abordagem da central de chamadas padrões de tentar cobrir todas as interações do cliente usando roteiros otimizados e padrões pode, algumas vezes, causar consideráveis frustrações – tanto do cliente quanto do agente de reservas – como revela o seguinte exemplo:

- **Posso ajudá-lo?** Eu gostaria de um carro para sair do centro da cidade de Manhattan para o aeroporto JFL no início desta tarde, por favor.
- **Qual é o seu nome?** Richard.
- **E o seu sobrenome?** Pawson.
- **Eu vejo que você já tem um registro anterior. Você ainda trabalha no CSC?** Sim.
- **Nesse caso eu preciso do número do projeto para a emissão da fatura.** Eu não estou com esse número no momento, posso pagar com cartão de crédito?
- **Certamente. Tipo do cartão de Crédito?** American Express.
- **Número do cartão?** 3791 xxxxxx xxxxx.
- **Data de vencimento?** 01.04
- **Nome como impresso no cartão?** Richard W Pawson.
- **Data da viagem?** Hoje.
- **Hora?** 2:30 da tarde.
- **Localização?** É o edifício Pfizer da East 42nd St – Eu não sei o número.
- **Eu posso ver para você, … um momento, … sim, é 234 da East 42nd St.**
- **E para onde você está indo?** Para o aeroporto JFK.
- **Em qual companhia aérea?** British Airways.
- **Qual é o número de vôo?** Eu não sei o número do vôo, é aquele que sai de Londres às 18:00hs.
- **Eu posso também verificar, … um momento, … sim, é o BA218.**
- **E finalmente, eu preciso de um contato telefônico.** Eu estou no Grand Hyatt na 42nd e Park Avenue. Eu não sei o número.
- **Eu posso verificar, ... sim, é 212 XXX XXXX. E qual é o seu apartamento?** 1634.
- **Obrigado, é tudo o que eu precisava. Só um momento por favor, ..., Desculpe-me senhor, parece que nós não temos nenhum carro disponível!**

O ponto é que o sistema deve estar apto a capturar a informação fornecida na primeira sentença e certificar se um carro está disponível primeiro – e só então perguntar todas as outras informações. Não é suficiente colocar um novo passo dentro do roteiro para verificar a disponibilidade o mais cedo quanto possível na

interação. Ao invés disso, o operador deveria estar apto a manipular cada chamada de maneira que satisfaça o cliente ou o particular problema que tem em mãos.

A tela exibida abaixo é de um simples protótipo que nós produzimos para demonstrar como projetar um sistema de reserva utilizando os naked objects para tratar desse assunto.

O protótipo usa apenas seis classes de objetos de negócio fundamentais e eles são brevemente detalhados.

As seis classes usadas no protótipo do ECS

A Reserva (*Booking*) conhece os detalhes de uma reserva do cliente. Sabe como verificar a disponibilidade e confirmar.

O Cliente (*Customer*) conhece os detalhes de identificação, e instâncias dos objetos freqüentemente usadas pelo cliente (por exemplo, Localizações, Telefones). Sabe como comunicar com o cliente.

A Cidade (City) conhece as Localizações normalmente usadas daquela cidade. Sabe como consultar as condições do tempo e de tráfico.

A Localização (*Location*) conhece o endereço da rua e uma abreviação (por exemplo, 'matriz'). Sabe como obter a direção.

O Cartão de Crédito (*Credit Card*) conhece os detalhes do cartão. Sabe como esconder o número, e como obter autorização para um dado valor. (Outros Métodos de Pagamento tal como Conta Empresa será adicionada posteriormente).

O Telefone (*Telephone*) conhece o número e a abreviação (por exemplo 'Meu escritório'). Sabe como fazer a chamada. (Objetos Email, Fax serão adicionados posteriormente, todos compartilhando uma interface comum).

A partir desse simples protótipo podemos ilustrar três dos quatro benefícios dos naked objects:

- Os naked objects dão maior poder ao usuário.
- Os naked objects podem acomodar melhor futuras mudanças dos requisitos de negócio.
- Os naked objects melhoram a comunicação entre desenvolvedores e usuários.

Primeiro, nós damos maior poder ao usuário. O protótipo permite ao agente construir uma reserva de várias formas, de acordo com o contexto particular ou preferências do cliente. Aqui existem apenas três cenários possíveis:

- Clicar com o botão direito do mouse sobre a classe Bookings (Reservas) e elecionar *New Instance* (nova instância). O método *Check Availability* (Verificar Disponibilidade) pode ser chamado em algum ponto do menu pop-up. Se todos os detalhes necessários forem fornecidos, este método irá retornar uma reposta sim ou não. Se não, ele irá dar uma indicação de

disponibilidade durante o dia. O objeto Booking pode também tem métodos para estimar o tempo da viagem e ajustar o melhor tempo apropriadamente, ou organizar melhor um vôo. (E devido ao Vôo também ser um objeto, ele pode ser facilmente projetado para monitorar a previsão de chagada a partir do website apropriado da companhia aérea).

- Vá para a classe *Customers* (Clientes) e recupere o objeto representando este cliente. Expanda a lista de recentes reservas do cliente e 'clone' um desejado, preenchendo apenas a nova data e hora. Alternativamente, arraste o *Customer* sobre a classe *Bookings* como um atalho – isso irá pré-popular a reserva com os detalhes do cliente.
- Pressione com o botão direito do mouse com o ponteiro sobre *City* (Cidade) onde a jornada começará, o qual oferece a você a opção de verificar a disponibilidade atual, e então a nova previsão do tempo e de tráfico (que o cliente pode querer saber a fim de decidir a hora da viagem). A Cidade pode então ser usada para exibir as localizações mais freqüentes reservadas por todos os clientes dentro da cidade, incluindo generalizações tais como 'centro da cidade' que pode subseqüentemente ser mais específico.

Segundo, o protótipo exibe como ele facilita acomodar futuras mudanças de negócio. Desde que a empresa freqüentemente introduz as reservas de carros nos extremos de um vôo, ele poderia no futuro também fazer reservas de vôos? Por uma taxa extra, por que não adicionar um serviço de porteiro para reserva de teatros ou jantares? Tudo isso pode ser associado à Reserva quando necessário. Ou, como um outro exemplo, atualmente os dois métodos de pagamento são cartão de crédito e conta empresa. Apenas um deles foi implementado no protótipo, mas ambos são subtipos de PaymentMethod. Futuros subtipos podem ser facilmente adicionados para habilitar, por exemplo, um serviço de vale presente.

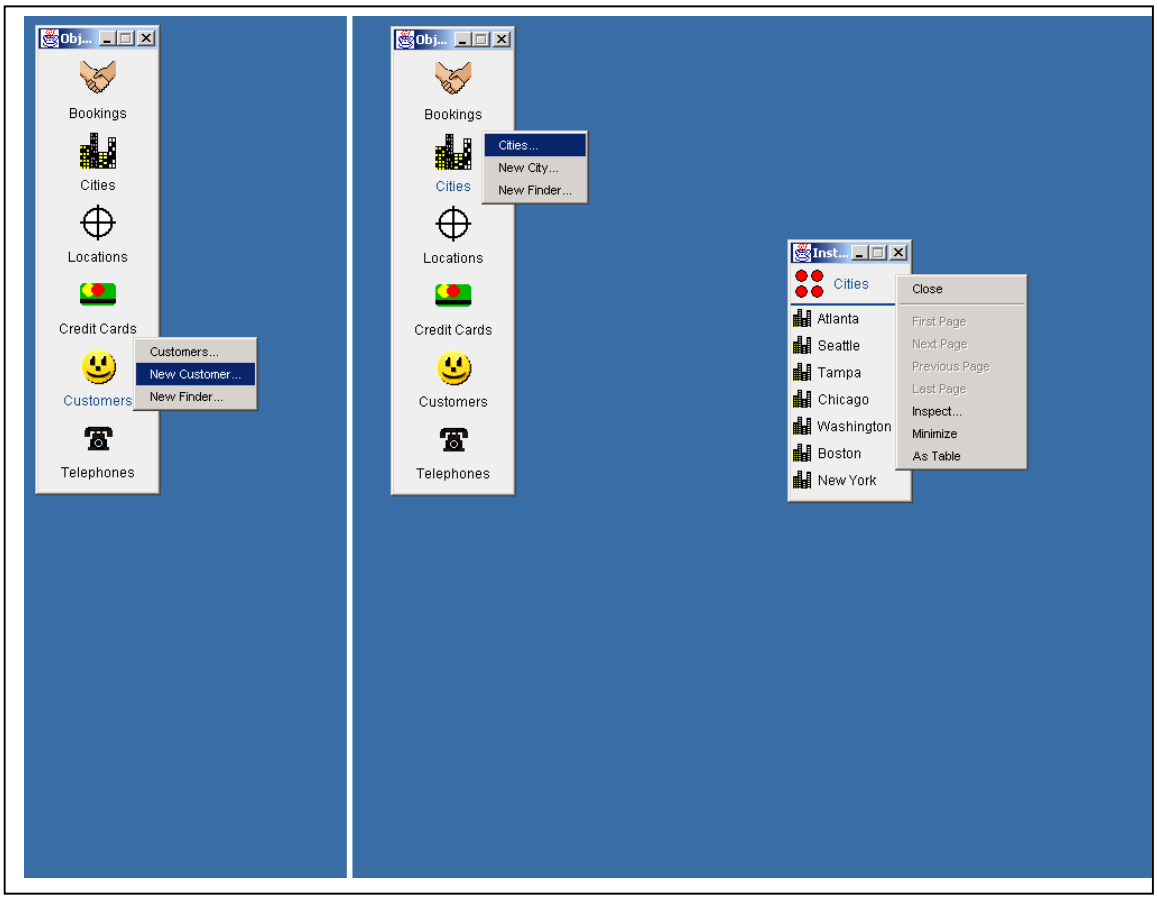
O terceiro grande benefício, de fornecer uma linguagem comum entre desenvolvedor e usuário, será ilustrado em algumas das próximas páginas.

Visões do sistema do usuário e desenvolvedor

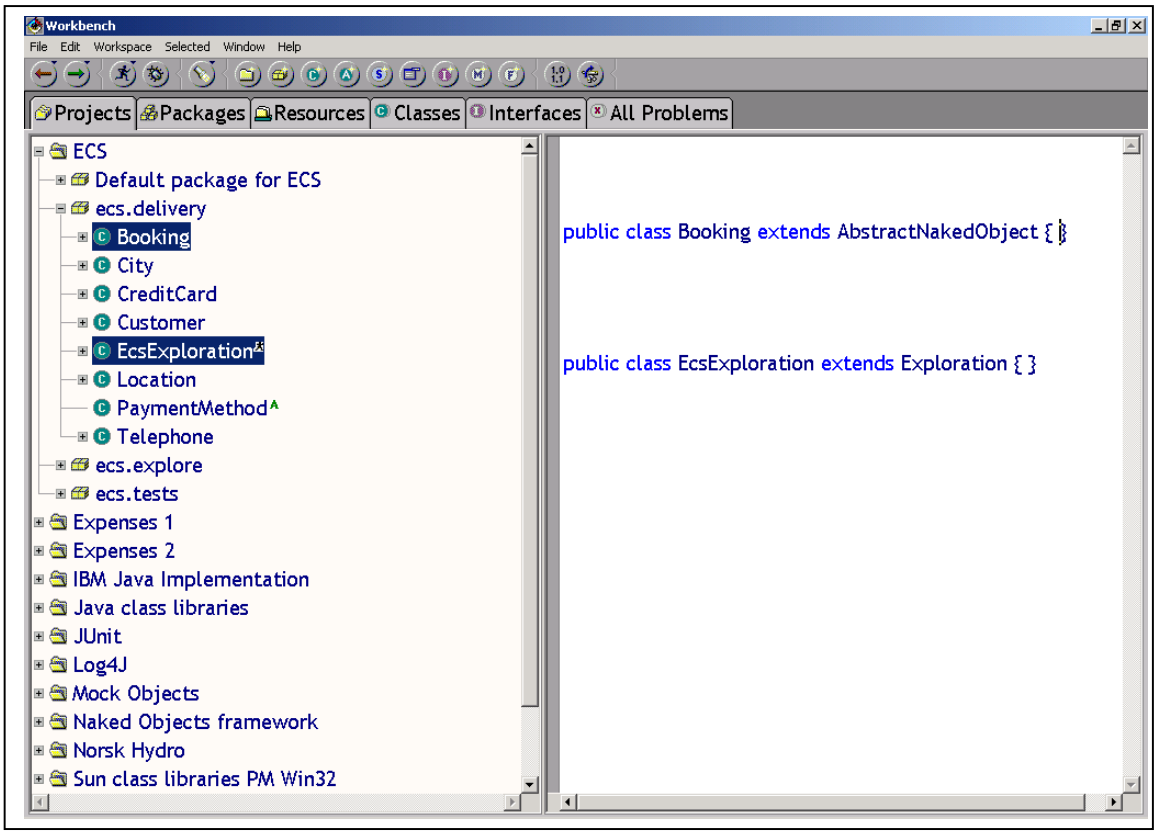
A seguir segue-se uma série de visões ilustrando diferentes aspectos do sistema da perspectiva de usuários e desenvolvedores. Não é um tutorial de programação. Ao invés disso, é uma tentativa de mostrar a próxima correspondência que existe entre as duas visões de um sistema de negócio desenvolvido com o framework Naked Objects. A partir dessas telas é possível obter uma sensação adequada de quão facilmente uma solicitação do usuário pode ser transformada em mudanças necessárias no código.

Classes

Uma aplicação ‘consiste’ de nada mais do que um conjunto de classes de objetos de negócio. Todas as operações de usuário tomam a forma de ações chamadas a partir de uma instância de uma dessas classes ou a partir das próprias classes. Toda classe que possa formar o ponto de início de uma atividade do usuário é mostrada na janela ‘Classes’ do usuário, exibida aqui no canto esquerdo da tela.



1. Existem seis classes de negócio que constituem o protótipo do sistema de reserva.
2. Ao pressionar o botão direito do mouse sobre qualquer ícone de classe surge um menu pop-up com os métodos de classe, incluindo um para criar uma nova instância dessa classe. Os métodos de classes genéricos exibidos aqui não requerem nenhuma programação desde que eles são automaticamente fornecidos pelo framework.
3. Outros métodos genéricos de classes permitem que o usuário encontre instâncias particulares ou exiba todas as instâncias dessa classe.
4. Este símbolo representa uma coleção de objetos: neste caso, todas as sete cidades do sistema.
5. Onde existirem mais instâncias do que puderem ser mostradas na janela, as coleções deveriam fornecer métodos (aqui cinzentos) para paginar os objetos.

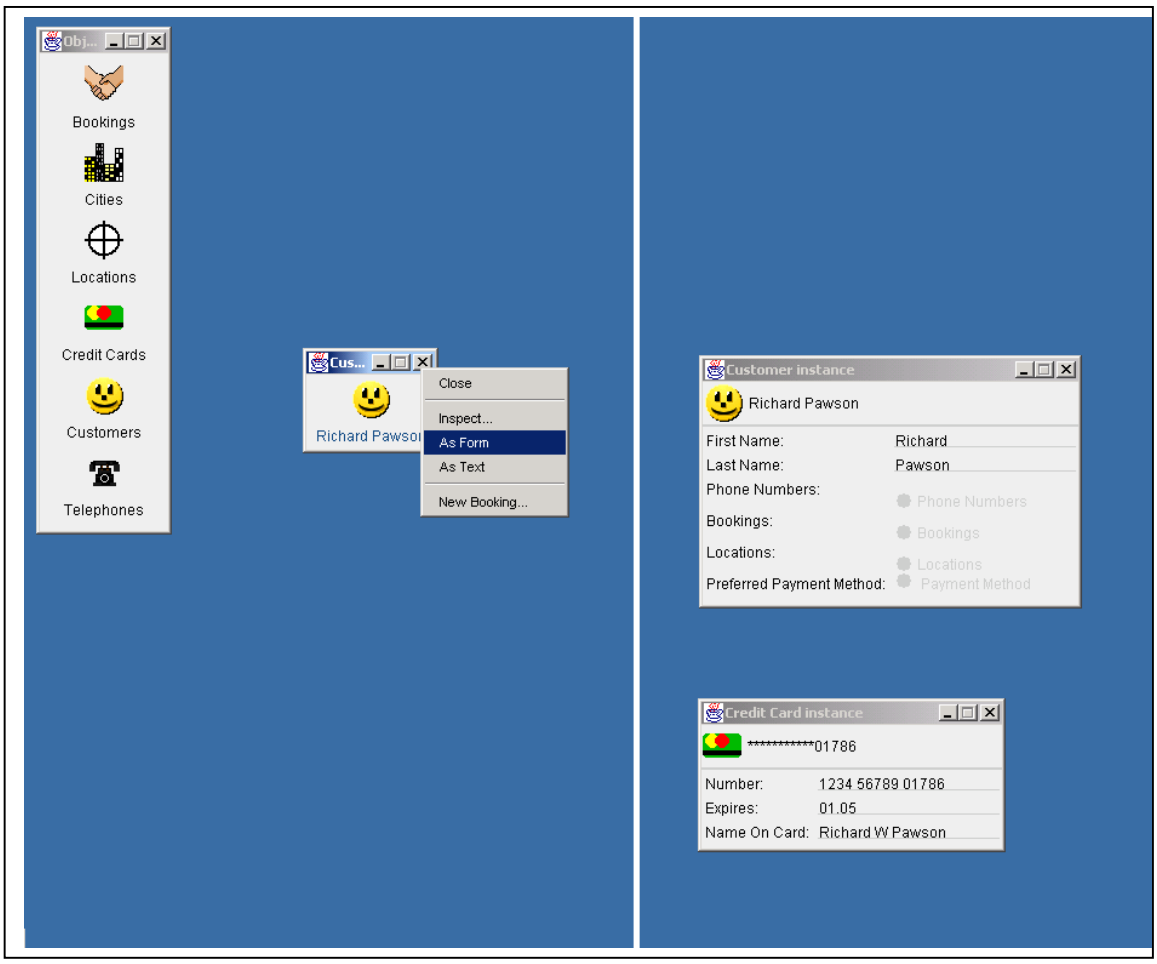


6. Os códigos desta aplicação são organizados num folder de projeto chamado esc.delivery.
7. Existe uma classe Java correspondente a cada uma das classes de objetos de negócio apresentado ao usuário. O rótulo sobre o ícone da classe de usuário (por default) automaticamente é derivado do nome da classe Java. Aqui, `CreditCard` foi convertido em *Credit Cards*. (Plurais irregulares devem ser especificados manualmente).
8. Cada classe de objeto de negócio deve implementar a interface `NakedObject` a fim utilizar o mecanismo de visualização e/ou de persistência. A maneira mais fácil de alcançar isso é fazer que cada classe de negócio herde da classe `AbstractNakedObject` fornecida com o framework.
9. O único código deste projeto que não é uma classe de negócio é `EcsExploration`, a qual é necessária para executar a aplicação. Tudo que essa classe precisa é fornecida pela classe `Exploration` do framework. Quando a aplicação é liberada o `EcsExploration` será traçada por um arquivo de configuração simples.

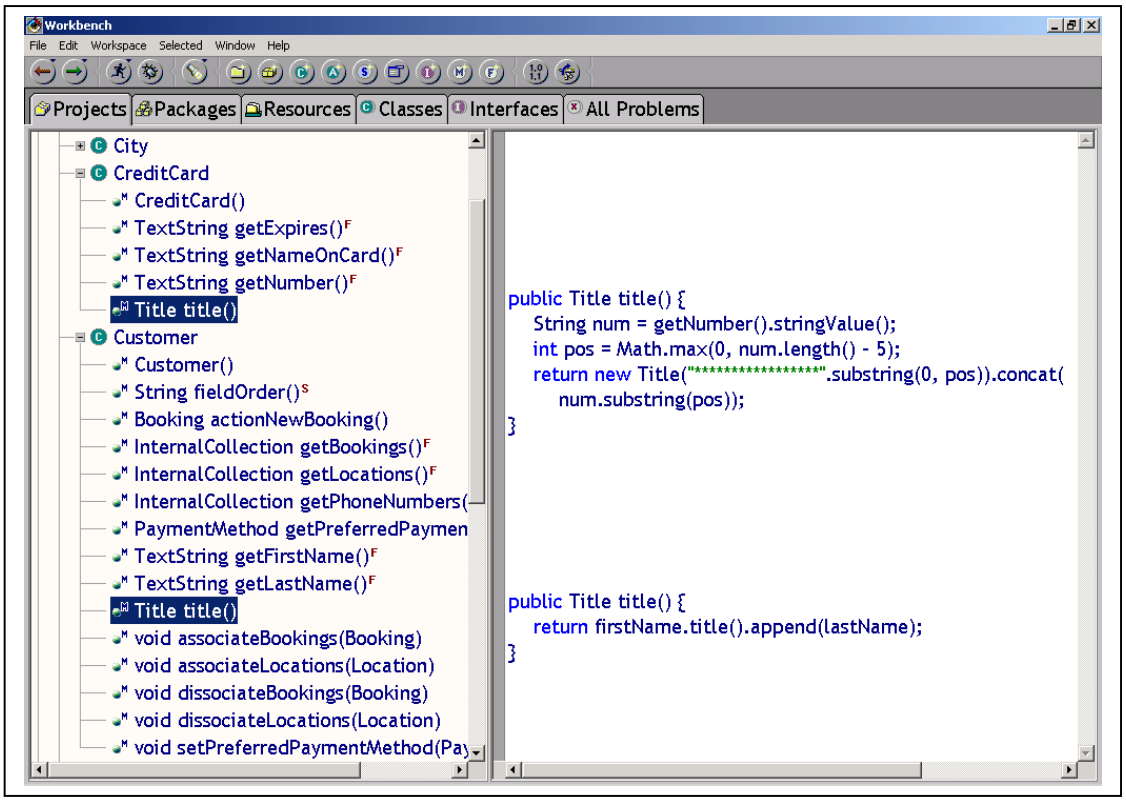
Instâncias

Para muitos cenários de negócio, o usuário do sistema lidará com instâncias individuais das classes de negócio, e algumas vezes com coleções de instâncias de mesmo tipo. Por default, uma instância usa o mesmo ícone usado pela sua

classe, mas tem um título individual. É possível também variar o ícone de acordo com a identidade ou o estado do objeto.



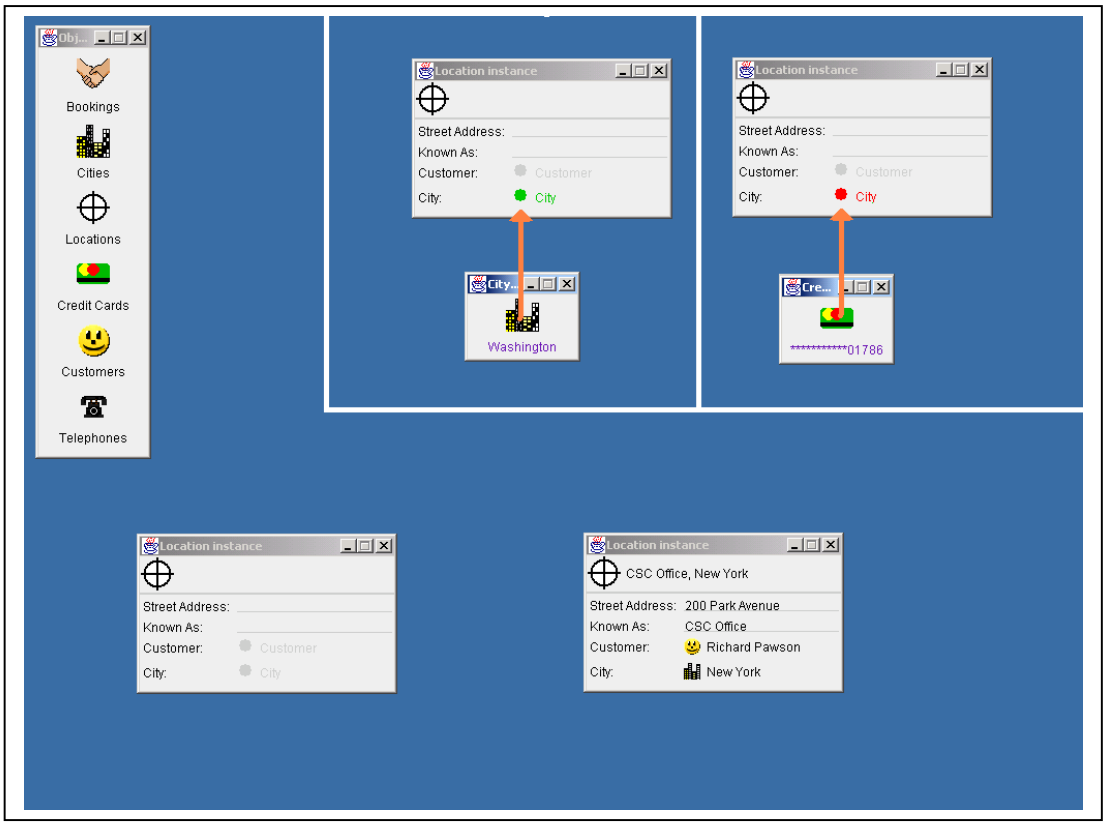
1. Por default, instâncias usam ícones de suas classes, mas instâncias têm títulos para identificá-los. Esta instância é exibida na sua visão de ícone.
2. As ações de menu exibidas aqui fornecem maneiras diferentes de ver o objeto. As opções de visualização oferecidas irão depender do tipo de objeto. Todas as opções de visualização são criadas automaticamente pelo framework.
3. A visão default é a visão de ‘Formulário’ exibida aqui.
4. Instâncias possuem títulos que são normalmente derivados da um ou mais atributos de identificação e/ou seu estado corrente.



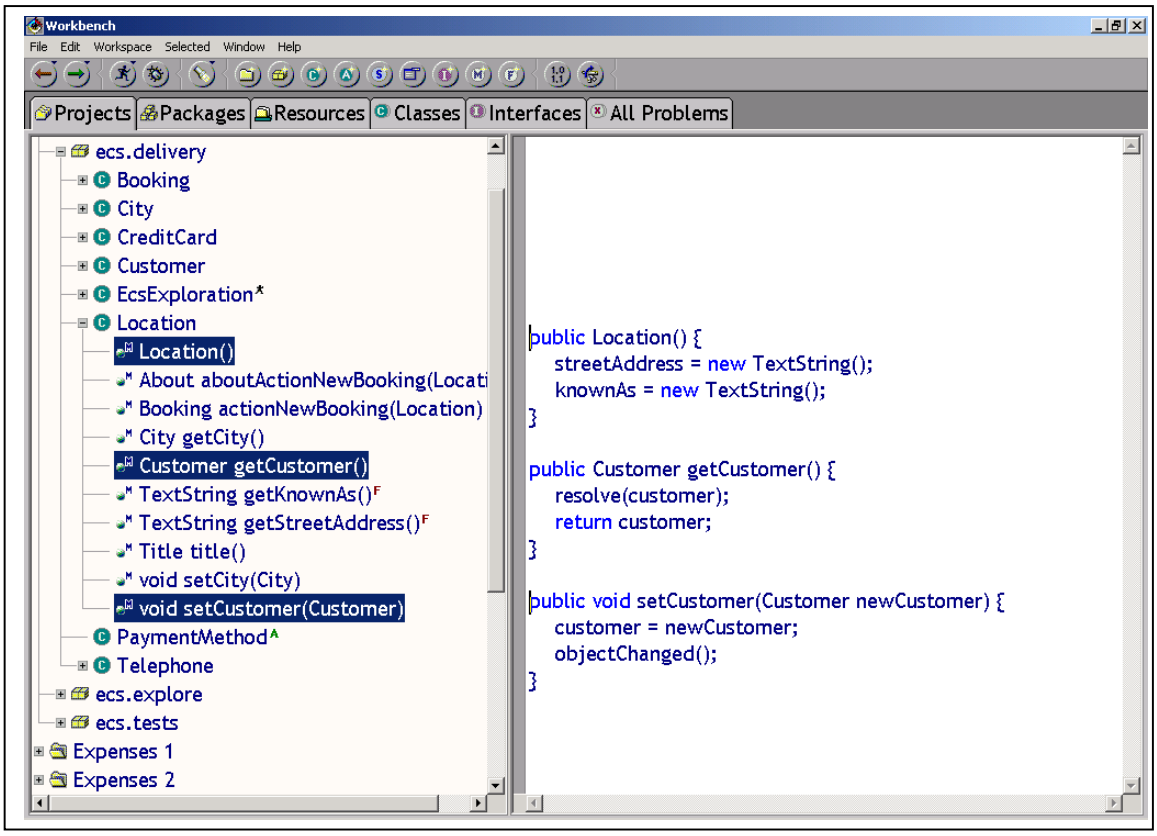
5. Cada classe é definida pelo conjunto de métodos que ele pode executar. Todos esses métodos liberam valores específicos de negócio, e resultam diretamente de um requisito do usuário. Os principais métodos técnicos necessários para gerenciar os objetos podem ser herdados do framework e nunca precisam ser vistos pelos desenvolvedores da aplicação.
6. Cada classe de negócio precisa de um método `title` para gerar o título que aparece próximo ao ícone. Isso normalmente é derivado de um ou mais atributos tais como nome, status, ou número de referência.
7. O método `title` deve retornar um `Title`. Aqui o `firstName` é transformado em um objeto `Title` (usando um método herdado), então o `lastName` é adicionado. O método `append` cuida de espaçar linhas e espaçar automaticamente.
8. O `CreditCard` tem um método `title` mais complexo que mascara tudo, menos os cinco últimos dígitos do número.

Campos

Abra uma visão de qualquer objeto e você verá um conjunto de campos. Alguns desses campos contêm valores simples (tais como datas, números ou cadeias de textos), que o usuário pode entrar ou editar. Outros campos irão conter objetos de negócio, exibindo ícones. Mesmo onde um ícone apareça dentro de uma outra visão do objeto, este ícone ainda representa um objeto totalmente funcional: você pode chamar seus comportamentos a partir daí ou abrir uma nova visão para ele.



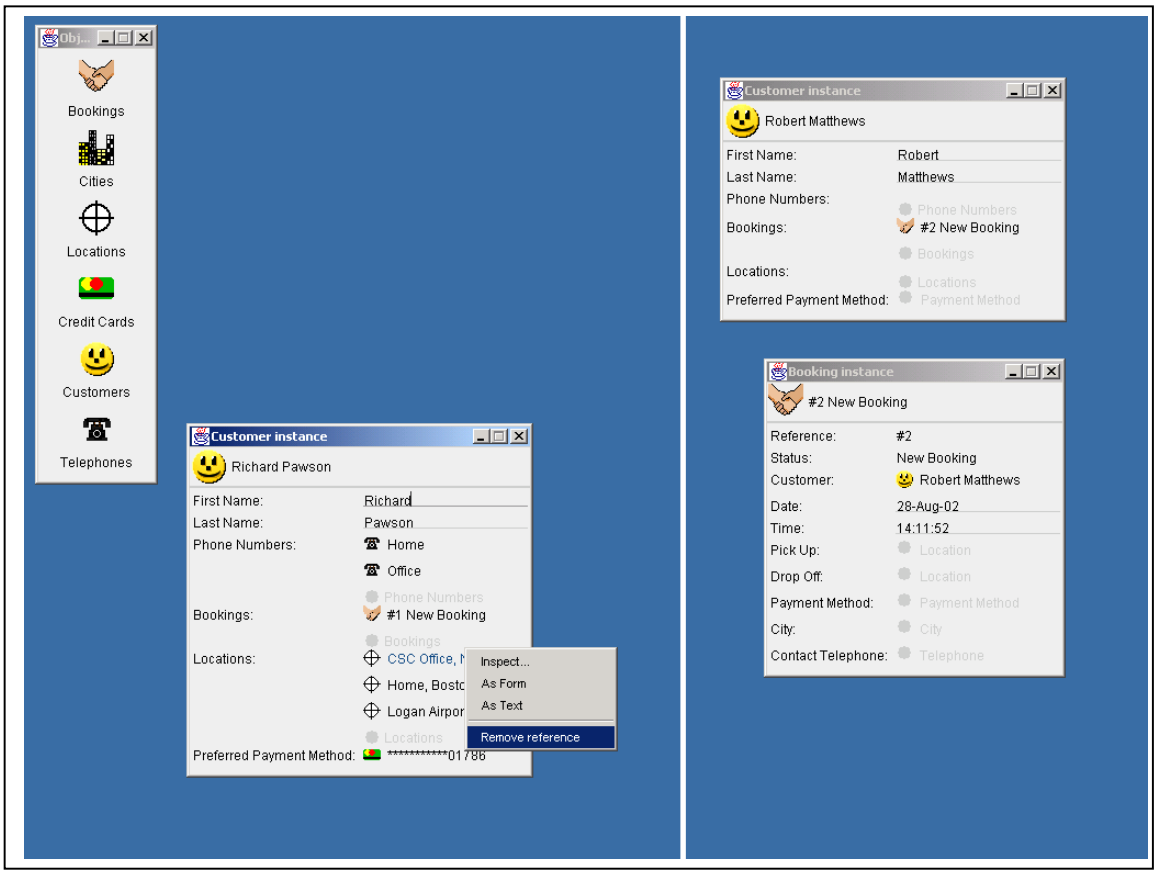
1. Um campo vazio ‘value’.
2. A bolinha verde indica que uma Cidade pode ser colocado aqui.
3. Este campo contém um outro objeto de negócio – uma Cidade.
4. Aqui o usuário teclou algum texto dentro do campo value.
5. Se você tentar arrastar o tipo de objeto correto dentro de um campo (aqui uma Cidade) a zona ficará verde. Um vermelho indica que o framework não deixará você soltar esse objeto aqui, ou porque o seu tipo é incorreto ou devido a alguma outra regra específica do programador.



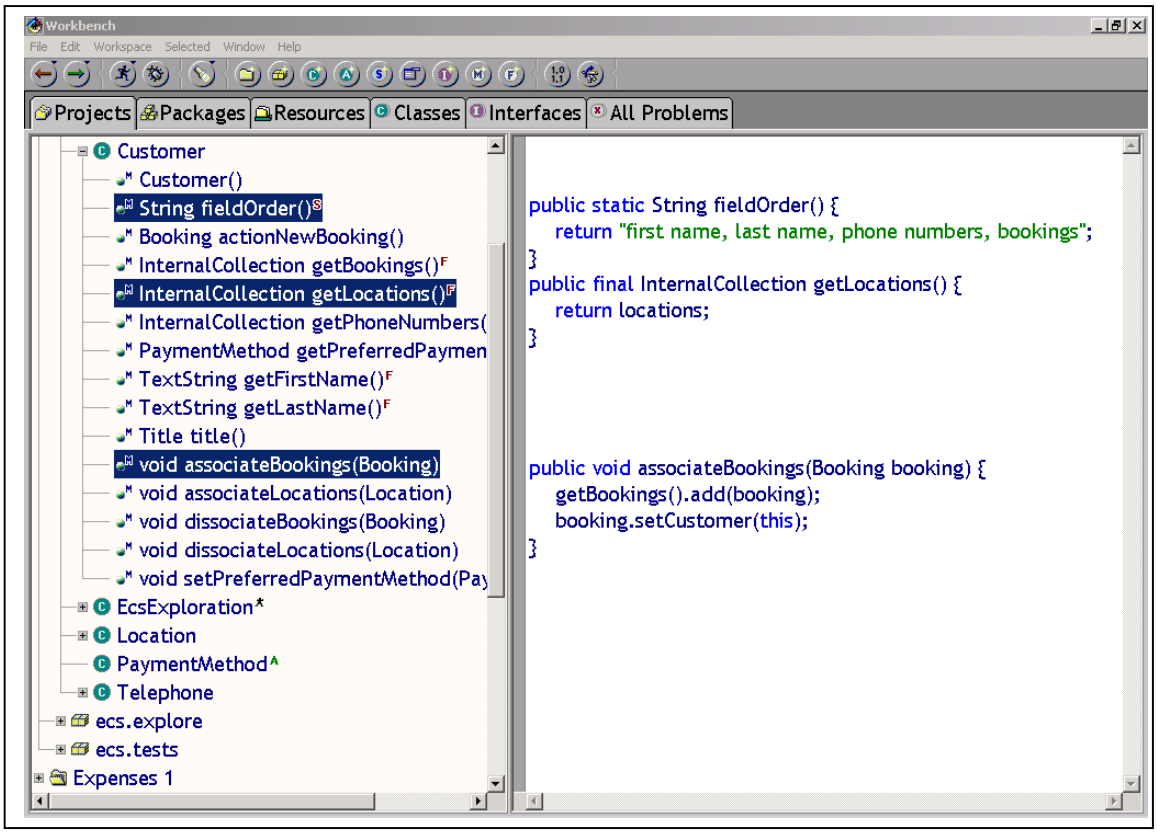
6. Cada campo na visão do usuário é determinado pela existência de um método `get` na definição da classe Java. O nome do campo é, por default, uma versão formatada do nome do método.
7. Um campo do objeto de negócio precisará de um método `get` e um correspondente método `set`.
8. Devido ao `setCustomer` precisar de um `Customer` como parâmetro no arquivo Java, o framework irá automaticamente fornecer a habilidade para o usuário soltar um Cliente sobre o campo correspondente campo na visão do usuário, mas irá desabilitar objetos de outros tipos exibindo o campo em vermelho.
9. O framework fornece um conjunto de classes `NakedValue` genérico incluindo `TextString`, `WholeNumber`, `Date` e `Money`. Tais campos requerem apenas um método `get` porque o valor do objeto por si só fornece os métodos para mudar seu conteúdo.
10. Os objetos `NakedValue` são normalmente iniciados dentro do método construtor para o objeto de negócio nos quais eles estão contidos.
11. O método `resolve` assegura que um objeto referenciado somente é recuperado a partir do armazenamento secundário quando se torna necessário.
12. O método `objectChanged` informa qualquer coisa que este objeto use, tal como o mecanismo de visualização ou de persistência que o status do objeto foi alterado.

Associações

Nós acabamos de ver como um campo pode conter uma referência para outros objetos de negócio. O Naked Object pode também manipular relacionamentos mais complexos tal como múltiplas associações, onde um objeto conhece instâncias múltiplas de um outro tipo, e associações bidirecionais, onde ambos os objetos conhecem um ao outro.



1. O *Cliente* pode ter múltiplas *Localizações* associadas.
2. Uma nova Localização pode ser associada soltando-a sobre a bolinha verde abaixo da coleção.
3. Clicar com o botão direito do mouse sobre qualquer membro dessa coleção oferece a opção de Remover a Referência.
4. Este é um exemplo de uma associação bidirecional. Quando um Cliente está associado com uma Reserva, uma referência para esta Reserva é automaticamente adicionada à lista de reservas do cliente.

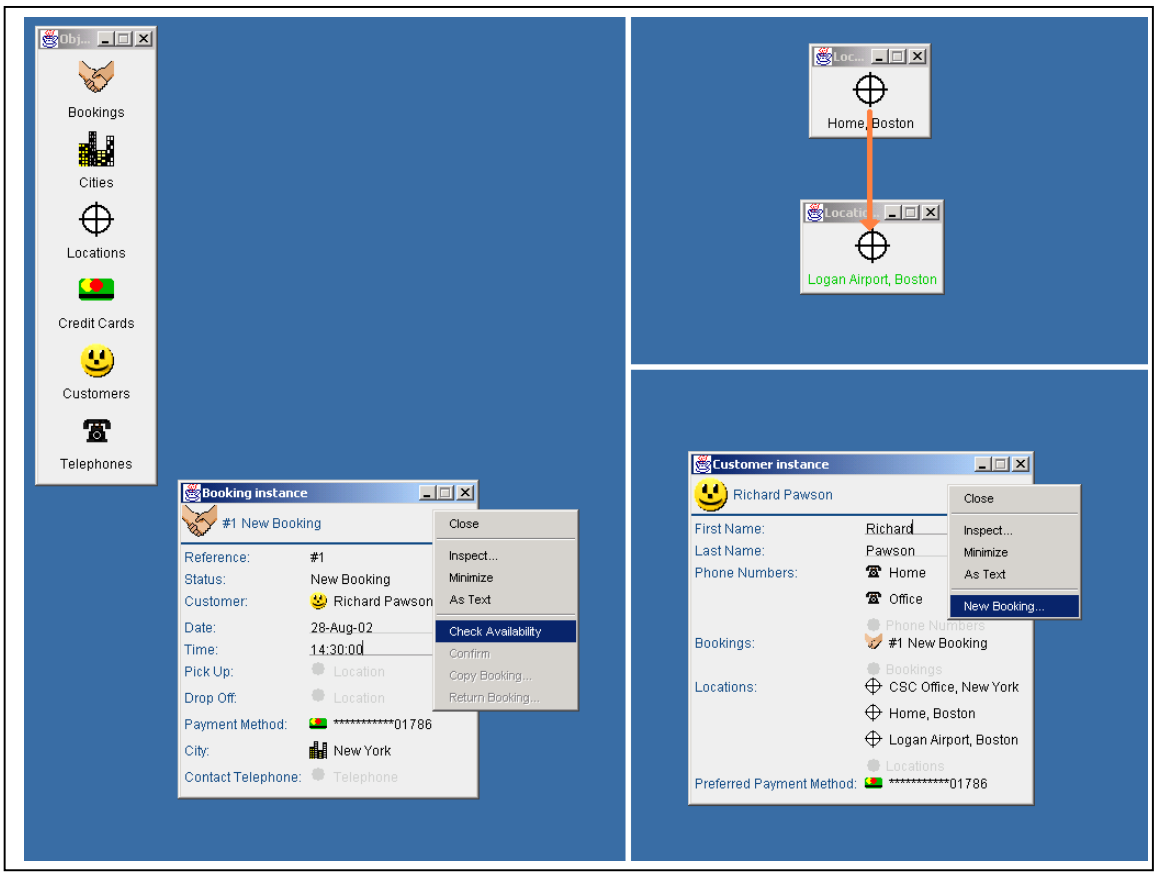


5. Uma associação múltipla é gerenciada por um `InternalCollection` (fornecido pelo framework) que pode guardar apenas objetos de um tipo especificado.
6. Uma associação múltipla precisa apenas de um método `get` para retornar o `InternalCollection`, que então fornece seus próprios métodos para acessar os objetos que ele contém.
7. Associações bidirecionais (seja simples ou múltipla) necessitam fornecer métodos `associate` e `dissociate`.
8. `associateBooking` sobre o `Customer` adicionam primeiro o novo `Booking` para a coleção interna de reservas, então configura o campo `customer` dessa reserva para que aponte para `this` (cliente). Numa associação bidirecional, um objeto gerencia a associação e o outro delega responsabilidade para ele.
9. Adicionar um método `fieldOrder` para um objeto nos permite controlar a ordem no qual os campos são apresentados ao usuário.

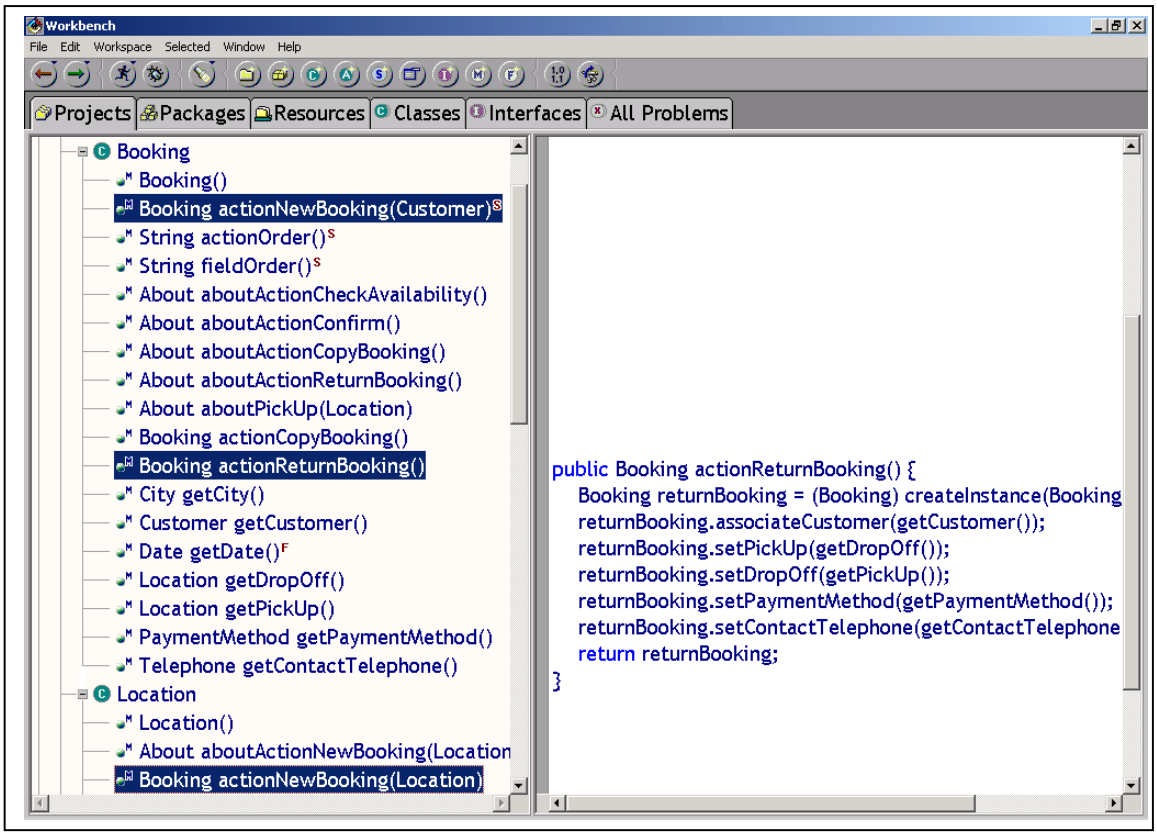
Comportamento

Os dois mecanismos principais pelos quais um usuário pode chamar um comportamento de negócio são por seleção de uma ação a partir do menu pop-up sobre o objeto de negócio e por soltar um objeto sobre um outro. (O último não é o mesmo que arrastar um objeto dentro de um campo vazio dentro de um objeto). É possível chamar comportamentos de negócio em nível de classe – através do

menu pop-up dessa classe – ou por arrastar um ícone de instância sobre o ícone de classe.



1. Na parte inferior do menu pop-up de um objeto estão os métodos de negócio que podem ser aplicados ao objeto.
2. As reticências ... seguindo a ação de menu indica que ela irá retornar um outro objeto como uma nova janela.
3. Arrastar *Home, Boston* diretamente sobre *Logan Airport, Boston* irá disparar a criação de uma nova Reserva que usa essas duas localizações como origem e destino respectivamente. Este atalho é muito útil quando ambas as localizações exibem uma lista de localizações freqüentemente utilizadas sendo de um Cliente.

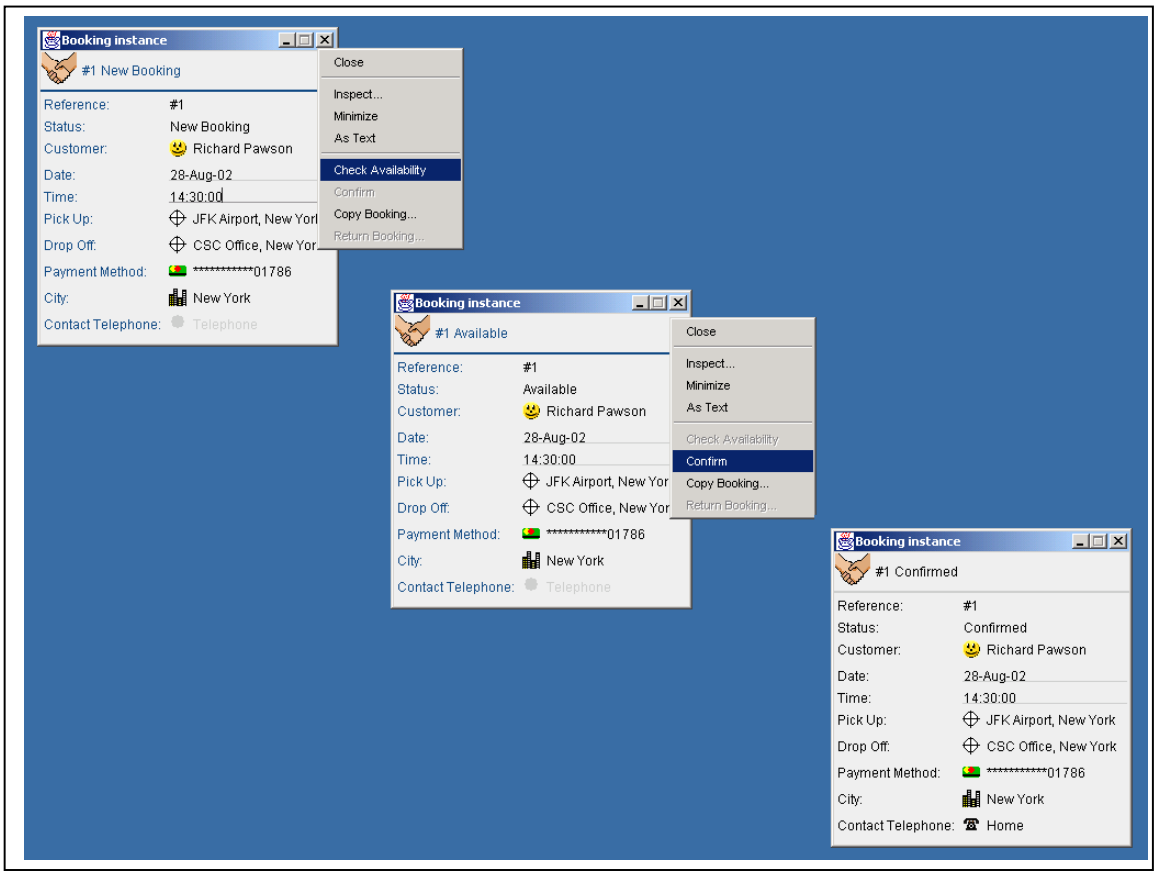


4. O nome do método `actionReturnBooking` é prefixado por `action` e formatado para gerar a opção de menu *Return Bookin*... automaticamente. Para a localização da linguagem é possível sobrescrever esta correspondência automática. A reticência na opção do menu reflete o fato que um objeto será retornado pelo método – neste caso um `Booking`.
5. O `createInstance` cria uma nova instância da classe `Booking`, chamada `returnBooking`. Usar apenas a palavra chave `new` de Java ao invés de `createInstance` não realiza toda a inicialização necessária.
6. Este código troca as localizações de origem e destino para a reserva retornada.
7. Se um método de ação precisar de um parâmetro de entrada (neste caso um outro `Location`) então este comportamento não será exibido no menu pop-up. Ao invés disso, ele será automaticamente chamado quando o usuário soltar um objeto do tipo requerido sobre esse objeto.
8. Este método, chamado quando o usuário solta uma localização sobre um outro, cria um novo `Booking` associado às duas localizações para os campos `pickUp` e `drop-off` respectivamente.
9. Se um outro objeto tiver chamado este método então o `Booking` criado recentemente será passado de volta para ele. Se o método for chamado pelo usuário a partir do menu, então o objeto retornado irá automaticamente exibir como uma nova janela.

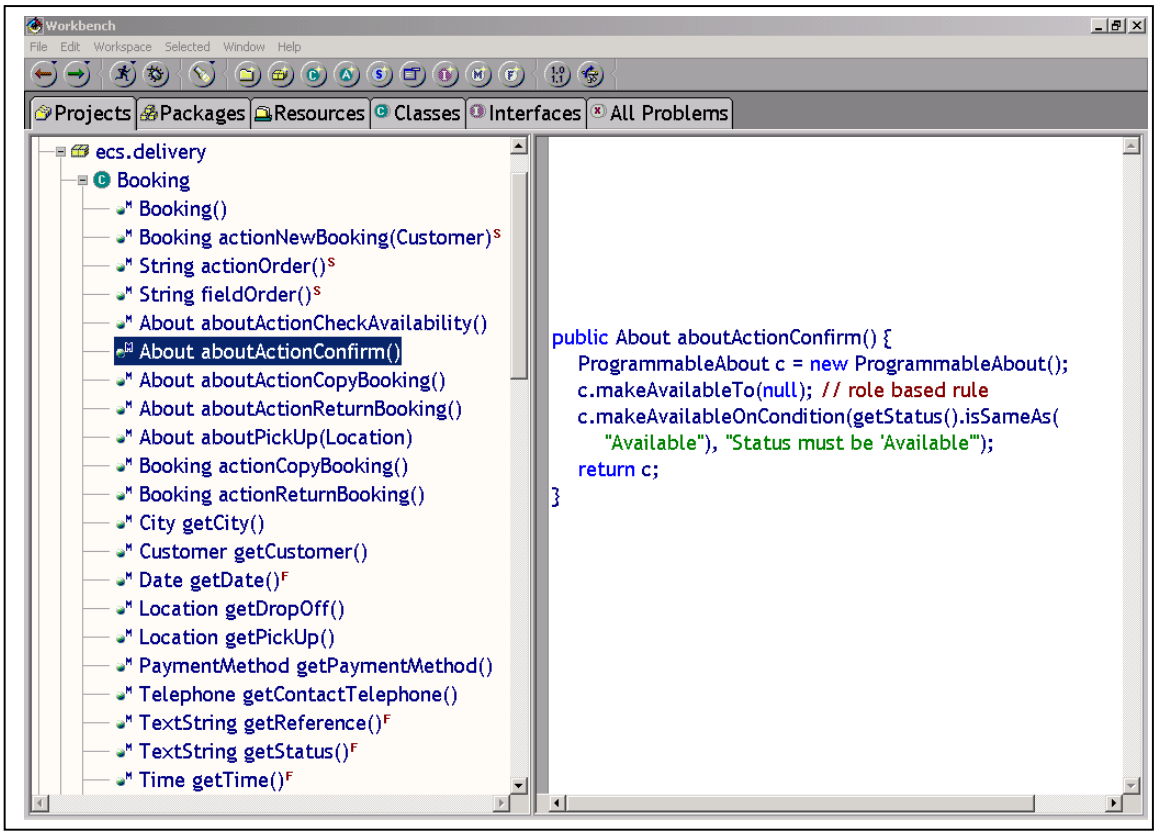
10. O `actionNewBooking` é um método estático: ele pode ser chamado a partir do ícone de classe, não dos ícones de instância. Este método pega um `Customer` como um parâmetro e retorna um `Booking`. Em outras palavras, soltar um cliente sobre o ícone *Bookings* e irá gerar uma nova reserva para esse cliente.

Controle

Os naked objects dão maior poder ao usuário, mas isso não implica na ausência de controles. Diferentes usuários precisarão de diferentes objetos, diferentes campos dentro desses objetos e diferentes comportamentos sobre esses objetos. Será também necessário reforçar certas regras de negócio, tal como prevenir uma ação executada a menos que o objeto esteja no correto estado. No Naked Objects estas formas de controle são implementadas usando métodos e objetos About.



1. Aqui, a opção Confirm (Confirmar) está esverdeada, mas o usuário pode Verificar a Disponibilidade (*Check Availability*).
2. O *Check Availability* resultou na mudança do campo status da reserva para *Available* (disponível). Por causa disso, o usuário agora tem a opção para Confirmar a reserva.
3. A reserva agora tem status Confirmado (*Confirmed*).



4. O método `actionConfirm` tem um método `aboutActionConfirm` correspondente que determina se o primeiro pode ser acessado. Se o acesso não é permitido então o mecanismo de visualização ficará verde nessa opção de menu.
5. Os métodos `about` retornam um objeto `About`, que contém dados sobre a disponibilidade de um método, classe ou instância no qual se aplica. Formas genéricas de `About` são fornecidas com o framework.
6. A disponibilidade pode ser determinada pelas regras de negócio ou pela autorização do usuário. A determinação integral pode ser subcontratada para um servidor de autorizações.
7. Aqui, `aboutActionConfirm` vê se o campo status do objeto `Booking` está como `Available`. Se estiver, *ActionConfirm* irá se tornar acessível ao usuário.
8. Aqui, é fornecida uma razão para a indisponibilidade do método. Na versão atual do framework, esta mensagem aparece na janela de log do sistema. Numa atualização futura, será feito melhor uso desta capacidade, talvez na forma de balões opcionais de ajuda.

Programando com Naked Objects

Nesta seção nós iremos ver como escrever um sistema de negócio usando framework Naked Objects. A maioria do exemplo de código desta seção é da aplicação Serviços de Carro Executivo (ou ECS), o código completo que é distribuído com o framework Naked Objects.

Nós iremos inicialmente verificar a anatomia de um naked object – a interface que deve ser apresentado aos usuários externos. As classes naked objects são escritos em Java e obedecem a certas convenções de codificação naturais adotadas pela estrutura da linguagem Java. Devido à padronização de nomenclatura e dos métodos de estrutura, o framework pode reconhecer os campos e comportamentos de um objeto tornando-os disponíveis, identificáveis e manipuláveis pelo usuário.

Nós então veremos como tornar suas classes naked objects disponíveis aos usuários na forma de uma aplicação padrão simples para exploração e teste.

Na terceira parte verificamos os mesmos objetos disponíveis como parte de um sistema de negócio finalizado. Esta seção cobre os assuntos de compartilhamento de objetos entre múltiplos usuários, persistência (normalmente via um banco de dados), e manipulação de transações.

Na quarta parte iremos escrever testes unitários e de aceitação. Nós mostramos como o Naked Objects viabiliza a escrita dos seus testes de aceitação antes que você comece a escrever algum código operacional.

Na última parte veremos detalhes de codificar funcionalidades de negócio dentro de um objeto de negócio. Isso inclui a manipulação de objetos de valor e coleções dentro de seus métodos, construção de títulos mais complexos para um objetos, e uso do objeto `About` para implementar controles de negócio.

A fim de prosseguir com esta seção você precisará ter alguma familiaridade com a linguagem Java. Nosso objetivo ao desenvolver o framework foi o de construir classes no padrão definido por Java, fazer uso de boas práticas reconhecidas e minimizar o número de novas restrições. A maioria das preocupações ao usar os objetos – apresentação, distribuição e persistência – foi deixada para o framework, liberando você para codificar apenas os objetos de negócio.

Nesta seção tratamos os naked objects da perspectiva do usuário. Por exemplo, falamos sobre métodos `action...` de um parâmetro como um meio de permitir que um objeto seja solto sobre um outro. Essa maneira de descrever os naked objects torna-se natural porque os objetos que estamos definindo são objetos que o usuário manipula. Mas é importante entender que a convenção de nomenclatura exigida pelo framework Naked Objects não é de fato, dirigida à interface do usuário. As convenções de codificação são um meio de expor o objeto, deixando-o

livre, tornando-o pelado. Então o naked objects pode ser visto pelo sistema, e assim, apresentado ao usuário.

Além do mais, o estilo de interface gráfica que mostramos através deste livro é o gerado pelo primeiro mecanismo de visualização que escrevemos. Esperamos escrever um outro mecanismo de visualização que tome os mesmos objetos de negócio e apresente-os com estilos um pouco diferente – talvez via uma página HTML – apenas um browser de interface, ou mesmo via um sistema sintetizador e reconhecimento de voz.

Se você ainda não tiver usado o Naked Objects, sugerimos que antes de trabalhar nesta seção, siga as seguintes instruções passo-a-passo para obter e instalar o framework e então desenvolver uma aplicação muito simples a partir do zero.

Notas para programadores Java experientes

Se você é um programador Java experiente então você estará apto a trabalhar nesta seção rapidamente. Mas saiba que a codificação com o Naked Object difere um pouco na maneira geral de escrever classes Java:

- Todos os campos usados pelo usuário precisam ser do tipo `org.nakedobjects.object.Naked`. O framework não sabe como apresentar ou persistir tipos padrões da linguagem Java. O framework fornece vários tipos `Naked` genéricos tal como `TextString`, `Money`, `WholeNumber` e `Date` (todos parte do pacote `org.nakedobjects.object.value`), o qual inclui estende os tipos fornecidos pela linguagem Java.
- É importante acessar variáveis através de seus métodos de acesso (`get...` e `set...`) ao invés do acesso de forma direta. O mecanismo de persistência conta com tais métodos para armazenar e recarregar os dados do objeto.
- Para criar um objeto com valores já atribuídos e persistentes, use o método fábrica `createInstance`. Novas instâncias dos objetos de negócio não devem ser criadas usando apenas o operador padrão `new` da linguagem Java – isso não irá criá-las apropriadamente. Além disso, um construtor de objeto persistente será chamado toda a vez que ele for restaurado.
- Todo objeto persistente deve ter um único ID que não muda entre instanciações. Os métodos `equals` e `hashCode` baseiam-se nesse ID e não devem ser mudados.

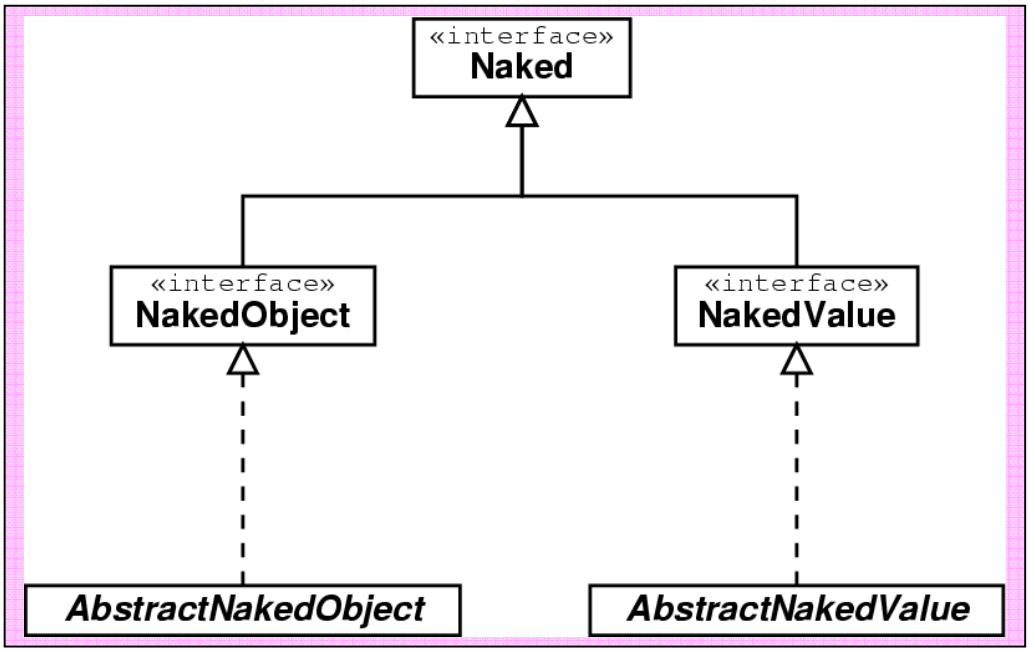
A anatomia de um naked objects

Um sistema Naked Object é apenas uma coleção de classes que são disponibilizados ao usuário. Para se tornar visível ao usuário, um objeto deve ser do tipo `org.nakedobjects.object.Naked`. Esta interface é definida dentro do framework Naked Object junto com duas subinterfaces. A primeira é

`org.nakedobjects.object.NakedValue`, para objetos de valor, que armazenam valores de dados simples (tais como textos, números, valores monetários e datas) e são usados somente dentre de outros objetos. O framework fornece várias classes prontas para o uso que implementam essa interface, as quais nós iremos introduzir nesta seção. Programadores podem adicionar novas classes `NakedValue` se quiserem. A segunda sub-interface é `org.nakedobjects.object.NakedObject`. Ela é usada por objetos que precisam ser referenciados em múltiplos contextos, os quais são normalmente objetos de negócio.

A maneira mais simples de assegurar que os seus objetos de negócio estão de acordo com a definição `NakedObject` é torná-los subclasses do `org.nakedobjects.object.AbstractNakedObject` - uma classe fornecida pelo framework. Se você não puder fazer isso, porque o seu objeto de negócio deve herdar de alguma outra hierarquia, então você terá que implementar os métodos necessários pelo próprio `NakedObject`.

A `AbstractNakedObject` implementa todos os métodos da interface `NakedObject` exceto o `title`, o qual iremos discutir em breve. Assim, a interface fornece uma base para construir seus objetos de negócio. Nós usaremos esta classe durante as nossas discussões.



Se uma classe de objetos de negócio é definida como uma subclasse de `AbstractNakedObject`, então você deve também fazer o seguinte para que o framework esteja apto a acessar e manipular esses objetos, e assim disponibilizá-los aos usuários.

- Declarar a classe como pública.
- Assegurar que a classe tenha um construtor sem nenhum parâmetro (normalmente chamado de construtor default).
- Declarar público todos os métodos que serão disponibilizados ao framework.
- Implementar o método abstrato `title` da superclasse tal que ele retorne uma referência não nula. O método `title` irá ajudar os usuários a distinguir cada objeto de outras instâncias de um tipo específico – mas pode também ser usado para outros propósitos, tais como o propósito de busca ou geração de relatórios.

As classes naked objects

Aqui está uma declaração de classe para uma classe de objetos de negócio escrito usando o framework Naked Objects. Ela é apresentada sem qualquer corpo de método, e com membros requeridos em negrito. O construtor sem nenhum parâmetro é apresentado para reforçar o ponto de que ele é sempre necessário (embora se você deixar de declará-lo, Java irá fornecer um para você).

```
import org.nakedobjects.object.AbstractNakedObject;
import org.nakedobjects.object.Title;
import org.nakedobjects.object.control.About;

public class Parcel extends AbstractNakedObject {
public static final long serialVersionUID = 1L;

public static About aboutParcel() {}

public Parcel() {}

public About about() {}

public void created() {}

public Title title() {}
}
```

A variável de classe é o número da versão para esta classe que tem a propriedade de ser seriada, que é usada quando o objeto é seriado (como acontece quando ele é transferido através da rede). Embora exista pouca coisa que possa provocar falha na seriação, o número serial default pode ser diferente em diferentes JVMs. Assim é boa prática configurar esta variável para cada nova classe. (No entanto, para manter as coisas simples, nós não mostramos isso nos exemplos que se seguem). Como o framework Naked Objects usa um algoritmo de seriação de “granularidade” alta, este número serial pode ser iniciado (com algum valor arbitrário) e deixado, até a mudança da classe ocorrer.

Os dois métodos `about...` são usados para controlar a acessibilidade da classe e de suas instâncias.

O método `created` somente é chamado após a criação de uma nova instância lógica da classe e não todas as vezes que o objeto lógico é recriado em memória pelo mecanismo de persistência. Por contraste, o construtor será chamado todas as vezes que Java instanciar este particular objeto persistente, o que acontece todas as vezes que o objeto é recuperado do armazenamento persistente. Assim, é no método `created` o local no qual você deve colocar qualquer código para iniciar um objeto de negócio.

Tornando uma classe não instanciável

Por default, permite-se que usuários criem novas instâncias de qualquer classe de negócio. Para prevenir que isso aconteça, você pode adicionar um método `about` estático em sua classe. Por exemplo, na aplicação ECS o conjunto de objetos `City` é fixado através do seguinte código:

```
public static About aboutCity() {
return ClassAbout.UNINSTANTIABLE;
}
```

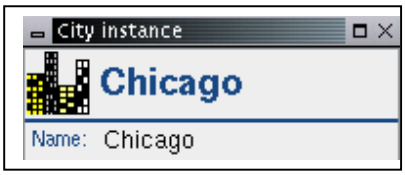
(`org.nakedobjects.object.control.ClassAbout` é um tipo especial de objeto `org.nakedobjects.object.control.About` fornecido pelo framework).

Por conseqüência, o menu pop-up do ícone *Cities* irá mostrar a opção *New Instance...* desabilitada. A operação de arrastar a classe sobre o desktop ou sobre um campo para criar um objeto não irá funcionar.

O `About` pode também ser usado para proibir acesso, tornando a classe invisível a certos tipos de usuários, ou a todos os usuários.

Tornando um objeto somente-leitura

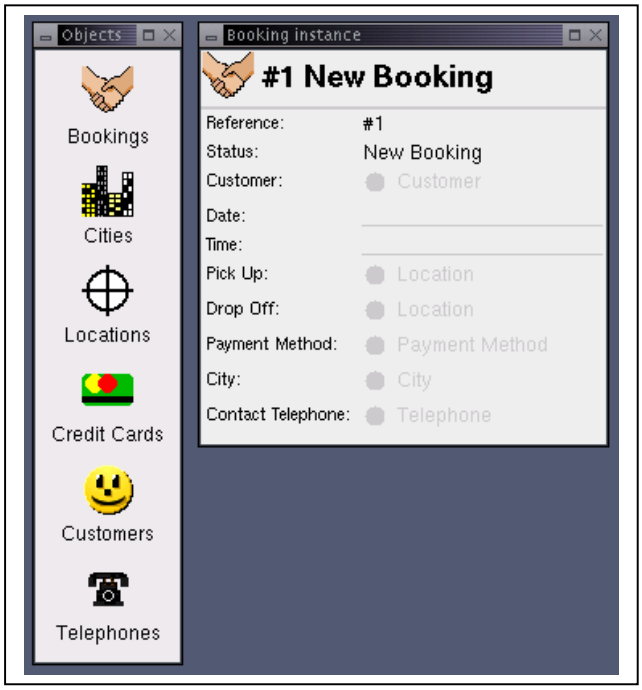
Também é possível controlar o acesso a instâncias individuais. Podemos deixar um objeto imutável, tal que os campos sejam configurados como somente leitura, e podemos tornar um objeto inacessível, tal que não apareça na tela. Isso é feito através do método do objeto `about` (em oposição aos métodos da classe `about` tal como o `aboutCity` do exemplo acima). Seguindo o método, tomemos novamente a classe `City`, e vamos tornar qualquer instância da cidade não editável:




```
public About about() {
return ObjectAbout.READ_ONLY;
}
```

Nomes de classe

As classes são normalmente apresentadas ao usuário em dois lugares quando o Naked Objects está executando. O primeiro, e mais óbvio, é na lista de classes. O segundo é em cada campo vazio, onde a classe indica o tipo de objeto que pode ser colocado naquele campo. A tela abaixo mostra ambos os casos. Note que o campo *Customer* recebe um objeto `Customer` e que ambos os campos *Pick Up* e *Drop Off* recebem objetos `Location`.



Os títulos apresentados pelas classes são automaticamente gerados a partir do nome da classe, sem o nome do pacote e adicionando espaços na frente dos subseqüentes caracteres maiúsculos, para separar as palavras, e adicionando um 's' no final. Se o nome da classe se tornar um plural irregular, então você pode usar o método `pluralName` para especificar manualmente a versão do plural. Por exemplo, o seguinte código mostra o método de configuração para retornar o plural correto da classe `Calf`:

```
public static String pluralName() {
return "Calves";
}
```

Você pode também especificar um título da classe a ser mostrado aos usuários de que seja diferente do nome da classe Java, usando o método `singularName`. Isso é aconselhável em linguagens que usam acentos e ligaduras (sobreposição de caracteres, por exemplo æ). Quando o código fonte Java é compilado usando

Unicode ao invés de ASCII, é possível usar ambos os identificadores, mas não é recomendado porque identificadores são facilmente propensos a erros ortográficos ou corrompidos quando arquivos de código fonte são transferidos. Ao invés disso, dê aos seus identificadores nomes simples em ASCII, como mostra o exemplo abaixo, que embute o 'ä' germânico dentro da string plural como um caractere unicode:

```
public class City extends AbstractNakedObject {
public static String singularName() {
return "Stadt";
}

public static String pluralName() {
return "St\u00e4dte";
}
}
```

Campos

Quando o mecanismo de visualização do framework retrata um objeto de negócio em uma de suas visões (mais comumente a visão 'form'), ele olha para ver se o objeto tem algum método de acesso público disponível que retorne um objeto do tipo `org.nakedobjects.object.Naked`. Se houver, ele mostra esse objeto num campo.

Esses métodos de acesso têm muitas utilidades: eles são usados pelo mecanismo de persistência, e podem ser chamados por outros objetos. Mas é normalmente conveniente pensar que eles, pelo menos inicialmente, tornam visível um atributo interno ao usuário.

Os campos podem ser divididos em três categorias: valor, associação e múltiplas associações. Todos eles podem ser vistos no exemplo de tela abaixo, onde os valores aparecem com caracteres sobre a uma linha e são editáveis, e as associações mostradas coma ícones. Para associações múltiplas, o campo pode mostrar mais de um ícone.

O rótulo, ou o nome do campo é gerado a partir do nome do método acesso (o nome do método sem o seu prefixo 'get', e com espaços adicionados onde uma letra maiúscula é encontrada). Um método de acesso normalmente compartilha o mesmo nome que a variável que ele acessa, mas nem sempre. Lembre-se que é o nome do método e não o nome da variável que é usado pelo framework para rotular o campo apresentado ao usuário.

Valores

Os valores de dados simples e não compartilhados são restritos à classe do tipo `org.nakedobjects.object.NakedValue`. O framework fornece `TextString`, `Date`, `Time`, `WholeNumber` (no pacote `org.nakedobjects.object.value`) e vários outros. Use essas classes para modelar todos os valores que podem precisar ser acessados de fora do objeto de negócio – por exemplo, para mostrar ao usuário ou tornar persistente num banco de dados. Programadores podem definir novos tipos `NakedValue`, por exemplo para manipular unidades científicas de medida. Você pode usar outras classes e primitivas Java tal que `java.lang.String` ou `int` dentro de um método para propósitos estritamente locais, mas não pode exibi-los diretamente via o mecanismo de visualização.

Cada objeto valor deve ser declarado e iniciado, e deve ser disponibilizado através de métodos de acesso:

```
public class Customer extends AbstractNakedObject {
private final TextString lastName;
private final TextString firstName;

public Customer() {
firstName = new TextString();
lastName = new TextString();
}
```

```
public TextString getLastName() {
return lastName;
}

public TextString getFirstName() {
return firstName;
}
}
```

As variáveis `lastName` e `firstName` são ambas objetos `TextString` e são usadas para armazenar informações textuais simples. Todos os objetos valor são mutáveis e devem ser declarados como privados ou finais, de modo que um objeto valor específico referenciado por um campo nunca seja trocado. O valor que ele contém (por exemplo, o sobrenome) pode mudar, mas apenas pelos seus métodos de mudança próprios fornecidos. Esta convenção ajuda a assegurar que objetos valor não sejam inadvertidamente compartilhados entre objetos de negócio tentando copiar um valor de um campo para outro.

Desde que eles são marcados como final, essas variáveis devem ser iniciadas antes que o objeto que a contém possa usá-lo. Isso deve ser realizado antes da declaração, ou como nós apresentamos, dentro de um construtor. Todas as classes de objeto valor têm um construtor sem nenhum parâmetro, assim eles podem ser instanciados sem que você tenha que fornecer um valor inicial.

Como todos os campos que contêm um objeto valor são marcados como final, apenas um método `get...` é requerido. A beleza dos objetos valor é que o framework cuida completamente deles. Se o usuário quiser mudar um valor então o framework pede ao objeto valor para que ele se altere. Se a interface do usuário precisa conhecer quais são valores possíveis existentes (digamos, para o objeto valor `org.nakedobjects.object.value.Option`) então o framework pergunta isso diretamente ao objeto valor. Como programadores, nosso trabalho estará concluído quando usamos um tipo específico de objeto valor e o declaramos como descrito.

Podemos fazer com que os campos de valor sejam apenas de leitura, assim os campos não poderão ser editados. Isso é indicado ao usuário pela ausência da linha verde sob o texto do campo. O campo pode ser iniciado a qualquer momento chamando um método `setAbout` do objeto valor e passando um objeto `About` como parâmetro – depois disso ele não poderá ser mudado. Os objetos `org.nakedobjects.object.control.About`, adequados para tornar um campo somente-leitura ou somente-escrita, podem ser obtidos a partir da classe `org.nakedobjects.object.control.FieldAbout` e são chamados `READ_ONLY` e `READ_WRITE` respectivamente. Normalmente, objetos valor são iniciados com somente-leitura quando eles são criados, como mostra o seguinte código tirado da classe `Booking`:

```
public Booking() {
reference = new TextString();
```

```
reference.setAbout(FieldAbout.READ_ONLY);
status = new TextString();
status.setAbout(FieldAbout.READ_ONLY);
}
```

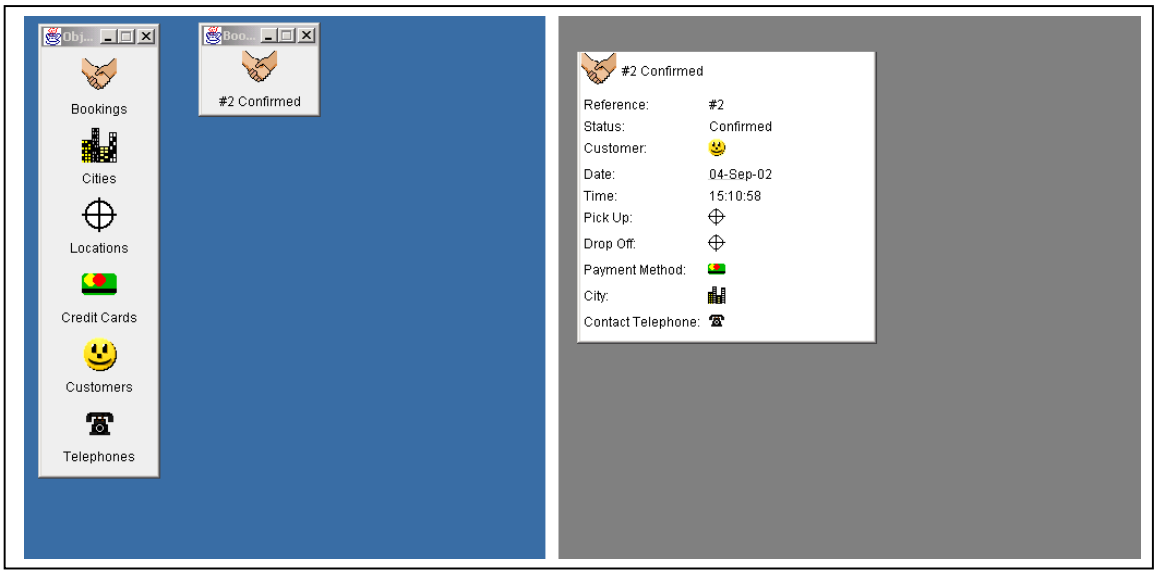
Associações um-para-um

Uma associação um-para-um ocorre quando um objeto de negócio tem um campo que contém uma referência para um outro objeto de negócio. (Um objeto de negócio é um objeto do tipo `org.nakedobjects.object.NakedObject` – qualquer outro tipo de objeto é ignorado pelo framework). Esta associação é essencialmente obtido apenas implementando os métodos de acesso padrão `get`**`Variable`** e `set`**`Variable`**. O seguinte código, tomado da classe `Booking`, mostra uma variável de referência e seus dois métodos de acesso. Você irá ver também que os métodos `getDropOff` e `setDropOff` possuem chamadas específicas adicionadas dentro dele:

```
public class Booking extends AbstractNakedObject {
private Location dropOff;

public Location getDropOff() {
resolve(dropOff);
return dropOff;
}

public void setDropOff(Location newDropOff) {
dropOff = newDropOff;
objectChanged();
}
}
```

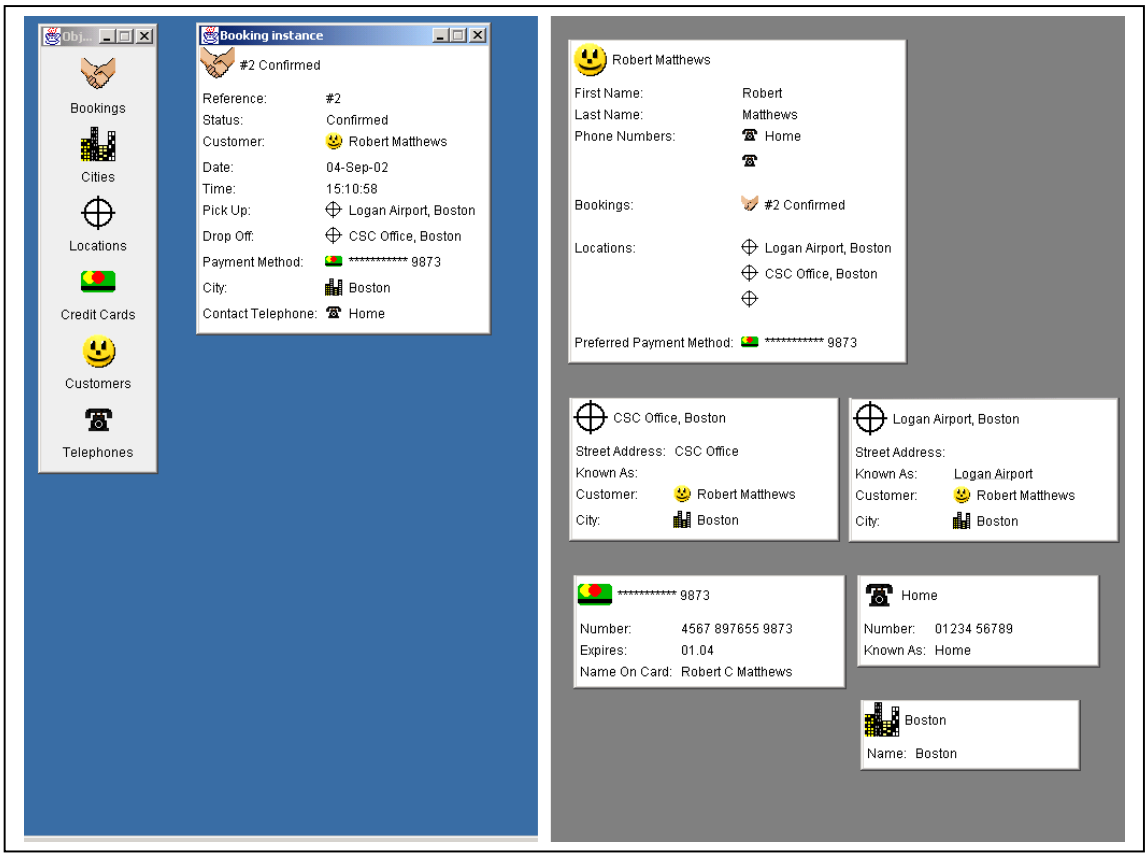
No método `getDropOff`, nós podemos ver uma chamada `resolve`, que é um método estático fornecido na hierarquia de superclasse. O propósito de chamar `resolve` é assegurar que o objeto referenciado (neste caso o `Location` usado no campo `DropOff`) realmente existe, é totalmente formado e está em memória. O fato de que o objeto `Booking` ter todos os seus dados carregados não implica que todos os outros objetos que ele referencia estejam também carregados: se o framework operasse desse jeito, a carrega de um único objeto poderia tomar muito tempo! Em vez disso, o framework empenhar-se em carregar objetos do repositório só quando eles são necessários.

Isso é mais bem explicado com um exemplo. Na imagem abaixo, do lado esquerdo (com o fundo azul), mostra a visão do usuário. Do lado direito (com fundo cinza) é uma ilustração do que acontece na memória de trabalho:



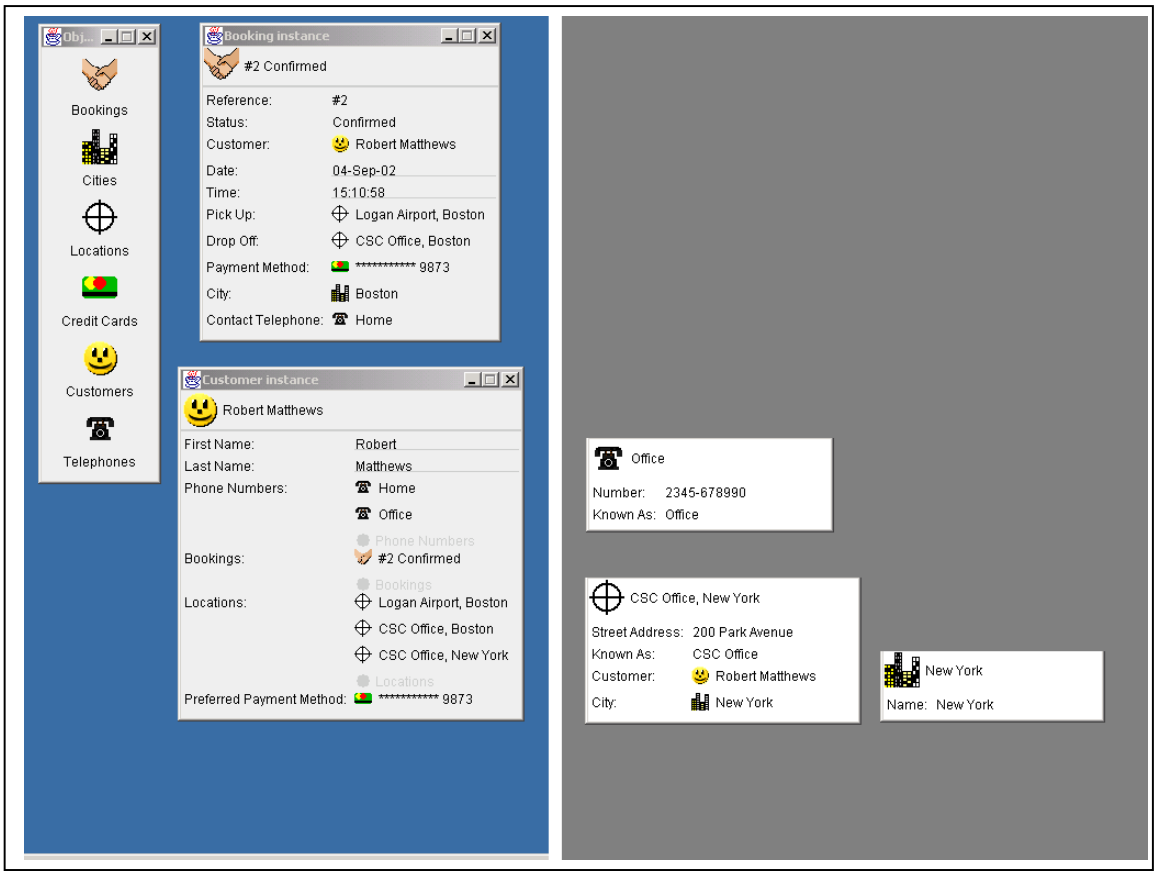
Na primeira tela, o usuário recuperou a reserva *#2 Confirmed* do repositório e o visualizou como um ícone. Na outra nós podemos ver que um objeto reserva foi instanciado na memória, e que os campos de valor, que são uma parte integral do objeto, foram recuperados do repositório. Neste caso, esses campos de valor fornecem informação suficiente para que o título do objeto reserva seja gerado (especificamente a partir dos campos `Reference` e `Status`). Para cada um dos objetos associados com essa reserva, uma nova instância do tipo apropriado foi criado dentro do campo associação, mas nenhum desses objetos foi ainda resolvido. Nós retratamos isso graficamente usando ícones sem títulos.

O usuário agora abre o objeto reserva para mostra a visão de formulário. Todos os campos de valor e todos os objetos associados são mostrados agora:



A fim de mostrar os títulos de cada um dos objetos associados (tal como o cliente para esta reserva) o framework resolve automaticamente esses objetos. (O objeto reserva ainda existe na memória, mas não é mostrado por razões de espaço). Cada um desses objetos tem agora cada um dos campos valores recuperados do repositório, mas onde os objetos contiverem referências (associações) a outros objetos, eles serão instanciados, mas não por si só resolvidos. Se o objeto associado já tiver sido recuperado, então esta referência é usada. Em nosso exemplo isso aconteceu com o telefone *Home*, com as duas primeiras localizações e o cartão de crédito, porque todos eles foram usados pela reserva. O segundo telefone e a terceira localização não foram usados ainda e assim, existe somente como esqueletos de objetos.

O usuário agora abre uma visão formulário do objeto cliente, neste caso como uma nova janela:



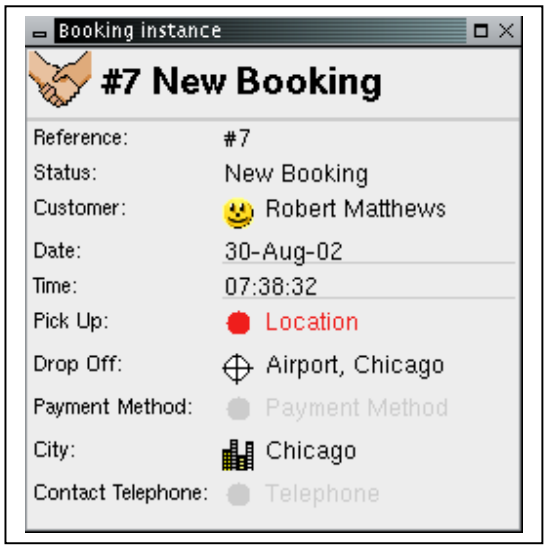
No lado direito podemos ver que este objeto agora foi resolvido em memória. Muitos objetos referenciados por ele já estão disponíveis quando eles tiverem sido usados anteriormente, exceto para o telefone *Office* e localização *CSC Office* que agora deve ser lido. Note que o objeto localização tem um título que usa tanto o campo valor *Known As*, quanto o campo associação *City*. A geração do titulo então força a recuperação automática do objeto *New York*.

Em conseqüência, assessores de campos que referenciam outros objetos de negócio (tal como o `getDropOff` visto na última peça de código) devem começar com uma tentativa explícita para resolver o objeto referenciado. O método `resolve` verifica a referência e se é válida, mas que ainda não foi resolvida, o framework lança uma requisição de resolução para o repositório do objeto. O repositório do objeto responde carregando os dados que faltam para dentro do objeto. Alternativamente, se o argumento é `null`, ou se o objeto referenciado já tiver sido resolvido, então a requisição é simplesmente ignorada. Portanto, quando este método retornar, podemos estar seguros de que o objeto referenciado foi resolvido e pode ser usado com segurança. Após a referencia ter sido resolvida ela é retornada normalmente a quem chamou o método de acesso.

No método `setDropOff` acima, após realizada a associação, o `objectChanged` é chamado. Isso notifica o framework que o objeto foi alterado. Em resposta o

framework atualiza a versão persistente do objeto e notifica os outros observadores desse objeto tal que suas visões possam ser atualizados. Se você esquecer de incluir a chamada do `objectChanged` dentro de um método `set...`, então você pode achar que as mudanças no objeto não foram persistidas, sua visualização não será atualizada apropriadamente, e; se o objeto estiver sendo compartilhado, seus usuários estarão trabalhando com informações não atualizadas.

É comum controlar campos de associação tal que eles possam se tornar inacessíveis ou somente-leitura, ou que eles aceitem somente certos objetos. Quando uma associação é tornada inacessível, o campo que seria normalmente mostrado não aparecerá na visão. Se o campo é somente-leitura ele não permitirá que objetos (de seu tipo) sejam soltos dentro dele, tornando-se vermelho se o usuário tentar fazer isso.



Um campo de associação é controlado através de um método `about...` cujo nome é baseado no nome do campo sem o prefixo 'get' ou 'set'. Esse método deve aceitar um parâmetro, que deve ser do mesmo tipo que o especificado nos métodos `get...` e `set...`, e retornar um objeto `About`. O seguinte exemplo torna o campo *Drop Off* (como identificado no método `getDropOff`) somente-leitura:

```
public About aboutDropOff(Location location) {
return FieldAbout.READ_ONLY;
}
```

Além do controle que é exercido sobre os métodos de acesso, o objeto `About` pode fornecer um nome que é usado para trocar o rótulo do campo. Este mecanismo pode ser usado para localizar os nomes de campos quando o sistema é usado internacionalmente.

Associações um-para-muitos

Uma associação um-para-muitos ocorre quando um campo contém várias referências para outros objetos de negócio. Isso é obtido especificando que o campo pode conter `org.nakedobjects.object.collection.InternalCollection` – uma coleção especializada fornecida pelo framework Naked Object. O `InternalCollection` é por si só um tipo de `org.nakedobjects.object.NakedObject`, e pode dessa forma ser 'visto' por outros elementos do framework tais como os mecanismos de visualização e persistência. O `InternalCollection` se torna uma parte composta do objeto de negócio e é responsável por gerenciar as referências a outros objetos de negócio. Ele fornece métodos para adicionar e remover referências a objetos de negócio e assegura que todos eles obedecem ao tipo de negócio especificado. Numa futura atualização ele irá fornecer métodos adicionais para ordenar e outras operações.

O exemplo a seguir, tirado da classe `Customer`, mostra duas coleções declaradas com os métodos de acesso apropriados. As coleções são configuradas para manter vários objetos `Location` e `Telephone`:

```
public class Customer extends AbstractNakedObject {
private final InternalCollection locations;
private final InternalCollection phoneNumbers;

public Customer() {
locations = new InternalCollection(Location.class, this);
phoneNumbers = new InternalCollection(Telephone.class, this);
}

public final InternalCollection getLocations() {
return locations;
}

public final InternalCollection getPhoneNumbers() {
return phoneNumbers;
}
}
```

As variáveis `org.nakedobjects.object.collection.InternalCollection` são marcadas como final para assegurar que eles permaneçam parte do objeto cliente e nunca sejam trocados. Os conteúdos das coleções são alterados chamando métodos do próprio objeto da coleção. Um método de acesso normal `get...` é declarado, mas como está marcado como final, não existe nenhum método `set...`.

Como a coleção é uma parte que compõem o objeto cliente, ela é criada quando o cliente é criado. O próprio construtor de `InternalCollection` precisa que nós especifiquemos o tipo dos objetos que ele irá conter, como um objeto `java.lang.Class`; e em qual objeto ela pertence. (Diferentemente de outros

campos, o tipo não pode ser especificado dentro da declaração de variáveis, nem colocá-lo como parte da assinatura do método).

Como ocorre com objetos valor, o framework toma cuidado para gerenciar coleções. Sempre que o usuário adicionar um objeto ao campo que usa um `InternalCollection`, ou um método adicionar um elemento na coleção, o framework verifica o tipo, adiciona o objeto à coleção, e notifica o repositório do objeto tal que o dado persistente possa ser atualizado. Sempre que o usuário usar um elemento desse campo, ou um outro método acessar os elementos da coleção, esses elementos serão resolvidos antes que eles sejam disponibilizados.

Associações um-para-muitos também podem se tornar inacessíveis ou somente-leitura, da mesma forma que as associações uma-para-um. Serão adicionados ao framework no futuro próximo, controles adicionais para adicionar e remover objetos específicos de coleções. Quando uma associação é marcada como inacessível o campo em que seria normalmente mostrado não será adicionado à visão. Quando marcado como somente-leitura, o campo não permitirá que objetos (mesmo de tipos corretos) sejam soltos nele; o a 'bolinha' cinza sobre a tela indicará que o objeto não poderá ser solto naquele lugar.

O controle do campo de associação uma-para-muitos é feito através de um método `about...` cujo nome é baseado no nome do campo sem o prefixo 'get'. Este método deve retornar um objeto `About` e não ter nenhum parâmetro. O seguinte exemplo deve tornar o campo *Locations* (como identificado no método `getLocations`) somente-leitura:

```
public About aboutLocations() {
return FieldAbout.READ_ONLY;
}
```

Associações bidirecionais

Até agora nós consideramos como um objeto conhece um outro objeto – uma associação de um caminho. Normalmente isso é suficiente, mas algumas vezes o objeto que está sendo referenciado também precisa conhecer o objeto que o referencia. Tome, por exemplo, um sistema de pedidos. Um objeto pedido normalmente conterá uma referência para o cliente que fez o pedido. No entanto, pode também ser útil estar apto a obter diretamente de um objeto cliente o seu pedido atual, talvez até o acesso a todos os seus pedidos. Para permitir isso, o cliente necessita manter uma referência ao objeto pedido. Isso pode ser obtido adicionando um campo à classe cliente, contendo ou um único objeto pedido, ou uma coleção de pedidos.

Manter referências em ambos os objetos de um relacionamento permite que o usuário encontre objetos associados a partir de um ou outro objeto como ponto de partida. Isto é, a associação entre os objetos é bidirecional.

Em princípio, isso é simples de implementar: você define os campos e disponibiliza os métodos de acesso em ambos os objetos, cada um contendo referências a objetos de outro tipo. No entanto, a fim de especificar tal relacionamento o usuário deve primeiro ter soltado um objeto dentro do campo apropriado do segundo objeto, e então soltar o segundo objeto de volta dentro do campo correspondente do primeiro objeto.

Isso seria tedioso e propenso a erros. A solução óbvia seria modificar os métodos `set...` tal que eles chamassem um ao outro, fazendo com que ambas as referências sejam resolvidas numa única operação `set...`. No entanto, você precisa assegurar que você não provocará um loop infinito pela chamada repetitiva dos métodos set dos objetos. Você deve também precisar escrever a funcionalidade necessária para remover ou limpar um relacionamento quando necessário. Este processo complexo fica pior quando um dos lados é uma `InternalCollection`.

O framework Naked Objects simplifica tudo isso através do uso opcional de um par de métodos `associate`**`Variable`** e `dissociate`**`Variable`** para associar e desassociar a associação respectivamente. Se esses dois métodos estiverem presentes, eles serão chamados preferencialmente ao método set, ou ao acesso à coleção interna para adicionar uma referência a ele ou removê-lo. Esses novos métodos devem corresponder ao método `get`**`Variable`** no nome e aceitar um parâmetro. O parâmetro deve ser o tipo do objeto de negócio retornado pelos métodos de acesso, ou se o campo contiver um `org.nakedobjects.object.collection.InternalCollection`, o tipo com o qual a coleção foi inicializada.

O relacionamento entre os objetos `Customer` e `Booking` na aplicação ECA mostra como isso funciona. O seguinte código tirado da classe `Customer` mostra a `InternalCollection` sendo declarada, inicializada e disponibilizada. Ele também mostra que os métodos de associação `associateBookings` e `dissociateBookings` foram especificados esperando que um objeto `Booking` como o tipo especificado quando a coleção interna foi inicializada.

```
public class Customer extends AbstractNakedObject {
private final InternalCollection bookings;

public Customer() {
bookings = new InternalCollection(Booking.class, this);
}

public final InternalCollection getBookings() {
return bookings;
}
```

```
public void associateBookings(Booking booking) {
getBookings().add(booking);
booking.setCustomer(this);
}

public void dissociateBookings(Booking booking) {
getBookings().remove(booking);
booking.setCustomer(null);
}

}
```

Com os dois métodos de associação no lugar, soltar um objeto *Booking* sobre a 'bolinha' verde do campo *Bookings* dentro de um objeto *Customer* ira fazer com que o método `associateBookings` seja chamado ao invés do método `getBookings`.

Quando o método `associateBookings` é chamado ele assume a responsabilidade de adicionar o objeto `Booking` na coleção de bookings do cliente, e assim finalizando o caminho de ida. Além disso, esse método pede ao booking para que o seu campo cliente o referencie, fechando o caminho de volta.

O método dissociate é usado para fazer o inverso, removendo o booking da coleção e configurando o campo cliente do cliente como `null`, i.e. removendo os caminhos de ida e volta.

Nós provavelmente também queremos estar aptos a iniciar esta associação bidirecional a partir do outro objeto – isto é, quando um objeto *Customer* é solto no campo *Customer* de *Booking*, então o campo cliente é configurado com esse objeto cliente e o objeto *Booking* adiciona-se ao conjunto de booking desse cliente.

O seguinte código mostra os métodos "associate" e "dissociate" que adicionamos à classe `Booking`. Por simplicidade, implementamos somente chamando o método correspondente da classe `Customer`. Isso evita a duplicação de código que gerencia a associação. Como na classe `Customer`, os métodos de acesso permanecem como são. Eles são chamados pelos métodos de associação, e são também usados para persistir o objeto:

```
public class Booking extends AbstractNakedObject {
private Customer customer;

public void associateCustomer(Customer customer) {
customer.associateBookings(this);
}

public void dissociateCustomer(Customer customer) {
customer.dissociateBookings(this);
}
```

```
public Customer getCustomer() {
resolve(customer);
return customer;
}

public void setCustomer(Customer newCustomer) {
customer = newCustomer;
objectChanged();
}
}
```

Agora, ao soltar um objeto *Customer* sobre a 'bolinha' verde do campo *Customer* de um objeto *Booking* irá fazer com que o método `associateCustomer` seja chamado ao invés do método `setCustomer`. Isso chamará o método `associateBookings` (do objeto `Customer`) que nós vimos anteriormente. Como antes, esse método adiciona o objeto booking à sua coleção de bookings, acerta o caminho de volta, e então chama o método `setCustomer` do objeto `Booking`, acertando o caminho de ida.

Associações bidirecionais são codificadas da mesma maneira se elas forem um-para-um, uma-para-muitos, ou muitos-para-muitos. A única diferença é se nós adicionamos ou removemos elementos de uma coleção ou um único objeto.

Os prefixos 'add' e 'remove' podem ser usados ao invés de prefixos 'associate' e 'dissociate'. Métodos usando esses prefixos são lidos mais facilmente quando usados dentro com uma coleção interna. Por exemplo, a classe `Customer` pode igualmente ser escrita como:

```
public class Customer extends AbstractNakedObject {
private final InternalCollection bookings;

public Customer() {
bookings = new InternalCollection(Booking.class, this);
}

public void addBookings(Booking booking) {
getBookings().add(booking);
booking.setCustomer(this);
}

public void removeBookings(Booking booking) {
getBookings().remove(booking);
booking.setCustomer(null);
}

public final InternalCollection getBookings() {
return bookings;
}
}
```

Os métodos `associate` e `dissociate` podem ser melhores usados para outros propósitos, tal como a de configurar dois campos de uma só vez. Por exemplo, no objeto `Booking` quando uma `Location` é solta sobre os campos *Pick Up* ou *Drop Off*, então assim como configurar o campo alvo, o campo *City* do booking também será configurado com a mesma que cidade, `City`, que está contido no objeto `Location` que foi solto. Isso obtido através do seguinte código:

```
public void associateDropOff(Location newDropOff) {
setDropOff(newDropOff);
setCity(newDropOff.getCity());
}

public void associatePickUp(Location newPickUp) {
setPickUp(newPickUp);
setCity(newPickUp.getCity());
}
```

Campos derivados

Se um campo precisa ser derivado dinamicamente a partir de outros campos dentro do objeto, então é melhor usar o prefixo 'derive' ao invés de 'get'. Isso tornará o campo somente-leitura e não-persistente, evitando qualquer potencial problema que possa surgir quando o mecanismo de persistência tentar armazenar ou recuperar este campo do repositório.

No seguinte exemplo uma data de vencimento 'due' é calculada como sendo 14 dias após a data acordada no campo `ordered`. O importante aqui é que um novo objeto `Date` é criado a partir de um existente. Esta cópia é então usada para fazer cálculos e é essa cópia que é retornada pelo framework. O princípio aqui é que por criar um novo objeto valor você evita corromper os campos de outros objetos.

```
public Date deriveDue() {
Date due = new Date(ordered);
due.add(14,0,0);
return due;
}
```

Ordenando campos

A ordem dos campos quando um objeto é exibido é baseado no vetor de métodos de acesso que é produzido pelo mecanismo de reflexão da linguagem Java. Isso significa que a ordem depende de como o JVM que você está executando coleciona os dados sobre a sua classe, e você não tem nenhum controle direto sobre isso. Você pode, no entanto, especificar uma ordem dentro de sua definição de classe, a qual o mecanismo de visualização interpreta. Isso é realizado adicionando-se um método chamado `fieldOrder` que retorna uma lista de `String`

com nomes de campos separados por vírgulas. Eles precisam ser os nomes refletidos e não os nomes de métodos. Por exemplo, a string 'Pick Up' deve ser usada para referenciar o método `getPickUp`.

O seguinte exemplo da classe `Booking` especifica uma ordem para o seus campos:

```
public static String fieldOrder() {
return "reference, status, customer, date, time, pick up, drop off, payment method";
}
```

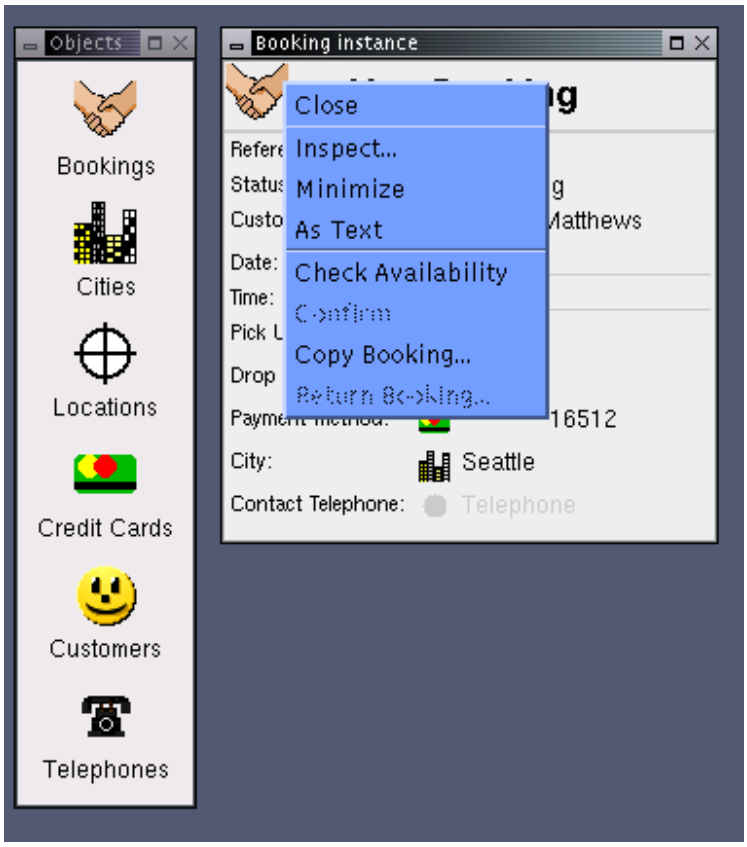
Qualquer nome da lista de que não tiver associação com um campo será ignorado, e os campos que não estão na lista serão colocados depois de todos os campos que estão especificados.

Comportamentos

Qualquer método dentro de um naked object que esteja prefixado com ‘action’, e que retorne ou um tipo `NakedObject` ou `void`, torna-se disponível ao usuário. Métodos definidos dessa maneira e que não tenham nenhum argumento são exibidos no menu pop-up do objeto como ilustra o exemplo abaixo. O nome usado no menu default para o método é despojado de seu prefixo e possuirá espaços onde caracteres maiúsculos forem encontrados.

Métodos que tiverem um único parâmetro (que deve ser do tipo `NakedObject`) tornam-se disponíveis através do mecanismo de arrastar e soltar.

Quando um desses métodos de ação retorna um valor não nulo, o framework irá tentar exibir esse objeto retornado ao usuário – normalmente em uma nova janela. Isso é indicado no menu pop-up por reticências (...) adicionado ao nome, por exemplo, a opção *Return Booking...* exibida abaixo irá mostrar um novo objeto, já a opção *Check Availability* não.



Métodos de instância

Um método de ação típico (tirado da classe `City`) é mostrado abaixo. Ele cria um novo objeto `Location` e configura seu campo cidade para referenciar o objeto cidade que o chamou. Esse objeto recentemente criado e persistido é então retornado para o usuário. Esse método é apresentado como *New Location...* sobre o menu pop-up de cidade e é chamado quando esse item de menu é selecionado.

```
public Location actionNewLocation() {
Location loc = (Location) createInstance(Location.class);
loc.setCity(this);
return loc;
}
```

O próximo método (tirado da classe `Location`) cria um novo objeto booking. Esse método precisa de um outro objeto location, fornecido como argumento, tal que ele possa configurar tanto o campo pick up quanto o campo drop off do booking. Esse método é chamado soltando um objeto location sobre um outro.

```
public Booking actionNewBooking(Location location) {
Booking booking = (Booking) createInstance(Booking.class);
Customer customer = location.getCustomer();
booking.setPickUp(location);
booking.setDropOff(this);
if (customer != null) {
booking.setCustomer(customer);
booking.setPaymentMethod(customer.getPreferredPaymentMethod());
}
booking.setCity(location.getCity());
return booking;
}
```

No exemplo de tela abaixo, você pode ver o objeto arrastado, cujo título e rótulo foram mudados para cor púrpura; o objeto arrastado cujo título e rótulo está em cor verde para indicar uma operação válida; e o novo booking resultante com seus campos *Customer*, *Pick Up*, *Drop Off* e *City* configurados usando a informação dos dois objetos envolvidos no método chamado.

Com muita freqüência, métodos de ação são usados para mudar o estado de um objeto. Este exemplo da classe `Booking` muda o objeto valor `status`, então associa as localizações `pickUp` e `dropOff`, e o método `paymentMethod` mais recentemente confirmado é usado no booking para alterar diretamente o objeto cliente – tal que ele possa facilmente ser reutilizado num futuro booking daquele cliente. Note que este método não retorna nada; ele somente muda o estado dos dois objetos:

```
public void actionConfirm() {
getStatus().setValue("Confirmed");

getCustomer().associateLocations(getPickUp());
getCustomer().associateLocations(getDropOff());
if (getCustomer().getPreferredPaymentMethod() == null) {
getCustomer().setPreferredPaymentMethod(getPaymentMethod());
}
}
```

Desabilitando métodos

Métodos de ação podem estar indisponíveis ou por falta de autorização apropriada do usuário, ou porque o objeto não está num estado apropriado, ou devido a alguma outra regra de negócio. A disponibilidade de um método `action...` pode ser controlado adicionando um método `about...` correspondente. Este método deve retornar um objeto `About`, ter o mesmo nome do método com o prefixo 'about', e ter exatamente a mesma lista de parâmetros. O seguinte código mostra dois métodos `action...` e seus métodos `about...`. O primeiro torna o método `NewLocation` disponível se o campo `city` contiver alguma coisa. O segundo desabilita o método `NewBooking` se a localização usada para chamar o método de ação tiver a mesma localização do objeto que chamou, isso é, se o usuário tentar arrastar e soltar o objeto localização sobre ele mesmo.

```
public Location actionNewLocation() {}

public About aboutActionNewLocation() {
if(city.isEmpty()) {
return ActionAbout.DISABLE;
} else {
return ActionAbout.ENABLE;
}
}

public Booking actionNewBooking(Location location) {}

public About aboutActionNewBooking(Location location) {
return ActionAbout.disable(location.equals(this));
}
```

Quando um método `about...` desabilita um método de ação sem nenhum parâmetro, esse método fica na cor cinza no menu do objeto.

Quando ele desabilita um método com um parâmetro, o framework irá destacar o objeto com a cor vermelha quando o usuário tentar soltar um objeto sobre ele mesmo (mesmo que o tipo esteja correto).

Método de ordenação

A ordenação dos métodos listados, particularmente para itens de menu exibidos no menu pop-up de um objeto, baseia-se no vetor de métodos de ação que é produzido pelo mecanismo de reflexão Java. A ordem, assim, é dependente de como o JVM em execução coleta os dados sobre suas classes, e você não tem acesso controle direto sobre isso. Você pode, no entanto, especificar uma ordem dentro de sua definição de classe, que o mecanismo de visualização pode então interpretar. Isso é realizado adicionando um método estático chamado `actionOrder` que retorna uma `String` listando os nomes derivados dos métodos separados por vírgulas. O método precisa que os nomes sejam refletidos e não dos identificadores de métodos. Por exemplo, a string 'Call Back' seria usada para referenciar o método `actionCallBack`.

O seguinte exemplo da classe `Booking` especifica uma ordem adequada para os seus métodos de ação:

```
public static String actionOrder() {
return "Check Availability, Confirm, Copy Booking, Return Booking";
}
```

Qualquer nome listado que não corresponda a um método será simplesmente ignorado. Os métodos que não são listados serão colocados no final dos métodos especificados.

Métodos de classe

Também é possível escrever métodos que são chamados de uma classe ao invés de serem chamados de uma instância específica. Isso pode ser usado para encontrar instâncias que correspondam a um dado critério – por exemplo, para encontrar booking que usam uma localização específica; ou para criar um objeto baseado em um outro, tal como uma cidade sendo usada para criar uma nova localização naquela cidade.

O mesmo princípio se aplica na definição de um método, mas o método é declarado como estático. O código abaixo ilustra um método de classe típico (dentro da classe `Booking`) que cria um novo booking para um cliente. Ele poderia ser chamado soltando um objeto cliente sobre a classe *Bookings*, resultando num novo booking visualizado numa nova janela:

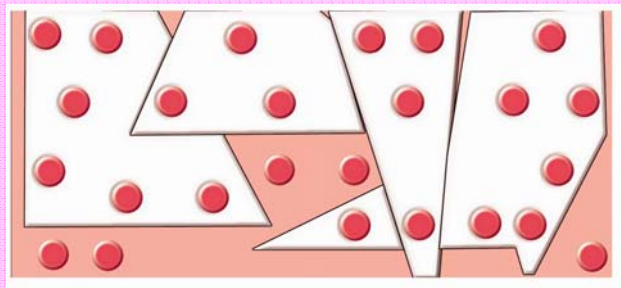
```
public static Booking actionNewBooking(Customer customer) {
Booking newBooking = (Booking) createInstance(Booking.class);
newBooking.setCustomer(customer);
newBooking.setPaymentMethod(customer.getPreferredPaymentMethod());
return newBooking;
}
```

Ordenando métodos de classe

Você pode ordenar métodos de classe da mesma forma que os métodos de instância, mas usando o método estático chamado `classActionOrder`.

Tornando objetos disponíveis ao usuário

Durante a prototipação você deseja traduzir a idéia de objeto de negócio tão rápido quanto possível dentro de uma forma que o usuário possa explorar. Para esse propósito o Naked Objects fornece uma classe simples chamada `org.nakedobjects.Exploration` que é usada para gerar um tipo de aplicação minimalista. Ele inicia o framework, permite que você registre as classes de negócio que você quer que o usuário veja, e então inicie a interface gráfica.



Uma aplicação Naked Objects não é um programa escrito com o propósito de se dedicar à interface com o usuário, ao invés disso, o propósito é de ser um 'portal' dentro do modelo de objetos de negócio – expressa um conjunto específico de classes de negócio, e possivelmente um conjunto específico de campos e comportamentos dentro dessas classes – disponibilizadas aos usuários num contexto de negócio amplo, porém definido. Em nosso exemplo de ECS, podemos ter um portal de agentes de booking, e um portal de despachantes que permite que um despachante de veículo compare as reservas confirmadas com os veículos disponíveis. Os despachantes devem receber acesso à classe Vehicle, mas não a classes PaymentMethod, ou a quaisquer métodos para gerar novas reservas.

Incorporando os naked objects dentro de uma simples aplicação

Seguindo o exemplo, o sistema ECS estende a classe `Exploration` para iniciar a exploração da aplicação. Aqui você pode ver o processo de registrar as classes de objetos de negócio (no método `classSet`) e como a aplicação é iniciada no método `main`:

```
package ecs.delivery;

import org.nakedobjects.Exploration;
import org.nakedobjects.object.NakedClassList;

public class EcsExploration extends Exploration {
public void classSet(NakedClassList classes) {
classes.addClass(Booking.class);
classes.addClass(City.class);
classes.addClass(Location.class);
classes.addClass(CreditCard.class);
classes.addClass(Customer.class);
classes.addClass(Telephone.class);
}

public static void main(String[] args) {
new EcsExploration();
}
}
```

Essa maneira de executar uma aplicação não faz uso de um mecanismo de persistência. Isso é intencional, pois facilita o teste da aplicação enquanto que as definições de objeto são rapidamente alteradas. É possível especificar um agente de persistência no teste de ambiente, mas pode ser que seja mais útil se a aplicação configurar um conjunto inicial de objetos conhecidos todas vez que a aplicação for iniciada.

Você pode fazer isso sobrecarregando o método `initObjects`, como apresentado no exemplo do ECS. Ele cria uma lista de sete cidades e dois clientes – uma demonstração mais realística precisaria de mais. (Os objetos devem ser criados via a classe `Exploration` caso contrário eles não seriam adicionados ao repositório de objetos (transientes) do framework e assim, não ficariam disponíveis ao usuário).

```
public void initObjects() {
String[] cities = {
"New York", "Boston", "Washington", "Chicago",
"Tampa", "Seattle", "Atlanta"
};

for (int i = 0; i < cities.length; i++) {
City newCity = (City) createInstance(City.class);
```

```
newCity.getName().setValue(cities[i]);
}

Customer newCustomer;

newCustomer = (Customer) createInstance(Customer.class);
newCustomer.getFirstName().setValue("Richard");
newCustomer.getLastName().setValue("Pawson");

newCustomer = (Customer) createInstance(Customer.class);
newCustomer.getFirstName().setValue("Robert");
newCustomer.getLastName().setValue("Matthews");
}
```

Ícones

As imagens de ícones são obtidas a partir de um diretório chamado images que está localizado dentro do diretório de trabalho – o diretório em que a linguagem Java está executando. Os arquivos de imagens são comparados com os nomes das classes com a adição de '16' ou '32' e uma extensão '.gif'. Por exemplo, City16.gif e City32.gif são as duas imagens da classe `City`. Os números se referem ao tamanho da imagem – 16x16 e 32x32 pixels respectivamente. O framework precisa de dois tamanhos de imagens.

Você pode também colocar essas imagens dentro dos recursos que serão disponibilizados ao JVM – colocá-los num arquivo '.jar' ao lado dos arquivos de classes quando você distribuir a sua aplicação. No entanto, eles devem ainda estar dentro de um subdiretório chamado images.

Criando uma demonstração padrão executável

Após escrever uma aplicação Naked Objects, seja como um protótipo exploratório ou como um sistema a ser entregue, é algumas vezes conveniente habilitá-lo para que seja compartilhado com outros na forma de uma demonstração executável independente e autocontida (uma que não precise de instalação a priori do framework, o gerenciamento do código fonte e de outros recursos, ou de qualquer serviço de infra-estrutura).

Para criar tal demonstração executável:

- Primeiro, assegure-se que você tenha criado uma aplicação usando a classe `org.nakedobjects.Exploration` como sua superclasse.
- Crie quaisquer instâncias requeridas programaticamente, como foi mostrado acima.

- Faça uma cópia do arquivo nakedobjects.jar fornecido no diretório lib da distribuição que você baixou. “Renomear” essa cópia de forma que ele indique que é da sua aplicação, por exemplo, booking-demo.jar.
- Crie um arquivo texto chamado manifest e coloque a seguinte linha dentro dele, mas especifique o nome completo da classe qualificada da sua aplicação.

  Main-Class: **ecs.delivery.EcsExploration**

- Extraia a classes de Log4J para que elas possam ser incluídas no seu novo arquivo jar.

  ```
  jar -xf log4.jar
  ```

- Atualize o novo arquivo jar tal que ele inclua suas classes e imagens da aplicação, as classes Log4J, e adicione os detalhes do arquivo manifest dentro do seu arquivo manifest. Seguindo o exemplo de booking, se as imagens estiverem no diretório images e todas as classes estiverem dentro do diretório ecs, então você irá executar o utilitário jar com as seguintes opções:

  ```
  jar -umf manifest booking-demo.jar images ecs org
  ```

- Agora, o único arquivo que será distribuído é o arquivo jar: booking-demo.jar. Para executar a aplicação usando a versão 1.2 ou posterior do Java, utilize o parâmetro `-jar`, por exemplo:

  ```
  java -jar booking-demo.jar
  ```

  Para facilitar, coloque essa linha dentro de um arquivo ‘.bat’ ou num script. De fato, se o Microsoft Windows estiver configurado apropriadamente, a aplicação poderá ser executada com um duplo-clique sobre o ícone do arquivo.

  Se uma das versões mais recentes do Java não estiver disponível, então a aplicação pode também ser executada usando a versão 1.1. No entanto, a biblioteca de classes Java terá que ser carregada manualmente e a classe de execução ser especificada da seguinte forma:

  ```
  java -cp <path-to-java>\lib\classes.zip;booking-demo.jar
  ecs.delivery.EcsExploration
  ```

  (Este comando deverá estar numa única linha).

Construindo um sistema multiusuário

Muitos sistemas de negócio exigirão que os naked objects sejam capazes de ser compartilhados entre os múltiplos usuários, e que tais objetos sejam persistentes. Essas características existem no framework e são brevemente explicados aqui. As funcionalidades descritas estão todas componentizadas e capazes de serem trocadas por implementações alternativas. Para mais informações sobre como configurar componentes alternativos, ou escrever um novo, você deve procurar na documentação fornecida dentro da distribuição do Naked Objects e em nosso web site.

Tornando os naked objects persistentes

Os naked objects são persistidos através de um repositório de objetos, o qual normalmente faz interface com um mecanismo de persistência tal como um banco de dados. Qualquer classe que implemente a interface `org.nakedobjects.object.NakedObjectStore` pode servir como um repositório de objetos. Em tempo de escrita, cinco repositórios de objetos alternativos estão disponíveis (ou como parte do framework ou acessíveis de nosso website) junto com a documentação sobre como utilizá-las:

- **Repositório de objetos XML**. Ele armazena cada instância do naked object como um arquivo no formato XML separado. Seu uso somente é adequado durante a prototipação, quando o número de instância é muito pequeno. Sua vantagem é que os formatos de arquivos podem facilmente ser lidos pelos programadores, assim como por outras ferramentas. Ele é razoavelmente robusto, mesmo quando ocorrerem mudanças nas definições de objetos durante a prototipação – você pode normalmente adicionar ou remover campos sem ter que alterar os arquivos de dados existentes.
- **Repositório de Objetos Seriados**. Ele armazena cada instância como um arquivo separado, usando a seriação do Java. Seu uso somente é adequado para prototipação ou para aplicações com pequeno volume de dados. No entanto, é mais eficiente que o repositório XML, especialmente para objetos complexos.
- **Repositório de Objetos SQL**. Ele funciona bem com qualquer banco de dados relacional compatível com o JDBC e é adequado para muitas aplicações de negócio. Se você está desenvolvendo uma nova e simples aplicação, então o Repositório de Objeto SQL pode gerenciar todo o processo de configuração do banco de dados e criar as tabelas necessárias de forma transparente: o programador pode simplesmente definir os objetos de negócio assumindo que eles serão armazenados e recuperados a partir de um banco de dados relacional quando necessário. Se a aplicação tiver que usar tabelas relacionais existentes, ou armazenar seus objetos numa

forma conhecida para que outras aplicações possam acessar, então um programador pode manualmente especificar uma série de objetos de mapeamento. A especificação manual dos objetos de mapeamento pode também ser necessária em aplicações muito grandes ou complexas, por razões de desempenho.

- **Repositório de Objeto EJB**. Este repositório de objeto foi escrito por Dave Slaughter do Safeway, e permite que uma aplicação Naked Objects tire vantqagens das capacidades do Enterprise Java Beans. O EJB fornece um apoio muito bom para escalabilidade, transações e segurança. Ao usar este repositório de objeto, o Naked Objects pode interoperar com outros sistemas construídos sobre essa infra-estrutura EJB.
- **Repositório de Objeto Transiente**. Ele fornece a funcionalidade de um repositório de objeto (que é requerido pelo framework para que ele possa ser executado), mas sem persistência, ou seja, os estados dos objetos são mantidos enquanto a aplicação estiver executando e não são salvos entre chamadas do framework. Ele é uma parte integrante do framework e é usado por default quando se executa uma aplicação `org.nakedobjects.Exploration`. Ele permite que definições de objetos sejam mudadas facilmente quando os naked objects são inicialmente manipulados. É conveniente também para testes, quando não se espera nenhum objeto quando a aplicação é iniciada.

Nós esperamos que com o tempo, muito mais repositórios de objetos estejam disponíveis ao Naked Objects, assim como versões melhoradas dos existentes.

Quando o framework é executado como um servidor, ele usa o repositório de objetos persistentes que está especificado no arquivo de configuração. A menos que você precise personalizar o mapeamento, é possível trocar o repositório de objetos apenas editando uma linha desse arquivo – nenhuma mudança precisa ser realizada no código dos naked objects. O arquivo de configuração default fornecido como parte da distribuição especifica o Repositório de Objeto XML.

Como um desenvolvedor, existem três maneiras que você pode usar um repositório de objetos:

- Você pode especificar um repositório de objetos existente e usá-lo de forma transparente.
- Você pode especificar um repositório de objetos existente e fornecer seu próprio mapeamento ou outra forma de controle personalizado sobre a persistência.
- Você pode escrever um novo repositório de objetos, normalmente como um pacote de um banco de dados existente ou outros mecanismos de persistência. (Os requisitos para um repositório de objetos são detalhados no API do framework e discutidos em nosso website).

O framework pressupõe que qualquer mudança que o usuário faça em qualquer objeto, seja pela alteração direta de seus atributos ou associações, ou pela chamada de um método de ação, é imediatamente tornada persistente. Por essa razão, os naked objects em nossos exemplos não possuem ações explícitas de 'salvar' ou 'tornar persistente'. O projeto do framework Naked Objects confia que seja possível seguir essa abordagem em muitas aplicações e ainda atingir níveis desejados de desempenho. A rapidez é também consistente com o estilo expressivo dos sistemas que o Naked Objects produz.

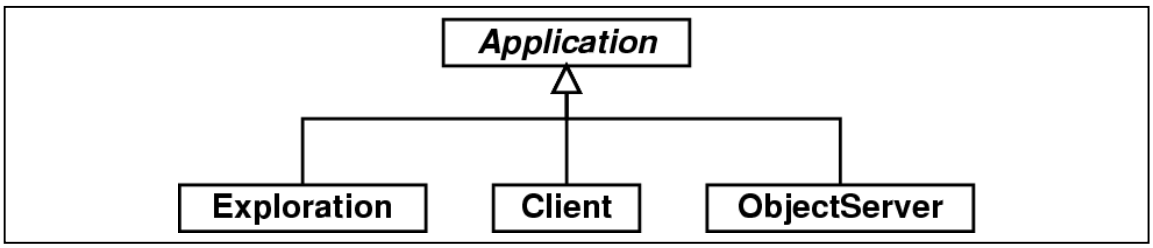
É possível postergar o processo de tornar um objeto persistente, criando objetos transientes e subseqüentemente tornando-o persistente. Por exemplo, um objeto pedido pode ser criado e preparado garantindo que todos os itens estejam disponíveis antes de adicionar o objeto ao repositório de objetos. Infelizmente, uma vez que o objeto tenha se tornado persistente, quaisquer outras mudanças realizadas nesse objeto serão persistidas imediatamente.

Numa futura atualização, o framework também atenderá o salvamento explícito de objetos, onde isso se fizer necessário, seja por razões de desempenho ou outros motivos.

Compartilhando Naked Objects entre múltiplos usuários

É possível executar o Naked Objects como um conjunto de aplicações discretas, cada uma ligada diretamente ao mecanismo de persistência via um repositório de objetos dentro do cliente. Isso é conhecido como arquitetura 'cliente gordo'. Isso não é recomendado a menos que você precise preservar a compatibilidade com uma arquitetura de cliente gordo existente, ou sua aplicação envolva um muito processamento sobre poucas instâncias de objetos.

Normalmente, seus clientes irão se comunicar com um servidor de objetos compartilhado, que por sua vez fala com o repositório de objetos para acessar o mecanismo de persistência. As classes Naked Objects `org.nakedobjects.Client` e `org.nakedobjects.ObjectServer` são subclasses de `org.nakedobjects.Application` (como é a classe `org.nakedobjects.Exploration` usada durante a prototipação).



Os termos ‘cliente’ e ‘servidor’ não tem nenhuma implicação com a localização física. Uma configuração comum é ter os clientes executando sob a plataforma do usuário e o servidor executando numa máquina central. Mas é também possível executar tanto o cliente quanto o servidor sobre um computador PC para propósitos de prototipação. Igualmente, você pode executar alguns clientes e o servidor sob uma plataforma compartilhada, com clientes comunicando com a plataforma do usuário via um servidor web. Assim, o cliente e servidor poderiam ser entendidos tão somente como duas camadas de uma arquitetura n-camadas.

Quando um servidor de objetos é utilizado, cada naked object usado nos clientes será replicado a partir do servidor. O framework mantém as duas réplicas sincronizadas. Se dois usuários estiverem manipulando (ou apenas visualizando) o mesmo objeto, então as duas réplicas cliente daquele objeto estarão ligadas ao mesmo objeto que está no servidor. Se um usuário alterar um objeto, então as alterações serão imediatamente propagadas para todas as réplicas cliente. (Esta é a razão do porquê é importante incluir a declaração `objectChanged` em seus métodos).

Tudo isso acontece de forma transparente. O desenvolvedor não tem que escrever qualquer código específico para gerenciar a replicação. Isto causa a sensação de que os naked objects existem em apenas um lugar. A configuração disso pode ser tão simples quanto executar o cliente em cada plataforma do usuário e avisá-lo da localização do servidor e depois, especificar o conjunto de definições das classes naked objects que serão disponibilizados. Tudo isso é feito no arquivo de configuração. Numa futura atualização, será possível configurar o servidor tal que o cliente possa obter a informação de configuração de classes e os arquivos necessários de classes e imagens diretamente desse servidor – através disso elimina-se a necessidade de atualizar manualmente os clientes quando novas classes naked objects forem introduzidas.

O cliente e o servidor Naked Objects são aplicações completas. Os objetos replicados são totalmente funcionais: o código de cada um dos métodos de objetos existe tanto na réplica cliente quanto na do servidor. É possível executar todas as funcionalidades de negócio tanto no objeto cliente quanto no objeto servidor. No entanto, se o cliente souber (através do arquivo de configuração) que ele tem um servidor, então ele irá automaticamente subcontratar a execução dos métodos da versão que está no servidor. Isso assegura que qualquer mudança realizada nos objetos seja persistida e que todos os clientes que usam esse objeto sejam imediatamente notificados. Em teoria, seu mecanismo de persistência poderia realizar isso com funções diretas de alertas, mas isso provavelmente necessitaria muita programação manual. O framework Naked Objects assegura que esses alertas e atualizações automáticas ocorram independentemente do mecanismo de persistência que você estiver usando. A segunda razão para delegar a execução dos métodos para a réplica que está no servidor é que ele provavelmente estará mais próximo do repositório de objetos e assim, produzir alto desempenho se o método envolver operações de recuperação e atualização de objetos persistentes.

Mantendo a integridade transacional

Se o seu mecanismo de persistência tiver a habilidade de gerenciar transações (como em muitos bancos de dados relacionais, por exemplo), então o Naked Objects pode fazer total proveito disso. Por default, assume-se que todas as ações explícitas do usuário, tais como a de selecionar uma ação a partir do menu pop-up, alterar um campo, ou arrastar e soltar um objeto, sejam transações atômicas mesmo que a ação dispare mudanças em múltiplos campos ou em outros objetos. Quando cada ação ocorre, o framework automaticamente envia uma mensagem 'start transaction' para o repositório de objetos, e o conclui com uma mensagem 'end transaction'.

Fornecendo segurança e autorização

Em tempo de escrita, o Naked Objects não tem um modelo explícito para segurança e autorização. Os objetos e métodos `About` permitem que o desenvolvedor controle o acesso em qualquer classe, método ou atributo específico. Isso fornece a base sobre o qual um modelo compreensivo de segurança/autorização pode ser construído. Nós esperamos que tal modelo fique disponível no Naked Objects num futuro próximo. Por hora, se você desejar usar um servidor de autenticação/autorização, então você precisará estender a classe `org.nakedobjects.object.control.About` tal que ela delegue tal responsabilidade para esse servidor.

Executando um sistema multiusuário

Executar Naked Objects num ambiente multiusuário é tão simples quanto executar o programa servidor e então executar alguns clientes. Como os clientes normalmente são executados em máquinas separadas, eles precisam saber conde encontrar o servidor. Eles também precisam saber quais classes naked objects estão disponíveis ao usuário.

Executando o servidor

A classe `org.nakedobjects.ObjectServer` do framework é usado para executar o framework como um servidor e é iniciado usando o seguinte comando:

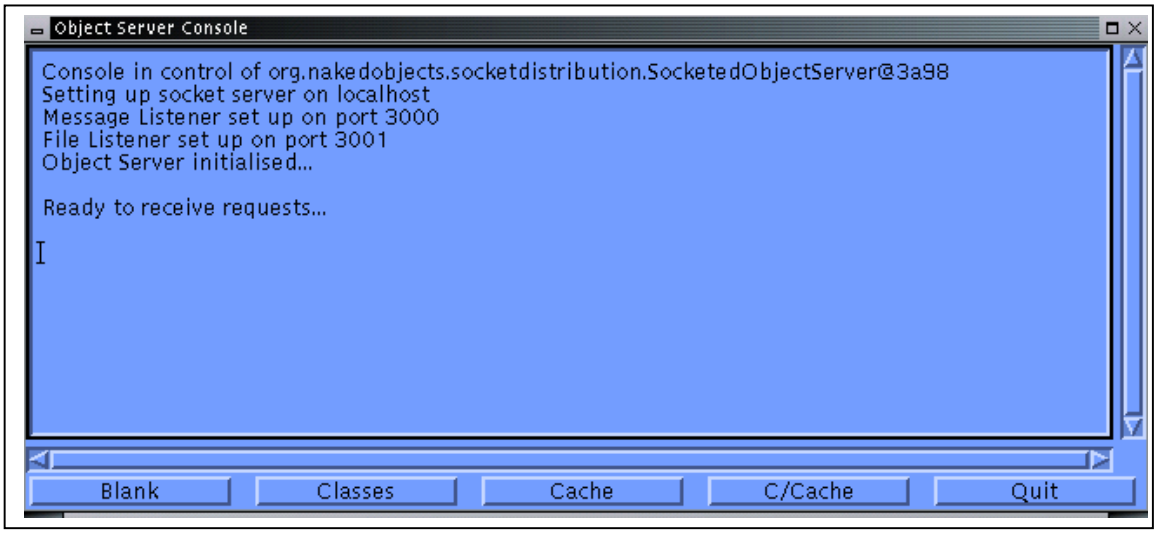
```
java -cp nakedobjects.jar:log4j.jar:<object class path>
                    org.nakedobjects.ObjectServer
```

(Este comando deve estar numa única linha).

O Log4J é um framework de log do Apache, que é usado para registrar a execução do framework. O arquivo Log4J.jar, o framework em si, e o diretório contendo as classes naked objects, devem todos possuir o classpath, a menos

que eles já estejam especificados na variável CLASSPATH do sistema. Além disso, uma cópia do 'server.properties' deve estar no diretório de trabalho, isto é, o diretório onde o comando acima é executado. Esse arquivo pode ser encontrado no diretório 'conf' da distribuição Naked Objects.

Assumindo que o arquivo de configuração original 'server.properties' tenha sido copiado, quando o servidor iniciar, ele irá instalar o repositório de objetos default, um serviço básico de rede, e um console gráfico para monitorar e controlar o servidor. Todos esses aspectos do servidor podem ser alterados. Detalhes completos de configuração foram incluídos na distribuição do Naked Objects. A seguinte tela de console permite que você monitore os clientes desative o servidor.



Executando o cliente

Antes de executar o cliente, você deve especificar onde o servidor está e dizer ao cliente quais as classes que estarão disponíveis ao usuário. Isso é feito editando o arquivo de configuração do cliente, 'client.properties'. Esse arquivo, que está no diretório 'conf' da distribuição do Naked Objects, deve ser copiado para o diretório de trabalho. A linha:

```
nakedobjects.socketed-proxy.address=localhost
```

deve ser alterada, trocando o localhost com o endereço de IP do servidor ou pelo seu host name. Por exemplo, em nosso servidor local a linha ficaria:

```
nakedobjects.socketed-proxy.address=192.168.1.8
```

As classes naked objects que são disponibilizadas ao usuário são especificadas anexando os nomes das classes completamente qualificadas, separados por ponto e vírgulas, à linha:

```
nakedobjects.classes=
```

Por exemplo, para configurar o cliente para que use as classes desenvolvidas para a aplicação ECS, as seguintes linhas deveriam ser vistas no arquivo de configuração. (As barras invertidas no final de cada linha indicam que a propriedade é continuada na próxima linha).

```
nakedobjects.classes=\
                org.nakedobjects.example.booking.Customer;\
                org.nakedobjects.example.booking.Booking;\
                org.nakedobjects.example.booking.Location;\
                org.nakedobjects.example.booking.Telephone;\
                org.nakedobjects.example.booking.CreditCard;\
                org.nakedobjects.example.booking.City
```

Antes de executar o cliente, o diretório de imagens e seu conteúdo, devem ser copiados para o diretório de trabalho. Os arquivos de classe necessários devem também ser copiados e colocados no diretório de trabalho (idealmente num subdiretório específico).

A classe `org.nakedobjects.Client` do framework é usado para executar o framework como uma aplicação cliente. O caminho da classe é configurado da mesma forma que o servidor, tal que o framework, Log4J e as classes naked objects possam ser encontrados:

```
java -cp nakedobjects.jar:log4j.jar:<object class path>
                org.nakedobjects.Client
```

(Este comando deve estar numa única linha).

A aplicação terá a seguinte aparência.

Enriquecendo comportamentos de objetos

Tendo visto a anatomia básica dos naked objects e como disponibilizar aos usuários, devemos agora construir mais funcionalidades dentro dos métodos desses objetos, muitos dos quais relacionados com a manipulação de atributos e associações de objetos. Como antes, você deve adotar algumas conversões simples em seu código Java para esses métodos a fim de permitir que o framework realize sua tarefa. No entanto, devemos também ver que o framework fornece várias características úteis que facilitam a tarefa de escrever seus métodos de negócio.

Acessando campos com segurança

Na programação orientada a objetos, considera-se uma boa prática, sempre acessar uma variável interna através de métodos `get...` e `set...`. Em Naked Objects isso é especialmente importante quando lidamos com variáveis que envolvem outros objetos de negócio. Os métodos de acesso cuidam dos assuntos de persistência e notificação, tal que o acesso a uma variável através de seus métodos `get...`, assegura que o objeto será apropriadamente carregado e que mudar uma variável através de seus métodos `set...`, assegura que as mudanças

serão persistidas e que qualquer visualização desse objeto, seja na tela do usuário ou sobre telas de usuários, permanecerá consistente.

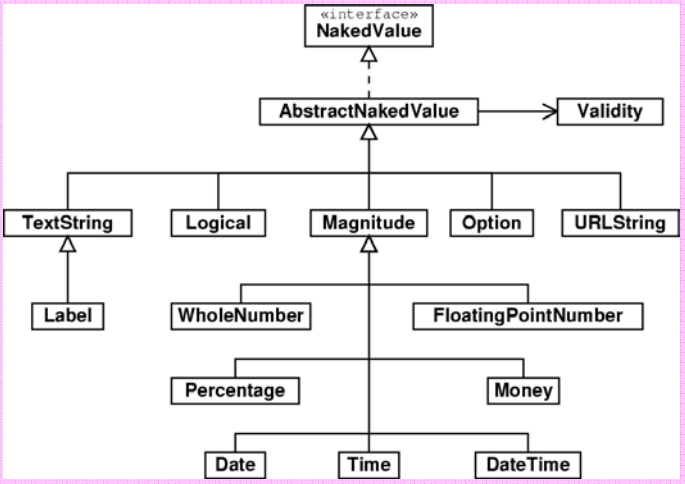
O seguinte método da classe `Booking` mostra essa prática. Cada associação é realizada como um `set...`, enquanto que cada acesso é realizado usando um `get...`. Note também que o campo cliente é configurado usando um método `associate...` para configurar também a associação na classe `Customer`.

```
public Booking actionCopyBooking() {
    Booking copiedBooking = (Booking) createInstance(Booking.class);
    copiedBooking.associateCustomer(getCustomer());
    copiedBooking.setPickUp(getPickUp());
    copiedBooking.setDropOff(getDropOff());
    copiedBooking.setPaymentMethod(getPaymentMethod());
    copiedBooking.setContactTelephone(getContactTelephone());
    return copiedBooking;
}
```

Os objetos de valor e as coleções internas não precisam ser acessados estritamente via seus métodos de acesso, já que eles tomam cuidado com os assuntos de notificação e persistência. No entanto, é uma boa prática sempre usar os métodos de acesso, pois fornece consistência ao seu código.

Manipulando objetos de valor

Os objetos de valor são usados para manipular valores de dados simples e genéricos (tais como datas, strings, caracteres, ou temperaturas) que pertençam a um único objeto de negócio. Todos os objetos de valor devem implementar a interface `org.nakedobjects.object.NakedValue`.

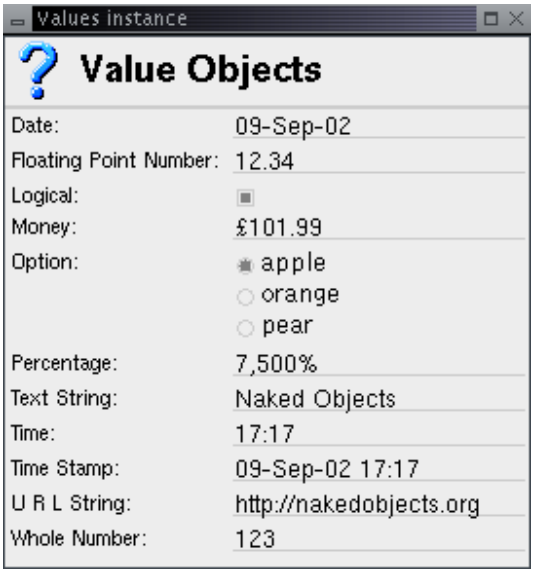


Todos os valores devem implementar a interface `NakedValue`, que é normalmente realizada estendendo `AbstractNakedValue`. A classe abstrata `Magnitude` fornece a habilidade principal de comparar objetos de valor com base em valores numéricos.

Os seguintes valores de objetos são identificados dentro do framework como subclasses da classe `org.nakedobjects.object.value.AbstractNakedValue` e novos tipos podem ser adicionados se necessário. Todos são auto-explicativos exceto a classe `Option`, que permite que o objeto assuma um valor de um conjunto de valores numéricos. O atual mecanismo de visualização retrata isso aos usuários como um conjunto de 'radio buttons' ou uma caixa de seleção 'drop-down', dependendo de quantas opções existirem. As seguintes classes são membros do pacote `org.nakedobjects.object.value`:

- `Date` - 3-Jun-02 ou 21/3/01
- `FloatingPointNumber` - 1,234.5 ou 0.125
- `Logical` - set to True ou False
- `Money` - £5,120.50 ou $10.99
- `Option` - Saloon|SUV|Minivan|Coupe
- `Percentage` - 0.1% ou 99.99%
- `TextString` - Alan McDonald
- `Time` - 10:50 AM ou 14:15
- `TimeStamp` - 3-Jun-02 10:50 AM ou 21/3/01 14:15
- `URLString` - http://www.nakedobjects.org/downloads.html
- `WholeNumber` - 18 ou 1,200

A tela abaixo apresenta e exemplifica como o atual mecanismo de visualização atua em cada um desses objetos de valor para o usuário:



Cada uma dessas classes possui três construtores comuns e um conjunto principal de métodos. O construtor sem nenhum parâmetro cria o objeto com seu valor default. Os defaults são os valores zero para tipos numéricos, a data e hora atual para tipos temporais, e uma string vazia para o tipo string. O construtor de um parâmetro recebe ou um objeto existente do mesmo tipo e copia o seu valor,

ou um tipo Java convencional que pode prontamente ser mapeado no valor do tipo de objeto. Os principais métodos são:

- `public Title title()` retorna um título string – como um objeto `Title` – formatado de acordo com o padrão `Locale` de Java.
- `public boolean isEmpty()` determina se o objeto está vazio: ele não contém um valor.
- `public void clear()` limpa o valor para que ele fique vazio.
- `public void reset()` redefine o objeto para o seu valor default.
- `public boolean isValid()` determina se um valor é atualmente válido de acordo com a estratégia `Validity` do objeto. Por exemplo, a estratégia `PositiveValue`, aplicada a um objeto `WholeNumber`, assegura que o número seja sempre positivo.
- `public void setValidity(Validity strategy)` associa uma estratégia `Validity` específica.
- `public void setAbout(About newAbout)` associa uma estratégia `About`. Isso é usado para controlar a acessibilidade do objeto de valor.
- `public void parse(String text) throws ValueParseException` tenta converter a `String` Java especificada para o tipo do objeto de valor.

Além desses métodos comuns, cada tipo tem vários métodos `typeValue` e `setValue` que aceitam e retornam vários tipos de dados relacionados. Por exemplo, a classe `Money` inclui os métodos `doubleValue` e `intValue` para converter o valor monetário em tipos `double` e `int` do Java. Ele também tem `setValue(double)` e `setValue(Money)` para configurar o valor usando valor `double` do Java ou de um outro objeto `Money`. Abaixo estão todos os métodos de conversão da classe `Money` seguido por um exemplo de seus usos:

```
public short shortValue();
public int intValue();
public long longValue();
public float floatValue();
public double doubleValue();

public void setValue(double amount);
public void setValue(Money value);

Money m = new Money();
m.setValue(8.4);
float f = m.floatValue();
```

Os tipos numéricos também incluem um conjunto de métodos para executar operações aritméticas básicas. Assim, por exemplo, um `FloatingPointNumber` pode ser adicionado a um outro `FloatingPointNumber` ou um `double` Java usando um método `add` sobrecarregado. Eles são fornecidos para simplificar a codificação de operações matemáticas, evitando a necessidade de conversão repetidas vezes entre tipos primitivos Java e tipos `NakedValue`. Apresentam-se a seguir as quatro operações básicas (sobrecarregadas para aceitar valores `double` e objetos

`FloatingPoint`) que estão disponíveis para objetos `FloatingPoint` seguidos por exemplos de suas utilizações:

```
public void add(double value);
public void add(FloatingPointNumber number);
public void subtract(double value);
public void subtract(FloatingPointNumber number);
public void multiply(double value);
public void multiply(FloatingPointNumber number);
public void divide(double value);
public void divide(FloatingPointNumber number);

FloatingPointNumber f = new FloatingPointNumber();
f.setValue(1.5);
FloatingPointNumber g = new FloatingPointNumber();
g.setValue(1.5);
f.add(g);
f.multiply(2.0);
double d = f.doubleValue();
```

Além das funções aritméticas, os tipos numéricos também possuem métodos para comparação com outras instâncias. Assim, por exemplo, um objeto `Percentage` pode ser comparado a um outro objeto `Percentage` para ver se ele é maior usando o método `isGreaterThan`. Esses métodos são conjuntamente implementados nas classes de objetos de valor e a classe `Magnitude` a partir da qual eles são estendidos. No código a seguir apresentam-se todos os métodos de `Magnitude`, dos quais `isEqualTo` e `isLessThan` devem ser implementados pela subclasse:

```
public boolean isEqualTo(Magnitude magnitude);
public boolean isLessThan(Magnitude magnitude);
public boolean isLessThanOrEqualTo(Magnitude magnitude);
public boolean isGreaterThan(Magnitude magnitude);
public boolean isGreaterThanOrEqualTo(Magnitude magnitude);
public boolean isBetween(Magnitude minMagnitude, Magnitude maxMagnitude);
public Magnitude max(Magnitude magnitude);
public Magnitude min(Magnitude magnitude);
```

A classe `TextString`, que é provavelmente o objeto de valor mais utilizado, fornece métodos tais como `contains`, `endsWith`, `isSameAs` que pode ser chamado tanto de maneira *case sensitive* quando *case insensitive*. Todas as coparações no exemplo abaixo retornarão `true`:

```
public boolean isSameAs(String text);
public boolean isSameAs(String text, Case caseSensitive);
public boolean contains(String text);
public boolean contains(String text, Case caseSensitive);
public boolean startsWith(String text);
public boolean startsWith(String text, Case caseSensitive);
public boolean endsWith(String text);
public boolean endsWith(String text, Case caseSensitive);

TextString t = new TextString();
```

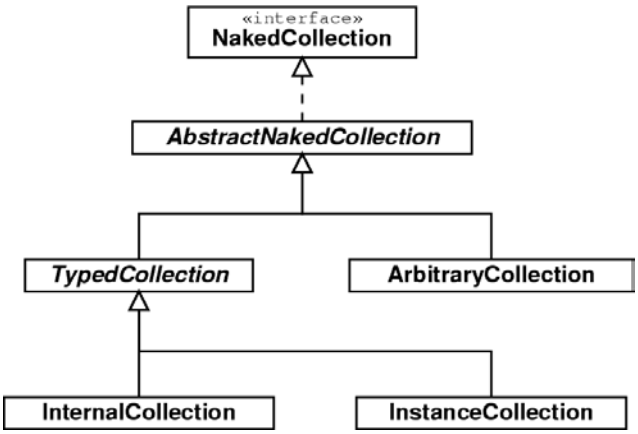
```
t.setValue("Form 1");
t.isSameAs("Form 1");
t.startsWith("form", Case.INSENSITIVE);
t.contains("1");
```

Todas as classes de valor e seus métodos estão totalmente catalogados na especificação da API, que pode ser encontrada na distribuição do Naked Objects.

Manipulando coleções

As coleções dentro do framework Naked Objects são utilizadas para armazenar vários objetos naked objects num único conjunto não ordenado. Algumas coleções são também tipificadas tal que elas somente aceitem um subtipo específico de `org.nakedobjects.object.NakedObject` – ou seja, um tipo específico de objetos de negócio. Existem três tipos de coleções usados dentro do framework (declarado no pacote `org.nakedobjects.object.collection`). Todos são baseados na mesma coleção abstrata, mas diferem no tipo de objeto que eles podem armazenar e na razão de eles serem usados. De acordo com a freqüência de uso, são eles:

- `InternalCollection`, uma coleção de associações, que é uma parte que compõem de um objeto de negócio. Os objetos que ela contém são do tipo especificado quando a coleção é criada. Isso foi descrito na seção associações um-para-muitos.
- `ArbitraryCollection`, uma coleção genérica não-tipada de objetos que o usuário cria para seus próprio uso. Esse tipo de coleção é criado quando alguma coleção existente é copiada.
- `InstanceCollection`, uma coleção de instâncias de um tipo específico, gerado diretamente pelo mecanismo de persistência. Por exemplo, na visão do usuário, o menu pop-up da classe oferece a opção de listar todas as instâncias da classe, ou encontrar instâncias que combinem critérios dados. Se a combinação não for encontrada eles irão ser retornados como uma `InstanceCollection`.



As superclasses abstratas `AbstractNakedCollection` fornecem métodos para adicionar e remover elementos de uma coleção. Você acessa os elementos da coleção através de uma `Enumeration`, que dá acesso a todos os objetos da coleção tal que você possa iterar sobre ela. Existem também alguns métodos utilitários que verificam o tamanho e o conteúdo de uma coleção.

- `public void add(NakedObject element)` adiciona um novo elemento para a coleção, primeiro chamando o método abstrato `canAdd` da coleção e verificando se o objeto pode ser adicionado. A `InternalCollection` implementa o `canAdd` tal que apenas objetos do tipo especificado, e que ainda não estejam na coleção, possam ser adicionados. O mesmo método na classe `ArbitraryCollection`, verifica somente a existência do objeto. A `InstanceCollection`, por outro lado, cancela qualquer adição realizada pelo programador.
- `public void remove(NakedObject element)` remove um elemento específico existente. Da mesma forma que o método de adição, o `remove` chama o método `canRemove` para verificar se o objeto existe e determinar se todos os elementos podem ser removidos exceto em `InstanceCollection` onde isso nunca é permitido.
- `public Enumeration elements()` gera uma `Enumeration` que pode ser usada para iterar através da coleção, como exemplificado abaixo:

```
Enumeration e = employees.elements();
while(e.hasMoreElements()){
        Employee emp;
        emp = (Employee)e.nextElement();
        :
        :
}
```

- `public boolean contains(NakedObject element)` verifica se uma elemento específico existe na coleção, retornando `true` se existir.
- `public boolean isEmpty()` determina se a coleção está vazia e retorna `true` se estiver.
- `public int size()` determina a quantidade de elementos existentes na coleção.

Criando objetos persistentes

Devido à maneira de como os objetos são persistidos dentro do framework, operadores `new` do Java não devem ser usados para criar novos naked objects. Ao invés disso, crie um novo objeto persistente usando o método utilitário `createInstance` fornecido pela superclasse `org.nakedobjects.object.AbstractNakedObject`. Como a maioria dos métodos de fábrica (factory) ele retorna uma instância usando uma referencia da superclasse – nesse caso, `org.nakedobjects.object.NakedObject` – e então deve ser convertido (usando cast) para o seu verdadeiro tipo antes que ele possa ser usado. Esse método toma um objeto `java.lang.Class`, que especifica qual classe de naked objects será criado. O exemplo abaixo cria um objeto `Location`:

```
Location home;
home = (Location) createInstance(Location.class);
```

Algumas vezes é necessário criar naked objects não-persistentes (ou 'transientes'), seja para usar fora do espaço de objetos compartilhados ou como parte de um longo processo de criação, onde o usuário colocará o objeto posteriormente no espaço compartilhado. Um objeto transiente pode ser criado usando o método utilitário `createTransientInstance`. Ele irá retornar um objeto transiente devidamente inicializado. Como no método `createInstance`, ele irá precisar que sua referência seja convertida (cast) para o seu tipo real. Note também que é ilegal (porque não dizer ilógico) que qualquer objeto persistente referencia um objeto transiente. Assim, não tente configurar um campo dentro de um objeto persistente usando esse método.

Ao chamar `makePersistent` de um objeto transiente fará com que o objeto, e todos os objetos nele contidos, persistam. O seguinte exemplo mostra um objeto não-persistente sendo criado e então sendo persistido:

```
Location away;
away = (Location) createTransientInstance(Location.class);
   :
   :
away.makePersistent();
```

O seguinte método exemplifica a criação de um objeto tomado da classe `City`:

```
public Location actionNewLocation() {
   Location loc = (Location) createInstance(Location.class);
   loc.setCity(this);
   return loc;
}
```

Como esses dois métodos estão disponíveis, deve-se evitar usar o operador `new`. Se, no entanto, você não tiver classes derivadas de `org.nakedobjects.object.AbstractNakedObject`, então você terá que criar um

objeto dessa maneira. Assim, é de sua responsabilidade chamar o método `created`, bem como chamar o método `makePersistent` se o objeto tiver que ser persistido.

Inicializando objetos persistentes

Quando certos tipos de objetos lógicos são criados pela primeira vez, você precisará dar aos seus campos de valor valores iniciais específicos ou associar outros objetos específicos. Em Java isso poderia ser feito dentro de construtores o qual é chamado somente uma vez durante a existência do objeto. No entanto, no framework Naked Object, sempre que um objeto lógico é recuperado do repositório de objetos, ele é reinstalado usando o construtor de classe. Isso significa que o construtor será chamado mais que uma vez durante a 'vida' do naked object. Você pode sem querer redefinir todos os campos toda a vez que um objeto é recuperado do repositório, sobrescrevendo outras mudanças.

Assim, a inicialização do objeto lógico não deve ser realizada no construtor, mas sim no método `created`. Esse método é chamado somente pelo framework quando um objeto é criado, usando os métodos fábrica que nós vimos na última seção. Isso é o que verdadeiramente acontece quando o usuário seleciona a opção de menu *New Instance...*

O seguinte código inicializa o campo *quantity* quando o objeto lógico é instanciado. O objeto de valor já deve existir nesse ponto, podendo ter sido inicializado durante a declaração da variável ou dentro do construtor do objeto.

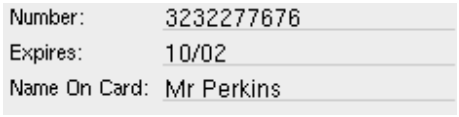
```
public void created() {
    quantity.set(5);
}
```

Isso não significa que o construtor nunca deve ser usado. Os objetos de valor e coleções internas ainda precisam ser criados antes que eles possam ser acessados, inclusive todas as vezes que um objeto é recriado a partir do repositório persistente.

Construindo um título a partir de múltiplos campos

Todos os objetos `org.nakedobjects.object.Naked` têm um título que pode ser usado quando o objeto é apresentado ao usuário. Esses títulos são gerados pelo método `title`, que retorna um objeto `org.nakedobjects.object.Title` contendo um título textual.

Para objetos de valor, ele é uma versão formatada do valor, por exemplo, "10:50 am" ou "£3.50". Para objetos de negócio o título é alguma forma de identificador.



Se um objeto de negócio tiver um campo que identifique unicamente a sua instância, tal como o número de um pedido, então o método `title` pode simplesmente obter o `Title` do objeto a partir desse campo de identificação. O seguinte método, da classe `City`, faz com que o título seja o objeto valor nome para cada cidade.


```
public Title title() {
   return name.title();
}
```

Em outros casos um título do objeto de negócio será criado derivado de vários atributos. O objeto `Title` fornece um conjunto de construtores e métodos úteis que auxiliam na criação de títulos a partir de objetos `String`, ou de outros naked objects (normalmente aqueles objetos que estão dentro de seus próprios campos), e de outros objetos `Title`. Por exemplo, para a classe `Customer`, nós poderíamos criar o título a partir do nome e sobrenome. Um objeto `Title` pode anexar vários objetos, na verdade pode sobrecarregar estilos, convertendo números, strings e outros naked objects para strings antes de anexá-los, e apresentar todos os espaçamentos para você. Considere o seguinte método:

```
public Title title() {
   Title title = firstName.title();
   Title fullTitle = title.append(lastName);
   return fullTitle;
}
```

Assumindo que o valor contido em `firstName` é 'Robert' e no `lastName` é 'Matthews', então a chamada `append` irá resultar no título 'Robert Matthews'. Se, no entanto, `firstName` estiver vazio então o título irá corretamente exibir 'Matthews' (sem nenhum espaço indesejado). Da mesma forma, se `lastName` estiver vazio, então o resultado será 'Robert' sem nenhum espaço indesejado.

Objetos `String` simples podem também ser anexados para formatações adicionais. Mudando o exemplo acima:

```
public Title title() {
                    Title title = lastName.title();
                    Title fullTitle = title.append(",", firstName);
                    return fullTitle;
}
```

altera a ordem do nome e sobrenome e produz as três variações corretas: 'Matthews, Robert', 'Matthews' e 'Robert'. A string de junção é adicional ao espaço delimitador.

(Se você não quer espaços entre os elementos de um título, use o método `concat` ao invés de `append`).

Uma outra coisa útil é lembrar que cada um desses métodos retorna o objeto que requerido. Isso permite aos métodos `append` e `concat` sejam encadeados, encurtando o seu método `title`. Essa é a versão resumida do código anterior:

```
public Title title() {
   return lastName.title().append(",", lastName);
}
```

Lembre-se, também, que não podemos garantir que as variáveis de referência a outros objetos de negócio referenciem alguma coisa (eles poderiam ser `null`) ou não serem resolvidos (eles não contêm a informação requerida). Por essa razão é importante não usar uma variável de referência diretamente, e verificar antes se a referência não é `null` antes de usá-las. Esta é uma outra razão do motivo pelo qual você deve usar o método `get...` de campo até para operações internas. Você pode então usar a referência retornada (`null` ou não) como um parâmetro para um dos construtores ou métodos de `Title`. Dessa maneira, a classe título irá verificar a referência e se ela estiver vazia, tratará de maneira natural sem lançar um `NullPointerException`.

Os métodos `Title` que tomam o valor `String` default como seu último parâmetro irá exibir esse valor quando a referência passada é `null`. O seguinte exemplo demonstra isso. Esse método irá funcionar mesmo que o campo `Order` não tenha um valor e mesmo que o dado do objeto `Order` não tenha sido carregado ainda.

```
public Title title() {
   return new Title(getOrder(), "New Order");
}
```

Especificando métodos About para controlar o acesso

Nós vimos vários exemplos de controlar acesso a classes, objetos, campos e método, usando um método `about...` que corresponda à classe, objeto, métodos de acesso ou métodos de ação. Todos esses métodos retornam um objeto `org.nakedobjects.object.control.About`. O objeto `About` fornece quatro peças de informação sobre a classe, objeto, campo ou métodos para uso pelo mecanismo de visualização ou qualquer outro serviço. São elas:

- Se ele deve ser acessível ao usuário corrente.
- Se ele pode ser usado enquanto o objeto estiver no seu estado atual.
- O nome dele deve ser conhecido (se ele precisa diferenciar o nome que é automaticamente gerado pelo framework do nome da classe ou método).
- Uma descrição de classe, objeto ou método que ele está controlando.

Todos os objetos `About` são derivados da interface `About`. Essa interface declara quatro métodos que representam a informação acima mencionada:

- `canAccess`
- `canUse`
- `getName`
- `getDescription`

Os dois últimos retornam objetos `String`, enquanto que os dois primeiros retornam um objeto `org.nakedobjects.object.control.Permission`. Um objeto `Permission` é usado ao invés de um simples boolean porque além de especificar se o acesso/uso é permitido ou desabilitado, ele pode também fornecer uma razão, na forma de uma mensagem textual. Isso foi feito para simplificar uma vez que a classe `Permission` é subclassificada como `org.nakedobjects.object.control.Allow` e `org.nakedobjects.object.control.Veto`, que simplesmente guarda uma `String` de razão e indica o estado pelo seu tipo.

Ao usar a interface `About` você pode definir sua própria classe `About` e exercer controle completo sobre seus objetos. No entanto, as classes `About` somente-leitura fornecidas pelo framework serão suficientes para a maioria dos propósitos.

Objetos About somente-leitura

A maneira mais simples de aplicar controle é usar objetos `About` somente-leitura. Todas as classes `About` básicas (tais como `org.nakedobjects.object.control.ClassAbout`, `org.nakedobjects.object.control.FieldAbout` e `org.nakedobjects.object.control.ActionAbout`) têm restrições de disponibilização pública (tais como `UNINSTANTIABLE`, `READ_ONLY` e `ENABLE`, que determinam a usabilidade das classes, campos e métodos respectivamente).

Para fornecer alguma flexibilidade, essas classes também possuem métodos estáticos que fornecem um objeto `About` baseado num flag que você fornece. Por exemplo, `ActionAbout` tem métodos `enable` que tomam um `boolean` e retornará o objeto `ENABLE` se o flag é `true` e o objeto `DISABLE` se `false`. Existe também um método complementar chamado `disable`. Isso nos permite desabilitar um método sob uma condição especificada, como exemplificada abaixo:

```
public About aboutActionSell() {
   return ActionAbout.disable(getCustomer() == null);
}
```

Construindo objetos About

Quando você precisa de um `About` para fazer mais que uma simples alteração (por exemplo, você quer mudar o nome de um campo ou método), então os objetos somente-leitura `About`, que discutimos até agora não são apropriados na medida em que eles não fornecem qualquer informação adicional conveniente. Isto é mais complicado quando existem várias influências que de certa forma determinam se algo é permitido ou não.

Por exemplo, considere a classe `Location`. Ela tem um método `actionNewBooking(Location)` que permite que duas localizações sejam usadas para criar um novo `Booking` usando duas localizações como os pontos de partida (pick-up) e destino (drop-off). Esse método deve somente ser permitido se os dois objetos `location` forem diferentes, e que eles também sejam da mesma cidade. Se nós implementarmos o método `about...` usando os objetos estáticos `About` (como ilustra abaixo) então os usuários irão saber quando eles podem soltar um objeto, mas não saberão o porquê:

```
public About aboutActionNewBooking(Location location) {
   boolean differentLocations = !equals(location);
   boolean sameCity = (getCity() != null) && getCity().equals(location.getCity());

   return ActionAbout.enable(differentLocations && sameCity);
}
```

Precisamos estar aptos a incluir uma razão do porquê alguma coisa está desabilitada. Isso é satisfeito pelos objetos `Permission` que mencionamos anteriormente. Eles são usados dentro da classe `org.nakedobjects.object.control.ProgrammableAbout`, que nos permite a construir um objeto `About` por explicitamente ajustar seu estado ou condicionalmente modificá-las quando condições são verificadas. Por exemplo, aqui está uma versão alternativa do método anterior usando essa classe:

```
public About aboutActionNewBooking(Location location) {
   boolean differentLocations = !equals(location);
   boolean sameCity = (getCity() != null) && getCity().equals(location.getCity());

   ProgrammableAbout about = new ProgrammableAbout();
   about.makeAvailableOnCondition(differentLocations,
         "Two different locations are required");
   about.makeAvailableOnCondition(sameCity,
         "Locations must be in the same city");
   return about;
}
```

Usando os mesmos dois flags, adicionamos essas condições ao objeto `ProgrammableAbout` usando o método `makeAvailableOnCondition`. Ele não faz nada se o flag for `true`, mas se for `false` então esse método assegura que uma chamada subseqüente ao `canUse` irá retornar um objeto `Veto` que inclui o texto especificado como a razão, ou parte da razão, argumentando que essa opção não está disponível. Essa informação será exibida pelo usuário quando a operação de soltar é inválida e descreve se as localizações são os mesmos ou são de cidades diferentes.

A versão seguinte estende o método tal que o `About` agora contenha uma descrição do método. Ele está alinhado com a nossa filosofia de que a documentação deve ser parte do código e não escrito e mantido separadamente. Ele poderia ser usado (embora não nesse momento) dentro do framework para descrever a opção para que o usuário aponte onde ele pode querer usar. Alternativamente, ele poderia ser usado por um programa que automaticamente gere a documentação completa de uma aplicação. Esta versão também mostra o método `changeNameIfAvailable` que atribui o nome do `About` tal que ele reflita a ação a ser desempenhada. Usando esta versão de método, o nome irá apenas ser mudada se a ação é permitida:

```
public About aboutActionNewBooking(Location location) {
    boolean differentLocations = !equals(location);
    boolean sameCity = (getCity() != null) && getCity().equals(location.getCity());

    ProgrammableAbout about = new ProgrammableAbout();
    about.setDescription("Giving one location to another location creates " +
          "a new booking going from the given location " +
          "to the receiving location.");
    about.makeAvailableOnCondition(differentLocations,
          "Two different locations are required");
    about.makeAvailableOnCondition(sameCity,
          "Locations must be in the same city");
    about.changeNameIfAvailable("New booking from " + location.title() +
          " to " + title());
    return about;
}
```

Áreas de controle

O seguinte resumo apresenta os aspectos de um naked objects que podem ser controlados e como isso deve ser feito.

- **Uma classe**
  Adicionar um método estático `aboutClass` para a classe.
  Use `ClassAbout` para tornar uma classe não-instanciável.
- **Um objeto**
  Adicionar um método `about` para a classe.

Use `ObjectAbout` ou `ProgrammableAbout` para tornar uma instância somente-leitura.

- **Um campo**
  Para objetos de valor associados a um `About` usando o método `setAbout` do objeto de valor.
  Para associações, adicionar o método `about`**`Variable`** para a classe adequando o método `get`**`Variable`** que você deseja controlar.
  Use `FieldAbout` para tornar um campo somente-leitura ou o `ProgrammableAbout` para tornar um campo somente-leitura ou mudar o nome de um campo.
- **Um método de ação**
  Adicione um método `aboutAction`**`Method`** para a classe adequada ao método `action`**`Method`**.
  Use ActionAbout para desabilitar o método ou o `ProgrammableAbout` para desabilitar o método ou mudar o nome da opção.

Escrevendo testes

A prática moderna de desenvolvimento de sistemas demanda agora que os testes sejam parte integral do processo de desenvolvimento, conduzido tão cedo quando possível, e automatizado o quanto possível. No mundo Java, o framework Junit tem sido o propulsor mais significativo desse conceito.

O código a seguir mostra uma abordagem claramente óbvia para escrever testes de unidade para Naked Objects usando Junit. Ele testa a associação bidirecional entre um booking (reserva) e o customer (cliente), e a opção confirmar.

```
public void testBookingForCustomer() {
    Booking booking = new Booking();
    Customer customer = new Customer();

    booking.associateCustomer(customer);

    assertEquals(customer, booking.getCustomer());
    assertTrue(customer.getBookings().contains(booking));

    assertTrue(booking.aboutActionCheckAvailability().canUse().isAllowed());
    booking.actionCheckAvailability();
    assertEquals("Available", booking.getStatus().title().toString());
}
```

O código abaixo reescreve isso usando objetos de visão ‘mock’ (simulação), que são fornecidos como parte do framework Naked Objects. As visões mock representam os objetos dentro do código de teste da mesma maneira que as visões gráficas representam os objetos dentro do GUI básico, isto é, eles permitem que objetos sejam manipulados como se um usuário estivesse trabalhando interativamente.

Como acima, nós iniciamos criando dois novos objetos, um booking e um customer. Por hora, no entanto, nós criamos o novo booking obtendo uma visão mock da classe booking (usando o nome 'Booking' que normalmente seria exibida sob o ícone da classe), e então executando o equivalente do evento de pressionar o botão direito do mouse sobre o menu da classe e chamar a opção *New Instance...*. Nós fazemos o mesmo para o customer:

```
public void testBookingForCustomer() {
    View booking = getClassView("Bookings").newInstance();
    View customer = getClassView("Customers").newInstance();

    booking.drop("Customer", customer.drag());

    booking.assertFieldContains("Customer", customer);
    customer.assertFieldContains("Bookings", booking);

    booking.rightClick("Check Availability");
    booking.assertFieldContains("Status", "Available");
}
```

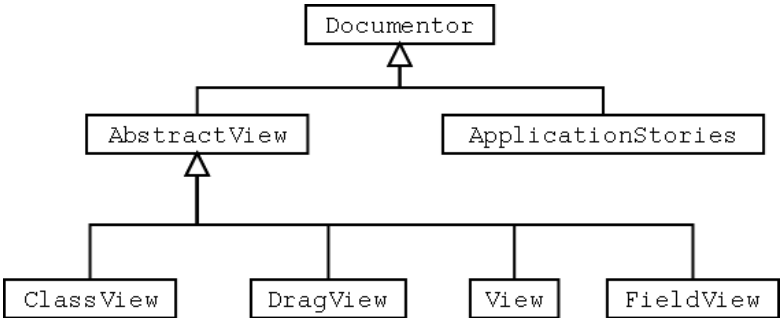
A vantagem dessa abordagem sobre o exemplo Junit anterior não está apenas na simplicidade do código de teste, mas nas visões mock que testam a funcionalidade da perspectiva do usuário. Assim, nós não mais temos que nos preocupar em chamar o método `set...` ou `associate...` ; nós podemos apenas pedir que a visão mock simule o soltar de um objeto sobre o campo requerido e deixar o framework decidir qual método precisa ser chamado. O framework irá agora também verificar o método `about...`, se existir um, e se a mudança do campo estiver desabilitada uma exceção será lançada. Da mesma forma, se uma visão mock é solicitada para chamar um dos métodos de ação do objeto de negócio, ela somente fará depois de primeiro verificar o método `about...` correspondente para ver se o método de ação está disponível naquele contexto.

O exemplo também mostra alguns métodos `assert...` adicionais para a verificação dos objetos simples. Esses métodos nos permitem especificar campos usando nomes como eles devem aparecer ao usuário. Isso significa que nós podemos escrever testes antes que tenhamos escrito os métodos que queremos testar, e antes de compilar esses testes sem erros. Quando o teste estiver em execução, se nenhum método for encontrado com o nome especificado, então uma exceção é lançada. Alguns programadores gostam de usar esse estilo de teste unitário executável para guiar seu processo de desenvolvimento – tratando cada exceção um após o outro.

Simulação usando visão mock

O framework de testes do Naked Objects faz uso de três tipos de visões mock, correspondendo às classes, objetos, e campos de objetos, cada um simulando o que pode ser realizado usando a interface gráfica. Assim, objetos podem ser arrastados, soltos e clicados com o botão direito do mouse, e objetos de valor

editados. Todas essas classes de visão são partes do pacote `org.nakedobjects.testing`.



Visão de classe

Visões de classe são criadas pelo framework de teste quando cada classe é registrada. Uma vez que uma `ClassView` é recuperada, ela pode ser usada para criar uma nova instância e acessar aqueles existentes.

- `public View newInstance()` retornam uma nova instância do tipo representado pela `ClassView`. Esse método é um substituto para todas as maneiras de criar um objeto, tal com selecionar a opção *New Instance...* ou arrastar o ícone da classe sobre o desktop.
- `public View instances()` retorna uma coleção de instâncias como se gerado pela opção *Instances...*.
- `public View findInstance(String pattern)` retorna a primeira instância encontrada do tipo representado cujo título combina, em todo ou em parte, com o texto `pattern`. Isso imita as várias maneiras de um usuário encontrar e então selecionar uma instância com base num único título. Isso inclui selecionar um ícone que já existe no desktop, ou encontrar manualmente através da coleção de instâncias retornadas pela opção do menu da classe *Instances...*.
- `public View drop(DragView draggedObject)` toma uma objeto solto e chama o método `action...` da classe para esse tipo de objeto. Isso corresponde a soltar um objeto sobre um ícone de classe.

O código a seguir simula três ações de classe: a criação de um objeto `Location`; a recuperação de um objeto `Customer` com 'Pawson' no título; e o soltar de um objeto *customer* sobre sobre o ícone de classe *Bookings*.

```
View location = getClassView("Locations").newInstance();
View customer = getClassView("Customers").findInstance("Pawson");
View booking = getClassView("Bookings").drop(customer.drag());
```

Visões de objeto

Uma vez que um objeto esteja disponível através de uma visão, então todos os gestos que estão disponíveis ao usuário através da GUI (tais como arrastar e soltar, ou clicar o botão direito do mouse e selecionar uma ação do menu) podem ser imitados.

- `public View rightClick(String optionName)` simula o click com o botão direito do mouse sobre um objeto e a seleção da opção nomeada de menu. O nome fornecido deve ser o nome da ação como visto pelo usuário, não no nome do método (por exemplo, “Return Bookin” para o método `actionReturnBooking`). Essa chamada verifica o método `about...` associado, se existir um, antes de chamar o método `action...`. Se o método `about...` desabilita a opção (isto é estiver desabilitada e exibida com a opção de menu em cor cinza) então um `IllegalActionError` é lançada. Se o método retornar um objeto então ele é colocado numa nova `View` antes de ser retornado.
- `public DragView drag()` imita a parte do arrastar de um gesto arrastar e soltar. Ele gera um objeto `DragView` que pode ser aceito pelos métodos `drop...`. Os objetos `DragView` podem parecer supérfluos, mas foi introduzido para facilitar a documentação desses gestos (como descrito na próxima seção).
- `public View drop(DragView dropObject)` imita a parte soltar de um gesto arrastar e soltar. Ele aceita uma visão gerada como um objeto `DragView` e chama o método ação relevante para esse tipo de objeto. O método `about...` associado, se existir, é chamado antes que o método `action...` seja chamado, lançando um `IllegalActionError` se a ação é desabilitada (em cada caso a zona de soltura aparece em cor vermelha). Como antes, esse método também colocará qualquer objeto retornado dentro de uma nova `View`.

As seguintes duas linhas de código simulam a seleção da opção *Return Booking...* sobre um objeto booking, e a soltura sobre uma instância de *location* sobre um outro:

```
View returnBooking = booking.rightClick("Return Booking");

View newBooking = location.drop(fromLocation.drag());
```

Coleções

Se o objeto na visão é uma coleção interna ao invés de um objeto de negócio, então use o método `public View select(String title)` para selecionar a primeira instância cujo título combina, em todo ou em parte, com o padrão do

`title`. Isso imita a seleção do usuário de um objeto a partir de uma lista, deslocando a lista se necessário.

Campos dentro de uma visão de objeto

Uma visão de objeto também tem uma série de métodos que permitem que seus campos sejam inicializados ou acessados. Para campos de valor, quando uma string é fornecida, será analisada pelo objeto de valor. Para campos de associação, esses métodos imitam a soltura de outros objetos e o arrastar de ícones. Quando algum desses métodos for chamado com um nome de campo inválido, um `IllegalActionError` será lançada.

- `public void fieldEntry(String fieldName, String entry)` configura o campo especificado usando a string `entry`. A entrada tem que ser entendida pelo objeto de valor e deve ser entrado exatamente da mesma maneira que o usuário deveria entrar – isso é, ele deve ser entrado no formato correto.
- `public void drop(String fieldName, DragView dropObject)` tenta configurar o campo nomeado tal que ele referencie o objeto contido na visão de soltura. Isso irá chamar o método `set...` ou `associate...` apropriado, ou irá acessar a coleção se ele é uma associação um-para-muitos, e adicionar o item. Um `IllegalActionError` irá ser lançado se o objeto for do tipo errado ou o campo já contiver uma referência.
- `public void removeReference(String fieldName)` remove a referência existente a partir do campo especificado.
- `public void removeReference(String fieldName, View object)` remove a referência existente, que combina o objeto da visão especificada, a partir do campo especificado que usa uma coleção interna.
- `public DragView drag(String fieldName)` cria um `DragView` que referencia o objeto envolvido pelo campo nomeado.
- `public DragView drag(String fieldName, String title)` cria um `DragView` que referencia o primeiro objeto dentro da coleção interna envolvida pelo campo nomeado que combina, em todo ou em parte, com o texto especificado no `title`. Um `IllegalActionError` será lançado se a coleção não contiver um item com o título especificado.

O seguinte exemplo configura os campos *First Name* e *Last Name* de um objeto customer, solta um objeto location dentro da lista de *Locations*, e arrasta o objeto location *Airport* da mesma lista e solta-o dentro do campo *Pick Up* de booking:

```
customer.fieldEntry("First Name", "Richard");
customer.fieldEntry("Last Name", "Pawson");

customer.drop("Locations", location.drag());

booking.drop("Pick Up", customer.drag("Locations", "Airport"));
```

Teste de unidade

Para criar um teste de unidade baseado na visão você deve subclassificar a classe `org.nakedobjects.testing.NakedTestCase` ao invés da usual `junit.framework.TestCase`. Para configurar a aplicação de simulação, as classes que são usadas devem ser registradas chamando o método `registerClass`.

O código a seguir foi tirado do código de teste de unidade do exemplo ECS. Ele mostra como o teste é configurado e inicializado. A chamada ao método `init` dentro método `main` configura o framework Naked Objects para trabalhar localmente:

```
package ecs.tests;

import junit.framework.TestSuite;
import junit.textui.TestRunner;
import org.nakedobjects.testing.NakedTestCase;
import org.nakedobjects.testing.View;
import ecs.delivery.*;

public class ECSUnitTests extends NakedTestCase {
   public static void main(java.lang.String[] args) {
      init();
      TestRunner.run(new TestSuite(ECSUnitTests.class));
   }

   public ECSUnitTests(String name) {
      super(name);
   }

   protected void setUp() {
      registerClass(Booking.class);
      registerClass(City.class);
      registerClass(Customer.class);
      registerClass(Location.class);
      registerClass(CreditCard.class);
   }
}
```

Como quando se usa Junit, cada teste deve ser escrito como um método público com o nome do método prefixado por ‘test’ e deve fazer uso de visões mock para simplificar a codificação. O seguinte exemplo, tirado dos testes de unidade ECS, é um método de teste que cria uma nova instância de `City` e configura e testa seu campo nome. Ele conclui verificando que o título de instância é o mesmo que o valor contido pelo campo nome:

```
public void testCity() {
        View city = getClassView("Cities").newInstance();
        city.testField("Name", "Boston");
        city.assertTitleEquals(city.getFieldTitle("Name"));
}
```

Este segundo exemplo teste o estado inicial de um objeto booking. Ele espera que tenha dois campos somente-leitura, um valor de status específico, e quatro associações que ainda não referenciam qualquer coisa. Devido a este status ele também especifica que a opção *Confirm* está desabilitada:

```
public void testBooking() {
        View booking = getClassView("Bookings").newInstance();
        booking.assertFieldReadOnly("Reference");
        booking.assertFieldReadOnly("Status");

        booking.assertFieldContains("Status", "New Booking");

        booking.assertCantRightClick("Confirm");

        booking.assertFieldContains("Customer", (View)null);
        booking.assertFieldContains("City", (View)null);
        booking.assertFieldContains("Pick Up", (View)null);
        booking.assertFieldContains("Drop Off", (View)null);
}
```

Como nos dois exemplos acima, muitos testes precisarão acessar os objetos de classe antes que qualquer teste útil possa ser realizado. A classe `org.nakedobjects.testing.ApplicationTestCase` fornece um método para obter a visão de classe envolvida por uma classe particular:

- `public ClassView getClassView(String name)` recupera uma visão de classe usando o nome da classe quando ele é conhecido pelo usuário (por exemplo, "Credit Cards" para a classe `CreditCard`).

Visões de objetos são usadas para simular ações de usuário como discutido na seção anterior. Métodos `assert...` adicionais são fornecidos com essas visões para ajudar a verificar campos e títulos:

- `public void assertTitleEquals(String message, String expectedTitle)` compara o título da visão, que é o texto fornecido pelo método `title` do objeto da visão, com o título experado. Ele lança um `junit.framework.AssertionFailedError`, que inclui a mensagem fornecida, se as strings resultantes forem diferentes.
- `public void assertTitleEquals(String expectedTitle)` é o mesmo que o método anterior, mas sem a mensagem de falha.
- `public void assertFieldContains(String message, String fieldName, String expectedValue)` verifica o campo nomeado para confirmar que ele contém o valor esperado. Se o valor do campo é diferente, então `AssertionFailedError` é lançado, e inclui a mensagem fornecida.
- `public void assertFieldContains(String fieldName, String expectedValue)` é o mesmo que o método anterior, mas sem a mensagem de falha.

- `public void assertFieldReadOnly(String fieldName)` verifica o campo nomeado para confirmar que ele não é editável. Se o campo é editável então um é `AssertionFailedError` lançado.
- `public void assertFieldContains(String message, String fieldName, View expected)` verifica o campo nomeado para confirmar que ele contém o mesmo objeto que a visão experada. Se o valor de `expected` é `null` então o espera-se que o campo também contenha `null`.
- `public void assertFieldContains(String fieldName, View expected)` é o mesmo que o método anterior, mas sem a mensagem de falha.
- `public void assertCantRightClick(String message, String option)` verifica que a opção nomeada não possa ser selecionada. Se a opção não existir ou estiver habilitada, uma `AssertionFailedError` é lançada, e inclui a mensagem fornecida.
- `public void assertCantRightClick(String option)` é o mesmo que o método anterior, mas sem a mensagem de falha.
- `public void assertCantDrop(String message, DragView dropObject)` verifica que o objeto especificado (contido pela visão) não pode ser solto sobre a visão corrente. Se o objeto solto puder ser solto sobre essa visão, uma `AssertionFailedError` é lançada, incluindo a mensagem fornecida.
- `public void assertCantDrop(DragView dropObject)` é o mesmo que o método anterior, mas sem a mensagem de falha.

Além dessas asserções explícitas existem três métodos que testam um campo associando um valor ou um objeto a ele e então, usando os métodos `assert` acima, confirmar que a informação foi armazenada apropriadamente. Como nos métodos anteriores, se o campo não existir então uma exceção será lançada:

- `public void testField(String fieldName, String value)` testa o campo atributo nomeado tentando configurá-lo para usar o valor especificado em `value`. Uma vez configurado, o valor envolvido pelo objeto é acessado e comparado com o mesmo valor.
- `public void testField(String fieldName, String value, String expected)` testa o campo valor nomeado tentando configurá-lo usando o valor especificado em `value`. Uma vez configurado, o valor envolvido pelo objeto é acessado e comparado ao valor em `expected`. Isso pode ser usado para compensar o objeto de valor que manipula o valor do conjunto antes de armazená-lo. Por exemplo, o objeto data permite que você especifique vários dias na adição de datas. Assim, se a data original for '5-Mar-01', então depois de configurá-lo usando o objeto contido na visão especificada e então configurado com o valor '+14' a data se tornará '19-Mar-01'.
- `public void testField(String fieldName, View setObject)` testa o campo de associação nomeado tentando configurá-lo usando o objeto contido na visão especificada e então comparar a referência envolvida pelo campo para a referência original.

Teste de aceitação

Enquanto os testes de unidade estão principalmente preocupados se um método foi escrito corretamente, os testes de aceitação do usuário estão preocupados se a funcionalidade pode ser combinada para liberar um resultado de valor ao usuário. Nós adotamos a abordagem de apoiar os testes de aceitação por uma série de 'estórias', que são realizadas através de visões mock. Além de verificar que a funcionalidade requerida está disponível, o teste de aceitação automaticamente gera documentação que explica como a estória deve ser realizada através da interface de usuário real. Isso é realizado através de métodos autocomentados que são fornecidos pelas visões. Desde que os testes de aceitação devem ser baseados nos requisitos especificados pelo usuário, essa documentação autogerada pode servir como uma proporção significativa do manual de treinamento do usuário.

A estrutura de um teste de aceitação

Antes de olhar como uma 'estória' é codificada, nós iremos ver como um teste é configurado e inicializado. O seguinte código foi tirado do código de teste de aceitação da aplicação ECS:

```
import org.nakedobjects.testing.AcceptanceTest;
import org.nakedobjects.testing.View;

public class ECSStories extends AcceptanceTest {
    public ECSStories(String name) {
        super(name);
    }
    public static void main(java.lang.String[] args) {
        ECSStories st = new ECSStories("ESC User Stories");
        st.start();
    }
    public void setUp() {
        registerClass(Booking.class);
        registerClass(City.class);
        registerClass(Customer.class);
        registerClass(Location.class);
        registerClass(CreditCard.class);
        registerClass(Telephone.class);

        adminCreateCities();
    }
    public void runStories() {
        story1BasicBooking();
        story2Reuse();
        story3ReturnBooking();
        story4CopyBooking();
        story5LocLoc();
    }
}
```

Testes de aceitação são escritos estendendo a classe `org.nakedobjects.testing.AcceptanceTest`. Essa classe tem um construtor que toma um nome para o suíte de testes. Sua classe derivada dessa forma precisa implementar um construtor que chama esse particular superconstrutor. Para iniciar o teste, chame o método `start` da superclasse.

O método `start` controla os testes e a criação da documentação. Após configurar o framework ele chama o método `setUp` onde você deve registrar o mesmo conjunto de classes que se tornará disponível ao usuário numa aplicação viva. Após esse método ter sido completado, a classe `org.nakedobjects.testing.Documentor` é inicializado com o nome do suíte de teste que foi passado para o superconstrutor. Isso cria o arquivo de documentação HTML, o qual se dá um nome de arquivo baseado no nome do suíte de teste (para o exemplo acima, o arquivo será ESC_User_Stories.html). Qualquer código executado a partir desse ponto será automaticamente documentado. Se existirem alguns objetos que precisem ser configurados, prontos para serem usados durante os testes, mas que não precisam ser descritos na documentação, então isso deveria ser parte do método `setUp` após alguma classe requerida tiver sido registrada. (A chamada a `adminCreateCities` no código acima é um exemplo). Uma vez que o documentador estiver em execução, o método `runStories` é chamado e cada um dos métodos de teste é chamado um após o outro, gerando a documentação quando eles executam. Se algum estaria falhar em algum ponto, então a exceção é passada ao usuário e o teste irá parar, porque em nossa abordagem de testes de aceitação, as estarias são consideradas dependentes uma das outras. Isso contrasta com o que acontece no Junit onde os testes são executados independentemente.

A estrutura de uma estória

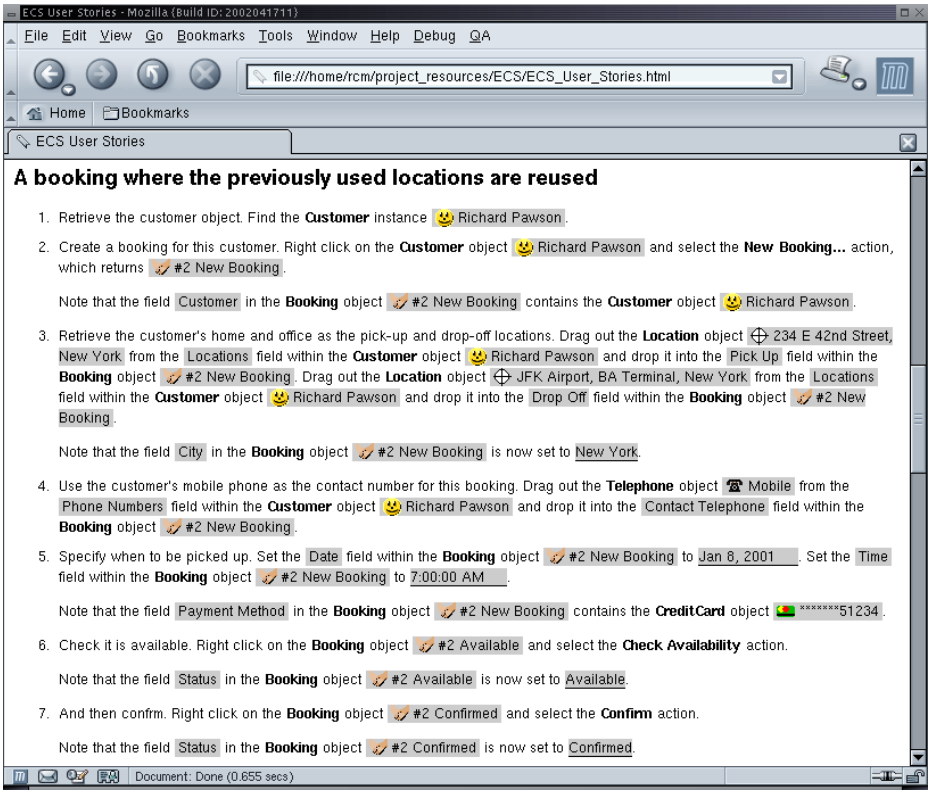
Existe um padrão comum na maioria das estórias que refletem tanto a maneira que o framework é usado quanto o como a documentação é gerada. Para ajudar a equipe de desenvolvimento a manter a trilha do que eles estão testando, e ajudar o usuário a decompor cada tarefa, as estórias podem ser divididas em passos lógicos, cada um pode incluir uma descrição da subtarefa.

Uma estória deve começar com uma chamada ao método `story`, para marcar essa estória como separado de algum anterior e atribuir um título. Cada estágio da estória é então iniciada pela chamada do método `step`, que pode se tornar uma string descrevendo o que será realizado depois. Então trazer as ações esperadas do usuário, tais como criar uma instância, arrastar e soltar objetos, selecionar opções de menu e editar campos. Após as ações do usuário é comum verificar o estado dos objetos para averiguar que tudo está prosseguindo como esperado – isto é, que os requisitos estão sendo atendidos. É também possível confirmar que certas ações não são permitidas em certos pontos.

Ambas as ações e as verificações são realizadas chamando métodos específicos sobre as visões. Quando eles executam, todos esses métodos adicionam descrições às documentações, detalhando passos, explicando como manipular os objetos, exibindo como seus estados mudam, e fazendo notas do que não pode ser realizado. Esse processo, assim, não só verifica a lógica do objeto de negócio, mas também dirige o usuário para o que pode e não pode ser realizado com ele. O seguinte código mostra o início da segunda estória de ECS e demonstra esse padrão comum:

```
public void story2Reuse() {
   story("A booking where the previous used locations are used");

   step("Retrieve the customer object.");
   View customer = getClassView("Customers").findInstance("Pawson");

   step("Create a booking for this customer.");
   View booking = customer.rightClick("New Booking");
   booking.checkField("Customer", customer);

   step("Retrieve the customer's home and office as the...");
   booking.drop("Pick Up",
      customer.drag("Locations", "234 E 42nd Street, New York"));
   booking.drop("Drop Off",
      customer.drag("Locations", "JFK Airport, BA Terminal"));
   booking.checkField("City", "New York");

   step("Use the customer's mobile phone as the contact...");
   :
   :

}
```

Quando o teste é executado e completo sem qualquer exceção, então ele terá também que ser documentado em HTML. A seguir apresentamos a saída dessa facilidade, usando CSS (*cascading style sheets*) com a página HTML para construir tais objetos, campos e menus que podem ser distinguidos:



Os seguintes métodos da classe `AcceptanceTest` estruturam a documentação:

- `public void story(String title)` cria uma nova estória e adiciona um título para o documento. Na versão atual isso é adicionado usando o tag <H2> (cabeçalho nível 2).
- `public void subtitle(String subtitle` adiciona um subtítulo ao documentoo usando o tag <H3> (cabeçalho nível 3).
- `public void step(String description)` adiciona um outro passo às instruções. Isso é adicionado como um tag <LI> (item de lista). Se o texto `description` não finalizar com um stop completo, então nós teríamos adicionado para evitar a descrição afetada pelo texto gerado. Na primeira chamada desse método, após uma chamada a `story` ou `subtitle`, uma tag <OL> (lista ordenada) é adicionada; o tag de fechamento é adicionado quando `story` o `subtitle` ou é chamado.
- `public void step()` adiciona um outro passo mas sem uma descrição. Ele funciona da mesma maneira que o método anterior.
- `public void append(String text)` adiciona qualquer outro texto para o passo corrente. Ele é adicionado sem qualquer tag tal que o texto é simplesmente concatenado as instruções quando eles permanecerem nesse ponto.

Quando o teste executar os métodos acima fornecem o documento com a sua estrutura, e as interações simuladas, gerando seus próprios comentários, descrevem as ações tomadas. Por exemplo, a declaração

`customer.rightClick("New Booking")` produz o texto "Right click on the Customer object 'Richard Pawson' and select the 'New Booking...' action, which returns '#2 New Booking'". (As telas anteriores exibem as saídas formatadas incluindo os ícones).

As visões do objeto mock também permitem que o estado de um objeto seja verificado. Por exemplo, a declaração `booking.checkField("Customer", customer)` deve adicionar ao "Note that the field Customer in the Booking object '#2 New Booking' contains the Customer object 'Richard Pawson'" documento. Esses métodos de verificação são fornecidos além dos métodos `assert...` que podem ainda serem usados embora ele não seja um teste de unidade, nem está baseado na classe `TestCase`. Se algum dessas verificações falharem, então uma `IllegalActionError` é lançada e o teste irá parar. Cada um desses passos adiciona uma explicação à documentação. O propósito é explicar ao usuário qual o efeito que os passos anteriores tiveram sobre o objeto e assim ao estado no qual ele se encontra.

Diferentemente desses métodos `check...`, os métodos `assert...` não produzem qualquer documentação.

Os seguintes métodos de verificação são encontrados na classe `org.nakedobjects.testing.View`:

- `public void checkTitle(String expected)` verifica que o título da visão (como gerado pelo método `title` do objeto) seja igual ao valor especificado em `expected`, adicionando uma nota à documentação para mostrar qual título deveria ser agora.
- `public void checkField(String fieldName, String expectedTitle)` verifica que o campo nomeado dentro da visão tem um título que combina com o valor especificado em `expectedTitle`. Ele adiciona uma nota à documentação mostrando qual texto deveria ser exibido no campo, ou como um valor ou como um título de objeto.
- `public void checkField(String fieldName, AbstractView expected)` verifica que o campo nomeado, dentro da visão, referencia o mesmo objeto que a visão especificada na referência `expected`. Ele adiciona uma nota à documentação mostrando qual o texto que deveria ser exibido na campo, ou com valor ou como um título do objeto.
- `public void checkCantRightClick(String option)` verifica que a opção nomeada não pode ser selecionada. Ele adiciona uma nota à documentação explicando que esta opção não está disponível. O método `append` de `AcceptanceTest` pode ser usado para dar uma razão.
- `public void checkCantDrop(DragView dropObject)` verifica que o objeto especificado (contido pela visão) não pode ser solto sobre a visão corrente. Ele adiciona uma nota à documentação explicando que essa operação não é possível. O método `append` de `AcceptanceTest` pode ser usado para dar uma razão.

Estudo de caso: Comércio e compra de varejo

A loja Safeway Stores é a quarta maior rede de supermercados da Inglaterra, com mais de 480 lojas que variam desde hipermercados a lojas de conveniências. Nos últimos dois anos a Safeway vem experimentando crescimento tanto no total de receitas quanto em lucro, disparado em parte pela nomeação de um novo chefe executivo e uma abordagem mais dinâmica de comercializar e valorar.

Escopo de negócio

Embora a Safeway tenha se comprometido a usar pacotes de software para os vários processos de negócio, ela está preparada também para desenvolver seu próprio sistema onde ela vê uma oportunidade de inovar ou diferenciar. Por exemplo, a Safeway foi a primeira da cadeia de lojas da Inglaterra a introduzir o auto-fechamento de conta para cliente, usando scanners de código de barra manual. E seu sistema de computação de reposição de prateleira portátil, o qual permite ao funcionário verificar preços, monitorar estoques e fazer pedidos no momento em que realizam a venda, venceu um prêmio da indústria quando foi introduzido em 2000.

Ainda que vários dos sistemas existentes da Safeway devam ser mantidos em Cobol, Java é a linguagem preferida por alguns novos desenvolvimentos e

integração de sistemas interativos. Existem desenvolvedores Java na maioria das áreas de aplicação, e também uma gerência central da Equipe de Serviços Java, a qual determina as melhores práticas e guias, realizando pesquisas e desenvolvimento, fornecendo assistência técnica e atuando como consultores durante o ciclo-de-vida de um projeto Java.

A gerência dessa equipe ficou interessada na abordagem Naked Objects no início de 2001. Eles inicialmente não viram o Naked Objects como uma abordagem de desenvolvimento, ao invés disso, como uma forma de treinar alguns de seus desenvolvedores Java para pensar mais em termos de orientação a objetos. (Muitos gerentes de desenvolvimento acharam que trocar por uma linguagem de programação orientada a objetos tal como Java não mudaria automaticamente a maneira das pessoas pensarem sobre o projeto de sistemas). Eles sentiam que o maior comprometimento com os princípios de orientação a objetos ajudaria a ver maiores benefícios a partir do investimento em tecnologias e habilidades em Java. Mais de 30 desenvolvedores receberam o treinamento básico em Naked Objects durante três meses – com a maioria reportando que tinham tido significativa melhora em seu entendimento sobre orientação a objetos. Mais ainda, a experiência criou considerável entusiasmo por responsabilizar-se por um exercício de desenvolvimento realístico usando o framework.

Oportunidade

Ainda sob a atuação do treinamento, um projeto candidato foi identificado na área de gerenciamento de promoções, conhecido como 'deal nominations'. A Safeway mudou sua política de preços dois anos atrás. Agora, ao invés de tentar manter uma política de ‘preço baixo todo o dia’, ela concorre com promoções especiais que oferecem até 50% de descontos sobre uma particular linha de comida e bebidas, projetada para trazer mais clientes para dentro da loja. Cada semana ele imprime e distribui 11 milhões de panfletos de propaganda coloridos de quatro páginas para eletrodomésticos. Para prevenir que competidores concorram com essas ofertas, o conjunto de promoções é constantemente alterado. As lojas são reunidas em grupos, e cada grupo oferece um pacote diferente contendo por volta de 40 promoções especiais a cada semana.

A implementação dessas promoções envolve o gerenciamento da cadeia de suprimentos para arcar com o grande aumento nas vendas dos itens em promoção, comunicação das mudanças de preços ao sistema EPOS das lojas, e imprimir e distribuir os panfletos promocionais, banners de estoque e etiquetas. Sistemas existem para gerenciar cada uma dessas atividades individualmente, mas o planejamento e coordenação geral dessas atividades são intensivamente manuais. Gerentes de promoções estão constantemente explorando as combinações de ofertas especiais com a intenção de atrair o número máximo de compradores que irão comprar itens regulares da loja, sem meramente encorajar ‘os apanhadores de cerejas’ que pegam as melhores ofertas e nada mais. Cada

oferta especial deve ser coordenada com o fornecedor para planejar a logística e, em alguns casos, compartilhar os custos.

Idealmente, esses gerentes precisariam de um sistema específico para determinar novos negócios, projetar vendas e disponibilidades, simular seu giro através do grupo de vendas, e então coordenar sua execução através dos sistemas de cadeia de suprimentos e coordenação de preços. Tentativas anteriores realizadas pelo departamento de sistemas para analisar os requisitos para tal sistema não obtiveram sucesso. A atividade não se adequava muito bem às perspectivas esperadas na abordagem fortemente orientada a processos, tal como os sistemas de gerenciamento da cadeia de fornecimento. O 'deal nominations' foi muito mais que uma atividade de solucionar problemas: todo particular negócio pode começar com uma proposta de um fornecedor. Ou pode ser iniciado preenchendo um 'buraco' numa oferta de parceiros. O que foi necessário foi um sistema que deveria permitir que usuários construíssem múltiplas ofertas, simular seus efeitos, copiar e colá-los até que eles se sentissem bem. Então eles fizeram isso!

Abordagem

Uma equipe constituída por desenvolvedores e representantes da área de negócio foi montada e deram apenas quatro meses para projetar uma prova de conceitos usando o Naked Objects. Para contextualizar, as tentativas anteriores tiveram que ser deixadas de lado, pois não passaram de um exercício de obtenção de requisitos com base no papel e lápis, o qual foi abandonado porque os usuários

não ficaram convencidos de que o documento resultante realmente capturava o que eles precisavam.

O novo exercício não fez uso desse trabalho anterior: começou-se do zero. Após apenas algumas horas gasta discutindo a dinâmica do negócio, a fim de dar aos desenvolvedores alguma familiaridade com o domínio, a equipe conseguiu corretamente identificar o conjunto de objetos de negócio que seria o melhor modelo para lidar com a área candidata. Foram sugeridas em torno de vinte objetos candidatos, mas no final do primeiro dia, foram gradualmente reduzidos para menos de dez.

Na manhã seguinte os desenvolvedores já tinham traduzido a descrição dos objetos candidatos em Java, usando o framework Naked Objects, desenhando ícones sugeridos pelos representantes de negócio e reunindo alguns dados reais de Produtos, Loja, entre outros.

As próximas quatro semanas seguiram um padrão iterativo. A equipe toda reunia uma vez na semana e revisavam todo o modelo de objetos e o estado do protótipo, decidindo quais prioridades estariam na próxima iteração. Durante a semana tinham muitas pequenas iterações. Uma maneira particularmente efetiva de trabalhar foi ter uma pessoa representante do negócio sentada ao lado do desenvolvedor e evoluir o protótipo em tempo-real: adicionando novos atributos ou associações, novas subclasses e métodos de negócio simples. Para funcionalidades de negócio mais complexas (especialmente aquelas que envolviam pesquisa em coleções de objetos ou navegação ao longo da cadeia de comandos) os desenvolvedores deveriam trabalhar sozinhos, ou em pares.

Durante esse período, houve constante demanda por demonstrações, tanto pelos membros da equipe, quanto pelas outras partes que tinham ouvido falar sobre a abordagem radical do projeto e que queriam conhecer melhor. O gerente de projeto assumiu o papel de apresentador chefe, registrando e gerenciando um conjunto de scripts de demonstrações correspondentes a um cenário de use-case específico. Além de seduzir a equipe, as demonstrações também serviram a um propósito importante de continuamente validar o modelo de objetos.

Além disso, em várias ocasiões durante esse período exploratório, a equipe foi cobrada para que identificasse os possíveis cenários. Não existiam requisitos, nem mesmo prováveis extensões futuras. Os cenários eram puramente hipotéticos, relacionando futuras mudanças na organização de negócio, estratégia e relacionamentos, bem como cenários orientados à tecnologia. Embora não tivessem explorado em detalhes, a equipe foi questionada para que explorasse brevemente quais mudanças esse novo cenário poderia afetar o modelo. Idealmente, a resposta seria que as mudanças estariam limitadas a apenas uma classe de objetos, ou talvez a criação de uma nova classe que implementasse uma interface existente, tal que poderia substituir um objeto existente em algum contexto.

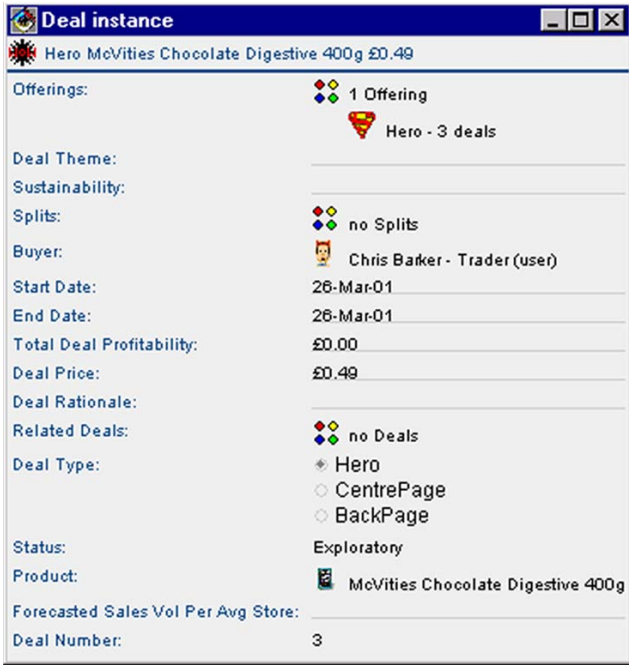
Objetos de negócio chaves

A seguir, são exibidas as principais classes de objetos de negócio identificadas durante a exploração do sistema e rudemente implementado no protótipo.

Usuário. O usuário conhece os papéis que um usuário pode exercer, como comunicar com essa pessoa e as várias informações relacionadas ao RH.

Produto. Existe uma instância do produto para cada produto no estoque da Safeway (existem mais 40.000). Além dos dados do fornecedor, Produtos possuem um ou mais imagens do produto, usado na propaganda e marketing. O objeto Produto poderia também conhecer a interface com o sistema de cadeia de fornecimento.

Promoção. Uma promoção (tal como um desconto ou promoção leve-dois-pague-um) pode ser atribuída a qualquer produto. Promoções são inicialmente hipotéticas quando os impactos sobre a cadeia de suprimentos e na receita são simulados, só depois é que eles são formalmente propostos. Uma promoção sabe como gerenciar seu próprio processo de aprovação.



Oferta. Uma Oferta é normalmente composta de múltiplas promoções. Além de conhecer como implementar a alteração do preço nos locais que serão aplicados, a Oferta deve conhecer como produzir o 'folheto' impresso promocional que será distribuído às casas locais, e os materiais visuais de promoção nas lojas. A Oferta tem várias subclasses. A oferta 'Hero', por exemplo, é normalmente uma coleção de itens com grandes descontos que devem ser apresentados em primeira página de um folheto promocional.

Grupo. Localizações de lojas são gerenciadas como grupos regionais. Ofertas são normalmente rotacionadas entre grupos, tal que cubra o máximo de demanda de um particular produto.

Localização. Uma localização modela uma particular loja ou subloja tal como uma loja de conveniência. Como Produto, Localização deve saber como conversar com outros sistemas que fornecem informações específicas de localizações ou funcionalidades.

Fornecedor. Fornecedor de alimentos detalha e sabe como se comunicar com os fornecedores e com o sistema de gerenciamento de fornecedores.

Papel. Sabe quais classes de objetos e quais métodos desses objetos que interpretando esse papel pode acessar.

Avaliação – a perspectiva de negócio

Todos que estiveram envolvidos com o projeto exploratório ficaram felizes com o que obtiveram num curto espaço de tempo. O representante do usuário disse que o protótipo estava no caminho certo – no sentido de que ele fornecia muita flexibilidade à definição de promoções e construíam Ofertas de diferentes maneiras, adequado à realidade de como eles trabalhavam. Todos fizeram comentários positivos sobre a maneira em que o Naked Objects facilitou o diálogo entre desenvolvedores e representantes de negócio. Este último não teve dificuldades em adotar a maioria das terminologias de objetos.

No final do exercício as capacidades do protótipo foram comparadas aos requisitos priorizados produzidos nas tentativas anteriores (sem sucesso) – que não haviam sido consultados até então. Todos os requisitos de maior prioridade tinham sido atendidos, e muito mais além. De fato, quando se questionou sobre as características do protótipo as quais eles consideravam ser às de maior prioridade para a implementação, se o projeto fosse prosseguir a Especificação e Implantação, as características de maior prioridade não tinham sido originalmente identificadas como requisitos durante o primeiro exercício. Por exemplo, um objeto Oferta, contendo várias Promoções – visto como uma coleção de ícones. Durante as primeiras duas semanas, alguém tinha perguntado se o ícone genérico representando uma Promoção poderia ser trocado por uma imagem fotográfica individual representando o Produto real que receberia o desconto. Essa capacidade foi adicionada ao framework Naked Objects permitindo que qualquer Oferta pudesse agora ser exibida como uma forma crua do folheto de propaganda que seria eventualmente impressa. O pessoal de marketing reportou que essa primeira visualização foi muito útil na avaliação da atratividade da oferta como um todo.

Avaliação – a perspectiva da TI

Desenvolvedores reportaram que acharam o framework Naked Objects mais fácil de adotar do que eles tinham anteriormente utilizado para desenvolver aplicações web. O Naked Objects permitiu que eles se concentrassem no problema de negócio, e sobre as disciplinas da boa programação, ao invés de simplesmente saber como usar a tecnologia.

Antes da introdução do Naked Objects em 2001, o mais novo Gerente de Serviços Java disse que tinha tido dificuldades de demonstrar a migração para Java do CICS/Cobol, para si próprio, aos seus colegas e aos executivos de TI. Ele pôde antever a Safeway repetindo o CICS/Cobol com um “visual bonito”, mas sem nenhuma vantagem significativa no projeto e custo nos desenvolvimentos futuros, na manutenabilidade esuporte. O Naked Objects mudou essa perspectiva.

O sistema Deal Nominations está agora esperando que uma decisão seja tomada; mas parece não existir questões que a abordagem Naked Objects é a melhor maneira de implementá-lo.

O segundo projeto

Nesse meio tempo, um outro grupo da Safeway tinha visto o protótipo do sistema de promoções Deal Nominations e pensaram que a abordagem poderia ajudá-los com um outro problema de negócio difícil. O Naked Objects foi inicialmente visto como uma forma de facilitar a modelagem de requisitos ao invés de ser vista como uma possível arquitetura para o sistema. No entanto, conforme a explanação progredia, ficava claro que os usuários adoravam o conceito. Os gerentes de TI também reconheceram que esse sistema era um candidato ideal para testar integralmente o framework Naked Objects: o sistema promoveu alto valor ao negócio, mas tinha uma base de dados pequena.

Naquele tempo, o framework Naked Objects não possuía serviços necessários para implementar sistemas reais. No entanto, a Safeway disponibilizou seu melhor desenvolvedor Java para explorar possibilidades. Em pouco tempo ficou claro que o mapeamento objeto/relacional requerido entre Naked Objects e bancos de dados existentes na Safeway podiam ser realizados usando Enterprise Java Beans (EJB) e XML. O resultado culminou nacriação de Repositório de Objetos EJB.

A fase de Exploração levou quatro semanas. Após gastar três semanas de Especificação, a fase de Liberação começou. Apenas as definições de objetos foram levadas em consideração. Todo código Java necessário para a atualização foi então reescrito do zero, adotando uma abordagem mais rigorosa para testes. A primeira versão ficou pronto para os testes de usuário após 90 dias, o que foi extraordinário dado que isso incluiu o treinamento do desenvolvedor, parada do Natal, e atrasos causados por mudanças e problemas iniciais com o framework e o middleware. O desempenho inicial foi ruim. No entanto, era porque o servidor EJB estava funcionando numa máquina separada da de banco de dados. Quando o banco de dados foi portado para o mainframe, o sistema como um todo executou tão rápido quanto “qualquer coisa que use o mainframe executado sobre o CICS”. A Safeway incorporou posteriormente as características de teste do Naked Objects e auxiliou no desenvolvimento de novas características para o framework.

Um processo de desenvolvimento

A nossa experiência em aplicar o framework Naked Objects para vários problemas de negócio sugeriu que tais projetos são melhores concebidos e gerenciados em três fases distintas: exploração, especificação e liberação. Para as fases de especificação e liberação era possível usar vários métodos existentes.

Embora muitos métodos prescrevam ou permitam alguma forma de exploração, nossa versão é diferente. O que se consegue nessa fase não é apenas o melhor entendimento dos requisitos e possibilidades de negócio, mas um esboço do modelo de objeto que foi testado contra cenários de negócio, através da aplicação de um protótipo. Você irá ver isso em ação nas seguintes páginas.

A especificação e liberação são mais convencionais. Durante a especificação, os requisitos de negócio são formalmente especificados e priorizados, versões planejadas, custos estimados, e implicações de infra-estrutura identificadas. Na fase de liberação, o sistema é desenvolvido, integrado, testado e disponibilizado. Você pode usar qualquer método moderno para esta fase, mas se você escolher XP, o Naked Objects irá facilitar algumas de suas disciplinas. Embora nenhum código seja aproveitado do protótipo exploratório, o esboço completo do modelo de objetos permite que desenvolvedores tenham maior confiança em adotar a disciplina rígida ‘uma estória de cada vez’ do XP durante a liberação, sabendo que subseqüente refatoramento será principalmente aplicada a métodos, não à fronteira do objeto. E o framework de testes do Naked Objects facilita a aplicação da disciplina ‘codificar primeiro os testes’ tanto para testes unitários quanto para testes de aceitação.

Dentro de cada uma dessas fases as atividades são fortemente iterativas. Você pode também iterar entre fases de exploração, especificação e liberação. No entanto, isso gera o risco de diluir o poder da abordagem e nós recomendamos que você os trate como três fases distintas e seqüenciais, a menos até que você se sinta confortável com a abordagem Naked Objects.

A fase de exploração

A fase de exploração tem dois propósitos principais: explorar requisitos de negócio e explorar representações de objetos do domínio de negócio. A abordagem naked objects é única em poder dar sinergia a essas duas atividades: explorar requisitos de negócio claramente ajuda você a identificar objetos candidatos, mas construir um modelo de objetos na forma concreta para o uso também o ajuda a explorar requisitos de negócio.

Note que nós dissemos ‘explorar’ e não ‘capturar’ requisitos de negócio. Em nossa experiência, os requisitos especificados por representantes do usuário após explorar os naked objects são muito diferentes daqueles que eles poderiam ter especificado, se eles tivessem especificado no início do projeto – e diferente também daqueles que eles poderiam ter especificado usando formas mais comuns de prototipação. A interação direta com o modelo de objetos parece sugerir novos requisitos e até novas possibilidades de negócio. Esse é o motivo do período de exploração ter que preceder qualquer forma de especificação de requisitos.

Preparando para a exploração

As pessoas normalmente perguntam: ‘Como você pode começar um projeto sem primeiro ter alguma idéia do escopo dos requisitos, a fim de fazer uma estimativa inicial dos custos envolvidos no projeto?’ Essa linha de pensamento conduz a falhas em projetos de desenvolvimento de sistemas. É muito difícil obter patrocinadores de negócio e representantes do usuário abertos a especificar o que eles querem de um novo sistema. Isso não é justo – como alguns métodos de desenvolvimento de sistemas parecem assumir – porque o usuário tem dificuldades de articular os requisitos. O motivo disso é que a introdução da nova tecnologia dentro da atividade de negócio altera o que é possível, mas na maneira quase impossível de imaginar. Solicitar que usuários especifiquem todos os seus requisitos de um sistema no papel e priorizar tais requisitos sem ter visto qualquer coisa implementada, ou mesmo simulada, apenas atrai futuros problemas.

Definir o caso antes da ação

Todos projetos começam com um período de exploração quando se usa a abordagem Naked Objects, o qual possui um tempo limitado; uma vez que você tenha tido alguma experiência usando essa abordagem, você pode fixar o preço sem precisar saber qualquer coisa sobre a aplicação.

‘Definir o caso antes da ação’ para um projeto Naked Objects deve, assim, significar definir o caso para realizar uma fase de exploração. Nós recomendamos que você escreva este simples parágrafo ou apresente num slide esclarecendo um dos mais convincentes razões para fazer a exploração: um ambiente regulador de mudanças, uma nova linha de negócio, uma necessidade súbita e dramática para reduzir custos, ou simplesmente uma visão convincente de uma nova maneira de trabalhar. Se você não puder escrever um simples parágrafo que convença o suficiente para justificar um exercício exploratório de quatro semanas com preço-fixo, então o projeto provavelmente não valerá a pena.

Embora isso dê coragem, é mais honesto recusar fazer qualquer estimativa do custo geral de implementação antes que a fase de exploração seja realizada, do que fazer uma estimativa crua e ter que ajustá-la por um fator de três, ou dez, quando os verdadeiros requisitos e complexidades emergirem. Nós não somos os primeiros a dizer isso. No entanto, uma das grandes vantagens da abordagem

Naked Objects é que o cliente obtém muitas vantagens da exploração: não apenas uma especificação, ou estudo de viabilidade, ou um conjunto de diagramas UML estáticos, mas um protótipo funcional com o qual eles podem se divertir, e um modelo de objetos que pode também indicar novas percepções do próprio negócio.

Antes que você comece o estágio da especificação você irá não apenas ter uma clara idéia do que é necessário e do que é possível, mas também dos custos envolvidos na implementação. Se necessário você pode então escrever uma justificativa mais detalhada para prosseguir com a fase de liberação.

Formando a equipe de exploração

A equipe de exploração inclui desenvolvedores de sistemas e representantes do usuário trabalhando juntos. Devido aos desenvolvedores serem os escritores de código e fazerem modificações diretamente na frente dos usuários, eles precisam ser fluentes em suas ferramentas: na linguagem de programação Java, no framework Naked Objects, e no ambiente de desenvolvimento escolhido. (Treinar desenvolvedores para explorar um problema de negócio hipotético pode ser muito efetivo, mas nós não recomendamos fazer isso numa fase real de exploração de negócio até que eles tenham atingido alguma fluência com as ferramentas).

Os representantes do usuário precisam de um grande e profundo conhecimento de seu próprio domínio de negócio, e confiar naquilo que eles representam. Idealmente eles serão selecionados devido a sua disposição em explorar novas maneiras de trabalhar, e devem ser usuários que confiam e sejam experientes na tecnologia de informação. Idealmente, o grupo incluirá ao menos uma pessoa que será um usuário diário do sistema que será proposto. Algumas vezes, o representante do usuário será uma pessoa de TI tal como um gerente de contas, analista de negócio ou consultores (algumas vezes conhecido como ‘especialistas da área’). No entanto, entenda que o representante do usuário existe para fornecer conhecimentos amplos e profundos sobre o negócio, e para representar os usuários, não para desempenhar algum tipo de papel transitório entre usuários e TI.

A equipe de exploração também precisa de uma pessoa de modelagem de objetos experiente para cuidar de todas as sessões de modelagem. Algumas abordagens de projetos de objetos enfatizam a idéia de um ‘facilitador’: alguém treinado em facilitar grupos, que possa assegurar que todos contribuíram com idéias, que nenhuma idéia seja rejeitada de imediato, que ninguém domine o grupo, e assim por diante. Mas a modelagem de objetos é um exercício de projeto, não um exercício de ‘brainstorming’. Certamente, deve ser um projeto participativo, tal que o líder de modelagem de objetos tenha uma razoável habilidade interpessoal. Mas ele pode querer dar resposta imediata a qualquer sugestão: “Sim, isso ajuda”, “Não, isso não é um objeto”, entre outras.

Uma boa dica é que o líder de modelagem de objetos seja suficientemente competente e seguro para fazer todo o modelo de objetos sozinho. Isso não implica que o exercício de equipe seja apenas uma simulação. A equipe irá sem dúvidas fazer sugestões que o líder de modelagem poderia não lembrar. Existem também, certas atividades de modelagem que a equipe faz melhor que um indivíduo – algumas das quais são descritas neste capítulo. Mesmo em períodos dinâmicos onde o líder de modelagem está fazendo muitas decisões de projeto diretamente, simplesmente pressione-o para pensar em voz alta na frente do grupo e justificar tais decisões. Isto irá provavelmente aprimorar a modelagem, assim como o entendimento do grupo. Finalmente, um outro membro da equipe pode e dará respostas imediatas sobre algumas dessas decisões por aplicar mentalmente as suas próprias necessidades e cenários de negócio. Se conduzido corretamente, cada membro da equipe sentirá que fez contribuições úteis e terá um forte sentimento de ser dono do modelo resultante, mas ele não deverá ter a ilusão de que o projeto seja um processo democrático.

Mais uma coisa sobre o líder de modelagem de negócio: em nossa opinião essa pessoa precisa ter experiência em programação, e numa linguagem de programação orientada a objetos. Isso não significa que o líder de modelagem de objetos tenha que codificar (embora eles possam fazê-lo), mas que eles precisam ter grande familiaridade com padrões de projeto orientados a objeto adequado para o nível de programação. Os naked objects permitem que haja sinergia entre a análise orientada a objetos e programação orientada a objetos de maneira que poucas abordagens fazem. A experiência em programação não precisa ser em Java – a sintaxe básica é muito fácil de aprender uma vez que você tenha programado em uma outra linguagem orientada a objetos. Nós observamos (e outros também) que muitas pessoas de modelagem aprenderam suas técnicas da programação em Smalltalk, não a partir de algum método. Se você aprendeu bem a linguagem Java, isso pode ser bom, mas infelizmente muitas instituições e autores ainda ensinam Java como ela fosse uma linguagem procedural.

Outros papéis são também necessários na equipe de exploração. Dependendo da aplicação, pode ser apropriado ter alguém que tenha uma ampla familiaridade com os sistemas existentes para o qual o novo sistema poderá se comunicar, particularmente com bancos de dados existentes. Quanto mais você fizer uso de dados realísticos durante a fase de exploração, a mais efetiva ela será. No estudo de casos da Safeway, até o protótipo tinha uma forma simples de persistência que permitia que a equipe de exploração trabalhasse com um conjunto grande e realístico de objetos Produto e Localização. Isso foi importante, porque a lógica do projeto tinha que gerenciar a complexidade em vários ajustes e/ou promoções especiais de preços potencialmente conflitantes. Se usássemos um conjunto de pequeno de dados teria sido difícil entender esses assuntos.

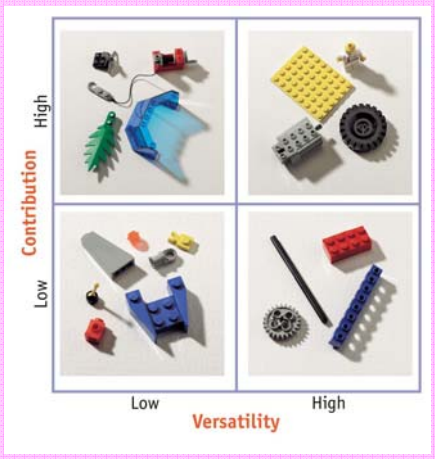
A equipe de exploração terá muitas responsabilidades – documentar o modelo, capturar os cenários de testes, projetar ícones, dar demonstrações – muitas das quais serão descritas posteriormente em detalhes. Tais responsabilidades podem ou não ser traduzidas em papéis especializados: cada equipe encontrará seu

próprio estilo. Você precisa encontrar o equilíbrio correto entre o poder da propriedade coletiva, como defendido pelo Extreme Programming, e a eficiência de ter especialistas disponíveis.

Preparação para avançar

Antes que a equipe inicie a explorar o domínio de negócio escolhido, é importante que se estabeleça uma visão comum do que será o sistema resultante. Por exemplo, um dos princípios do Extreme Programming (XP) é o da 'metáfora do sistema': todos os membros da equipe devem ser em mente uma visão comum do que será o sistema que pode ser usado para informar decisões de projeto. A metáfora de uma planilha eletrônica é citada como um exemplo em muitos livros de XP. Todavia, muitos praticantes de XP relatam que eles têm dificuldades de identificar uma metáfora adequada, ou que a metáfora tem contribuído muito pouco para resolver casos reais de projeto.

Os naked objects são, por si só, uma metáfora muito forte. Quando alguém entende um sistema projetado com os naked objects, ele passa a ter um modelo mental muito claro do que ele pode traduzir dentro de um novo domínio. Assim, nós recomendamos que todos os membros da equipe sejam introduzidos a exemplos de sistemas naked objects antes de começarem a explorar novos domínios de problemas. Idealmente isso envolveria ler os estudos de casos deste livro e exercitar as demonstrações que você pode obter de nosso website. Se isso não for praticável, então inicie a fase de exploração com um conjunto de demonstrações para toda a equipe.



Um exercício de preparação útil é obter membros da equipe que classifiquem uma coleção de peças Lego de acordo com a sua 'versatilidade' (isto é, o número de modelos diferentes que você pode usar) e 'contribuição' (o valor médio que cada peça adiciona ao modelo). A mesma analogia pode ser aplicada aos objetos de negócio. Num exercício de exploração Naked Objects, estamos procurando identificar apenas objetos que pertencem ao quadrante superior direito: aqueles que o usuário poderia empregar na solução de uma ampla faixa de problemas e ainda manter seu valor comportamental. Projetos de sistemas clássicos orientados a tarefas tendem a focar no quadrante superior esquerdo, enquanto a programação clássica orientada a objetos foca nos quadrantes inferiores.

É possível estender a metáfora genérica do naked object em uma variedade de formas mais específicas. As metáforas que nós temos utilizado em vários projetos incluem a da Assistência Social, Gerenciamento Visual da Rede, a Bancada, e o Catálogo de Peças.

Em outra área onde nós achamos que essa preparação trás benefícios é a do ambiente físico. Os workshops de exploração são atividades intensas – pouca iluminação, ventilação, ou níveis de ruído ambiental provocam estresse e reduzem sua efetividade. Tente usar uma sala com luz natural, preferivelmente nos dois lados da sala [Alexander1977]. Organize as mobílias tal que todos os participantes possam ver uns aos outros: uma mesa aberta em U é muito melhor do que uma mesa longa como uma mesa de diretoria.

Um monte de quadro branco e/ou lugares para armar flipcharts são sempre úteis, mas faça uso tanto quanto possível de projeções eletrônicas. Você precisará fazer demonstrações da evolução do protótipo, e irá permitir que múltiplos desenvolvedores vejam a evolução da estrutura do código. Onde possível, gostamos de ter dois projetores e dois monitores, de forma que possamos ver tanto a visão do desenvolvedor quanto do usuário do sistema ao mesmo tempo.

Nós também recomendamos que você tente capturar as idéias que surgem das sessões de projeto diretamente em forma eletrônica. A menos que uma secretária tenha escrita perfeita, a eletrônica fornece legibilidade. É um meio também de gerar saídas que possam ser instantaneamente disponibilizadas a todos os participantes assim que o workshop finalizar. Mas a grande razão é que a captura com projeção de notas eletronicamente facilita a edição de notas e permite reestruturá-los durante o progresso das discussões. Nossa preferência é usar um processador de textos tal como o Word, embora nós tenhamos visto uma variedade de ferramentas de 'mapa mental' usadas para esse propósito. Em ambos os casos, o usuário deve ser fluente na ferramenta escolhida.

Planejando e definindo a programação

A exploração é um exercício iterativo, com cada iteração composta de três principais atividades:

- Definir objetos e suas responsabilidades.
- Construir os objetos dentro de um protótipo.
- Usar o protótipo para explorar cenários.

É possível e até desejável, planejar a fase de exploração como um conjunto fixo de iterações formais, talvez quatro dos cinco deles, e programar as três atividades dentro de cada iteração. Mas eles também podem ter muitas iterações informais, onde todas as três atividades podem ser de responsabilidade de um único indivíduo no curso de alguns minutos e onde os contornos entre as atividades não são claros e nem importantes.

Em circunstâncias ideais, durante a fase de exploração toda equipe estaria trabalhando em tempo integral sobre o projeto e estariam fisicamente ocupando a mesma localização. Com tal configuração e com desenvolvedores experientes na abordagem Naked Objects, duas semanas poderia ser um tempo de exploração suficiente. Existe pouca necessidade para planejamento formal quando a equipe pode decidir suas prioridades e estilo de trabalho diário.

Tais circunstâncias ideais raramente aparecem. É normalmente difícil obter representantes do usuário que se ocupe em tempo integral em qualquer projeto de sistemas. Em nossa experiência é muito mais importante obter os representantes do usuário corretos, os quais estarão mentalmente engajados mesmo se eles não puderam estar em tempo integral, do que insistir num representante do usuário em tempo integral e no final ficar com alguém 'perdido' no negócio. Mas então existe a necessidade de um plano levemente formal: a equipe toda se reúne formalmente durante três dias, por quatro horas talvez. A sessão inicia com uma demonstração ensaiada de cenários finalizados, uma revisão do modelo de objetos como um todo e então uma discussão dos novos caminhos a desvendar ou problemas a resolver. Entre essas reuniões, representantes do usuário e desenvolvedores se

reúnem para trabalhar num determinado aspecto do modelo e protótipo, ou para especificar e testar novos cenários.

Com tal abordagem, quatro semanas é um tempo mais do que suficiente. E se for pela primeira vez que você está projetando sistemas com essa abordagem ou se existir muitos diferentes stakeholders envolvidos, então permita um pouco mais de tempo. Note, no entanto, que nenhum dos projetos descritos neste livro teve uma fase de exploração maior do que seis semanas. Não se iluda dizendo que o domínio do problema é especialmente complexo. Os naked objects tem sucesso sobre problemas aparentemente complexos! Algumas aplicações de negócio que foram propostas pareciam ser assustadoramente complexas. Entretanto, o principal problema é que tais aplicações não se encaixavam no paradigma orientado a processos que muitos métodos de desenvolvimento assumem.

Nós devemos agora olhar cada um dos três elementos de uma iteração.

Definindo objetos e suas responsabilidades

A exploração inicia indo direto aos objetos. Não tente capturar use-cases antes de olhar para os objetos chaves. Nosso objetivo é sempre obter uma primeira, crua, iteração do modelo de objetos de negócio completo no final de todo o primeiro dia de exploração.

Você não precisa de qualquer preparação formal ou documentar entradas antes de começar, você irá contar apenas com o conhecimento inerente da equipe. Gaste umas duas horas só para conversar sobre o domínio de negócio selecionado: o que acontece, qual é a dificuldades, como as coisas mudam. Isso é informal, não estruturado e não documentado. O propósito é realizar o aquecimento das pessoas, deixá-los confortável uns com os outros, e familiarizados com o domínio.

Identificação inicial dos objetos

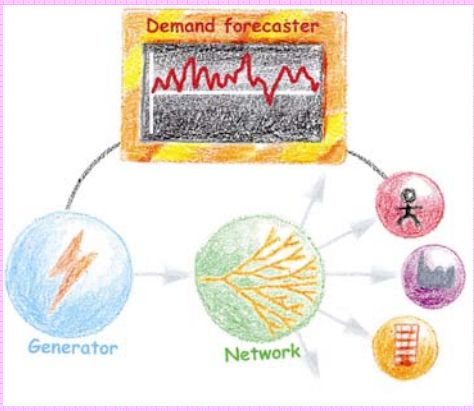
Uma técnica a muito estabelecida para identificar objetos candidatos [Abbott1983] [Booch1994] é escrever uma descrição muito informal do domínio de negócio e da contribuição do sistema e, então, sublinhar todos os substantivos ou frases substantivadas. Provavelmente, durante as nossas discussões informais, começamos a fazer alguns mentalmente, destacando frases substantivadas. Entretanto nós achamos desnecessário reconhecer isso formalmente ou documentar qualquer coisa dessa discussão.

Portanto, após umas duas horas de discussão, o líder da modelagem de objetos convida todos da equipe a iniciarem sugerindo os objetos candidatos, e capturá-las como um mapa mental ou documento texto estruturado. (Se você decidir ter um secretário, então essa pessoa e o líder da modelagem de objetos devem ter

pré-acordado o formato, e cuidadosamente ensaiado o processo de capturar e editar idéias de objetos). Nesse estágio, a ênfase estará em obter muitas sugestões e manter um alto nível de energia. No entanto, o líder da modelagem de objetos deve sentir se livre para fornecer respostas imediatas a sugestões, e adicionar várias de suas próprias sugestões.

Este exemplo, desenho tirado de um exercício de exploração realizado por um serviço público de energia elétrica Norte Americana, ilustra como o modelo de objetos pode evoluir rapidamente nas primeiras poucas horas.

O primeiro corte do modelo de objetos caracterizou um objeto Network conectando Geradores a Aparelhos Domésticos, Negócios e Indústrias. Um sistema de previsão de demanda separado que equilibra as cargas:



A equipe rapidamente percebeu que a responsabilidade da distribuição para previsão de demanda de cada consumidor – na forma de um objeto Perfil de Potência ativo – deveria aprimorar a agilidade de negócio. E por trocar a responsabilidade para a liberação da eletricidade num objeto Contrato, ao invés do Network, a empresa podia com maior facilidade vender eletricidade para fora de seu principal território.

Nós nunca vimos um grupo que não entrasse nessa tarefa inicial de sugerir objetos com entusiasmo e efetividade. Uma hora é usualmente suficiente para gerar uma boa lista de candidatos – normalmente entre 20 a 40.

Reduzindo a lista

Após um breve intervalo, inicia-se o refinamento da lista. O alvo claramente estabelecido é reduzir a lista para entre cinco e dez objetos de negócio 'principais'. Uma razão para isso é o princípio freqüentemente citado sobre o número de idéias que pessoas podem manter em suas cabeças durante uma hora. O framework Naked Objects encoraja essa postura porque, ao menos para o protótipo, todas as principais classes de objetos de negócio irão aparecer nas janelas de classes sobre a tela, e dez é próximo do limite visual antes que a tela se apresente irremediavelmente complexa. O simples ato de forçar para que a lista seja reduzida também encoraja maior abstração na modelagem.

Para a lista reduzida, você pode passar rapidamente por uma série de testes simples, ou casualmente ou formalmente:

É uma classe instanciável? Essas classes candidatas representam um tipo de entidade de negócio dos quais existem várias instâncias? Os naked objects concretos podem ser mais fácil de ver. Pergunte para a equipe: Deveria, em algum momento, existir dois ou mais desse tipo de objeto exibidos na tela simultaneamente? Alguma vez o usuário poderia dizer (apontando na tela): 'Não este, aquele!' Se sim, então você tem uma classe de entidade instanciável e ela pode ser mantida. Um outro teste é simplesmente perguntar como cada instância é identificada de forma única: em termos do Naked Objects, qual é o título do

objeto? Se um candidato falhar nesse teste, então você pode ter uma função que talvez deva aparecer como um método sobre um dos outros objetos. Via de regra, seja cauteloso com objetos candidatos que possuam verbos no infinitivo. CalcularTaxaDeJuros é provavelmente uma classe não instanciável, já TaxaDeJuros pode ser.

Você pode pensar num ícone para ele? Nós insistimos que todos os objetos candidatos tenham um ícone, desde a primeira sessão de modelagem, e a equipe concorde com esse ícone que irá representar um objeto dessa classe. Ícones são necessários para simplificar o protótipo a fim de distinguir os diferentes tipos de objetos. Ícones discutidos desde o início também fazem com que a equipe mantenha o foco na natureza concreta dos objetos. O desacordo sobre os ícones sugeridos algumas vezes revela uma diferença fundamental no entendimento da definição do objeto. Se o projeto progredir para a fase de liberação então é importante desenhar um bom conjunto de ícones que sejam aceitáveis para um amplo conjunto de usuários. Os ícones devem ser tanto esteticamente apelativos individualmente quanto formar um conjunto visualmente coerente quando juntos. Mas durante a exploração não existe necessidade de ser rigoroso, e é freqüentemente oportuno usar ícones das bibliotecas padrão. (A distribuição do Naked Objects inclui uma modesta biblioteca de ícones disponível para ser usada na exploração).

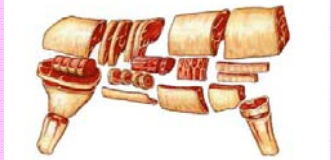
Existe muita conversa sobre a necessidade dos ícones serem intuitivos. Eles certamente precisam ser visualmente distintos, tal que nenhum usuário possa se confundir facilmente. E eles precisam ser 'associativos', uma vez que a pessoa saiba quais ícones estão associados a cada tipo de objeto de negócio, eles possam facilmente memorizar essa associação. Mas não existe necessidade que usuários estejam aptos a entender o significado de um ícone sem que alguém explique, e pesquisas mostram que essa meta é sempre impossível de ser alcançada.

Existe algum sinônimo óbvio? Considerando a maneira que a lista foi gerada, é provável que existam candidatos que são ou idênticos ou fortemente sobrepostos. (Caso e Solicitação foram exemplos iniciais na exploração do DSFA). As diferenças podem ser sutis. A melhor estratégia é tentar combinar um dentro do outro. Até quando as duas sugestões chegam de membros diferentes da equipe, nós achamos que eles normalmente concordam em aceitar tentativas de união, desde que uma nota seja mantida da distinção original. Usar um esboço eletrônico para colar notas ajuda consideravelmente. Talvez, posteriormente, prove que seja necessário dividir o objeto composto, embora seja mais provável achar que a divisão acorre ao longo de linhas diferentes.

“Dividir o problema seguindo as suas articulações naturais” é a meta dos modeladores de objetos. O princípio nem mesmo é novo: em Sócrates de Phaedrus de Platão aconselha que nós devemos observar a articulação natural de um problema, “não mutilar alguma parte como um inexperiente açougueiro”.

Considere os diagramas mostrando os diferentes padrões para dividir uma carcaça de boi na França, Inglaterra e Estados Unidos. Por que existem tais diferenças nesses três padrões, quando todos têm uma meta comum de maximizar o rendimento da carcaça? Uma explicação está na culinária. Pense na França e você pensa na caçarola; a Inglaterra tradicionalmente amam seus assados; enquanto que os Americanos preferem os grelhados. Os três padrões diferentes refletem de certa forma essas preferências.





No entanto, a culinária está mudando. Poucas famílias Inglesas regularmente se sentam para comer um tradicional assado; até os Franceses estão se sentando menos desejando gastar menos horas do dia cozinhando. A influência da fartura criou demanda para cortes mais exóticos. Uma conseqüente complicação é que a estrutura da indústria está mudando: muitos dos cortes atuais deixaram o açougue local para as facilidades de comida processada central. Isso fornece grande economia de escala, mas tem tendido a reduzir a flexibilidade: o cliente deve escolher entre os diversos cortes pré-empacotados ao invés de estar apto a pedir ao açougueiro por um particular corte ou peso.

Uma das maiores cadeias de supermercados da Inglaterra adotou uma ‘arquitetura’ totalmente nova com base nos 10 maiores construtores conhecidos como ‘primals’. Carcaças são liberadas aos supermercados já cortados nos 10 primals, e o gerente de alimentação local decide como melhor subdividir cada primal. Assim, a nova arquitetura não apenas atende o fato que condições mudaram, mas que eles continuam a mudar. Esses supermercados podem dizer que encontraram o corte natural? Nós duvidamos que eles declarem que o novo sistema é perfeito, mas é interessante notar que os primals seguem o grupo de músculos da carcaça que mais se aproxima dos padrões anteriores.

Essa é uma metáfora muito boa de negócio. Os ‘cortes’ de um negócio estão refletidas nas fronteiras organizacionais (intera e externa), definições de produtos e serviços, estratégias de canal e papéis individuais. Elevar a competição e mudar condições de fornecimento e demanda tem exigido novos conceitos que abreviam esses cortes. Abordagens como reengenharia de negócio tem tornado as organizações mais otimizadas para as condições do dia, mas não mais aptas a responder mudanças futuras.

Num projeto Naked Objects, o objetivo é que os objetos de negócio representem os 'primals' ou grupos de músculos de negócio, não as definições de estrutura ou produto/serviço da organização, particulares do dia.

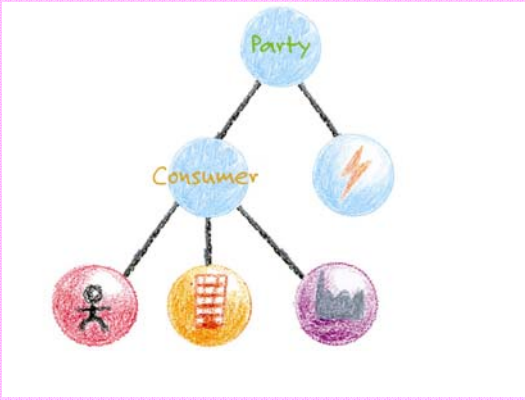
Por exemplo, uma empresa de seguro alternativo que estava implementando um novo conjunto de sistemas de contrato, inicialmente baseou sua representação de objetos na indústria – conceitos de negócio padrão de 'acordo' e 'facultativo' – os dois tipos principais de contratos de seguros alternativos, refletiram também na sua estrutura organizacional. Conforme o projeto progredia, a especificação dos dois elementos construtores foi modificada, e eles foram renomeados como 'portfólio' e 'risco individual' – pois a empresa entendeu como um corte mais natural. Esses dois novos conceitos puderam facilmente ser mapeados sobre as divisões 'acordo' e 'facultativo' do negócio para propósitos de continuidade, mas a nova representação abria possibilidades de construir novos produtos de seguro híbridos. Em outras palavras, o critério para decidir a qualidade da representação de objeto não estava adaptado para a um propósito imediato, mas a várias situações futuras que puderam ser expressas usando essa representação.

Não existe fórmula efetiva de encontrar os cortes naturais, a despeito dos melhores esforços de muitos pesquisadores para encontrar um. A única maneira de alcançar isso é manter reunidos os candidatos de objetos relacionados, então dividi-los posteriormente numa linha diferente para ver as possibilidades emergentes. As principais representações de objeto falham em encontrar cortes naturais devido aos projetistas que não acreditam no valor dessa disciplina.

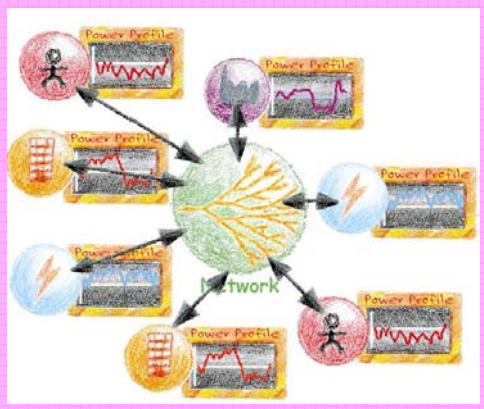
Um grupo de candidatos compartilha comportamentos comuns? Vários candidatos podem não ter sinônimo, mas eles podem compartilhar comportamentos comuns. Assim, você pode fazer uso de tipos de abstrações. ContaCorrente e ContaPoupança são ambos tipos de ContaBancária, por exemplo. Para reduzir a lista você pode inicialmente focar na definição ContaBancaria e mais tarde cuidar das implementações mais específicas desse tipo.

Cuidado! Evite aproveitar as discussões sobre hierarquia de heranças. Java distingue entre 'herança', onde uma classe herda os meios de realizar uma responsabilidade de suas superclasses, e 'implementação de interface' onde uma classe meramente possui o mesmo tipo, por meio da mesma interface externa compartilhada por outras classes. Este último é um poderoso construtor, e no contexto de uma sessão de modelagem de objetos de negócio, é atualmente mais poderoso do que o conceito de herança. Novamente, o Naked Objects faz uso disso obviamente: se as classes A e B implementam a mesma interface abstrata, isso significa que em qualquer lugar que você solte uma instância da classe A, você pode também, soltar uma instância da classe B. Assim, nós inicialmente pensamos num Pedido sendo associado a um Cliente, e então percebemos que Pedido pode ser de um Agente ou um Distribuidor. Tudo que precisamos fazer é

definir uma interface abstrata tal como ParceiroDeNegócio e assegurar que Cliente, Agente e Distribuidor implementem essa interface. Eles podem muito bem implementar outras interfaces comuns tais como Comunicável, com o significado de que nós sabemos chamar qualquer um deles para que forneça um endereço de comunicação.



Aparelhos Domésticos, Negócios e Indústrias são todos consumidores de eletricidade. Ambos, Consumidores e Geradores são Partes de negócio. No entanto, provavelmente seria melhor implementar essa associação como interfaces compartilhadas do que uma hierarquia de herança.



Implementar a interface Parte permite que Consumidores e Geradores sejam usados indistintamente em certos contextos. Por permitir que Perfil de Potência manipule demandas positivas e negativas, poderia ser possível um híbrido Gerador/Consumidor – alguma coisa que a desregulamentação elétrica irá encorajar.

Isso nos ajuda a evitar receber questões apaixonadas tal como se Cliente é uma subclasse de Parte, ou de Entidade; e Empregado uma subclasse de Parte, ou de Recursos? Nós fazemos pouco uso dessa forma de herança altamente abstrata. Nossa regra é somente usar a herança quando os usuários por eles mesmos apoiarem alegremente que a declaração ‘X é uma forma especializada de Y’. Se o

usuário meramente disser ‘X parece compartilhar coisas comuns com Y’ então use interfaces.

Um candidato é apenas um componente de outro? Alguma coisa pode ser um claro candidato para um objeto (é instanciável, possui responsabilidades reais, e não se sobrepõem a outros objetos), mas ele somente existe dentro do contexto de um outro objeto. O termo técnico é ‘agregação’. Um bom exemplo é o relacionamento entre Pedido e LinhaDePedido. Existem boas razões para tornar LinhaDePedido num objeto: ele possui responsabilidades reais tais como verificar o nível de estoque e aplicar o VAT (Taxa de Importação). Tornar uma classe instanciável facilita a adição de uma nova LinhaDePedido a um Pedido (novamente, o Naked Objects facilita a visualização desse conceito). Mas uma LinhaDePedido nunca precisa existir fora do contexto de um Pedido. LinhaDePedido é um objeto importante, mas ele não é um dos nossos objetos de alto-nível.

A agregação não deve ser confundida com associação. Um objeto Cliente pode conter referências a múltiplas Reservas, mas tais reservas não estão agregadas no Cliente: eles existem por seus próprios méritos.

Um simples exemplo de agregação é um objeto Produto que pode ter um UPC (Universal Produt Code). Bons modeladores irão indicar que é melhor modelar o UPC como um objeto do que, digamos, como uma string, porque isso permitiria que o UPC estivesse apto a imprimir um código de barras, consultar um banco de dados, entre outras coisas. Tudo certo, mas o UPC é muito mais um objeto secundário que pode ser tratado num outro momento do processo: ele não fará parte da lista dos cinco a dez objetos de negócio principais. Apenas mantenha uma nota disso dentro as responsabilidades apropriadas para o objeto Produto.

O objeto é um possível ponto de partida para uma atividade do usuário? Objetos que podem ser o ponto de início para um cenário de negócio são mais merecedores do termo objeto de negócio ‘principal’ do que outros. Em nosso exemplo de reserva de limosine, podemos certamente realizar o caso que um cenário de negócio pode iniciar com uma nova Reserva ou com um objeto Reserva existente, um objeto Cliente, uma Cidade, ou uma Localização. Seria menos provável começar com um CartãoDeCrédito, mas se nós enxergarmos CartãoDeCrédito como implementando um tipo mais abstrato de MétodoDePagamento, que poderia também ser estendido para cobrir ContaDaEmpresa, então MétodoDePagamento pode facilmente formar um ponto de início de um cenário.

Assim, no final do primeiro dia da fase de exploração, você terá como objetivo chegar a uma lista contendo entre cinco a dez classes de objetos de negócio principais. Cada um das quais será de interesse chave para usuários e poderá potencialmente formar o ponto de início para alguns breves cenários de negócio. Para algumas dessas classes, subclasses especializadas poderão já ter sido identificadas. Algumas delas também terão classes que nós chamamos de

‘secundárias’ ou ‘agregadas’. Você pode também ter identificado algumas necessidades para interfaces abstratas comuns as quais permitiriam que diferentes classes fossem usadas indistintamente em certos contextos (isto é, você poderá soltá-las dentro de um mesmo campo).

Nós normalmente achamos que durante o primeiro dia, podemos fazer um início útil definindo e refinando responsabilidades de objetos. De fato, isso normalmente irá reforçar ou contestar a nossa escolha inicial de objetos de negócio principal. Você poderia muito bem encontrar, por exemplo, que duas classes de objetos aparentam compartilhar várias responsabilidades e que devem ser fundidos. Ou você pode achar que especificou uma responsabilidade para um objeto que não parece possuir compatibilidade natural com o seu nome – uma boa indicação de que você deve considerar dividir esse papel em dois.

Responsabilidades de objetos podem, convenientemente, ser divididas em duas categorias: ‘saber o que’ e ‘saber como’.

Adicionando responsabilidades ‘saber o que’

Responsabilidades ‘saber o que’ envolvem os atributos (valores simples tais como: nomes, datas e preços) e associações a outros objetos de negócio.

Estritamente falando, nós estamos nos referindo apenas aqueles atributos e associações que são deliberadamente planejadas para serem acessíveis de fora do objeto. O Naked Objects ajuda a tornar isso claro. Qualquer objeto com um método público ‘get’, será automaticamente evidenciado aos usuários no momento de visualização desses objetos. De fato, como Wirfs-Brock disse, [Wirfs-Brock1989], existem atributos e associações que o objeto ‘sabe como’ comunicar-se com o usuário externo, e/ou permite que o usuário externo especifique ou modifique. Assim, de fato, todas as responsabilidades são realmente ‘saber como’. O valor de preservar alguma distinção artificial entre os dois tipos de responsabilidades é que ela potencialmente encoraja os modeladores a promover comportamentos importantes.

Em teoria, seria melhor começar especificando essas responsabilidades ‘saber como’ importantes: elas têm mais a ver com a nossa meta de completeza comportamental. Na prática, também achamos que isso seja plausível, mas os modeladores de objetos mais experientes acham que isso é difícil de fazer. Deixar que a equipe dê um candidato de objeto um pouco mais substancioso, na forma de alguns atributos e associações, ajuda a manter a motivação do processo de criação dos Naked objects.

Para identificar as responsabilidades de associação, questione a equipe: ‘Se você abrir um desses objetos, quais outros objetos você esperaria obter diretamente dele?’ Um engano comum cometido por modeladores inexperientes é incluir num objeto atributos de outros objetos. Assim, eles especificam um Pedido ‘sabe o

nome e o endereço do cliente', quando ele deveria 'saber o Cliente' (que por sua vez sabe seu nome e endereço). Encorajar a equipe a especificar associações antes dos atributos simples, ajuda a evitar esses enganos. Isso os ajuda a pensar em navegar nas coisas que eles podem encontrar fora de um Pedido ao invés de procurar como sendo parte de um Pedido.

Para cada associação, pergunte também se a associação precisa ser bidirecional ou não. Se uma Reserva precisa ter um Cliente associado, você quer estar apto, também, a obter a Reserva a partir do objeto Cliente? Algumas vezes você não verá a necessidade para isso até começar a caminhar através de alguns cenários. Se um objeto é agregado dentro de um outro, então não existe a necessidade de fazer isso: uma LinhaDePedido não precisa possuir uma referência explícita ao Pedido, porque ele nunca poderá ser acessado sem que seja pelo Pedido. Saiba também que embora seja tentador tornar todas as associações navegáveis em ambas as direções, isso pode incorrer em alguma sobrecarga em termos tanto do esforço de desenvolvimento quanto de desempenho (por causa das implicações de banco de dados), assim, não faça isso sem alguma razão.

Voltando para os atributos de dados simples, tente mantê-los o menor possível. Existe uma grande tentação de começar com uma lista grande de atributos simples de 'pouco valor', apenas porque eles são fáceis de pensar: nome, endereço, sexo, data de nascimento, cor do cabelo, entre outros. Tais atributos raramente adicionam valor e eles distraem a equipe das responsabilidades de maior valor.

Nós recomendamos que, na primeira iteração, pelo menos, você mantenha os atributos básicos ao mínimo necessário para identificar o objeto unicamente e convenientemente. Uma boa regra para uma implementação em Naked Objects é: adicionar apenas aqueles atributos que você precisa usar no método de título. Adicione outros atributos quando você precisar completar os seus cenários de teste operacional – mas apenas após questionar seriamente se esse atributo não poderia ser trocado por uma responsabilidade 'saber como'.

Adicionando responsabilidade 'saber como'

As pessoas acham difícil conceber as responsabilidades 'saber como'. Você irá provavelmente começar com apenas alguns deles, e então adicionar mais enquanto você considera os cenários de negócio. Uma maneira de começar é se familiarizar com outros exemplos. O modelo de objetos de negócio do DSFA seria um bom ponto de partida.

Uma técnica útil é olhar em cada uma das responsabilidades (isto é, as associações e atributos) e se perguntar se seria melhor que eles fossem iniciados como uma responsabilidade 'saber como'. Alguns exemplos são exibidos no painel.

Tente especificar responsabilidades de objetos normalmente iniciados com o que um objeto precisa saber. É uma boa disciplina questionar e perguntar o que o objeto deveria estar apto a fazer com o que ele sabe.

Algumas vezes pode ser importante levar esse princípio ao extremo. Por exemplo, numa ocasião nós desafiamos uma organização de um setor público sobre o porquê das definições de Cliente incluíam a data de aniversário como um 'saber o que'. A audiência ficou incrédula. A data de aniversário era necessária devido a vários propósitos; eles disseram: para a identificação (nós buscamos o Fred Smith correto?), para autenticação (a pessoa do outro lado do telefone é quem disse que é?), para determinar se um cliente é um adulto ou não. Nós questionamos que cada um desses usos teria que ser modelado como responsabilidade 'saber como'. Pode ser que o objeto use uma data de aniversário armazenada para esses propósitos, ou pode ser que não (o objeto pode delegar a responsabilidade para uma agência externa). De fato, é boa prática basear a identificação e a autenticação na mesma data. Mais ainda, se eles projetaram o objeto Cliente para estar apto a responder a questão 'O [cliente] tem acima de 18?', ao invés de 'Qual é a data de aniversário do [cliente]?', então eles não deixarão vulneráveis a mudanças do mau uso da informação de idade. (Se alguém pensa que essa linha de pensamento é artificial, então deve ler a legislação de privacidade eletrônica da Europa).

Evite a tentação de preencher a página com responsabilidades que são automaticamente fornecidas pelo framework Naked Object. Não existe a necessidade, por exemplo, de especificar a cada atributo e associação uma correspondente responsabilidade 'saber como' para leitura e escrita – isso deveria ser assumido. A única razão de você especificar que o objeto Reserva 'sabe como

escolher um Lugar [no vôo]' é se o objeto Reserva por si mesmo soubesse fazer a seleção ao invés do usuário.

Da mesma forma, não existe necessidade de especificar a responsabilidade para o objeto se fazer persistente, ou gerenciar a autorização e segurança, distribuição ou controle de versão. Existem capacidades genéricas fornecidas pela infra-estrutura e presume-se aplicar a todas as classes de objetos de negócio.

Você pode também omitir as responsabilidades de classe genéricas. Todas as classes Naked Objects automaticamente fornecem as responsabilidades de classes genéricas (a menos que o programador as suprima):

- Criar uma nova instância de classe.
- Recuperar uma particular instância pela sua referência única.
- Listar todas as instâncias compatíveis com um conjunto de critérios.
- Lista de subclasses.

Você pode adicionar responsabilidades específicas de classe para uma classe de negócio – por exemplo, para criar um particular tipo de relatório sobre uma instância dessa classe – mas a necessidade para isso é mais provável que surja mais tarde na fase de exploração.

Lidando com conceitos de processo

Nós já deixamos claro que não concordamos com a prática de transformar use-cases em objetos, nem de dividir em papéis entre objetos Entidade ou Modelo e objetos Controladores. Usando Naked Objects, apenas a primeira categoria de objetos é permitida. Para colocar de outra maneira, devemos pensar que todas as classes de objetos de negócio devem ser persistentes.

Dito isso, é legitimo pensar em categorias amplamente diferentes, ou estereótipos de objetos de negócio [Wirfs-Brock'Characterizing your objects'] [Coad1999]. Em particular, nós freqüentemente descrevemos a distinção entre objetos intencional e não-intencional. Objetos não-intencionais são coisas como Produto, Cliente, Empregado e Localização. O estado desses objetos irá mudar no tempo, e esse estado irá ser persistente, mas as mudanças não ocorrem numa particular direção. Elas podem ser entendidas como aleatórias.

O estado de um objeto intencional geralmente muda numa direção pré-ordenada. Assim, um Pedido pode ir do estado de Confirmação, para Confirmado, para Lançado, para Faturado. O estado pode ocasionalmente retroagir, ou o Pedido pode ser finalizado prematuramente, mas existe uma clara direção pretendida. Em geral, apesar do estado de um objeto não-intencional ser apenas a aglomeração de seus vários atributos e associações, um objeto intencional tem maior probabilidade de ter um estado explicitado, representado por um único campo que pode tomar um valor dentre um conjunto finito de valores pré-definidos. É também

freqüentemente apropriado modelar o comportamento dos objetos intencionais usando diagramas de transição de estados, que especificam as condições sob às quais o objeto irá mover de um desses estados pré-definidos para um outro. Além disso, algumas pessoas descrevem tais objetos como 'com estado' ao invés de intencionais.

Portanto, um objeto intencional não é apenas um objeto de processo, ou um Controlador, com um outro nome? Existe uma importante diferença. Em nossa abordagem tanto objetos não-intencionais quanto intencionais são objetos Entidade. Por definição, todos eles são persistentes. Eles continuam a existir como objetos mesmo quando eles atingirem os seus estados finais pretendidos. Essa não é a única idéia. Nós temos tido muitas conversas com modeladores experientes de objetos e com autores bem conhecidos que chegaram na mesma conclusão: Transferência, Saque e Depósito (numa aplicação bancária) não são use-cases ou objetos Controladores, mas Entidades genuinamente persistentes. Mas esse tratamento não é o mesmo encontrado na maior parte da literatura de objetos.

Os Naked Objects tomam vantagens de sua maneira clara de pensar. Na maioria das abordagens de modelagem de objetos, uma vez que uma transação, um processo ou use-case estiver completado, não existe mais como nos referir explicitamente a ele. Numa aplicação desenvolvida usando Naked Objects, tais atividades são exibidas como ícones: você pode abri-los e inspecioná-los, e chamar qualquer ação válida que esteja disponível. Assim, mesmo se uma Transferência (bancária) tenha sido realizada, você pode examiná-la e decidir reverte-la, alterando seu imposto, ou notificando o cliente de sua finalização com sucesso.

Qualquer atividade de negócio cujo verbo que descreve a atividade, puder ser facilmente mudado para um substantivo, é o primeiro candidato para um objeto intencional. Dessa forma, os usuários podem ter um requisito de ajustar (verbo) os preços. Mas eles também falam sem problemas 'fazer o ajuste de preços'. Essa é uma dica para pensar sobre AjusteDePreços como um objeto entidade instanciável, exibido como um ícone na tela pelo mecanismo de visualização do Naked Objects. Uma outra boa dica, como outros indicam, é que verbos que podem ser facilmente modificados para substantivos estão normalmente expressos nos formulários de papel de um sistema manual.

Construindo os objetos de um protótipo

A segunda das três principais atividades durante a exploração é a construção de objetos de um protótipo. Seu objetivo é obter suas idéias de objeto implementadas no framework Naked Objects tão rápido quanto possível, para ajudar a equipe a visualizar os atributos, associações e (o mais importante) os comportamentos dos objetos.

Para cada um dos objetos de negócio que você identificou, você precisará:

- Definir uma classe NakedObject.
- Adicionar um ou mais campos de valor que possam ser usados para identificar cada instância dessa classe (por exemplo, um campo nome ou referência).
- Especificar as associações mais óbvias entre o objeto de negócio novo e qualquer outro tipo de objeto de negócio já especificado.
- Escreva um método título que irá identificar uma instância de objeto ao usuário.
- Adicione uma nova classe para o conjunto de classes numa aplicação teste.
- Especifique dois ícones (um maior e o outro menor) a serem associados à classe.

Uma vez que isso estiver realizado para cada uma das classes identificadas, então você pode iniciar o framework e usar a aplicação. Isso irá fornecer respostas imediatas, permitindo que você confirme ou questione suas decisões de projeto, e indique futuras possibilidades. O próximo passo é enriquecer a aplicação, por exemplo:

- Adicionando futuros campos contendo objeto de valor ou objetos de negócio associados.
- Considerando se cada campo for somente-leitura ou permitir atualizações e remoções.
- Adicionando métodos de objeto e talvez métodos de classes adicionais.
- Adicionando métodos *about* para controlar acesso individual, classes, campos e métodos baseados nas regras de negócio ou nível de autorização do usuário.

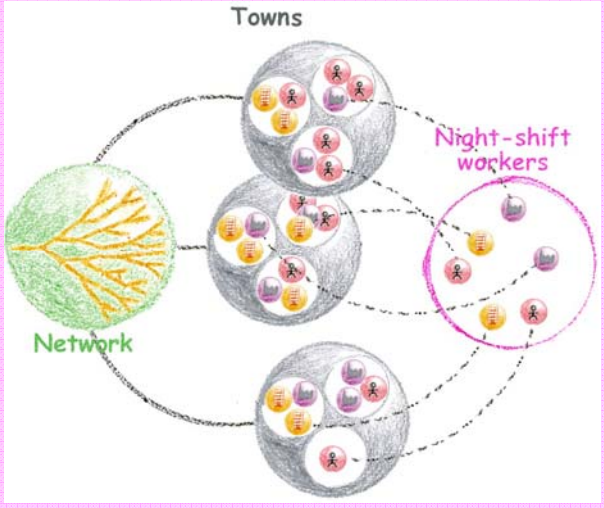
Algumas vezes, a prototipação ocorre em ‘off-line’. Traduzir uma lista inicial de objetos candidatos (produzidos no primeiro dia) dentro de um protótipo funcional pode levar várias horas de programação. O mesmo pode ocorrer quando, nas iterações subseqüentes, novas classes de objetos de negócio ou responsabilidades são identificadas, ou existir um significante ‘refatoramento’ do modelo de objetos, ou reinstalação de responsabilidades.

No entanto, mais mudanças sugeridas pela equipe – um novo subtipo, atributo, associação ou simples método – podem ser programados em minutos, e o Naked Objects torna tais mudanças visíveis, instantaneamente, ao usuário. Conseqüentemente, faz sentido tentar fazer a maior parte dessas mudanças quanto possível em tempo real. Observamos, com freqüência, que programadores dizem aos usuários: ‘Isso provavelmente levará alguns minutos: você pode voltar mais tarde se quiser’, somente para ouvir uma resposta do usuário: ‘Eu fico – eu estou adorando isso!’. O resultado positivo é que normalmente é possível passar por dez a vinte iterações com os usuários num único dia.

Uma das razões de ‘prototipar’ em tempo real é que isso encoraja a equipe toda a explorar mais de uma abordagem para um assunto de negócio. Exercícios de modelagem de negócio podem, com freqüência, cair numa situação desagradável de debates sobre duas maneiras diferentes de representar um domínio do problema [Riel1996]. Com Naked Objects, a maioria desses debates pode ser eliminada simplesmente oferecendo protótipos para ambas as abordagens e então mostrar aos usuários na tela. Quando confrontado com os dois modelos, possuindo ícones tangíveis que podem ser arrastados e soltos, pressionados com o botão direito do mouse e abertos para visualizar seus conteúdos, é surpreendente a maneira como a solução da modelagem superior chega rapidamente – se existir um – tornando-a óbvia.

Usando o protótipo para explorar cenários de negócio

A terceira atividade principal de exploração é usar o protótipo para explorar cenários de negócio. Durante cada iteração, formal ou informal, o protótipo evoluído deve ser testado contra os cenários de negócio. Existem dois tipos principais de cenários, operacional e ‘o que acontece se’.



Por visualizar possibilidades de objetos de negócio na interface de usuário, a companhia elétrica viu a possibilidade de criar novas comunidades de Consumidores com Perfil de Energia comum (ou complementares) – literalmente arrastando e soltando ícones sobre a tela. Essas comunidades receberam ofertas de desconto especial, ou puderam vender para outras companhias de energia.

Cenários Operacionais

Cenários operacionais são visualizações de uso específico do sistema. Eles cobrem tanto as tarefas que ocorrem com freqüência quanto os usos específicos, tais como: 'Mary Cahill tem três filhos menores de 16 anos, dos quais os dois mais novos são gêmeos, mas um está vivendo com a mãe de Mary'. 'Exercitar quanto a Pensão para Crianças está devendo' ou 'Nós temos um desconto de 15% em todas as cervejas, vinhos e alcoólicas em nossas lojas do hipermercado, e uma oferta promocional de "1,00 dólar" nos vinhos australianos em todas as lojas'. 'Resolver algum conflito de preços'. A melhor maneira de testar o poder do modelo de objetos emergente é aplicar vários desses cenários específicos do que apenas enfatizar tarefas freqüentemente repetitivas.

Os cenários operacionais sempre devem ser especificados em termos concretos. Isso previne que usuários formulem questões funcionalmente impossíveis. Por exemplo, considere um requisito de usuário ou estória parecida com 'O sistema deve estar apto a identificar e resolver ajustes de preços em conflito originários de diferentes fontes'. O critério de identificar conflitos pode ser simples (por exemplo, dois ajustes para diferentes quantidades aplicadas ao mesmo produto em períodos de tempo sobrepostos), mas a sua resolução pode não ser simples. Forçar o usuário a escrever um ou mais casos de teste concretos, cada um dando um exemplo de um conflito e especificar exatamente como um particular conflito seria resolvido irá, ou esclarecer o algoritmo ou convencer o usuário de que o conflito somente poderá ser resolvido por um ser humano.

Um cenário operacional pode muito bem destacar a necessidades de novas classes, subclasses de objetos, ou novos métodos e associações. Alguns cenários serão prototipados durante a exploração – outros podem simplesmente ser marcados para serem considerados na fase de Especificação. Em tais casos, é normalmente apropriado adicionar, digamos, novos nomes de métodos ao objeto, tal que esses cenários sejam exibidos como ações no menu pop-up do usuário, mas sem necessariamente escrever quaisquer códigos necessários para implementar esses métodos.

Os cenários operacionais devem ser documentados, pois eles formarão um conjunto útil de orientações durante a fase de especificação. Aqueles que foram deixados para que a especificação seja realizada em uma particular fase serão codificados mais formalmente como um ou mais testes de aceitação executáveis do usuário (descrita na seção Liberação). Quando você dominar a abordagem Naked Objects, você poderá decidir capturar os cenários operacionais de forma correta desde o início – nas isso não é essencial.

Demonstrações

Cenários operacionais são úteis como scripts de demonstrações. Durante a exploração, a regra é demonstrá-lo o quanto antes e com freqüência. Tais demonstrações podem ser internas à equipe, trazendo todos na velocidade normal sob a última versão do modelo de objetos de negócio; ou podem estar fora da equipe de exploração para testar uma idéia em particular ou para ajudar a construir um suporte político. Sabendo que sistemas construídos a partir dos naked objects 'parecem' muito diferentes dos sistemas convencionais, nós achamos que o primeiro projeto numa organização conquiste muita atenção, com demandas freqüentes de demonstrações até para aqueles que não estão envolvidos nesse domínio de negócio. Você precisa estar pronto para demonstrar o protótipo de desenvolvimento a qualquer momento e normalmente encarregamos um particular membro da equipe como sendo o especialista de demonstração (embora isso possa ser combinado com outros papéis ou responsabilidades). Em alguns casos, até temos uma máquina dedicada para demonstrações, atualizando a demonstração no final de cada dia quando todas as várias modificações estiverem integradas.

Você precisa desenvolver uma demonstração de 'tour geral' que mostre, grosso modo, cada um dos objetos conhecidos e saber como fazer. O restante das demonstrações são normalmente baseados nos cenários de teste operacional. Aqui, novamente, a natureza concreta dos cenários ajuda. Eles liberam o demonstrador de ter que pensar em dados de exemplos realísticos. (Usar dados que não sejam reais em demonstrações deixa a platéia muito confusa – a maioria dos não programadores não identifica, naturalmente, clientes ou produtos quando chamados de Foo e Bar!).

Paradoxalmente, ainda que uma das motivações, ao fornecer aos usuários o acesso aos naked objects, seja a de permitir que eles tenham maior escolha em como cuidar de um particular problema, nós recomendamos que demonstrações sejam fortemente roteirizadas. Interrupções contínuas na demonstração de um caso de teste como no seguinte comentário 'em vez disso, é claro que eu poderia ter feito isso um pouco diferente', deixam as pessoas confusas. Faça isso somente quando sua platéia questionar sobre as possibilidades, ou sugerir que a maneira como você acabou de abordar alguma coisa vai contra a intuição, ou que não é a melhor maneira. Uma outra abordagem é selecionar uma faixa de scripts de demonstrações para mostrar problemas que estão sendo cuidados de várias maneiras diferentes, ou de diferentes pontos de início.

Cenário 'o que acontece se'

Os cenários 'o que acontece se' testam o modelo de objetos contra futuras mudanças no negócio. Tais mudanças podem surgir de alguma nova capacidade tecnológica tais como smart cards, nova geração de telefones móveis, ou assinaturas digitais. Eles podem ser afetados por novas legislações governamentais, regulamentos industriais, ou regras e políticas internas. Ou eles

podem refletir mudanças no relacionamento entre fornecedores e parceiros, e com alguns clientes em particular. O painel mostra alguns exemplos da vida real desses cenários.

Alguns exemplos de cenários 'o que acontece se' durante as sessões de exploração:

A idéia não é assegurar que o modelo de objetos arque com todos os cenários hipotéticos 'o que acontece se' sem modificações, mas a de assegurar que as modificações requeridas sejam fáceis de identificar e que sejam localizados em uma ou talvez duas classes de objetos, ao invés de estar espalhado em todo o sistema.

Para ser mais específico, o objetivo do teste de cenários 'o que acontece se' não é assegurar que você tenha especificado todas as responsabilidades que irão sempre necessitar de um objeto: novas responsabilidades serão adicionadas ao longo da vida do sistema. O objetivo de testarem cenários 'o que acontece se' é assegurar que você tenha definido corretamente a fronteira entre os objetos. Em outras palavras, que você dissecou o problema de forma natural.

O resultado da exploração

O principal resultado da fase de exploração é o protótipo funcional. Devido ao Naked Objects exibir os principais objetos de negócio e seus principais comportamentos visíveis e acessíveis aos usuários, o protótipo é por si só a melhor documentação do modelo de objetos de negócio. A correspondência um-para-um entre o que o usuário vê na tela e as definições de classe Java subjacente eliminam a necessidades da maioria das documentações geradas pelas abordagens convencionais. Essa é uma das razões do porque participantes

acham tão divertido e recompensador. Em contraste, muitas documentações produzidas nas fases iniciais das abordagens convencionais são na verdade contraprodutivos: são difíceis de escrever, ingrato de ler, e muito difícil de manter sincronizado durante a evolução do projeto.

Além do protótipo, você deve documentar as responsabilidades de alto nível das classes de objetos de negócio. Enquanto algumas responsabilidades são traduzidas diretamente em atributos visíveis, associações e métodos de ação, outras não são traduzidas tão diretamente assim. Algumas responsabilidades terão sido identificadas, mas não simuladas no protótipo. E algumas responsabilidades existirão para orientar a direção do desenvolvimento futuro (um exemplo típico é a responsabilidade requerida por um objeto Cliente de 'estar apto a comunicar [com o cliente] usando seu canal preferido' – embora no protótipo, e talvez a primeira implementação liberada, cuide apenas dos canais correio e telefone).

Essas responsabilidades devem ser obtidas na forma que sejam legíveis. O modelo de objetos de negócio da DSFA fornece um bom exemplo de tal documentação. Uma outra abordagem é usar a XML para documentar os objetos e suas responsabilidades, com tags específicos para nomes de Classe, Métodos e outros. Isso irá auxiliar na produção subseqüente de um documento navegável eletronicamente.

O desafio é manter essa documentação sincronizada com o sistema conforme ele evolui. Existem ferramentas caras e sofisticadas disponíveis para esse propósito, mas a nossa abordagem preferida é usar o código Java com sua própria documentação. Escreva responsabilidades de alto nível na forma de declarações comentadas no topo do arquivo de código para cada definição de classe. Se uma responsabilidade é completamente e obviamente clara somente por um método específico (tal como a responsabilidades do objeto Cliente de saber as suas Reservas), então a descrição da responsabilidade pode ser removida – pois seria redundante. Se uma responsabilidade não tiver sido implementada em código, ou existirem vários métodos em aberto, ou é mais abstrato do que a implementação atual (como em nosso exemplo acima do canal de comunicações), então a definição de responsabilidade deve ser mantida com um comentário.

Embutir tais definições de responsabilidades no código não garante que eles estejam consistentes com o código, mas é mais provável que fiquem. Além disso, você pode usar uma ferramenta tal como o JavaDoc para gerar automaticamente diretamente, através do código, alguma documentação eletronicamente navegável, incluindo tanto as declarações das responsabilidades de alto nível quanto os métodos ao nível de negócio existentes. Essa solução é muito mais efetiva.

E a UML?

Leitores podem perguntar porque nós fizemos pouco uso da UML (Unified Modeling Language) em nosso livro, quando sua notação é agora aceita como a forma padrão de documentar um modelo de objetos.

Capturar diretamente o seu modelo de objetos de negocio como definições naked objects permite que muitos relacionamentos de objetos sejam imediatamente visíveis aos usuários: abra algum objeto que você poderá ver os tipos de objetos com os quais ele se relaciona. Nós achamos que isso seja mais acessível aos usuários do que um diagrama de classes UML. E, naturalmente, a representação do naked objects é executável.

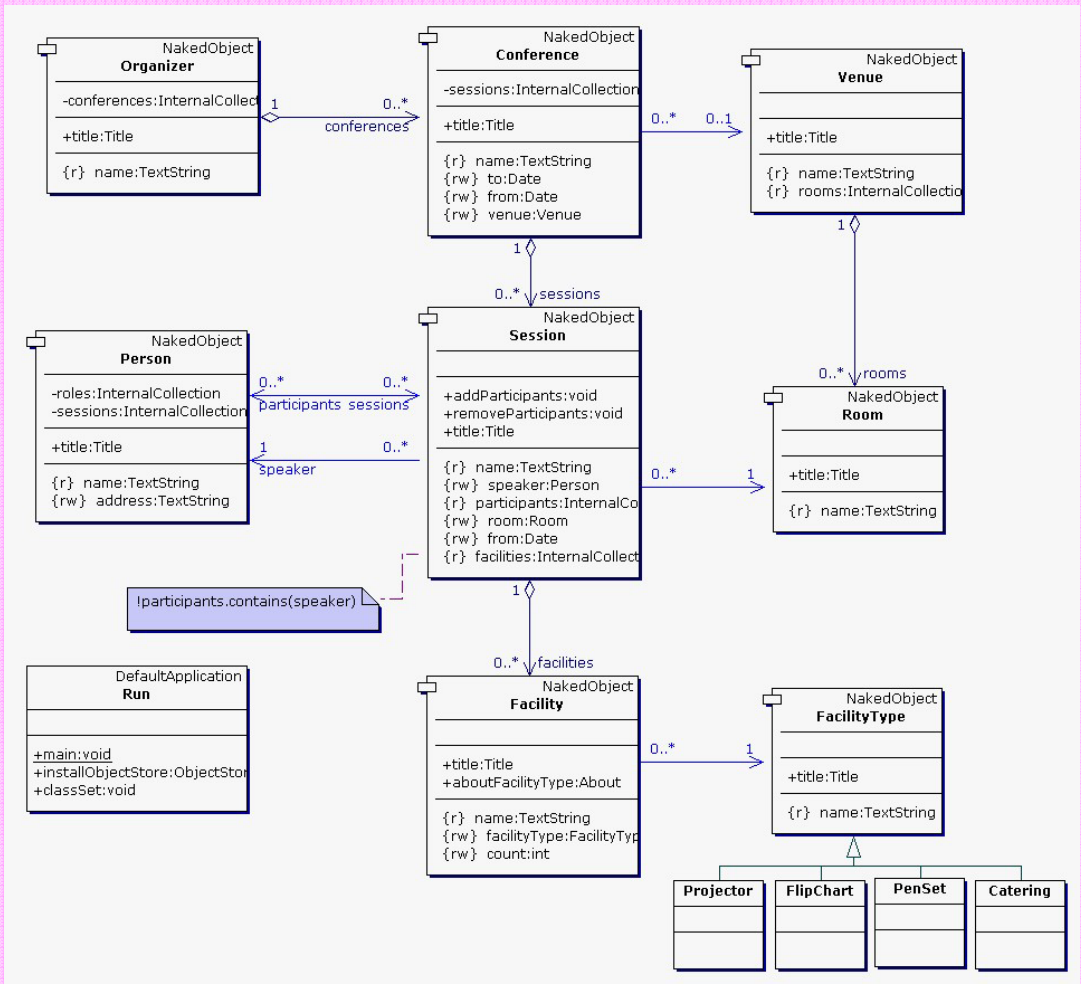
Reconhecidamente, a UML é muito mais rica do que o framework Naked Objects em sua habilidade de especificar diferentes formas e restrições sobre relacionamentos. A UML fornece a habilidade para visualizar diagramas de seqüência, diagramas de transição de estados e muitas outras formas úteis. Mas muitas pessoas não fazem uso de todas essas representações, e a menos que você tenha um conjunto de ferramentas sofisticadas, mantê-las todas sincronizadas pode ser um pesadelo.

Um outro argumento para usar a UML e a linguagem de restrições de objetos (Object Constraint Language - OCL) é que elas são linguagens neutras de programação. O Naked Objects é escrito em Java 1.1. Ele é compatível com o Java 1.2, 1.3 e 1.4 – mas não usa nenhuma das novas características. Essa escolha é consciente. Usar a última característica poderia facilitar a nossa tarefa de escrever o framework. Mas o Java 1.1 é altamente portável: existem implementações para uma vasta plataforma de hardware, desde os Palm Pilots até mainframes. Além disso, o Java 1.1 é agora um padrão público para todas as intenções e propósitos: existem disponíveis compiladores de código aberto, até mesmo máquinas virtuais Java (Java Virtual Machines - JVM) de código aberto. Em outras palavras, a linguagem Java usada para especificar uma aplicação Naked Objects é tão aberto quanto a UML.

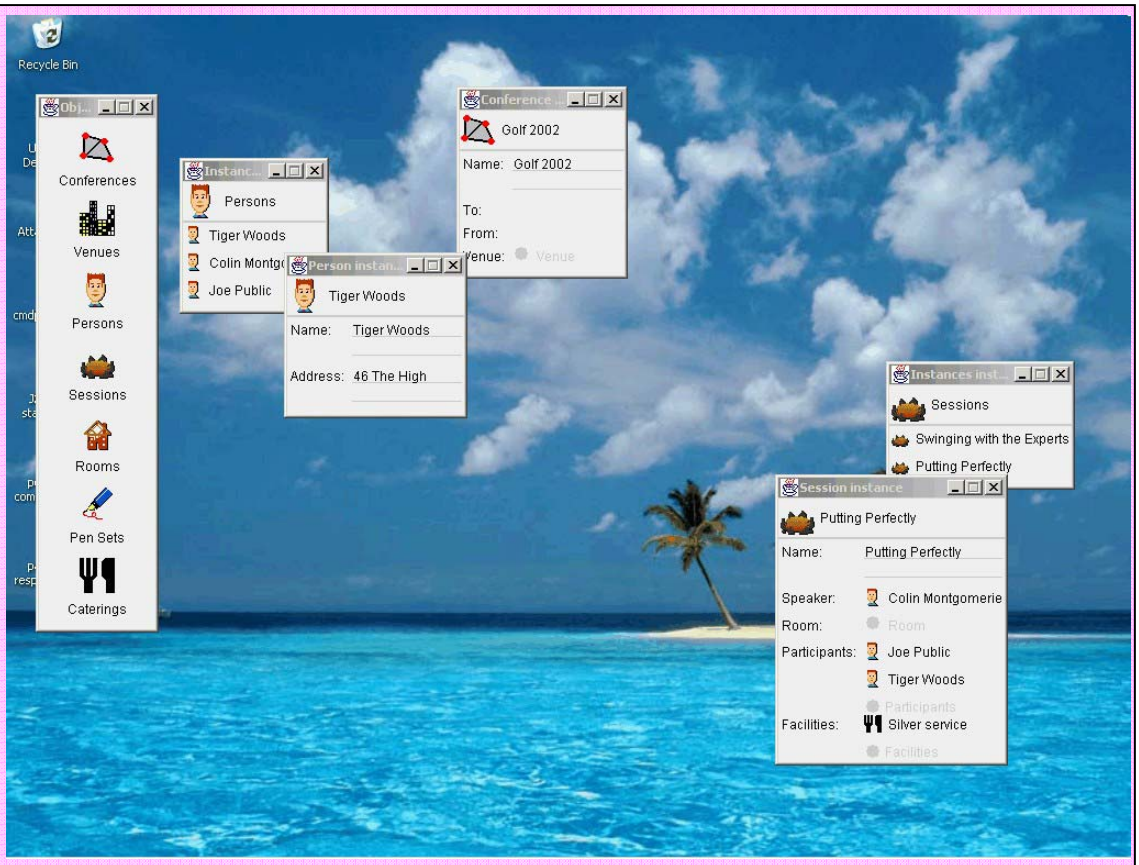
Existem outras vantagens de se especificar regras de negócio, relacionamentos e restrições diretamente usando uma linguagem de programação de padrão-aberto, simples e bem estabelecida, do que usar alguma forma de pseudocódigo. Ela encoraja uma correspondência próxima entre a modelagem de objetos e a programação orientada a objetos, a qual acreditamos que seja uma boa coisa. Por exemplo, nós achamos que muitos padrões orientados a objetos largamente considerados como boas práticas por programadores experientes (por exemplo [Gamma1995]) são altamente aplicáveis em nível de modelagem de negócio. Veja por exemplo o uso do padrão Strategy no Estudo de Caso: Atrasos e Cobranças – esse padrão no modelo de objetos de negócio corresponde diretamente ao mesmo padrão do código.

Uma coisa que pode mudar nossa visão sobre a UML é o recente surgimento de ferramentas (tais como TogetherJ) que pode transformar automaticamente diagramas UML em estruturas de código Java e fazer o inverso. Isso não só economiza tempo, mas o mais importante é que assegura que os dois permaneçam sincronizados.

Na conferência de OT2002, nós executamos um workshop no qual delegados tiveram apenas duas horas e meia para projetar um modelo de objetos para um sistema de administração de conferência e então construir um protótipo funcional utilizando o Naked Objects a partir desse modelo. Todas as quatro equipes de quatro pessoas tiveram sucesso e todos ficaram surpresos com o que eles alcançaram. A primeira equipe resolveu começar desenhando diagramas UML num flip chart, mas o protótipo resultante foi mais pobre do que os protótipos obtidos pelas outras duas equipes, as quais representaram suas idéias diretamente em Naked Objects. A quarta equipe desenhou diagramas UML usando o TogetherJ e geraram automaticamente, a partir desse diagrama, as definições de classes em Java, prontas para serem usadas pelo nosso framework. Muitos avanços subseqüentes foram feitos diretamente no código, mas o diagrama UML foi mantido atualizado na ferramenta. Esse foi o mais impressionante dos protótipos e suas telas são mostradas abaixo.



Este diagrama UML modela uma aplicação simples de gerenciamento de conferência.



O diagrama de classes UML foi automaticamente convertida num protótipo Naked Objects usando a ferramenta TogetherJ. (Agradeço a Dan Haywood por fornecer as telas capturadas).

Tais ferramentas mudam a natureza da UML. De fato, a UML agora, torna-se uma alternativa visual do código – pois ela esconde a complexidade e fazem com que as coisas como relacionamentos tornem-se mais visíveis. Nós gostamos dessa idéia. Não pensamos na modelagem e programação como duas atividades distintas: eles são visões diferentes da mesma coisa. Um outro movimento útil nessa direção é a recente pesquisa sobre máquinas virtuais UML. Na medida em que essa idéia progride, veremos potencialmente maior sinergia entre UML e Naked Objects.

Realmente, nós já temos claro que as telas do usuário exibidas através deste livro são resultados de apenas um dos muitos possíveis mecanismos de visualização. É concebível que um outro mecanismo seja um diagrama UML. Talvez alguém se inspire em escrever um.

A fase de especificação

Durante a especificação, os requisitos de negócio são formalmente especificados e priorizados, releases planejados, custos estimados e implicações de infra-estrutura identificadas. É importante indicar também que o comprometimento de desenvolver um sistema não é até a fase de especificação – o comprometimento inicial é apenas para obrigar a fase de exploração.

O estágio da especificação começa com a concordância sobre a funcionalidade de negócio requerida pelo sistema completo, ou com a concordância sobre a funcionalidade requerida pela primeira de uma série de atualizações planejadas. Depois disso, segue-se estimando os recursos necessários para desenvolver e testar a nova funcionalidade, integrá-la com alguns sistemas existente, fornecer qualquer que seja a nova infra-estrutura que possa ser necessária para treinar usuários, entre outras coisas. A maior parte desse planejamento é convencional, ainda que seja algo como uma magia negra, e nós sugerimos que você use o método de planejamento que funciona para você. No entanto, alguns praticantes indicam que o Naked Objects facilitam significativamente a estimativa – principalmente uma vez que você tenha executado um par de projetos sobre sua responsabilidade – porque todo o esforço de desenvolvimento está focado nos próprios objetos de negócio. Uma abordagem inteligente é classificar os vários métodos envolvidos, e que precisam ser escritos em termos de estimativa, usando o feedback para traduzir isso em recursos necessários. (Essa técnica tem muito mais em comum com o aspecto do ‘o jogo de planejamento’ da Extreme Programming).

Você vai achar que a fase de exploração facilita a tarefa inicial de especificar os requisitos. O patrocinador de negócio irá ter uma idéia muito mais clara do que eles querem. Algumas idéias já terão sido simuladas no protótipo, algumas apenas vislumbradas e registradas na forma textual. Além disso, mesmo no estágio de alto nível de planejamento, é possível entrar em acordo com o cliente sobre as capacidades requeridas em termos de objetos e responsabilidades desses objetos. Essa estrutura irá ajudar, ao menos um pouco, na tarefa difícil de estimar o esforço requerido de desenvolvimento.

Escrevendo user stories no estilo XP

Se você gosta da Extreme Programming (XP) então o nosso estágio de especificação pode ser executado de acordo com os princípios de 'planning game' (jogo de planejamento) do XP, no qual os requisitos de negócio são agora capturados como estórias de uma ou duas sentenças nos cartões de índice e então priorizados em releases. Com o protótipo orientado a objetos em mãos, é muito simples para os usuários, com um pouco de ajuda dos modeladores, especificar cada uma dessas estórias em termos de operações específicas sobre objetos específicos. Alguns exemplos são apresentados na figura abaixo.

Essas estórias no estilo XP foram geradas para o sistema ECS.

Nós organizamos esse grupo de sete estórias de forma que eles se referenciem em termos de funcionalidade e/ou complexidade. Nós achamos que encorajar usuários a especificar seus requisitos em ordem crescente de complexidade ajuda a evitar problemas iniciais em muitas situações complexas. De fato, é útil encorajar os usuários a expressar suas estórias do seu jeito desde o início: inicie com o caso mais simples possível, e então gradualmente adicione maior complexidade. (As estórias normalmente não seguem uma linha de execução simples de complexidade – podem existir várias linhas um pouco diferente numa atualização).

O conceito oficial do XP é que clientes devem priorizar as estórias de todas as atualizações, sem qualquer interferência dos desenvolvedores. Nós pensamos que neste caso, uma pequena ajuda do modelador principal irá produzir melhores resultados para ambas as partes. A elevação progressiva da funcionalidade, combinada com o fato de que todas as estórias são especificadas em termos de

operações sobre um modelo de objetos que foi testado contra múltiplos cenários durante a exploração, dá aos desenvolvedores muito mais segurança para adotar a disciplina programar uma estória de cada vez (program-one-story-at-a-time) do XP. Então, o refactoring entre estórias irá ocorrer normalmente ao nível de métodos, não nas fronteiras entre objetos, que é uma novidade muito boa para administradores de banco de dados.

A fase de liberação

Na fase de liberação, o sistema é desenvolvido, integrado, testado e liberado.

Nenhum código é levado da fase de exploração. O estilo de código adotado durante a exploração privilegiava a rapidez e era livre. Não havia absolutamente nenhuma ênfase no rigor de projeto e/ou teste; nem mesmo na validação das entradas, na validação das regras, ou prevenção de erros. A exploração assume ou que o protótipo somente será usado por um usuário especialista, ou que os erros não importam. Reutilizar esses códigos é criar futuros problemas.

O modelo de objetos e as definições de alto nível das responsabilidades que evoluíram durante a exploração são utilizados. Você pode até querer manter as assinaturas dos métodos, especialmente onde eles são reflexos diretos de certas responsabilidades, mas todo o código interno desses métodos deve ser totalmente apagado.

Nós achamos que deixar que os desenvolvedores preservem essas definições e assinaturas de métodos de classes, no ambiente de desenvolvimento, ajuda a reduzir a tentação de reaproveitar o código. Eles fornecem um modelo útil do projeto global e, psicologicamente, isso ajuda os desenvolvedores a superar o sentido de que eles estão começando tudo novamente com uma folha de papel em branco. Muitos realmente recebem com prazer a chance de escrever o código novamente a partir do zero de maneira mais disciplinada.

Se você usa um ambiente que pode manter uma representação UML sincronizada com o código, em ambas as direções, então você pode pensar na UML da forma como foi passada pela exploração para a liberação – desde que você também adote alguma convenção para registrar as declarações de alto nível das responsabilidades de objetos. Se você não tiver uma ferramenta, então nós achamos melhor que você não use a UML, ao invés disso registre as responsabilidades na forma de comentários textuais no topo de cada arquivo de classe Java.

Durante a liberação, a codificação das funcionalidades de negócio pode envolver a criação de algumas novas classes específicas não modeladas explicitamente, mas ao menos prevista durante a exploração. Pode também envolver a codificação de algumas novas classes agregadas que ficam inteiramente dentro de um dos

objetos de negócio. Mas na essência, a atividade de codificação na fase de liberação irá consistir de escrever métodos dos objetos de negócio.

Testes antes da codificação

Quando for codificar esses métodos, recomendamos fortemente que adote a disciplina 'testes antes da codificação': antes de você começar a escrever os métodos, você escreve uma ou mais testes de unidade executável que irá verificar se o método está corretamente implementado. Esses testes são continuamente executados durante o ciclo de desenvolvimento para assegurar que novos erros não tenham sido introduzidos no sistema. Usando o framework Junit é possível chamar esses testes num toque de um botão. Esta abordagem permite executar testes de unidade várias vezes por dia durante o desenvolvimento. Nós fizemos várias extensões no Junit para facilitar a sua aplicação no contexto do framework Naked Objects.

A Extreme Programming (XP) busca aplicar esse princípio para escrever testes executáveis honestos não apenas para testes unitários, mas também para testes de aceitação. Em XP, quando uma particular estória (ou requisito) está para ser implementada, uma breve descrição é desenvolvida através de discussões diretas entre desenvolvedor e usuário. Eles também escrevem em conjunto um ou mais testes de aceitação executáveis para essa estória. Por escrevê-los de forma que sejam executáveis, os desenvolvedores podem executar esses testes com freqüência durante o desenvolvimento da estória, para obter uma indicação do progresso, e poder executá-los como testes de regressão após subseqüente refatoração [Fowler2000]. O papel principal dos testes de aceitação em XP, no entanto, é como uma medida do valor liberado: quando todos os testes de aceitação executar, a estória é considerada implementada e os jogadores vão para o próximo passo.

O Naked Objects facilita a adoção dessa particular prática da XP. A codificação dos testes de aceitação executáveis para sistemas com interfaces gráficas de usuário (GUIs) é geralmente reconhecida como sendo muito difícil [Kaner1997]. Existem muitas ferramentas que podem capturar e reproduzir eventos de teclado e mouse para uma operação real do usuário, mas essa abordagem de teste tem muitos problemas [Groder1999]. Qualquer mudança no layout ou estilo da interface do usuário irá exigir que esses testes tenham que ser novamente registrados, e quase sempre isso também ocorre quando a aplicação é portada para uma outra máquina que não seja a de testes. Para piorar, do ponto de vista do XP, é que a gravação e reprodução de testes somente podem ser realizadas após o sistema ter sido desenvolvido. Algumas ferramentas fornecem uma linguagem de script ao nível de GUI que, em teoria, poderia permitir que scripts de testes sejam escritos antecipadamente. No entanto, como no projeto de sistemas convencionais permanece o problema de que é muito difícil para o usuário imaginar uma interface de usuário que ainda será implementada para possibilitar a escrita de scripts de teste com detalhes suficientes.

Escrevendo testes de aceitação executáveis

O framework Naked Objects aproveita os testes escritos em termos de ações de alto nível do usuário [Finsterwalder2001]. Quando os usuários chegarem para executar uma estória durante a liberação, eles podem usar o protótipo de exploração. Como as interações do usuário tomam uma forma padrão, os usuários podem especificar a implementação de alguma estória em termos de operações diretas sobre objetos de negócio (instâncias ou classes) que eles usaram durante a manipulação do protótipo. O protótipo está longe de estar completo, assim, algumas estórias irão exigir atributos, métodos e associações que não existem, mas achamos que os usuários não têm dificuldades de imaginar extensões dos objetos concretos que estão a sua frente. Isso é muito menos verdade se o protótipo tomar a forma 'scriptizada' padrão.

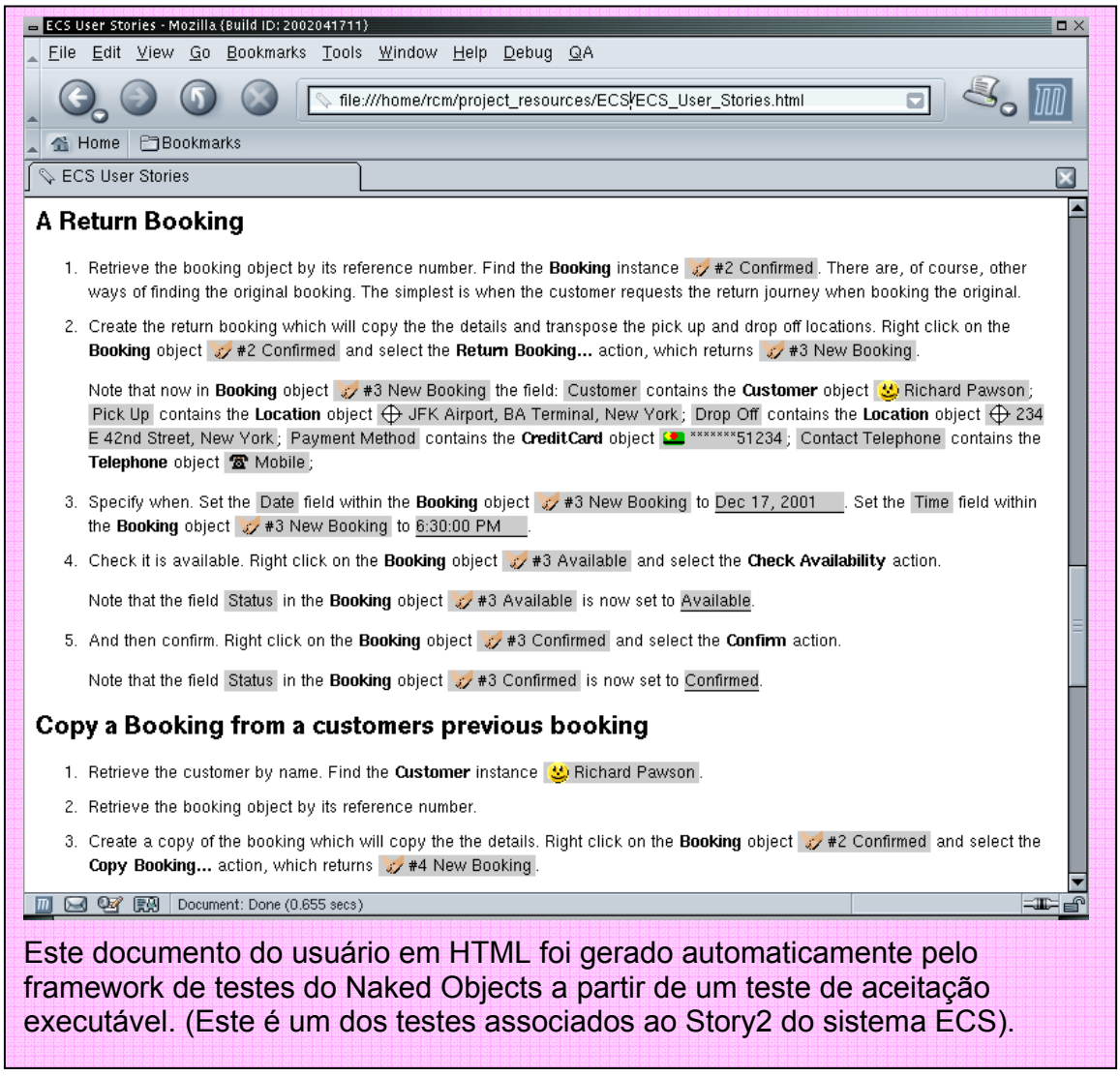
Então, quando uma nova estória for iniciada, um usuário e um programador sentam-se e juntos, escrevem a tarefa numa linguagem formal que consiste de operações estilo substantivo-verbo sobre os objetos e classes de negócio. (Note que isso é meramente uma definição de um conjunto de ações que um usuário pode escolher para seguir. Não é uma definição para um procedimento executável que irá, eventualmente, formar uma parte do sistema).

Nossa idéia original foi definir uma linguagem especializada e restrita à língua Inglesa para escrever esses scripts de teste de aceitação, usando XML. Essa linguagem poderia então ser facilmente convertida em testes executáveis Java. No entanto, rapidamente ficou claro que essa linguagem restrita à língua Inglesa era muito próxima à linguagem Java que nós tanto gostamos de trabalhar. Em teoria, o usuário poderia escrever a versão restrita à língua Inglesa sozinho. Na prática, nós achamos que independentemente da linguagem, um usuário e um programador trabalhando juntos é, no final de contas, mais efetivo e rápido. Portanto, trocar para um framework Java simples não impede o processo.

O programador captura detalhes da narrativa, ao vivo, como uma seqüência de métodos sobre classes de teste especializadas fornecida como parte do framework Naked Objects. Essas classes de teste simulam a interação entre o mecanismo de visualização do framework e os objetos de negócio. Quando os testes de aceitação de uma estória são finalizados, o programador pode então iniciar o projeto e codificação da funcionalidade necessária escrevendo testes de unidade para cada um dos métodos a serem criados ou alterados. Os testes de aceitação são executados de maneira muito similar aos testes de unidade sob o Junit. Da mesma forma que na abordagem Junit para testes de unidade, nós achamos que alguns programadores usam os testes de aceitação executável para guiar o trabalho: em outras palavras, eles cobrem os erros abandonados pelos testes em ordem seqüencial. Isso é uma questão de escolha pessoal.

Gerando automaticamente o manual de treinamento do usuário

Uma vez que você tenha um framework genérico para escrever testes de aceitação executáveis, outra coisa se torna possível: pode-se dar a essas classes de teste a habilidade de se 'autotraduzirem' num conjunto de instruções de usuário passo-a-passo, em língua Inglesa, para realizar a mesma tarefa manualmente. Essas instruções dizem ao usuário como realizar os mesmos testes de aceitação manualmente se ele desejar fazê-lo. Um exemplo dessa saída automática é apresentada a seguir.



Este documento do usuário em HTML foi gerado automaticamente pelo framework de testes do Naked Objects a partir de um teste de aceitação executável. (Este é um dos testes associados ao Story2 do sistema ECS).

Talvez o mais importante dessas instruções de usuários, em língua Inglesa autogeradas, seja a de representar uma porção significativa do manual de

treinamento do usuário do sistema em desenvolvimento. Afinal de contas, diferentemente dos testes de unidade (os quais se preocupam principalmente com a veracidade técnica), os testes de aceitação do usuário representam cenários que os usuários esperaram encontrar, algumas delas rotineiras e algumas delas excepcionais; e um manual de treinamento deve ter instruções explícitas sobre como lidar com tais cenários no sistema.

Algumas pessoas perguntaram se esse uso de scripts não está em divergência com a abordagem de tratar o usuário como um solucionador de problemas ao invés de um seguidor de processos. É importante entender que esses scripts são simulações do que um usuário faz. Eles não são executados dentro da própria aplicação, mas dentro do framework de testes que fica fora da aplicação. E até mesmo quando usado para gerar páginas de um manual de treinamento, esses scripts não dizem que 'você deve realizar essa estória dessa maneira', mas que 'você tem uma maneira possível de realizar essa estória'. Existem muitos scripts alternativos para uma mesma estória e podem ter muitas maneiras de realizar a mesma estória que não estão descritas como testes de aceitação formal. Alguns podem dizer que isso implicará em testes incompreensíveis. Isso é necessariamente verdade para os sistemas orientados a eventos, os quais efetivamente implicam em alguma GUI. Mas a facilidade com que você pode agora escrever testes de aceitação executável está em praticar possibilidades de conduzir testes mais completos, o que é, normalmente, o caso na maioria do desenvolvimento de sistemas comerciais.

O manual de treinamento também precisaria de outras coisas, incluindo uma introdução conceitual da aplicação e uma explicação dos vários objetos de negócio com seus métodos. Além disso, deve existir alguma explicação genérica do ambiente do usuário, equivalente às instruções genéricas para alguma aplicação baseada em Windows ou baseada num browser web. (Nós forneceremos uma versão atualizada dessa introdução genérica em nosso website).

Além de economizar trabalho, autogerar a documentação de treinamento do usuário a partir dos testes de aceitação executável garante que ele esteja consistente com a operação do sistema. É como se eles estivessem escrevendo a página do manual de treinamento quando estiverem praticando uma particular estória, ao mesmo tempo em que nós usamos uma versão executável dessa página como nosso teste de aceitação.

Estudo de caso: Atrasos e Cobranças

Este estudo de caso está baseado num breve projeto exploratório realizado por um grande banco inglês. Como em qualquer grande banco, 'Atrasos e Cobranças' é uma atividade principal de negócio. Em algum momento ela terá dezenas de milhares, talvez centenas de milhões, de produtos em atrasos, desde uma conta corrente que tenha excedido seu limite de retirada acordado, a um empréstimo onde o pagamento do total esteja em atraso. Muitas situações como essas não são corretas: elas podem ter surgido de um simples esquecimento ou enganos da parte do cliente, ou possivelmente um erro administrativo cometido pelo próprio banco, e que serão retificados em alguns dias. Em outro extremo do espectro, o departamento de atrasos e cobranças lida com delitos graves padrão: falências e fraudes.

Background de negócio

O volume do desvio das transgressões secundárias regulamenta a prioridade no sistema automatizado. Esse banco tem um sistema sofisticado para gerar uma série de cartas automatizadas, de acordo com a severidade. A redação dessas cartas não é trivial. O sistema constantemente persegue a efetividade de cada modelo de carta e testa novos 'candidatos' contra a carta 'campeã' para um dado estado de atraso.

Se a carta falhar em corrigir a situação, então em um certo ponto o caso irá chamar a atenção de um funcionário de atrasos e cobranças. O limite para essa transferência talvez se baseie em critérios fixos tais como a quantidade e duração do atraso, ou num perfil de avaliação de risco do cliente. O funcionário irá decidir quais novos cursos de ação iniciar. Isso pode envolver a busca por negociar um acordo para pagamento gradual dos atrasos, reestruturar o débito, re-processar os bens (para um empréstimo com garantia), ou persistir na dívida através de da corte judicial. Isso pode até envolver a descobertas de que o devedor faleceu e a iniciação de uma reivindicação contra o estado.

Enquanto a geração automática de cartas é um modelo de sofisticação técnica, o suporte de TI para as intervenções de um funcionário é mínimo – pouco mais que uma planilha eletrônica e um processador de textos. Idealmente, um sistema de atrasos e cobranças deveria de forma transparente integrar a automatização nos primeiros estágios de um caso com estágios posteriores intrinsecamente manuais. Tal integração deve facilitar o encaminhamento de um simples caso de ponta a ponta e monitorar o relacionamento entre táticas iniciais e aquelas posteriores. Ela deve também permitir que funcionários intervenham no início de alguns casos e coloque outros de volta em piloto automático por um período.

Abordagem

Nós trabalhamos com uma equipe pequena de gerentes de negócio e TI para explorar um possível projeto para tal sistema usando Naked Objects. A equipe não teve dificuldade de identificar certos objetos de negócio principais que caracterizam o sistema, incluindo Clientes, Funcionários, Conta, Transação e Caso.

Em algum dado momento, cada caso de atraso estará sujeito a uma particular 'estratégia', embora essa estratégia mude de acordo com o como o caso progride. No início do exercício, os gerentes de negócio referenciaram 'milhares' de estratégias. No entanto, no período de duas semanas a equipe abstraiu para apenas 13 estratégias principais, cada uma das quais tinham ramos e variações:

- Identificar a razão e tentar identificar situações de não-atraso (por exemplo, erros técnicos).
- Estender instrumentos (por exemplo, estender débito, oferecer novo produto de empréstimo).
- Gerar automaticamente cartas de acordo com a severidade.
- Aceitar a promessa verbal do cliente de solucionar numa data específica.
- Fazer contato direto com o cliente e estabelecer circunstâncias.
- Criar e monitorar um novo arranjo de pagamento (após o período).
- Achar um cliente desaparecido.
- Restabelecer a confiança.
- Tomar ação legal.
- Cancelar dívida.
- Terceirizar a dívida.
- Reivindicar os bens (falência / morte).
- Monitorar com atenção a conta após a reabilitação.

Um dos principais desafios que nós enfrentamos foi que tanto os gerentes de TI quanto de negócios envolvidos tinham uma visão do mundo muito fortemente orientada a processos. Eles assumiram que essas estratégias seriam implementadas usando uma máquina workflow ou alguma outra forma de mecanismo de script de alto nível, e que deveriam descrever dados e talvez até algumas formas limitadas do comportamento, a partir dos principais objetos de negócio. No entanto, desde que parte do propósito desse exercício era avaliar o potencial da abordagem Naked Objects; a equipe concordou, com certa relutância, em pensar nessas estratégias como verdadeiros objetos. (De fato, as 13 estratégias deveriam ser subclasses de uma classe genérica Estratégia). Diferentemente, digamos, do objeto Cliente, esses objetos de Estratégia têm um sentido de direção: eles devem ser projetados para seguir uma seqüência de estados conhecidos, mesmo que essa seqüência possa algumas vezes voltar atrás. Isso pode ser útil na modelagem de objetos, pois faz distinção simples entre objetos 'com direção' e objetos 'sem direção': o primeiro pode ser pensado como um processo rudimentar, mas é importante entender que eles não estão no 'topo' dos outros objetos da maneira como os processos são usualmente concebidos.

De fato, os objetos ‘com direção’ são tão válidos quanto imaginá-los como os objetos ‘sem direção’ ‘situados internamente’.

A prototipação do sistema usando o framework Naked Objects não ajuda apenas a equipe a vislumbrar esse conceito, mas também a enxergar os benefícios dessa forma de pensar. Agora, ao invés de pensar nas estratégias como verbos sobre um menu de nível superior, eles são vistos como substantivos – representados como ícones. Mudar a estratégia a ser aplicada a um particular caso de atraso pode ser visualizado, tanto metaforicamente quanto literalmente, como soltar um novo objeto Estratégia dentro do objeto Caso. Mais ainda, quando um funcionário analisa um caso, ele pode ver imediatamente a história do caso resumido como uma lista de ícones representando as Estratégias tentadas anteriormente. Um duplo-clique sobre algum desses ícones deverá permitir a inspeção de como a Estratégia realmente progrediu e foi finalizada. É muito difícil obter esse tipo de visão a partir de um sistema workflow. Cada objeto Estratégia é capaz de executar uma seqüência de ações e responder a eventos tais como nova Transação sobre uma Conta que está em atraso. Adicionalmente, cada Estratégia terá sua própria condição de saída, que pode simplesmente incluir a passagem suficiente de tempo. Na saída, uma dada Estratégia pode substituir ele próprio com um outro diferente ou pode levar o Caso para a apreciação de um Funcionário para que tome uma decisão.

De fato, programadores com experiência em orientação a objetos irão reconhecer que isso é uma programação normal de padrão de projeto, coincidentemente chamado ‘Strategy’ (Estratégia) [Gamma1995]. No entanto, parece que tais padrões raramente são conhecidos pelos analistas de negócio. Isso levanta um outro ponto sobre o Naked Objects: eles facilitam a aplicação da riqueza e do poder expressivo de uma linguagem como Java ou Smalltalk para resolver problemas de negócio. Muitos métodos insistem que a análise seja independente de qualquer linguagem de programação. Isso seria um objetivo louvável exceto que o efeito final é que projetos resultantes falham ao tomar vantagens desses poderosos padrões de objetos.

Avaliação

Duas coisas surpreenderam a equipe. A primeira foi de que foi possível projetar um sistema sofisticado puramente em termos de objetos comportamentalmente ricos, sem mesmo ter que desenhar um conjunto de diagramas de processos. A segunda foi que a aceitação das restrições da abordagem Naked Objects rapidamente rendeu algumas percepções da natureza e estrutura das atividades de atrasos e cobranças que ninguém tinha visto anteriormente (especialmente, as abstrações que existiam dentro da noção de estratégias). Dada a quantidade muito limitada de tempo que os representantes de negócio podiam dedicar ao exercício, eles ficaram muito surpresos com a quantidade de trabalho útil que de fato conseguiram fazer. Um comentário foi que: “O tempo em que nós normalmente gastaríamos, apenas para discutir como iniciar a obtenção de

requisitos, nós construímos um modelo funcional simples do domínio de negócio que rendeu algumas novas percepções sobre como nosso negócio realmente funciona".

Segue-se uma breve descrição das responsabilidades das quatro classes naked objects.

Classe: Conta

Subclassificada em tipos diferentes de conta: empréstimo, cartão de crédito, empréstimo sem garantias, conta bancária, etc. O objeto atua como uma capa sobre o sistema de gerenciamento de contas, o qual irá realizar as responsabilidades transacionais e relatórios gerenciais. As responsabilidades específicas incluem:

- Identificar se ele está em atraso e se está apto a propor a quantidade e período de atraso.
- Antes que a Conta fique em atraso, ou cria-se um novo Caso, ou se um Caso já existir para esse Cliente, então se adiciona essa Conta naquele Caso.
- Exibir um histórico visual ampliável das contas de um período específico, apresentando o saldo planejado e o atual.
- Calcular o impacto de uma ou mais transações hipotéticas sobre a conta (principalmente usado quando se faz um acordo – isso idealmente deveria ser feito como uma planilha).
- Alertar o Caso ou quando houver uma mudança significativa na situação do atraso (em quantidade ou período) ou quando ocorrer alguma transação.
- Relatar atrasos na agência de créditos se apropriado.
- Congelar ou tornar as restrições temporárias.

Classe: Propriedade


As Propriedades normalmente irão existir porque elas são a garantia de um empréstimo (por exemplo, uma propriedade hipotecada) ou porque elas são partes de um contrato de empréstimo (por exemplo, um carro). As Propriedades também podem ser usadas para modelar qualquer propriedade física conhecida em casos de recuperação. As propriedades mais comuns serão modeladas como subclasses especializadas de Propriedade, tal que elas possam registrar seus próprios detalhes. As responsabilidades específicas incluem:

- Iniciar a recuperação física.
- Transformar em valor (iniciar a venda de uma propriedade).

- Avaliar a diferença entre o valor do patrimônio e o valor da dívida (por exemplo, aplicando a depreciação).

Classe: CasoDeAtraso

Responsável em gerenciar a carga de trabalho do processo de atrasos e cobranças e por produzir informações gerenciais. A maioria dos casos de atrasos, no entanto, não irão envolver a intervenção de um funcionário. As responsabilidades específicas incluem:

- Dividir, juntar ou incluir outros Casos.
- Trazer de volta a atenção de um funcionário.
- Enviar para um outro funcionário.
- Escolher a Estratégia inicial sobre a criação.
- Estratégia de alerta para eventos de mudança.
- Fornecer relatórios gerenciais sobre o estado atual.

Classe: Estratégia

Algumas Estratégias irão operar automaticamente – outros irão envolver um Funcionário. Cada estratégia tem um resultado desejado, o qual pode ser fixado ou pode ser determinado pelo funcionário. A Estratégia é um objeto ativo – não meramente um atributo. A Estratégia é um meio de alcançar um resultado, não uma seqüência de ações. As responsabilidades específicas incluem:

- Selecionar a versão a ser usada (a maioria das Estratégias possui versões ‘campeãs’ e ‘desafiantes’). Isso é feito pelo sistema, não pelo usuário.
- Tomar a próxima ação – quando e o que.
- Gerar a propriedade de Comunicação padrão para essa Estratégia.
- Instruir a Conta (por exemplo, bloquear saques).
- Responder a eventos tais como transação, comunicação do cliente, elevação do débito ou passagem do tempo.
- Terminar sozinho (ou manualmente por um Funcionário) e escolher a próxima estratégia (incluindo a recomendação para um funcionário decidir manualmente).
- Auto-avaliação – se funcionou ou não, inclusive por versão.

Estendendo o Naked Objects

O Naked Objects ainda está na infância. Planejamos estender o framework de várias maneiras. Devido ao framework ser de código aberto, esperamos que ele seja estendido por outros desenvolvedores também, de forma que nós não planejamos ou previmos. Nesta seção nós veremos algumas dessas possibilidades.

O Naked Objects é um trabalho em progresso. Nós incluímos neste livro apenas aqueles aspectos do framework que acreditamos estar razoavelmente estável. Futuras atualizações do framework podem conter incorretamente alguns detalhes exibidos nesta edição, mas esperamos que os princípios básicos sejam mantidos. Se você achar alguma coisa que parece não estar funcionando direito, então por gentileza, verifique em nosso website, onde você encontrará a versão mais atualizada de vários documentos que fazem parte deste livro.

Documentação adicional disponível em nosso website

O website é o principal repositório de informações sobre o framework Naked Objects e técnicas de como usá-lo. No website você já pode encontrar documentações adicionais sobre:

- A API (Application Programming Interface) completa para o framework. (Ela é incluída automaticamente na distribuição do framework).
- Códigos exemplo sobre vários aspectos de uso do framework.
- Tutoriais sobre usar o Naked Objects.
- Como personalizar o mecanismo de visualização existente: alterando a aparência (fontes, cores da janela, etc.); alterando os ícones padrões (usado quando nenhum ícone é especificado para uma classe naked particular); e usando o modo de depuração, o qual mostra cada número ID único do objeto em cinza próximo ao título do objeto.
- Como personalizar o servidor e clientes, incluindo: a especificação de protocolos alternativos pela qual a aplicação cliente faz solicitações ao servidor, ou alterações nos objetos que são atualizados pelo mecanismo de visualização; a especificação de um console de operador alternativo para o servidor; a especificação do nível de logging (via o framework Log4J) para ser usado na depuração.
- Como estender o framework, incluindo: a criação de novos objetos de valor; construindo novos visualizadores de objetos; construindo outros mecanismos de visualização; desenvolvendo e personalizando repositórios de objetos.

Algumas formas possíveis de estender o Naked Objects

O website é também o fórum através do qual iremos gerenciar o progresso das extensões do framework. O Naked Objects é totalmente aberto. Isso significa que você pode copiá-lo e usá-lo livremente e ter acesso ao código fonte se desejar estendê-lo. O núcleo do framework foi totalmente escrito por Robert Matthews, embora com contribuições e sugestões de vários desenvolvedores interessados. Alguns deles já começaram a escrever suas próprias extensões, tal como o Repositório de Objetos EJB de Dave Slaughter. Nós esperamos que com o lançamento deste livro, uma comunidade substancial de desenvolvedores cresça em torno do framework. Nós precisaremos de sua ajuda para que o Naked Objects atinja todo o seu potencial. No website você estará apto a encontrar mais detalhes das extensões que estão sendo desenvolvidos, planejados ou apenas considerados. Em breve, existirá uma lista de algumas principais direções em que achamos que as extensões irão tomar.

Novos objetos de valor e extensões

A maioria dos tipos de `NakedValue` pode receber comportamentos adicionais tanto para uso em funcionalidades de adicionais de negócio quanto para disponibilizá-los ao usuário via um mecanismo de visualização apropriado. Por exemplo, um objeto de valor `Date` deveria permitir a aritmética de data, e um objeto de valor `Money` deveria suportar a conversão monetária.

Existe também a necessidade de vários tipos novos de `NakedValue` incluindo uma forma de `TextString` que pode limitar o número de caracteres (para combinar com o esquema de banco de dados existente), e um outro tipo que pode suportar uma rica formatação de texto usando XML.

Visões adicionais

O mecanismo de visualização inicial é extensível e permite que novas visões sejam adicionadas. Novas visões podem ser escritas para os vários objetos de valor existentes (bem como para os novos), para exibi-los melhor e ajudar na entrada de dados. Um exemplo é um simples calendário para valores de data o qual permite que um usuário selecione uma data como um simples click.

Os naked objects podem ser exibidos de várias maneiras, por exemplo, todos os objetos de negócio podem ser visualizados como um ícone ou como um formulário. Outras visões também podem ser facilmente adicionadas, por exemplo, podemos fornecer algum tipo de visão sumário que ocupe menos espaço na tela. Além disso, a visão de tabela, como uma alternativa para a visão lista, precisa ser melhorada. Visões específicas dos supertipos genéricos e interfaces são também possíveis.

Para data, nós utilizamos muito pouco gráficos. Aqui, escopo é imenso. Objetos de valor do tipo numérico podem ser visualizados na forma de gráfico, tanto para entrada quanto para saída (muito parecido com as modernas planilhas eletrônicas que permitem que você desenhe gráficos e então calcule os números para ele – uma característica que mostra excelente perspectiva de como as projeções de venda são realizadas!). Mapas, esquemas, redes e outros layouts espaciais podem ser possíveis. O estudo de caso sobre Norsk Hydro mostra um exemplo de como tais capacidades podem ser adicionadas, sem violar a filosofia básica do Naked Objects (que a interface do usuário orientada a objetos deve ser automaticamente derivada 100% a partir das definições dos objetos de negócio).

Com relação à visualização gráfica, é preciso um visualizador de imagens junto com um tipo de objeto imagem.

Mecanismos de visualização adicionais

A interface de usuário gráfica de arrastar e soltar é apenas uma das várias formas possíveis dos mecanismos de visualizações. Outros mecanismos podem ser escritos trocando esta interface em particular. O benefício que o framework fornece é que um mecanismo de visualização específico pode ser usado para satisfazer um requisito específico do usuário, por exemplo, se o sistema precisar ser acessado por meio de um browser, sobre um modem lento, então um mecanismo de visualização HTML pode ser empregado ao invés da interface gráfica atual. Diferentes visualizadores podem ser usados ao mesmo tempo para exibir os mesmos objetos, permitindo que eles sejam executados em diferentes clientes, por exemplo, dentro de diferentes JVMs.

Novos repositórios de objetos

O Naked Objects é um sistema orientado a objetos e é menos complicado quando usado com algumas formas de mecanismo de persistência baseada em objetos. Nós gostaríamos de ver, para pequenos e médios projetos, mecanismos simples de persistência de objetos sendo usados para armazenar objetos, assim como interfaces para os bancos de dados orientados a objetos existentes atualmente.

No entanto, uma boa proporção de aplicações Naked Objects irão se basear nos bancos de dados existentes. Junto com os repositórios de objetos baseados em SQL e EJB, disponíveis no momento, outros repositórios de objetos são necessários a fim de atender demandas de aplicações específicas e manipular sistemas proprietários.

Serviços de infra-estrutura

Como mencionamos, nós não implementamos quaisquer mecanismos de segurança a não ser os objetos `About`. Isso precisa ser explorado e uma interface de autorização e autenticação precisa ser construído dentro do framework.

Padrões reutilizáveis para objetos de negócio

Você irá ver que existem muitos padrões comuns, até mesmo classes de objeto de negócio comuns, somente com a leitura dos cinco estudos de caso deste livro: Cliente, Caso, Comunicação, entre outras. A idéia de uma biblioteca de classes de objetos de negócio não é nova – existem várias publicações, bem como bibliotecas proprietárias disponíveis. Nós ainda não avaliamos nenhuma dessas bibliotecas para ver se elas podem ser usadas, ou adaptadas, para formar um conjunto aceitável de padrões naked objects de negócio, mas isso deve ser um exercício útil. A nossa principal preocupação é a extensão na qual os modeladores terão perseguido a idéia de completeza comportamental. Nós podemos, em paralelo, começar a fazer nossa própria coleção de padrões naked objects reutilizáveis.

Os sistemas naked objects são escaláveis?

Uma das questões que nós respondemos com freqüência é: “Os sistemas naked objects são escaláveis com relação ao número de usuários?”. Neste momento, a única resposta honesta é: “Nós não estamos certos”. Nós ainda não vimos quaisquer sistemas implementados usando Naked Objects que servisse a um grande número de usuários simultaneamente, embora nós tenhamos visto aplicações que envolvem grande quantidade de entrada de dados e deviam apresentar desempenho equivalente aos sistemas de transação de mainframe (por exemplo, no Safeway).

Como nós deixamos claro nesta seção, nós projetamos a arquitetura básica do naked objects com a intenção que ele escale para aplicações muito maiores. Mas nós não desenvolvemos ainda, todas as peças. Nós estamos seguros que quando grandes sistemas forem construídos, novos problemas irão emergir, os quais nós ainda não previmos e que nós precisaremos atacar com novas extensões ou modificações.

O fato de nós ainda não termos visto sistemas muito grandes desenvolvidos em Naked Objects não é tão surpreendente. Qualquer tecnologia significativamente nova envolve uma curva de aprendizagem: é correto e apropriado que organizações queiram iniciar aplicando-o para solucionar pequenos problemas, onde existam poucas variáveis. (De fato, muitas aplicações de Naked Objects

estão em áreas onde a abordagem tradicional de sistemas falhou gravemente e existe o argumento de não se ter nada a perder tentando uma abordagem radical).

Assim, nós recebemos a questão com prazer, uma vez que ela representa um desejo de entender se a arquitetura tem a capacidade de escalar com o aumento da demanda – abastecido pela elevação da experiência. Nós não temos nenhum interesse em responder a aqueles que usam tais questões meramente como uma desculpa para manter o *status quo*. Nós lembramos que os pioneiros do banco de dados relacional, arquitetura cliente-servidor, redes ponto-aponto, mensagens assíncronas e, naturalmente técnicas orientadas a objetos em geral tem enfrentado ceticismo similar.

Grande parte de nosso segredo repousa no fato de que Naked Objects é open-source. Enquanto a comunidade de desenvolvimento crescer, obteremos ajuda de pessoas que tem substancial especialidade nesta área. De fato isso já está acontecendo. Muitas pessoas nos dizem que eles consideram Naked Objects como fundamentalmente 'a maneira certa de projetar software' e tem oferecido seu apoio. Nós achamos isso muito motivador. Uma coisa que é verdade no desenvolvimento de software em geral e no desenvolvimento open-source baseado na comunidade em particular: onde existir uma vontade, existirá sempre um caminho.

A coisa mais animadora é que pessoas que possuem muita experiência em escalabilidade em objetos distribuídos têm nos apoiado. Nós estamos contentes em dar a última palavra neste livro para Oliver Sims, autor de 'Business Objects' [Sims1994], co-autor de 'Building Business Objects' [Eeles1998] e 'Business Component Factory' [Herzum2000], e uma autoridade altamente respeitada no campo de objetos distribuídos de larga escala e sistemas de negócio componentizados:

"Eu acho que Naked Objects é tão excitante quanto importante. Embora existam alguns conceitos técnicos sobre escalabilidade que precisam ser cobertos, eu não vejo nada nos conceitos fundamentais por trás do Naked Objects que impeça que ele seja escalável para um grande número de usuários. E, o mais importante, eu acredito que será possível cobrir esses assuntos de forma a não comprometer a fundamental filosofia do Naked Objects".

Estudo de caso: Mercado de Energia

A Norsk Hydro é um grande conglomerado multinacional, com sede na Noruega. As origens da companhia repousam na geração de energia hidroelétrica, mas ela é agora também a maior produtora de óleo e gás, e tem outros núcleos de negócio na produção de alumínio e fertilizantes – todos são negócios de energia intensiva. Por vários anos a Norsk Hydro teve que construir especialidades em comprar e vender energia dentro de muitos mercados nacionais onde eles estão estabelecidos.

A base do negócio

Em 2000 o mercado de eletricidade da Europa foi liberado para o mercado, permitindo que grandes produtores ou consumidores de eletricidade possam suprir ou fornecer além das fronteiras nacionais – expandindo a tendência que tinham. (Uma tendência similar ocorreu na indústria de gás natural na Europa).

A combinação de especialidades, escala e cobertura geográfica tanto no fornecimento quanto no consumo permitiu que a Norsk Hydro ganhasse a posição líder no mercado de energia na Europa. (Nota: Embora o colapso da Enron no final de 2001 ter gerado algum cinismo público sobre o mercado de energia, hoje é aceito que o desastre foi causado pelas irregularidades financeiras ao invés da natureza do negócio em si. O negócio de energia permanece forte e de fato, é um negócio muito importante).

Oportunidade

Dada a renovação da oportunidade, não havia nenhuma solução de TI empacotada para apoiar esta atividade. O mercado é normalmente conduzido por telefone, apoiado com fax e com uma grande planilha para análise. Isso funcionava para a fase inicial, mas claramente não estava a altura com a previsão de crescimento tanto em volume quanto em complexidade do negócio. Em algum ponto, um sistema sob medida para esse propósito seria necessário. O sistema precisaria ser muito ágil: capacidade de extensão em novas áreas geográficas e acomodação em territórios de granularidade fina dentro de países individuais. Ele deveria idealmente ser capaz de ser estendido dentro do negócio de outras formas de energia, especialmente gás. Isso sugeriu uma solução orientada a objetos. (A orientação a objetos foi inventada na Noruega em meados de 1960 e existe ainda uma grande consciência do valor de objetos naquele país do que em qualquer outro).

Um pequeno grupo do departamento de TI Estratégico ficou sabendo dos conceitos do Naked Objects. Aparte do compromisso óbvio com a orientação a objetos, eles viram duas outras vantagens. Uma foi sobre a habilidade de usar prototipação rápida para explorar os requisitos de um novíssimo domínio de

negócio e de rápida mudança. A outra vantagem foi sobre o potencial de construir um sistema que é muito 'expressivo' – capaz de sustentar o negócio em suas tarefas diárias de balancear suas 'posições' de rede. O negócio de energia gerencia um grande número de contratos de longa duração para compra de eletricidade dos produtores, venda para consumidores industriais e transporte via linhas de potência a um para outro. As partes do negócio incluem as subsidiárias da Norsk Hydro e companhias de energia terceirizadas. Esses contratos de longa duração pretendem balancear o fornecimento e demanda. Porém, muitos contratos permitem que consumidores especifiquem seus requisitos atuais, dentro de limites especificados, em dias antes do requerido. Pelo número de dias, o negócio tem que balancear essas variações com contratos adicionais de curta duração para compra, venda e transporte, incluindo o negócio sobre 'venda à vista' em vários centros principais tais como Amsterdam e Leipzig. Se existir uma diferença significativa entre o preço à vista em duas dessas moedas de troca e o negócio tem acesso à capacidade de transmissão excedente, então eles podem querer transferir tanto quanto eles puderam de uma localização a outra.

Assim, o negócio de energia é inerentemente uma atividade de solucionar problemas. Mesmo quando a rede é simples, envolvendo, digamos, apenas meia dúzia de nós representando os países do Norte da Europa, a posição do negócio atual é difícil entender sem algum tipo de representação visual. Um grupo de TI estratégico da Norsk Hydro percebeu que com o Naked Objects não só era possível produzir uma representação visual (mapa) da posição, mas que era possível executar todas as atividades de balanceamento através desse mapa.

Abordagem

Foram dadas apenas duas semanas para que uma equipe entendesse um projeto exploratório que tinha que incluir todas as seguintes atividades:

- Treinar alguns de seus desenvolvedores Java a usar o framework Naked Objects.
- Identificar os principais objetos de negócio que modelaria o negócio de mercado de energia.
- Construir uma simples prova de conceitos mostrando como um mercado deveria trabalhar com esses objetos, através de uma representação de mapa, para entender algumas atividades diárias típicas incluindo balancear uma posição de rede.

Além disso, foi necessário escrever uma extensão do framework Naked Objects que atendesse as necessidades da interface de mapa visual desejado. Deveria ser fácil de escrever uma interface de usuário personalizado que chamasse responsabilidades a partir dos objetos subjacentes. Mas nós queríamos encontrar uma outra abordagem, uma que mantivesse mais a filosofia do Naked Objects. No final da solução foi um tanto elegante. Nós definimos uma nova interface, `Spatial`. A fim de adequar essa interface, um objeto de negócio deveria ser capaz de

produzir um par de coordenadas representando a latitude e longitude. Então, nós estendemos o mecanismo de visualização tal que qualquer coleção de objetos pudesse ser visualizada pelo usuário na forma de um mapa, com os objetos individuais exibidos como ícones posicionados sobre o mapa. O fundo do mapa foi fornecido por um arquivo de imagem e especificado através do arquivo de configuração. O resultado foi que a aplicação como um todo pode ainda ser escrita sem os objetos de negócio conhecendo qualquer coisa sobre a interface do usuário, e sem ter que escrever qualquer código específico interface de usuário para a aplicação. Nós teremos algum trabalho para fazer o refinamento deste conceito, mas tal capacidade será liberada numa futura versão do framework.

Avaliação

O exercício foi considerado um grande sucesso. Além dos desenvolvedores Java, a equipe incluiu estrategistas de TI representando tanto o mercado de eletricidade quanto o negócio de gás. Os estrategistas de negócio de gás disseram que acharam a experiência de participar do processo de design divertido e recompensador. Pelo bem da velocidade, muito do desenvolvimento e depuração foi realizado ao vivo em frente desses representantes de usuário, embora eles não fossem programadores.

A prova de conceito foi levada para avaliação do negócio de energia. Se bem sucedido, espera-se que ela seja implementada em toda a escala.

Aqui está uma breve explicação de cada uma das classes de negócio junto com uma ilustração de responsabilidade. (Cada objeto tem muito mais responsabilidades).

Parte do Mercado

Pode ser uma subsidiária do Norsk Hydro ou um terceirizado. Sabe como criar um novo Contrato usando valores default dessa parte.

Cambio

(Subclasse de Parte do Mercado). Sabe como determinar o preço para um dado período.

Contrato

Conhece o a Parte do Mercado, Período e tipo do contrato (compra, venda, transmissão, curta duração, longa duração, etc.). Conhece o Requisito de Potência.

Requisito de Potência (volume)

Na sua forma mais simples ele é apenas um objeto de valor padrão – um número e uma unidade de medida (Megawatts), com a habilidade de realizar operações aritméticas normais. Modelado como um objeto de negócio, ele permite requisitos de potência mais sofisticados, por exemplo, limites de variação, ou perfil de variação diário.

Rede

Conhece as Localizações e Links que formam a Rede e como gerar uma Posição de Rede para um dado período (normalmente um dia). Trabalha a rede como um objeto de negócio permitindo que o negociante manipule a rede, as quais são ou geograficamente distinta, ou manipulam formas diferentes de energia (por exemplo, gás). Sabe como adicionar novas Localizações e Links.

Localização

Modela um nó da Rede. Conhece as Partes do Mercado que operam nessa Localização e como comprar ou vender no Cambio local se existir um. Sabe como criar uma Posição de Localização para um dado período.

Link

Modela uma linha de transmissão entre duas Localizações. Sabe como comprar ou vender a capacidade a partir das Partes do Mercado apropriadas.

Posição de Rede

Esta é a posição no sentido de mercado (por exemplo, longa, curta, balanceado) não geográfico. A Posição da Rede é realizado a partir das Posições de Localização individuais e Posições de Link para o mesmo período. Sabe como calcular o lucro da Posição.

Posição de Localização

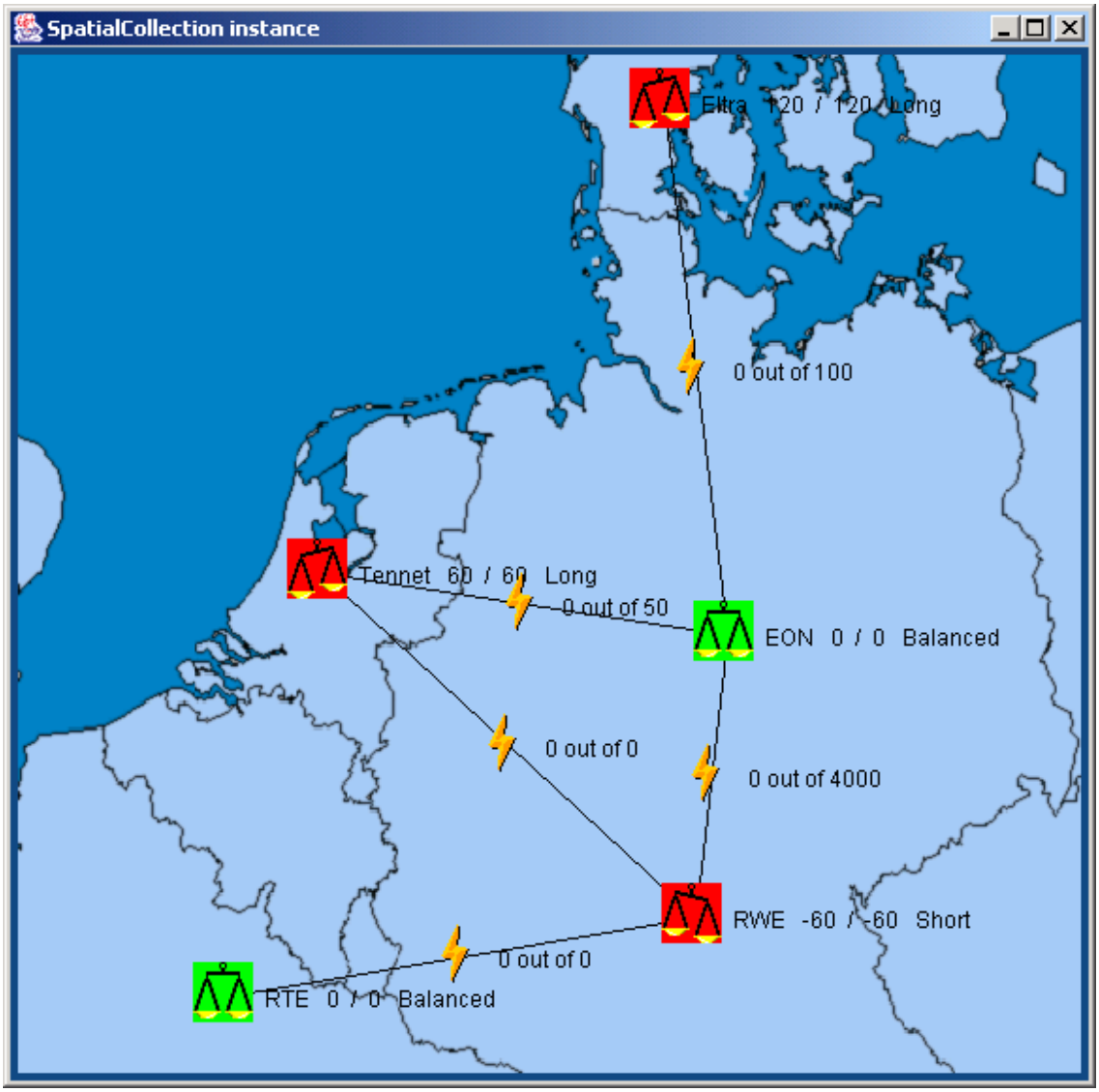
Sabe como encontrar e sumarizar Contratos ativos em sua Localização no período especificado.

Posição do Link

Sabe como calcular a quantidade de capacidade disponível a ser usada. Sabe como criar uma 'transferência' (uma instrução operacional para transmitir uma quantidade especificada de potência).

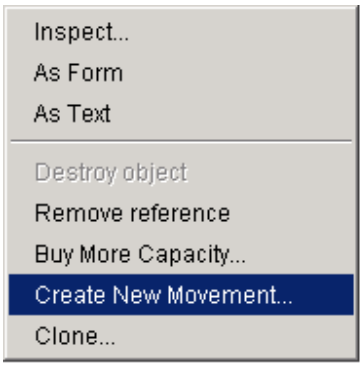
Um cenário típico de mercado de energia

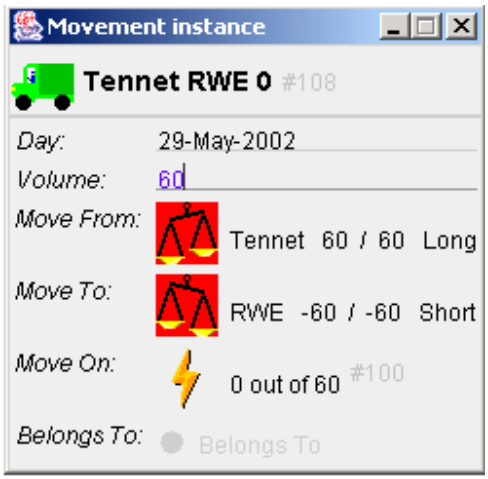
A Posição da Rede do dia exibe que nós temos 60 Megawatts (MW) 'longa' na localização Tennet e 60 MW curta na localização RWE:



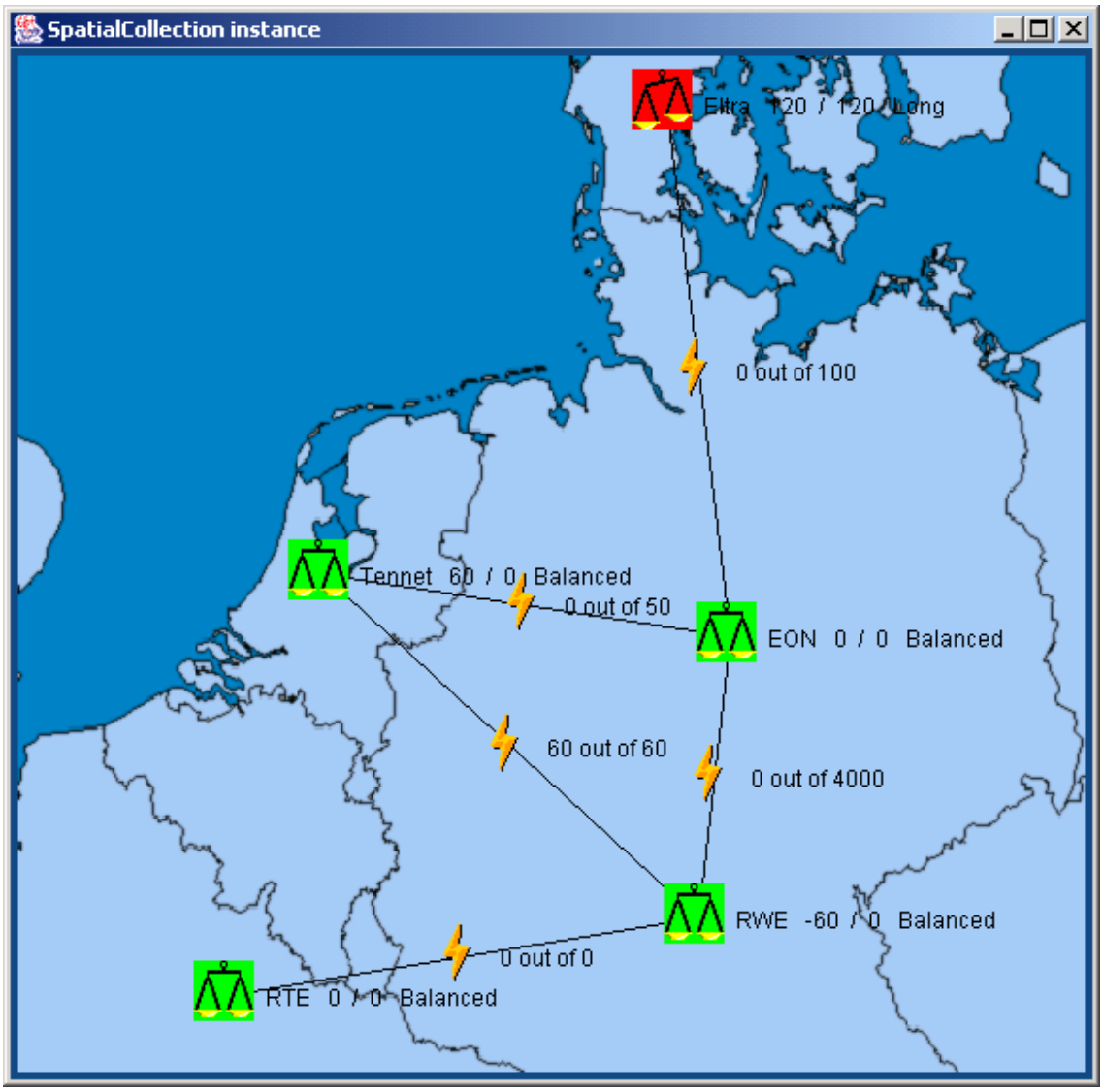
Clicando com o botão direito do mouse sobre a Posição de Link para o Link entre os dois, nós selecionamos ‘Comprar Mais Capacidade’, que cria um novo Contrato de ‘curta duração’ com o proprietário do Link, para o qual especificamos um Perfil de Potência de 60MW:

Nós agora temos a capacidade de transmissão para transmitir potência de onde temos longa para onde temos curta (isso pode envolver muitas compras e vendas com preços diferentes). Clicando, novamente, com o botão direito do mouse sobre esta posição de link nós podemos agora criar uma Transmissão instruindo o operador de linha a 'transportar' 60 MW:

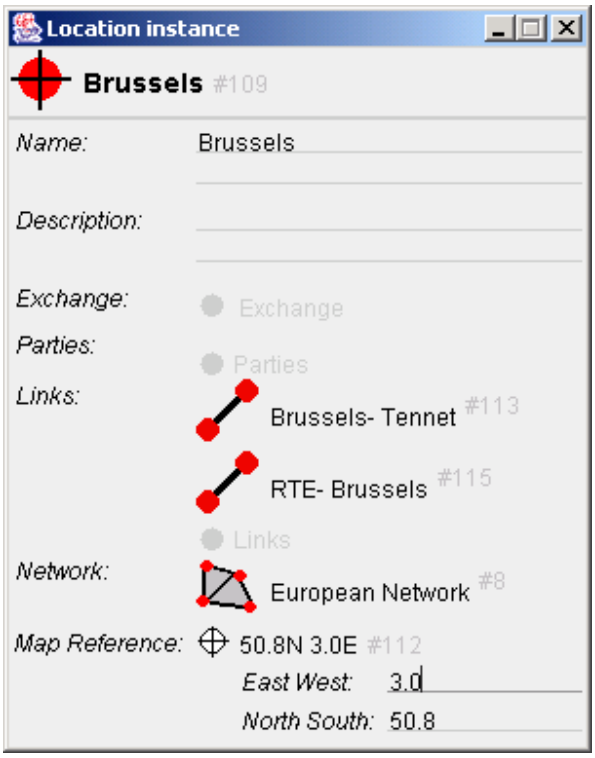




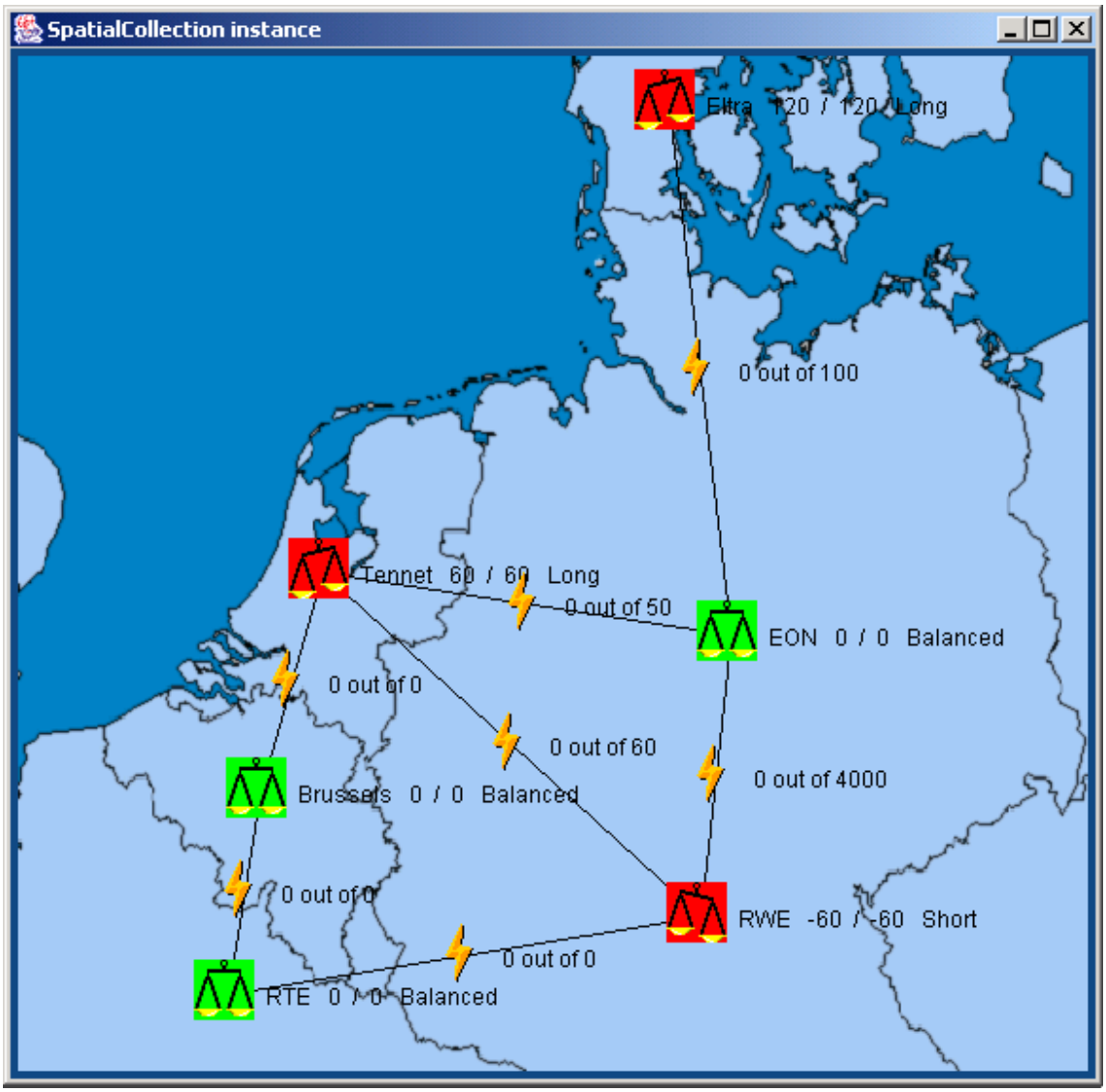
Os ícones verdes Posição de Localização (escalas) exibem que nós estamos agora balanceados na Tennet e RWE, mas ainda tem a Localização Eltra a balancear:



De acordo com o crescimento do negócio, novas Localizações, Links, e toda a Rede pode ser adicionada, apenas criando novas instâncias de objetos:



A localização recentemente adicionada e os links são imediatamente disponibilizados ao mercado:



Apêndice A: Os primeiros passos

Esta seção tem o objetivo de introduzir os aspectos básicos de implementação do Naked Objects, usando uma aplicação muito simples como um exemplo. Embora esta seção seja de particular interesse dos programadores Java, ela foi escrita com detalhes suficientes para que desenvolvedores proficientes em outras linguagens e outros profissionais de software possam entendê-la. Nós discutiremos os requisitos do software, onde obtê-lo e então veremos como codificar uma simples aplicação.

Para o exemplo, nós iremos trabalhar com ferramentas de comando de linha básicos ao invés de utilizar uma IDE (Integrated Development Environment). Isso nos permite ver os passos exatos necessários para escrever, compilar e executar a aplicação, as quais você pode posteriormente aplicar para o seu IDE específico.

Requisitos

Para executar o Naked Objects você deve estar apto a compilar e executar programas Java; o framework funciona com a versão Java 1.1 ou maior, mas para este exemplo nós iremos usar a versão 1.2. Além disso, o framework usa o Log4J do Apache, como parte do projeto framework de logging do Jakarta para log de atividades. Para desenvolver aplicações, alguma forma de ambiente de desenvolvimento é necessária; ou um IDE ou um JDK disponível da Sun e um editor de texto. Um editor de ícone também provou ser útil, embora nós o utilizemos durante este tutorial, uma vez que todos os ícones que iremos usar estão incluídos na distribuição do framework.

Obtendo o framework

O framework pode ser obtido como um arquivo zip da página 'downloads' do web site Naked Objects.

Uma vez que o arquivo zip tenha sido descarregado, seu conteúdo deve ser extraído numa localização conveniente a fim de facilitar o acesso a vários arquivos. Faça uma nota do nome do diretório extraído uma vez que ele inclui um número de versão e, assim, será diferente do diretório que nós estamos usando. Por exemplo, nós iremos usar a versão 0.8.0 e extrair os arquivos em nosso diretório raiz. Isso nos dá o seguinte caminho: c:\nakedobjects0.8.0 no Windows e /nakedobjects0.8.0 no Unix.

Preparação do projeto

Antes de iniciarmos, devemos criar um diretório de trabalho onde você poderá colocar seus arquivos. Neste exemplo, nós usaremos um diretório chamado

projeto, criado na raiz do nosso sistema de arquivos: C:\Project no Windows e /project no Unix.

Escrevendo uma aplicação

Para entender os princípios de codificar aplicações Naked Objects, iremos agora ver uma aplicação muito simples, porém completa. O requisito é para um sistema que nos permite rapidamente organizar os empregados em equipes, tal que eles possam trabalhar em projetos específicos. Quando um projeto é configurado, habilidades distintas são identificadas e indivíduos são associados ao projeto com base nesses requisitos.

Ele irá precisar de três classes cujos objetos representarão os projetos, os empregados e papéis (um papel como um conjunto combinado de habilidades). Essas classes precisam se relacionar uma com as outras: um projeto identificando os papéis requeridos, e cada papel identificando os indivíduos que estão reunidos em equipe.

Dessa forma, as classes que iremos definir nessa aplicação são: `Employee`, `Role` e `Project`. Cada objeto `Project` envolve uma coleção de objetos `Role` e cada `Role` envolve uma referência para um objeto `Employee`.

Nós começamos definindo as classes para os três tipos de objetos. Para cada classe, nós precisamos criar um arquivo .java e declarar dentro dele a classe com extensão de `AbstractNakedObject`.

Comece a descrição das três classes como a seguir:

Project.java

```
import org.nakedobjects.object.AbstractNakedObject;

public class Project extends AbstractNakedObject {
}
```

Role.java

```
import org.nakedobjects.object.AbstractNakedObject;

public class Role extends AbstractNakedObject {
}
```

Employee.java

```
import org.nakedobjects.object.AbstractNakedObject;

public class Employee extends AbstractNakedObject {
}
```

Agora nós consideramos quais campos cada objeto deve ter. Primeiro adicionamos campos que permitem ao usuário identificar os diferentes objetos (por exemplo, para que possamos ver que um objeto representa Dave e o outro representa John). Depois adicionamos campos que descrevem algum relacionamento entre objetos (por exemplo, todo papel é realizado por um empregado tal que um objeto `Role` irá precisar de um campo para referenciar um objeto `Employee`).

Objeto Empregado

Tomando inicialmente o objeto mais simples, um `Employee` precisa ter apenas um nome na medida em que isso seja suficiente para distinguí-lo. Para armazenar esse nome, adicionamos uma variável privada final, do tipo `TextString`, chamada `name` para a classe `Employee`. Um objeto de valor `TextString` armazena informações textuais simples. A variável é marcada como final para indicar que ela é parte composta de `Employee` e nunca deverá ser trocada. Como ela é final, nós a inicializamos assim que ela é declarada (alternativamente, isso poderia ser realizado dentro do construtor). Para tornar esse campo disponível a outros objetos e para que o framework possa exibi-lo, nós adicionamos um método de acesso que irá retornar a referência ao objeto de valor.

Employee.java

```
import org.nakedobjects.object.AbstractNakedObject;
import org.nakedobjects.object.value.TextString;

public class Employee extends AbstractNakedObject {
  private final TextString name = new TextString();

  public TextString getName() {
    return name;
  }
}
```

Objeto Papel

Depois, nós faremos uma coisa similar com a classe `Role` pois também requer um nome. Além disso, para este dado básico também precisamos referenciar um objeto `Employee` que está interpretando esse papel. Para fazer isso, nós declaramos um campo privado do tipo que nós queremos para manter uma referência; neste exemplo ele precisa ser uma instância da classe `Employee`. Como objeto de valor, precisamos também disponibilizar esse campo, assim definimos um par de métodos de acesso padrão (`getEmployee` e `setEmployee`).

Dentro desses dois métodos você irá ver duas chamadas obrigatórias para manter o objeto atual no repositório de objetos (nosso link para um mecanismo de persistência) e para qualquer outra visão do mesmo objeto (tanto sobre clientes

locais quanto remotos). O método `objectChanged` deve ser chamado nos métodos `set...` para quaisquer objetos de referência (em oposição aos objetos de valor que nós vimos anteriormente). Isso notifica o repositório de objetos que esse objeto precisa ser armazenado em outro lugar e também notifica a quaisquer outras visões que exibem esse objeto para que eles se atualizem.

Dentro dos métodos `get...`, `resolve` devem ser chamado para assegurar que o objeto que está sendo requisitado foi completamente carregado do repositório de objetos. Isso é necessário quando cada objeto é armazenado independentemente usando referências soft ao invés de um grande grafo de objeto.

Role.java

```
import org.nakedobjects.object.AbstractNakedObject;
import org.nakedobjects.object.value.TextString;

public class Role extends AbstractNakedObject {
  private final TextString name = new TextString();
  private Employee employee;

  public TextString getName() {
    return name;
  }

  public Employee getEmployee() {
    resolve(employee);
    return employee;
  }

  public void setEmployee(Employee employee) {
    this.employee = employee;
    objectChanged();
  }
}
```

Objeto Projeto

O `Project` é complexo em seu conteúdo, mas é na realidade é mais simples de codificar. Como um projeto envolver muitos papeis o objeto `Project` precisará armazenar um número de referências no mesmo tipo de coleção. Para criar um campo de coleção que é uma parte composta do objeto, nós fazemos uso da classe `InternalCollection` fornecida pelo framework e o declaramos como privado e final. Para inicializar a coleção, nós criamos um novo objeto, especificando o tipo de objeto que é permitido conter (objetos `Role`) e uma referência ao objeto no qual ele pertence (o seu projeto). Como no `TextString` de composição, este campo somente precisa de um método `get` (de fato, como ele é marcado como final, um método set não irá compilar).

Como nas duas classes anteriores, o `Project` deve também ter um campo nome e isso é adicionado como anteriormente.

Project.java

```
import java.util.Enumeration;
import org.nakedobjects.object.AbstractNakedObject;
import org.nakedobjects.object.collection.InternalCollection;
import org.nakedobjects.object.control.About;
import org.nakedobjects.object.control.ActionAbout;
import org.nakedobjects.object.value.Case;
import org.nakedobjects.object.value.TextString;

public class Project extends AbstractNakedObject {
  private final TextString name = new TextString();
  private final InternalCollection roles = new InternalCollection(Role.class, this);

  public TextString getName() {
    return name;
  }

  public InternalCollection getRoles() {
    return roles;
  }
}
```

Além dos dois campos que este objeto oferece, nós iremos adicionar o seguinte método, o qual será disponível para uso a partir do menu pop-up do objeto. Esse método é simples: ele cria um novo objeto `Role`; configura o campo `name` usando um valor de texto; e então adiciona um novo papel ao campo `roles` do projeto. Ele finaliza retornando o novo objeto papel, que, quando chamado através da interface gráfica, irá resultar na exibição de um novo objeto. (Reconhecidamente, este método é um tanto supérfluo, mas ele demonstra como é fácil servir comportamento de objeto ao usuário).

```
  public Role actionAddProjectLeader() {
     Role projectLeader = (Role)createInstance(Role.class);
     projectLeader.getName().setValue("Project Leader");
     roles.add(projectLeader);
     return projectLeader;
  }
```

No entanto, o método acima somente estará disponível quando nenhum líder de projeto tiver sido associado (como será o caso antes do método ser chamado pela primeira vez). Métodos como esse pode ser controlado por um método `about...` correspondente. Um método `about...` deve retornar um objeto `About`, que pode ser revisado posteriormente para determinar se o item de menu para o método `action` deve ser desabilitado ou não.

O método abaixo é compatível com o método `actionAddProjectLeader` pela similaridade do nome. Quando chamado, a coleção mantida no campo `roles` é

percorrido. Se um dos objetos `Role` tiver o nome 'líder de projeto' então um objeto `About` – conectado para desabilitar a opção – é retornado. Se nenhum dos papéis cominarem, então um objeto `About` – conectado para habilitar a opção – é retornado. Ambos objetos são recuperados a partir da classe `ActionAbout`.

```
public About aboutActionAddProjectLeader(){
   Enumeration e = getRoles().elements();
   while(e.hasMoreElements()){
        Role role = (Role)e.nextElement();
        if(role.getName().contains("project leader", Case.INSENSITIVE)){
           return ActionAbout.DISABLE;
        }
   }
   return ActionAbout.ENABLE;
}
```

Títulos

Antes que possamos compilar e executar a aplicação nós temos que implementar o método `title` para cada classe. Então quando os objetos forem exibidos, cada um deles terá títulos distintos, permitindo que o usuário distinga um objeto de outro. Cada um dos objetos que nós acabamos de definir tem um `name` campo que nós usamos como título de cada objeto adicionando o seguinte código para cada uma das três classes. Para implementar o método `title` devemos retornar um objeto `Title`, e a maneira mais fácil de obter um objeto `Title` disponível é pedir a um outro objeto `Naked`. Nós fizemos isso aqui perguntando ao campo `name` por seu objeto `Title`.

```
public Title title() {
  return name.title();
}
```

A classe `Title` é parte do pacote principal e precisa ser importado.

```
import org.nakedobjects.object.Title;
```

O código completo

As três classes resultantes são como seguem e devem ser salvos no diretório do projeto:

Project.java

```
import java.util.Enumeration;
import org.nakedobjects.object.collection.InternalCollection;
import org.nakedobjects.object.AbstractNakedObject;
import org.nakedobjects.object.Title;
import org.nakedobjects.object.control.About;
```

```
import org.nakedobjects.object.control.ActionAbout;
import org.nakedobjects.object.value.Case;
import org.nakedobjects.object.value.TextString;

public class Project extends AbstractNakedObject {
  private final TextString name = new TextString();
  private final InternalCollection roles = new InternalCollection(Role.class, this);

  public TextString getName() {
     return name;
  }

  public InternalCollection getRoles() {
     return roles;
  }

  public Role actionAddProjectLeader() {
     Role projectLeader = (Role)createInstance(Role.class);
     projectLeader.getName().setValue("Project Leader");
     roles.add(projectLeader);
     return projectLeader;
  }

  public About aboutActionAddProjectLeader(){
     Enumeration e = getRoles().elements();
     while(e.hasMoreElements()){
           Role role = (Role)e.nextElement();
           if(role.getName().contains("project leader", Case.INSENSITIVE)){
              return ActionAbout.DISABLE;
           }
     }
     return ActionAbout.ENABLE;
  }

  public Title title() {
     return name.title();
  }
}
```

Role.java

```
import org.nakedobjects.object.AbstractNakedObject;
import org.nakedobjects.object.Title;
import org.nakedobjects.object.value.TextString;

public class Role extends AbstractNakedObject {
  private final TextString name = new TextString();
  private Employee employee;

  public TextString getName() {
     return name;
  }

  public Employee getEmployee() {
     resolve(employee);
```

```
    return employee;
  }

  public void setEmployee(Employee employee) {
    this.employee = employee;
    objectChanged();
  }

  public Title title() {
    return name.title();
  }
}
```

Employee.java

```
import org.nakedobjects.object.AbstractNakedObject;
import org.nakedobjects.object.Title;
import org.nakedobjects.object.value.TextString;

public class Employee extends AbstractNakedObject {
  private final TextString name = new TextString();

  public TextString getName() {
    return name;
  }

  public Title title() {
    return name.title();
  }
}
```

Executando uma aplicação

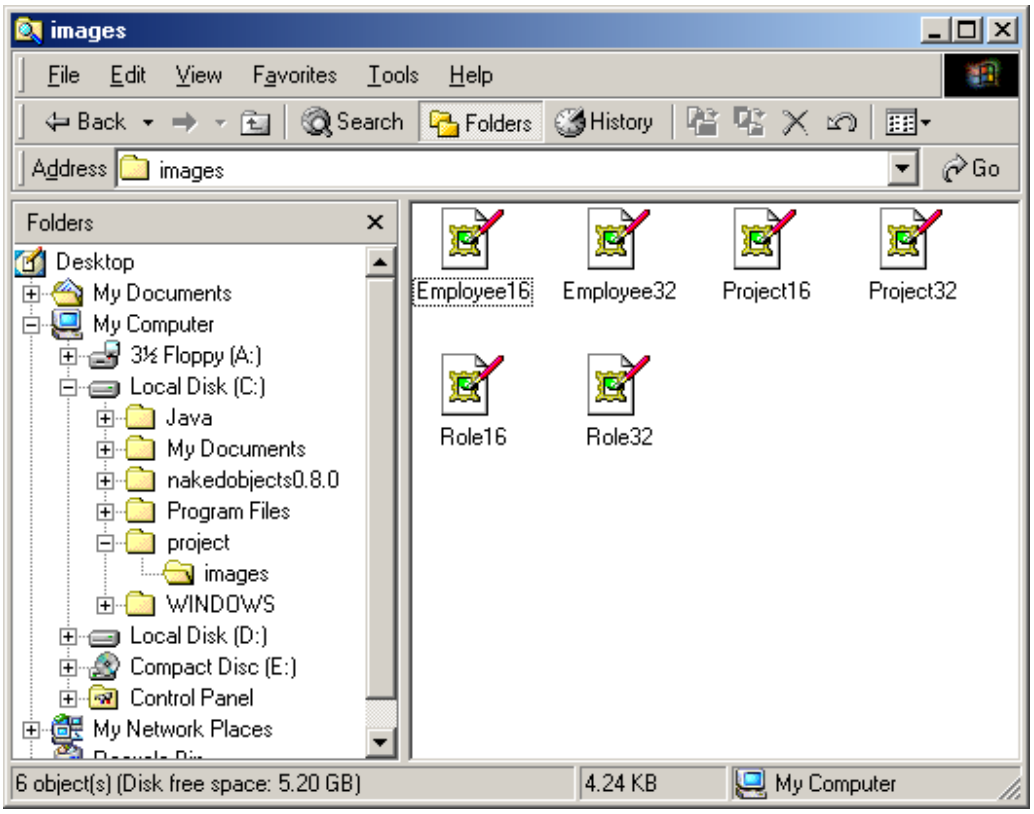
Após ter definido os nossos objetos de negócio, devemos configurar alguns ícones e escrever uma classe simples que fará com que essas classes fiquem disponíveis ao usuário. Os ícones são usados para representar nossos objetos sobre a tela e são apenas imagens, armazenadas em arquivos, que o framework pode exibir. Cada nome do arquivo de imagem é baseado no nome da classe e o arquivo deve estar num diretório chamado imagens e deve ter uma extensão.gif.

Configuração

Para configurar as imagens de ícones para esta aplicação, crie um diretório no diretório de projeto chamado ‘images’. Então copie seis imagens adequadas, a partir do diretório de biblioteca de ícones do diretório nakedobjects0.8.0, dentro do novo diretório images e as renomeie. Os nomes dos arquivos devem corresponder aos nomes das classes acrescidos com o tamanho da imagem (`16` ou `32` pixels).

As imagens sugeridas são HouseUnderConstruction, Hammer, e Man. Eles devem ser renomeados como Project, Role e Employee respectivamente. Por exemplo, Hammer16.gif irá se tornar Role16.gif e Hammer32.gif se tornará Role32.gif.

A janela abaixo exibe o novo diretório com os seis novos arquivos de imagens dentro dele:



Em Unix o comando cp pode ser usado para copiar e renomear cada arquivo de imagens, como mostrado abaixo:

```
mkdir images
cd /nakedobjects0.8.0/icon-library
cp HouseUnderConstruction16.gif /project/images/Project16.gif
cp HouseUnderConstruction32.gif /project/images/Project32.gif
cp Hammer16.gif /project/images/Role16.gif
cp Hammer32.gif /project/images/Role32.gif
cp Man16.gif /project/images/Employee16.gif
cp Man32.gif /project/images/Employee32.gif
```

A aplicação de teste

Para exibir essas classes num ambiente de teste, nós precisamos escrever uma classe simples que carregue o framework, configure-o, e então torne a classe disponível ao usuário. Por favor, note que esta classe é usada apenas para testar

a nossa aplicação; quando nós tornamos a aplicação disponível ao usuário final ele é configurado de forma diferente.

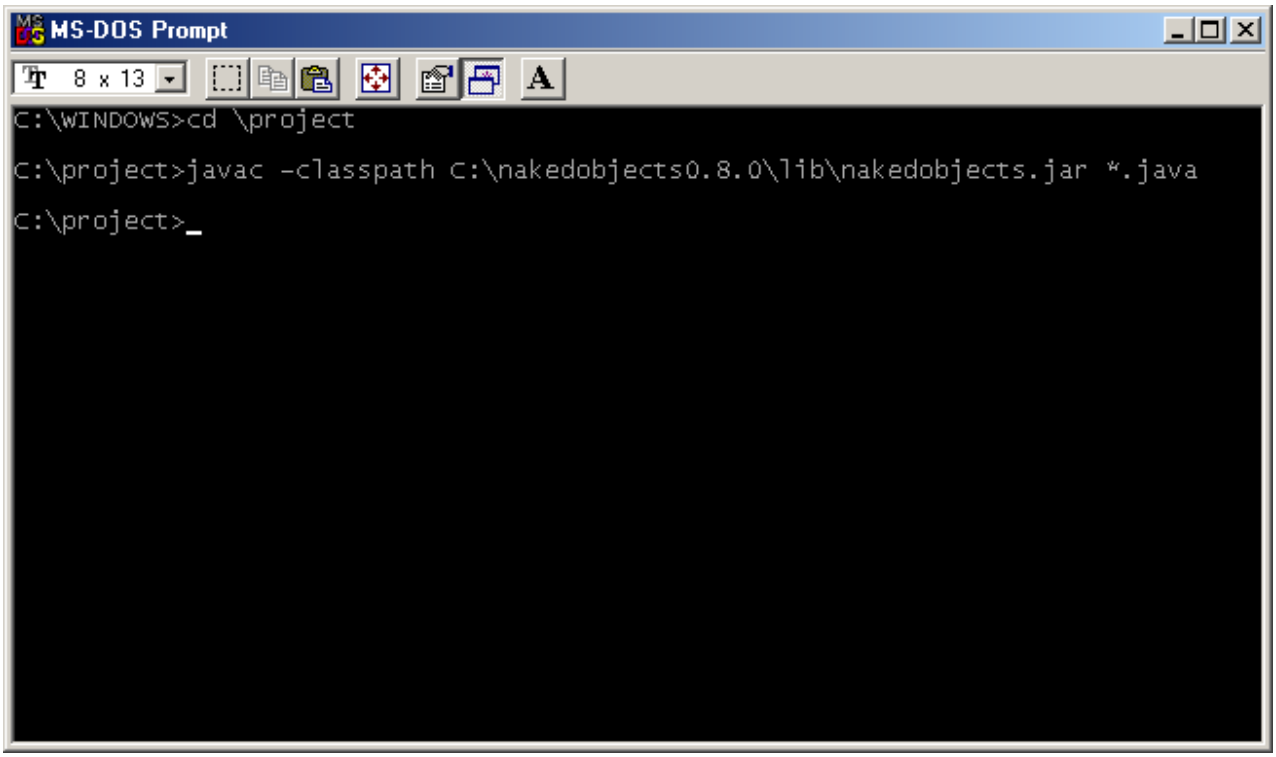
A classe a seguir é usada para carregar nossas três classes dentro do framework, cada qual disponível ao usuário.

Run.java

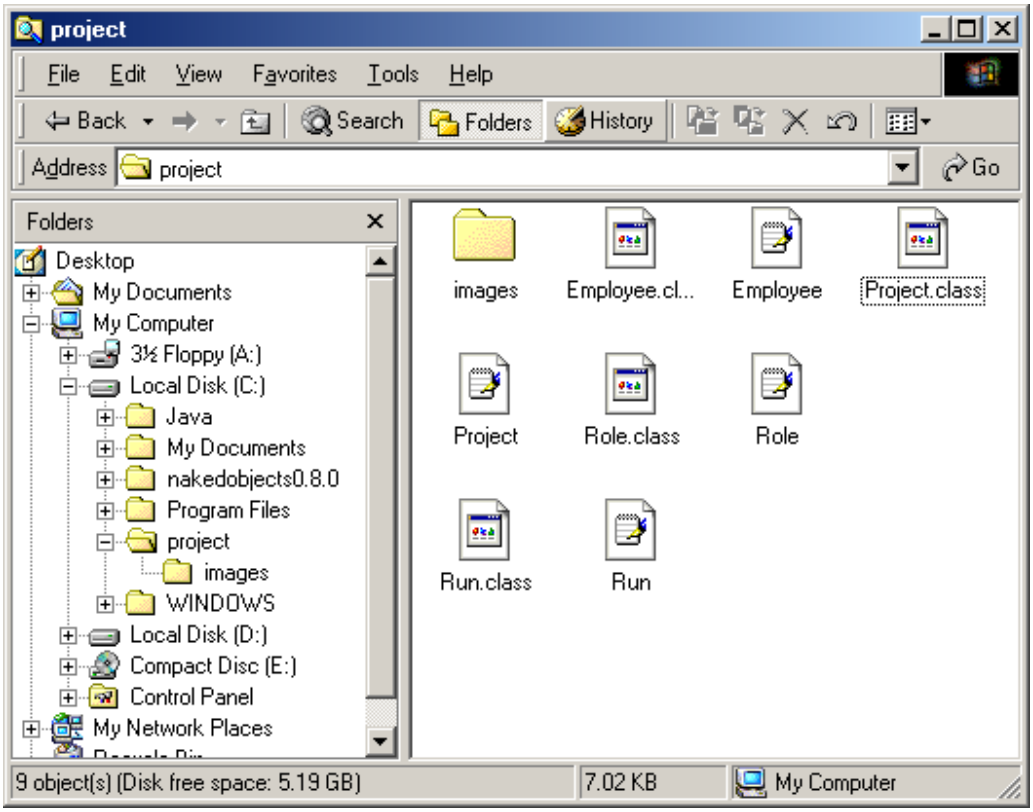
```
import org.nakedobjects.Exploration;
import org.nakedobjects.object.NakedClassList;

public class Run extends Exploration {
   public static void main(String args[]){
         new Run();
   }

         public void classSet(NakedClassList classes){
          classes.addClass(Project.class);
          classes.addClass(Role.class);
          classes.addClass(Employee.class);
         }
}
```

Compilação

Para compilar tanto as classes de objetos de negócio quanto a classe `Run` que nós precisamos para chamar o compilador Java, especifique a biblioteca do framework (nakedobjects.jar) na classe classpath. (Lembre que o nome do diretório usado, devido a versão, pode ser diferente). Para fazer isso sobre o Windows, execute uma janela MS-DOS e entre com os seguintes comandos.



Após isso, o diretório projeto deve conter os arquivos .java e .class e o diretório images como pode ser visto abaixo.



Em Unix, o comando é muito similar:

```
javac -classpath /nakedobjects0.8.0/lib/nakedobjects.jar:. *.java
```

A lista de arquivos é o mesmo exibido pelo Windows:

```
Employee.class
Employee.java
Project.class
Project.java
Role.class
Role.java
Run.class
Run.java
Images
```

Executando a aplicação

Para executar a aplicação (a classe `Run`), chame a máquina virtual Java (JVM) a partir do diretório de projeto. A classpath deve referenciar o framework, a biblioteca Log4j e o diretório corrente onde seus arquivos de classe residem. Para o Windows:

```
java -classpath
                c:\nakedobjects0.8.0\lib\nakedobjects.jar;c:\nakedobjects0.8.0\lib\log4j.jar;.
                Run
```

(Este comando deve estar numa única linha).

E no Unix:

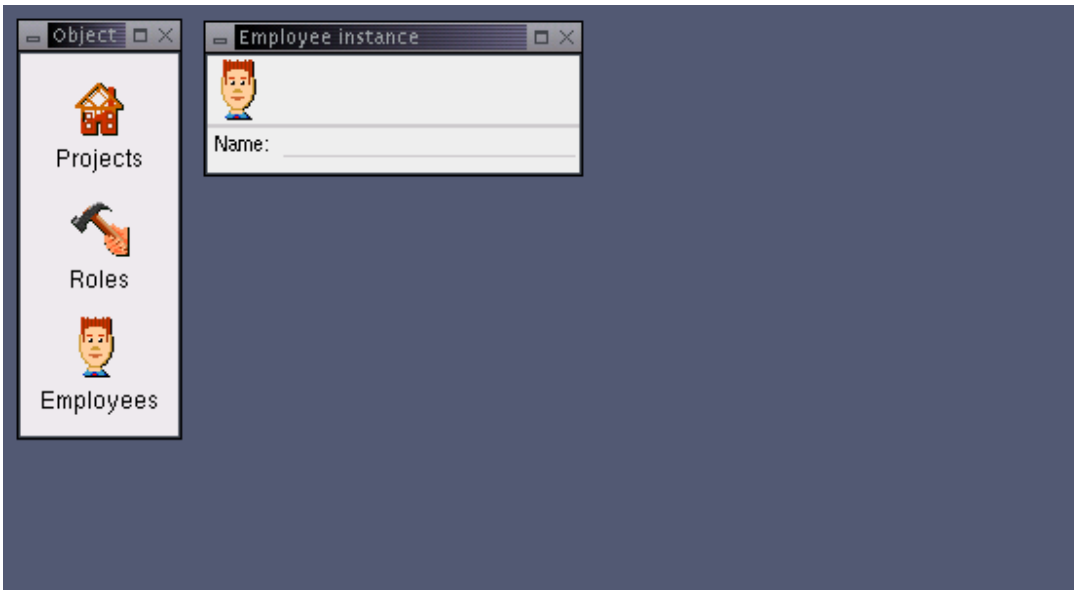
```
java -classpath
                /nakedobjects0.8.0/lib/nakedobjects.jar:/nakedobjects0.8.0/lib/log4j.jar:.
                Run
```

(Este comando deve estar numa única linha).

Uma pequena janela irá aparecer exibindo as três classes. As seguintes telas exibem essas classes em uso.

1. As janelas de classe pop-ups exibem as três classes que nós especificamos:

2. Clicando com o botão direito do mouse na classe Employees e selecionando New Employee... cria um novo objeto empregado:



3. Entrando com um nome dentro do campo Name dá ao objeto um título. O título, como definido pelo método title() retorna o valor de campo Name:

4. Da mesma forma, clicando o botão direito do mouse na classe Roles e selecionando New Role... cria um novo objeto role:

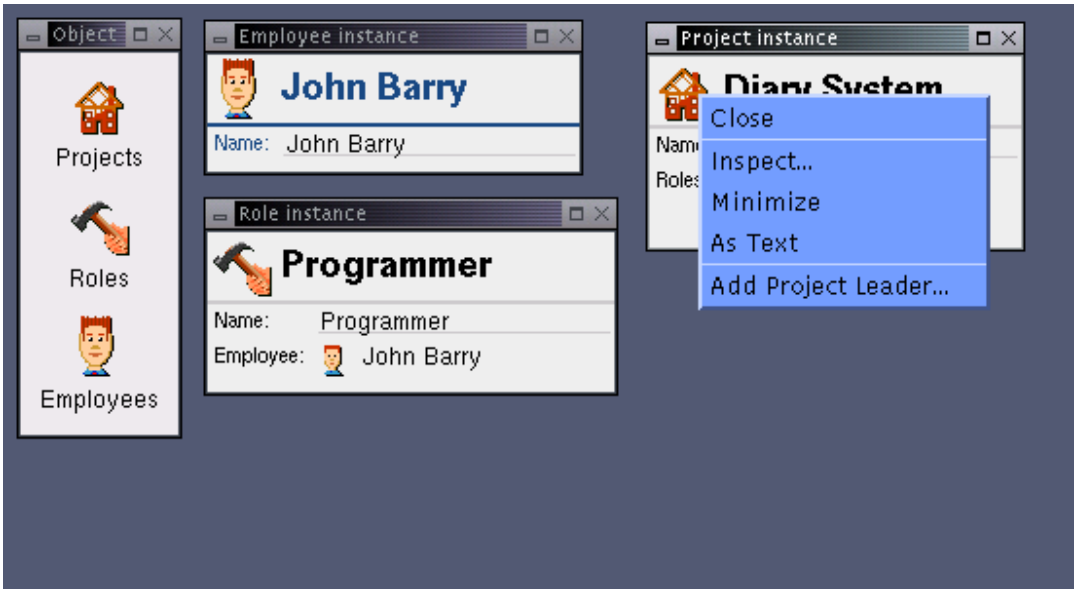
5. Entrando com um nome dentro do campo Name dá ao objeto Role um título:

6. Agora arraste o empregado John Barry e solte-o no campo Employee configurando um relacionamento entre os dois objetos: o role conhece o employee:

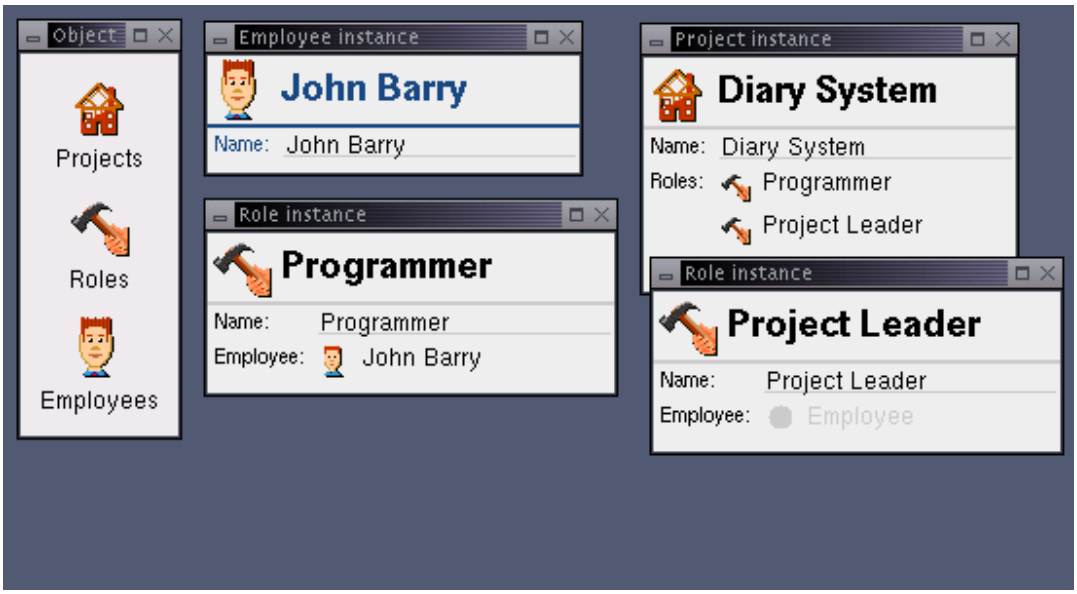
7. Como discutido, um projeto é o ponto de início normal e onde nós criamos e associamos um nome:

8. Arrastando o papel (role) Programmer sobre o campo Roles adiciona o nosso programador ao projeto:

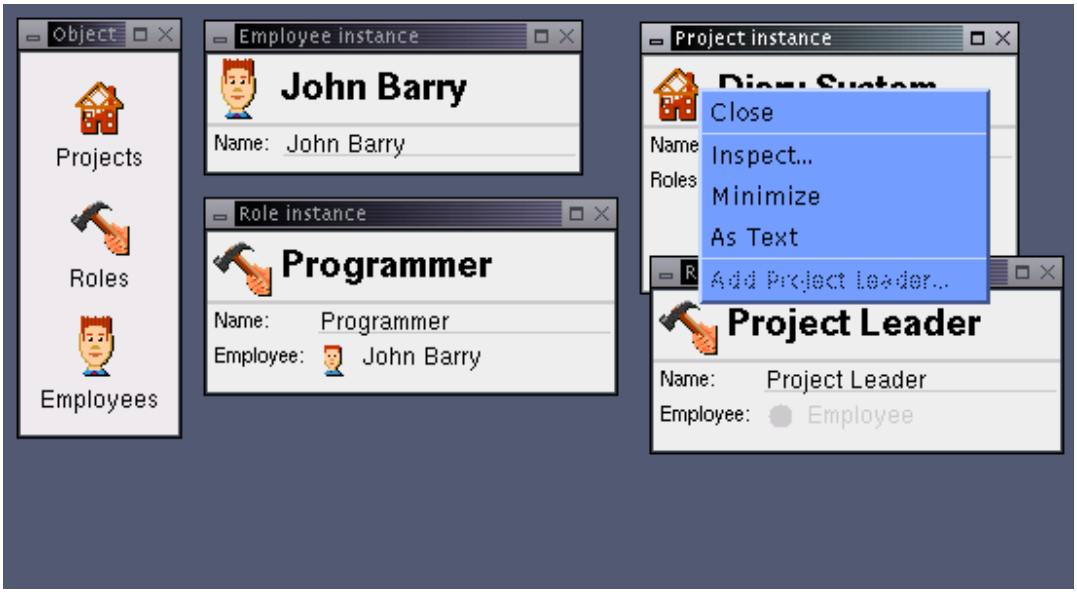
9. Clicando com o botão direito do mouse sobre o objeto Project exibe a ação que nós adicionamos: Add Project Leader...:



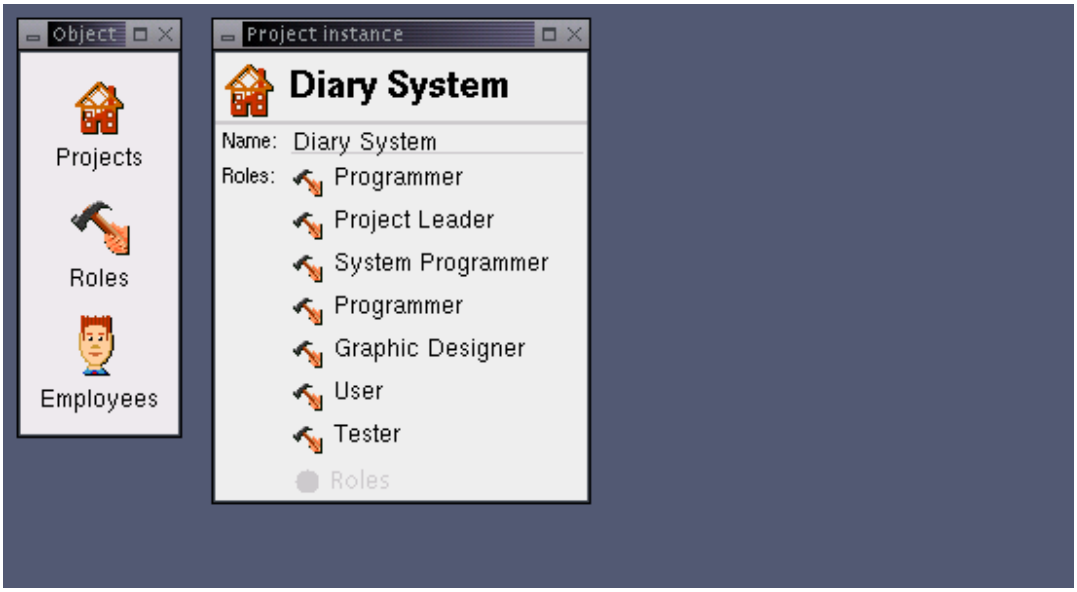
10. Ao selecionar essa opção, cria-se um novo objeto Role, rotulando-o como 'Project Leader' e adicionando-o ao campo Roles:



11. O mesmo menu pop-up agora exibe a opção desabilitada, pois o método encontrou um papel contendo um líder de projeto:



12. O estado final do projeto após todos os papeis terem sido adicionados:



Apêndice B: Exemplo de Código

A seguir apresentamos o código completo da classe `Booking` como definido na aplicação ECS. O código completo da aplicação pode ser baixado de nosso website website. Este exemplo exibe como um naked objects é definido dentro do framework e demonstra a maioria das técnicas cobertas neste livro.

Booking.java

```
package ecs.delivery;

import org.nakedobjects.object.AbstractNakedObject;
import org.nakedobjects.object.Title;
import org.nakedobjects.object.control.About;
import org.nakedobjects.object.control.ActionAbout;
import org.nakedobjects.object.control.FieldAbout;
import org.nakedobjects.object.control.ProgrammableAbout;
import org.nakedobjects.object.value.Date;
import org.nakedobjects.object.value.TextString;
import org.nakedobjects.object.value.Time;


/* Every business object is created by subclassing AbstractNakedObject
 * and implementing the title method */
public class Booking extends AbstractNakedObject {
    /* The version number is required for serialization, used when
    * passing objects between the client and server. */
```

```
private static final long serialVersionUID = 1L;

/* The values we want the user to be able to change, via the
 * keyboard, are declared as NakedValue objects. */
private final TextString reference;
private final TextString status;
private final Date date;
private final Time time;

/* All the associations between the booking and the other objects
 * are declared as other NakedObject objects. */
private City city;
private Customer customer;
private Telephone contactTelephone;
private Location pickUp;
private Location dropOff;
private PaymentMethod paymentMethod;

/* The constructor is commonly used to set up the object, specifically
 * so all the value field objects are immediately avaialable. */
public Booking() {
   /* Each value object is created via its default constructor. */
   reference = new TextString();
   /* The reference field is made read-only by assigning this FieldAbout. */
   reference.setAbout(FieldAbout.READ_ONLY);
   status = new TextString();
   status.setAbout(FieldAbout.READ_ONLY);
   date = new Date();
   time = new Time();
}

/* An aboutAction... methods control the action... method that matches it's
* name. */
public About aboutActionCheckAvailability() {
      /* A ProgrammableAbout can be used to check a number of conditions. */
   ProgrammableAbout c = new ProgrammableAbout();

   /* Checks conditions and adjusts the About accordingly: if the
   * argument is false then the About is altered so that a call
   * to canUse returns a Veto. */
   c.makeAvailableOnCondition(!getStatus().isSameAs("Available"));
   c.makeAvailableOnCondition(!getDate().isEmpty());

   return c;
}

public About aboutActionConfirm() {
   ProgrammableAbout c = new ProgrammableAbout();

   /* This version of the makeAvailableOnCondition method also adds a
   * message to the Veto. */
   c.makeAvailableOnCondition(getStatus().isSameAs("Available"),
                        "Status must be 'Available'");

   return c;
}
```

```
    public About aboutActionCopyBooking() {
        int sets = 0;

        sets += ((getCustomer() != null) ? 1 : 0);
        sets += ((getPickUp() != null) ? 1 : 0);
        sets += ((getDropOff() != null) ? 1 : 0);
        sets += ((getPaymentMethod() != null) ? 1 : 0);
        sets += ((getContactTelephone() != null) ? 1 : 0);

        /* An About can be conditionally created: if the argument is true then
         * the returned About enables the action; if false, it disables it. */
        return ActionAbout.enable(sets >= 3);
    }

    public About aboutActionReturnBooking() {
        ProgrammableAbout c = new ProgrammableAbout();

        /* A description can be added to the About to tell the user what
         * the action will do. */
        c.setDescription(
                "Creates a new Booking based on the current booking.  The new booking has the pick up
amd drop off locations reversed.");
        c.makeAvailableOnCondition(getStatus().isSameAs("Confirmed"),
                                "Can only create a return based on a confirmed booking");

        /* If the About is still allowing the action then the action's
         * name will be changed. */
        c.changeNameIfAvailable("Return booking to " + getPickUp());

        return c;
    }

    public About aboutPickUp(Location newPickup) {
        ProgrammableAbout c = new ProgrammableAbout();

        c.setDescription("The location to pick up the customer from.");

        if ((newPickup != null) && (getCity() != null)) {
            if (newPickup.equals(getDropOff())) {
                c.makeUnavailable(
                        "Pick up must differ from the drop off location");
            } else {
                boolean sameCity = getCity().equals(newPickup.getCity());

                c.makeAvailableOnCondition(sameCity,
                                        "Location must be in " +
                                        getCity().title());
            }
        }

        return c;
    }

    /* Zero-parametered action methods are made available to the user via
    * the object's pop up menu.  */
```

```
public void actionCheckAvailability() {
   /* The value field is accessed and its value changed. */
   getStatus().setValue("Available");
   objectChanged();
}

public void actionConfirm() {
   getStatus().setValue("Confirmed");

   /* The locations used are added to the customer's Locations field.
   * Note that the accessor methods are used to ensure that the objects
   * are loaded first. */
   getCustomer().associateLocations(getPickUp());
   getCustomer().associateLocations(getDropOff());

   if (getCustomer().getPreferredPaymentMethod() == null) {
      getCustomer().setPreferredPaymentMethod(getPaymentMethod());
   }
}

public Booking actionCopyBooking() {
   /* A new instance is created by calling the createInstance method.  This
   * ensures that the created method is always called and that the object
   * is made persistent. */
   Booking copiedBooking = (Booking) createInstance(Booking.class);

   copiedBooking.associateCustomer(getCustomer());
   copiedBooking.setPickUp(getPickUp());
   copiedBooking.setDropOff(getDropOff());
   copiedBooking.setPaymentMethod(getPaymentMethod());
   copiedBooking.setContactTelephone(getContactTelephone());

   /* By returning the object we ensure that the user gets it: it is
   * displayed to the user in a new window. */
   return copiedBooking;
}

/* One-parametered action methods are also available to the user and are
* invoked via drag and drop.
*
* When an action method is marked as static then it works for the
* class rather than the object.  To invoke this method the user must
* drop a customer onto the Booking class icon. */
public static Booking actionNewBooking(Customer customer) {
   Booking newBooking = (Booking) createInstance(Booking.class);

   newBooking.setCustomer(customer);
   newBooking.setPaymentMethod(customer.getPreferredPaymentMethod());

   return newBooking;
}

/* The recommended ordering for the action methods can be specified
* with the actionOrder method.   This will affect the order of the
* menu items for this object. */
public static String actionOrder() {
```

```
    return "Check Availability, Confirm, Copy Booking, Return Booking";
}

public Booking actionReturnBooking() {
    Booking returnBooking = (Booking) createInstance(Booking.class);

    returnBooking.associateCustomer(getCustomer());
    returnBooking.setPickUp(getDropOff());
    returnBooking.setDropOff(getPickUp());
    returnBooking.setPaymentMethod(getPaymentMethod());
    returnBooking.setContactTelephone(getContactTelephone());

    return returnBooking;
}

/* The associate method overrides the get/set and is called by the
* framework instead of the ordinary accessor methods.  They are used
* to set up complex or bidirectional associations.  This method delegates,
* to the other class, the work to set up a bidirectional link.  */
public void associateCustomer(Customer customer) {
    customer.associateBookings(this);
}

public void associateDropOff(Location newDropOff) {
    setDropOff(newDropOff);
    setCity(newDropOff.getCity());
}

public void associatePickUp(Location newPickUp) {
    setPickUp(newPickUp);
    setCity(newPickUp.getCity());
}

private long createBookingRef() {
    try {
        /* The object store provides the ability to create and maintain
        * numbered sequences, which are unique. */
        return getObjectStore().serialNumber("booking ref");
    } catch (org.nakedobjects.object.ObjectStoreException e) {
        return 0;
    }
}

/* The created method is called when the logical object is created,
* i.e. it is not called when an object is recreated from its persisted
* data. */
public void created() {
    status.setValue("New Booking");
    reference.setValue("#" + createBookingRef());
}

/* The dissociate method mirrors the associate method and is called
* when the user tries to remove a reference. */
public void dissociateCustomer(Customer customer) {
    customer.dissociateBookings(this);
}
```

```
/* The recommended order for the fields to be presented to the user
* can be specified by the fieldOrder method. */
public static String fieldOrder() {
   return "reference, status, customer, date, time, pick up, drop off, payment method";
}

/* Each association within an object requires a get and a set method.
 * The get method simply returns the associated object's reference after
 * it has ensured that the object has been loaded into memory.
 */
public City getCity() {
   resolve(city);

   return city;
}

public Telephone getContactTelephone() {
   resolve(contactTelephone);

   return contactTelephone;
}

public Customer getCustomer() {
   resolve(customer);

   return customer;
}

/* Each value field only has a get method, which returns the value's
 * reference.  No set is required as the value objects must be a integral
 * part of the naked object. */
public final Date getDate() {
   return date;
}

public Location getDropOff() {
   resolve(dropOff);

   return dropOff;
}

public PaymentMethod getPaymentMethod() {
   resolve(paymentMethod);

   return paymentMethod;
}

public Location getPickUp() {
   resolve(pickUp);

   return pickUp;
}

public final TextString getReference() {
   return reference;
```

```
    }

    public final TextString getStatus() {
        return status;
    }

    public final Time getTime() {
        return time;
    }

    /* The association has a set method so a reference can be passed to
     * the object to set up the association.  As the object has now changed
     * it also must be notified.*/
    public void setCity(City newCity) {
        city = newCity;
        objectChanged();
    }

    public void setContactTelephone(Telephone newContactTelephone) {
        contactTelephone = newContactTelephone;
        objectChanged();
    }

    public void setCustomer(Customer newCustomer) {
        customer = newCustomer;
        objectChanged();
    }

    public void setDropOff(Location newDropOff) {
        dropOff = newDropOff;
        objectChanged();
    }

    public void setPaymentMethod(PaymentMethod newPaymentMethod) {
        paymentMethod = newPaymentMethod;
        objectChanged();
    }

    public void setPickUp(Location newPickUp) {
        pickUp = newPickUp;
        objectChanged();
    }

    /* The title method generates a title string for the object that will
    * be used when the object is displayed.  This title should identify the
    * object to the user. */
    public Title title() {
        /* A Title object is normally retrieved from one of the object's
        * field (all Naked objects can return a Title).  All of the title's
        * methods return the same Title object. */
        return reference.title().append(status);
    }
}
```

Apêndice C: Código Cliché

Desenvolver software utilizando recursos de copiar de outros programas e colar incorre em alguns riscos, mas quando você está aprendendo, isso é uma abordagem pragmática e normalmente efetiva. Nós fornecemos aqui alguns exemplos do código 'cliché' normalmente usado em programas Naked Objects.

Classes de exploração

Para explorar um conjunto de classes que você desenvolveu, crie uma extensão de `Exploration` e adicione suas classe ao ClassSet no método classSet.

```
import org.nakedobjects.Exploration;
import org.nakedobjects.object.NakedClassList;

public class MyExploration extends Exploration {
   public void classSet(NakedClassList classes) {
      /* Add to the list all classes that are to made available to
       * to the user in the classes window */
      classes.addClass(NakedObjectClass1.class);
      classes.addClass(NakedObjectClass2.class);
      classes.addClass(NakedObjectClass3.class);
      :
   }

   public void initObjects() {
      NakedObjectClass newObject;
      /* create instances via the class and initialize them */
      newObject = (NakedObjectClass) createInstance(NakedObjectClass.class);
      newObject.getValueObject().setValue(value);
      newObject.setAssociation(object);
    }

   public static void main(String[] args) {
      new MyExploration();
   }
```

Naked Object

O seguinte modelo exibe a declaração de um objeto de negócio. Apenas o construtor, sem nenhum parâmetro, e o método `title` é exigido.

```
import org.nakedobject.object.NakedObject;
import org.nakedobject.object.Title;
import org.nakedobject.object.value.ValueType;
:
:
public class ClassName extends AbstractNakedObject {

   /* Makes the class uninstantiable: no new objects can be created */
   public static About aboutClassName() {
```

```
    return ClassAbout.UNINSTANTIABLE;
}

/* Gives the class a name */
 public static String singularName() {
    return "Singular Name";
}

/* Gives the class a plural name */
public static String pluralName() {
    return "Plural Name";
}

public Object() {
     /* Set a value's about so the value is uneditable */
    value.setAbout(FieldAbout.READ_ONLY);
}

/* This method is only called when the logical object is created.
 * The constructor will be called every time the object is
 * recreated (by the persistence mechanism). */
public void created() {}

/* Titles can be generated directly from both value and one-to-one
 * association objects, but not from one-to-many associations.  See
 * below for details of the Title class. */
public Title title() {
    return value/object.title();
}

/* Makes the object read-only */
public About about() {
    return ObjectAbout.READ_ONLY;
}

/* Value fields: TextString, Date, Time, DateTime, WholeNumber,
 * FloatingPointNumber, Percentage, Money, Option, URLString and     Label.
 * Should be marked as final and not have a set... method. */
private final ValueType value = new ValueType();

public ValueType getVariable() {
    return value;
}

 /* Association fields: any other class based on AbstractNakedObject.*/
private ClassName object;

public ClassName getObject() {
    resolve(object);
    return object;
}

public void setObject(ClassName object) {
    this.object = object;
    objectChanged();
}
```

```
/* The associate... and dissociate... method can override
 * the set... method and, hence, used to set up associations. */
public void associateObject(ClassName object) {
   setObject(object);
}

public void dissociateObject(ClassName object) {
   setObject(null);
}

/* The About object returned by    this method field determines
 * if the field can be changed. */
public About aboutObject(ClassName object) {
 return FieldAbout.READ_ONLY;
}

/* One-to-many association.  The collection object must be instantiated
 * with the element type and a reference to its parent, the owner.
 * This should be marked as final and only a set... method provided. */
private final InternalCollection objects = new InternalCollection(ClassName.class, this);

public InternalCollection getObjects() {
   return objects;
}

/* The associate... and dissociate... method
 * can override the get... method and, hence, are used
 * to set up associations.  The collection's add and
 * remove methods are used to change the collection. */
public void associateObjects((Object object) {
   objects.add(object);
}

public void dissociateObjects((Object object) {
   objects.remove(null);
}

/* The About object returned by this method determines if
 * objects can be added and removed from this field. */
public About aboutObjects() {
 return FieldAbout.READ_ONLY;
}

/* action methods */

/* Zero-parameter methods are show in the object's pop-up menu.  If the
 * method returns an object then this will be shown.*/
public void/Object actionOptionName() {
   :
   return object;
}

/* The about method determines whether the option will disabled or not. */
public About aboutActionOptionName() {
```

```
        return ActionAbout.enable();
    }

    /* One-parameter methods are made available through drag and drop.
     * If the method returns an object then this will be shown. */
    public void/Object actionOptionName(Object object) {
        :
        return object;
    }

     /* The about method determines whether the option will disabled or not. */
    public About aboutActionOptionName(Object object)(Object object) {
        return ActionAbout.enable();
    }
}
```

Title

Existem três maneiras básicas de criar objetos `Title`.

```
value.title()
association.title()
new Title("text")
```

Títulos podem receber sufixos, com a adição necessária de espaços, ou ter detalhes concatenados.

```
value.title().append("text");
association.title().append(value)
new Title("text").append(association)

value.title().concat("text");
association.title().concat(value)
new Title("text").concat(association)
```

Apêndice D: Biblioteca de ícone

Estes ícones foram incluídos com a distribuição do framework Naked Objects, tanto no formato pequeno (16x16 pixel) quanto no formato grande (32x32). Eles foram disponibilizados para sejam usados durante a fase de exploração de um projeto. Para um sistema de entrefas nós recomendamos que você use um conjunto que seja visualmente consistente.

Aircraft32.gif

Audiotape32.gif

Bargraph.gif

Basket32.gif

Bed32.gif

CallCentreAgent32.gif

Can32.gif

Car32.gif

Cheque32.gif

ChessPiece32.gif

City32.gif

Coins32.gif

CreditCard32.gif

Crosshairs32.gif

Factory32.gif

Fax32.gif

Folder32.gif

Globe32.gif

Hammer32.gif

Handshake32.gif

HouseUnderConstruction32.gif

KnifeFork32.gif

Letter32.gif

LightningFlash32.gif

Man32.gif

Map32.gif

MissedTarget32.gif

Missile32.gif

MobilePhone32.gif

Network32.gif

Pen32.gif

Phone32.gif

PiggyBank32.gif

QuestionMark32.gif

RadialPlot32.gif

RoundTable32.gif

Scales32.gif

Smiley32.gif

SpeechBubble32.gif

SteeringWheel32.gif

Taxi32.gif

Truck32.gif

Woman32.gif

Bibliografia

[Abbott1983] Abbott, R. J. (1983). 'Program design by informal English descriptions.' Communications of the ACM **26(11)**: 882 - 894.

[Alexander1977] Alexander, C., S. Ishikawa, et al. (1977). A Pattern Language. New York, Oxford University Press.

[Beck1999] Beck, K. (1999). EXtreme Programming EXplained., Addison-Wesley.

[Beck1989] Beck, K. and W. Cunningham (1989). A Laboratory for Teaching Object-Oriented Thinking OOPLSA '89, Association of Computing Machinery.

[Booch1986] Booch, G. (1986). 'Object-Oriented Development.' IEEE Transactions on Software Engineering **12(2)**: 211-221.

[Booch1994] Booch, G. (1994). Object-Oriented Analysis and Design: with Applications. Addison-Wesley.

[Brown2000] Brown, J. S. and P. Duguid (2000). The Social Life of Information. Boston, MA, Harvard Business School Press.

[Bruner1966] Bruner, J. (1966). Toward a Theory of Instruction. Cambridge, MA, Belknap Press/Harvard University Press.

[Clement1996] Clement, A. (1996). 'Computing at work: Empowering Action by Low-Level Users.' in Computerization and Controversy - Value Conflicts and Social Choices. R. Kling Ed. San Diego, CA, Academic Press.

[Coad1999] Coad, P. and E. Lefebvre (1999). Modeling in Color. Software Development. March 1999.

[Collins1995] Collins, D. (1995). Designing Object-oriented User interfaces. Redwood City, CA, Benjamin/Cummings.

[Dahl1966] Dahl, O. J. and K. Nygaard (1966). 'Simula -- an Algol-based simulation language.' CACM **(9)**: 671-678.

[Deligiannis2002] Deligiannis, I., M. Shepperd, et al. (2002). A Controlled Experiment Investigation of an Object-Oriented Design Heuristic for Maintainability. Bournemouth University, ESERG,.

[Dietel1999] Dietel, H. and D. P (1999). Java: How to Program. Third Edition, Prentice Hall.

[Eeles1998] Eeles, P. and O. Sims (1998). Building Business Objects. New York, John Wiley.

[Finsterwalder2001] Finsterwalder, M. (2001). Automating Acceptance Tests for GUI Applications. XP2001, Cagliari.

[Firesmith1996] Firesmith, D. (1996). 'Use Cases: The Pros and Cons.' in Wisdom of the Gurus. R. Wiener. New York, SIGS books.

[Fowler2000] Fowler, M. (2000). Refactoring. Addison-Wesley.

[Gamma1995] Gamma, E., R. Helm, et al. (1995). Design Patterns - Elements of Reusable Object Oriented Software. Reading, MA, Addison-Wesley.

[Garson1988] Garson, B. (1988). The Electronic Sweatshop - How Computers are Transforming the Office of the Future into the Factory of the Past. New York, Simon and Schuster.

[Groder1999] Groder, C. (1999). 'Building Maintainable GUI Tests.' in Software Test Automation. M. Fewster and D. Graham, Eds., ACM Press / Addison-Wesley.

[Hammer1993] Hammer, M. and J. Champy (1993). Reengineering the Corporation: A Manifesto for Business Revolution. Harper Collins.

[Herzum2000] Herzum, P. and O. Sims (2000). Business Component Factory. Wiley.

[Hiltzik1999] Hiltzik, M. (1999). Dealers of Lightning - Xerox PARC and the Dawn of the Computer Age. New York, HarperCollins.

[Holub1999] Holub, A. (1999). 'Building User Interfaces for Object-Oriented Systems.' Java World. July 1999.

[Hunt2000] Hunt, A. and D. Thomas (2000). The Pragmatic Programmer. Addison-Wesley.

[Hutchins1986] Hutchins, E., J. Hollan, et al. (1986). 'Direct Manipulation Interfaces.' in User Centered System Design: New Perspectives on Human-Computer Interaction. D. Norman and S. Draper, Eds. Hillsdale, NJ, Lawrence Erlbaum.

[IBM1991] IBM (1991). CUA-91 Manual: Common User Access-Advanced Interface Design Reference. IBM.

[Ingalls1997] Ingalls, D., T. Kaehler, et al. (1997). Back to the Future: The story of Squeak. OOPSLA'97, Association of Computing Machinery.

[Jacobson1992] Jacobson, I., M. Christersson, et al. (1992). Object-oriented Software Engineering: A Use Case Driven Approach. Reading, MA, Addison-Wesley.

[Jacobson1995] Jacobson, I., M. Ericsson, et al. (1995). The Object Advantage - Business Process Reengineering with Object Technology. Reading, MA, Addison-Wesley.

[Jacobson1999] Jacobson, I., J. Rumbaugh, et al. (1999). The Unified Software Development Process.Addison-Wesley

[Kaner1997] Kaner, C. (1997). 'Pitfalls and Strategies in Automated Testing.' IEEE Computer **30(4)**: 114-116.

[Kanigel1997] Kanigel, R. (1997). The One Best Way - Frederick Winslow Taylor and the Enigma of Efficiency. London, Little, Brown and company.

[Kay1990] Kay, A. (1990). 'User Interface: A Personal View.' in The Art of Human-Computer Interface Design. B. Laurel. Reading, MA, Addison-Wesley: 191-207.

[Kay1996] Kay, A. (1996). 'The early history of SmallTalk.' in History of Programming Languages. T. Bergin and R. Gibson. Reading, MA, Addison-Wesley / ACM Press: 511-589.

[Krasner1988] Krasner, G. and S. Pope (1988). 'A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80.' Journal of Object Oriented Programming **1(3)**: 26-49.

[Laurel1991] Laurel, B. (1991). Computers as Theatre. Reading, MA, Addison-Wesley.

[Levy1994] Levy, S. (1994). Insanely Great. New York, Penguin Books.

[Maloney1995] Maloney, J. and R. Smith (1995). Directness and Liveness in the Morphic User Interface Construction Environment. UIST, Pittsburgh, ACM.

[Meyer1988] Meyer, B. (1988). Object-oriented Software Construction. Prentice-Hall.

[Pawson1995] Pawson, R., J.-L. Bravard, et al. (1995). 'The Case for Expressive Systems.' Sloan Management Review **Winter 1995**: 41-48.

[Porter1985] Porter, M. (1985). Competitive Advantage: Creating an Sustaining Superior Performance. New York, Free Press.

[Raskin2000] Raskin, J. (2000). The Humane Interface. Reading, MA, Addison-Wesley / ACM Press.

[Riel1996] Riel, A. (1996). Object-Oriented Design Heuristics. Addison-Wesley.

[Roberts1998] Roberts, D., D. Berry, et al. (1998). Designing for the User with OVID. IndianApolis, Macmillan Technical Publishing / IBM.

[Rosson1989] Rosson, M. B. and E. Gold (1989). Problem-Solution Mapping in Object-Oriented Design. OOPSLA '89, New York, ACM.

[Rurnbaugh1991] Rurnbaugh, J., M. Blaha, et al. (1991). Object-Oriented Modeling and Design. Englewood Cliffs, NJ, Prentice-Hall.

[Rurnbaugh1999] Rumbaugh, J., I. Jacobson, et al. (1999). The Unified Modeling Language Reference Guide. Reading, MA, Addison Wesley

[Sharble1993] Sharble, R. and S. Cohen (1993). 'The object-oriented brewery: a comparison of two object-oriented development methods.' SIGSOFT Software Engineering Notes **18(2)**.

[Sharp1997] Sharp, A. (1997). Smalltalk By Example. McGraw-Hill.

[Shlaer1988] Shlaer, S. and S. J. Mellor (1988). Object-Oriented Systems Analysis - Modelling the World in Data. Yourdon Press.

[Shneiderman1982] Shneiderman, B. (1982). 'The Future of Interactive Systems, and the Emergence of Direct Manipulation.' Behaviour and Information Technology **1**: 237-256.

[Shneiderman1998] Shneiderman, B., Ed. (1998). Designing the User Interface. Reading, MA, Addison-Wesley.

[Sims1994] Sims, O. (1994). Business Objects: Ease of Programming for Client-Server. McGraw-Hill.

[Smith1995] Smith, R., J. Maloney, et al. (1995). The Self-4.0 User Interface: Manifesting a System-wide Vision of Concreteness, Uniformity, and Flexibility. OOPSLA '95, Association of Computing Machinery.

[Stabell1998] Stabell, C. and O, Fjeldstad (1998). 'Configuring Value for Competitive Advantage: On Chains, Shops and Networks.' Strategic Management Journal **19**: 413-437.

[Sutherland1963] Sutherland, I. (1963). Sketchpad: A Man-Machine Graphical Communication System. Spring Joint Computer Conference.

[Taylor1911] Taylor, F. (1911). The Principles of Scientific Management. New York, W.W. Norton and Co.

[Wirfs-Brock] Wirfs-Brock, R. 'Characterizing Your Objects.' SmallTalk Report **2(5)**.

[Wirfs-Brock] Wirfs-Brock, R. 'How Designs Differ.' Report on Object Analysis and Design **1(4)**.

[Wirfs-Brock1989] Wirfs-Brock, R. and B. Wilkerson (1989). Object-oriented Design: A Responsibility-Driven Approach. OOPSLA, New Orleans.