JAdaptiveForms

Prof. Alfredo Goldman

MAC 499

TRABALHO DE FORMATURA SUPERVISIONADO

Giuliano Mega
Peter Kreslins Junior

INTRODUÇÃO

Sistemas comerciais, de gestão empresarial, gerenciamento de escolas ou até mesmo sistemas de venda em balcão são fortemente dependentes de mecanismos de entrada de dado por meio de interfaces gráficas. Assim, gostaríamos de obter interfaces que sejam amigáveis e padronizadas para tais manipulações.

Do ponto de vista de desenvolvimento desses softwares, sabemos que a tarefa de produzir tais interfaces não é trivial. Mesmo existindo, atualmente, utilitários que ajudam nesse serviço - como construtores de interfaces gráficas por exemplo – é notório que os desenvolvedores queiram poupar ainda mais esforços nessa empreitada.

Com o intuito de minimizar esse esforço requerido para a construção de interfaces que recebem e exibem dados para o usuário, foi criado o JAdaptiveForms. Assim, podemos definir esse software como sendo uma ferramenta de definição de formulários de entrada de dados independente da fonte. Essa independência vem do fato de que JAdaptiveForms não requer um banco de dados ou tecnologia de persistência específica. Ele foi modelado para que consiga trabalhar em cima de qualquer modelo de persistência que siga padrões razoáveis de modelagem de dados.

Utilizando o conceito de modelos adaptativos de objetos, projetamos um software que se molda de acordo com definições escritas em arquivos XML. No decorrer deste documento, veremos os diversos componentes e conceitos que definem o JAdaptiveForms.

METODOLOGIA

Para desenvolver o JAdaptiveForms em duas pessoas algumas medidas foram necessárias, para que o projeto pudesse caminhar e para que as idéias pudessem ser bem entendidas pelos membros. Como, para o momento, este é um projeto conceitual, muito mais do que de implementação, várias semanas de discussão foram necessárias para que se chegasse a um acordo sobre a arquitetura do software. Abordaremos nessa seção, um pouco da técnica, por nós utilizada, para modelar e construir a arquitetura inicial do software.

O ciclo de desenvolvimento da arquitetura foi baseado em reuniões periódicas (semanais) para discussão dos problemas a serem enfrentados e as possíveis soluções. O ideal é que se tenha pelo menos uma reunião por semana e, assim, os problemas já discutidos e possivelmente solucionados não são esquecidos. Estimamos que tivemos por volta de 20 reuniões, todas com aproximadamente 4 horas de duração onde procuramos, entre outras coisas:

- pesquisar alguns softwares existentes que se aproximem da proposta do JAdaptiveForms;
- pesquisar domínios de utilização do software proposto;
- entender as necessidades dos softwares de entrada de dados (ponto-chave do nosso projeto)
- definir as funcionalidades desejáveis para o JAdaptiveForms;
- pesquisar as ferramentas que fazem uma camada entre o modelo de objetos e a persistência, pois desejávamos tornar o JAdaptiveForms independente da fonte de dados;
- desenvolvimento de uma arquitetura suficientemente flexível para acomodar relacionamentos entre entidades (clientes e seus pedidos de compra por exemplo);
- desenvolvimento de alguns exemplos de arquivos de configuração (XML)
- iniciar a codificação do software, respeitando a arquitetura

Por outro lado, sempre quisemos fazer desse software um utilitário para a comunidade de desenvolvimento. Para isso, desde o início, registramos a proposta de desenvolvimento, no site de apoio ao software livre, chamado SourceForge. Neste site, você tem a possibilidade de submeter uma idéia de software livre para apreciação. Uma vez aceito, seu projeto fica hospedado e você ganha muitos recursos, como CVS (sistema de controle de versão), página web, fórum de discussão, bug tracking, espaço para disponibilizar os arquivos, compiler farm (um conjunto de servidores rodando diversas plataformas onde você pode testar seu software), entre outros.

Com esses recursos, tivemos maior facilidade para desenvolver o software de forma independente, ou seja, cada um em sua casa. No entanto, procuramos aplicar o conceito de programação pareada sempre que possível. E a verdade é que esta metodologia de desenvolvimento realmente funciona: enquanto um codifica, o outro detecta erros quase que imediatamente. Além disso, as discussões acontecem enquanto você codifica, e isto enriquece o produto final.

Ao longo deste documento, descreveremos os principais conceitos criados ao redor do JAdaptiveForms e daremos uma visão geral de onde o projeto se encontra e para onde desejamos que ele vá.

FERRAMENTAS

Como já foi dito anteriormente, o uso do SourceForge facilitou muito a tarefa de dividir as responsabilidades pelo projeto. O sistema de controle de versões (CVS) permite que muitos desenvolvedores trabalhem em um mesmo projeto e ao mesmo tempo, mantendo um histórico de todas as alterações feitas e disponibilizando todos esses arquivos para download.

Além disso, utilizamos o ambiente de desenvolvimento Eclipse, que também tem licença de código livre. Ele pode ser integrado ao CVS e consegue auxiliar nas tarefas repetitivas como enviar os novos arquivos para o servidor, trazer as versões atuais para a máquina local, etc.

Com relação à arquitetura do sistema como um todo, nos inspiramos fortemente no modelo de objetos adaptativos, cujo principal “pregador” é o Joe Yoder (www.adaptiveobjectmodel.com). Nesse modelo, a principal idéia é que o desenvolvedor entenda que software muda ao longo do tempo. Assim, não basta criar um software que funcione agora, mas que em um ou dois anos precisará de uma reforma drástica. Assim, a proposta é que se desenvolva pensando em flexibilidade. E o JAdaptiveForms é uma inspiração desse modelo. Ele não é 100% adaptativo, mas podemos considerá-lo bastante flexível, na medida em que arquivos de configuração XML determinam o modelo de objetos que executarão naquele domínio de aplicação.

SOLUÇÃO

A solução parte do princípio que em toda tarefa repetitiva num domínio é possível identificar, em algum nível, um conjunto de participantes comuns que interagem e variam de forma previsível. Determinar tais participantes e a natureza de suas interações não é, todavia, um processo trivial.

Começamos com um modelo de semântica frouxa, num nível de abstração bastante alto, tendo como ponto de partida apenas os requisitos dados pelos casos de uso que

motivaram o projeto. Esse modelo foi refinado sucessivamente no decorrer do semestre e, à medida em que as reuniões foram realizadas, introduzimos novos casos de uso e novas extensões à semântica do modelo.

Ao final do processo obtivemos um modelo estável, flexível e completo, capaz de abarcar grande parte dos casos de uso propostos. A filosofia por trás do modelo reflete exatamente a direção esperada na solução – identificar os participantes, a natureza de suas interações e suas variações.

Problema

Antes de prosseguir com uma solução é necessário antes definir o problema; isto é, o que exatamente é um formulário de entrada de dados. Formulários de entrada de dados são janelas em programas que obedecem, em essência, a seguinte forma:

- Possuem campos onde o usuário pode digitar dados;
- possuem listas de onde o usuário pode selecionar e modificar dados

Muitas vezes, todavia, o usuário quer ser capaz de selecionar um item numa lista e modificá-lo num campo de texto. Isso constitui ao que vamos nos referir como uma *relação*. Existem muitos tipos de *relações* desejáveis num formulário de entrada de dados, mas elas se restringem, essencialmente, às seguintes:

- O usuário quer selecionar um *item* numa *lista* e modificá-lo em um ou mais campos de texto;
- o usuário quer *selecionar itens* numa *lista* e deseja *modificá-los na própria lista*;
- o usuário quer *selecionar itens* numa lista de tal maneira que, *numa outra lista*, sejam mostrados *itens relacionados à sua seleção* seguindo um *critério específico*.

Embora exista um sentido intuitivo nas palavras *lista, item, campo de texto, critério específico*, etc, é necessário definir exatamente o que são esses elementos, onde eles estão localizados no nosso modelo e quais são os seus aspectos funcionais fundamentais. É necessário ainda definir uma maneira de acomodar, no modelo, as várias modalidades de relações descritas acima (e algumas outras que serão discutidas mais à frente).

Listas, itens e *widgets*

Naturalmente, essas *listas* às quais fazemos referência devem conter informações sob a forma de *itens*. *Itens* são, essencialmente, informações que obedecem a uma estrutura específica, podendo conter múltiplos segmentos de dados a serem interpretados cada qual segundo uma semântica inerente ao próprio *item*. Isso quer dizer que os segmentos de dados que compõem os itens possuem, portanto, tipos.

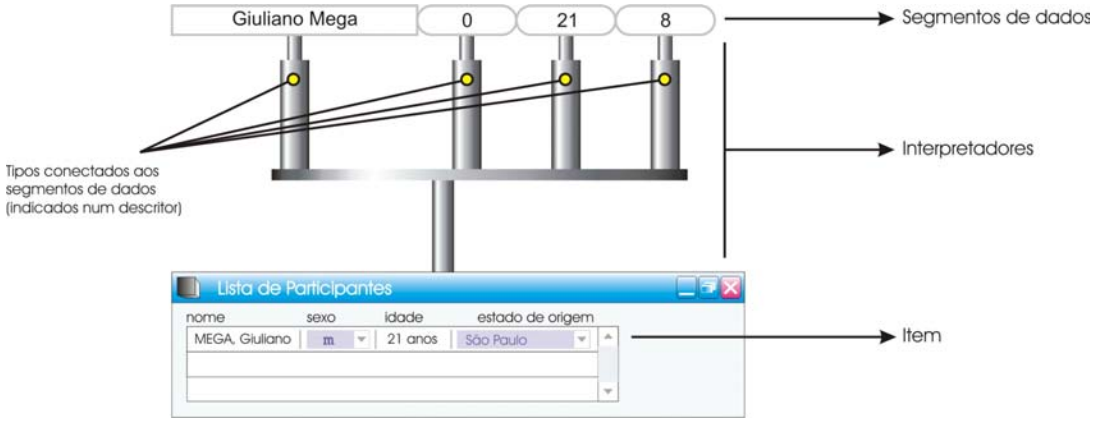

fig 1. Complexo de tradução

A partir do raciocínio desenvolvido concluímos, portanto, que um item é a composição de um conjunto de segmentos de dados e uma semântica associada, dada pelo tipo dos segmentos que o compõem. Chamaremos de tipo também ao conjunto de tipos dos segmentos de dados que compõem um item. A partir deste instante, iremos nos referir a esses segmentos de dados que compõem um item como sendo *campos*.

Listas são representações gráficas de um grupo de tamanho arbitrário de itens. Uma restrição fundamental que impusemos às listas é que elas comportam itens de um só tipo. Isso porque não é prática comum mesclar tipos diferentes em uma só lista (dada a própria estrutura de armazenagem que em geral fornece os dados para a lista) e, em função do aumento de complexidade que a ausência de tal restrição traria ao próprio modelo, optamos em favor de permitir um só tipo por lista.

Ainda não foi definido por qual maneira são gerados os dados que compõem um item. Certamente tais dados encontram-se armazenados em alguma base de dados, o que varia é o método de acesso à base de dados. Inicialmente, concebemos dois métodos como sendo os principais métodos de acesso nos grandes sistemas que requerem nossos formulários:

1. A uma base de dados que implemente SQL, via JDBC;
2. a uma base de dados arbitrária via RMI (inclusive EJB).

Inicialmente, nesta versão do software, vamos abordar somente a questão do acesso a bases de dados que implementam a linguagem de consultas SQL. Algumas idéias acerca da problemática envolvida no mapeamento de expressões de restrição em métodos de objetos serão discutidas, todavia.

Existe, certamente, uma identificação premeditada entre a uniformidade restrita e imposta nas listas (itens de um só tipo) e a uniformidade dos dados resultantes de uma consulta SQL. O elo está no operador de projeção π, que é o mesmo – isso significa que é possível "projetar" a consulta numa lista, a única questão que permanece aberta é o mapeamento (interpretação) dos valores obtidos na consulta nos *widgets*[1] correspondentes numa lista.

Cabe, neste instante, definir alguns aspectos com relação à composição dos formulários. Os formulários são compostos por duas classes básicas de widgets:

***Widgets* simples**: são aqueles capazes de exibir, num dado instante, apenas o conteúdo de um campo de um item. *Widgets* simples compreendem as *combo boxes*, *text*

[1] Referimo-nos a *widgets* como sendo quaisquer representações primitivas *Swing* na camada de apresentação, tais como *combo boxes*, tabelas, *radio buttons* e *text fields*.

fields, *check boxes* e todo e qualquer *Widget* para o qual for possível estabelecer um mecanismo de tradução direto a partir de um campo. Esse mecanismo de tradução é padronizado e definido para tipos de segmentos de dados (campos).

***Widgets* compostos**: são capazes de exibir múltiplos campos de múltiplos itens. São formados a partir da composição de vários *widgets* simples. São representados por tabelas, portando uma estrutura fixa em suas linhas (composição uniforme de widgets). Deve ser claro que as linhas provenientes de uma consulta podem ser diretamente mapeadas em *widgets* compostos.

fig. 2 – Mapeamento num *widget* composto

Este modelo de *Widgets*, embora possa parecer restritivo de início, é suficiente para atender às necessidades dos casos de uso.

Vamos introduzir agora as entidades que modelam e restringem o acoplamento entre os elementos já apresentados (alguns de maneira precisa, outros nem tanto). O foco passa a ser particularmente, a partir deste instante, a relação *model-view* e os mecanismos de sincronização; isto é, a relação entre os *widgets* (simples e compostos) e a abstração das projeções da base de dados.

Relações, schemas e domains

Naturalmente deve existir uma abstração que represente o conjunto total dos dados que podem ser manipulados (traduzidos, exibidos e modificados). Para construir essa abstração, optamos por um objeto que representasse uma visão particular de uma base de dados, podendo originar-se numa consulta ou chamada RMI. A esse objeto demos o nome de **schema**.

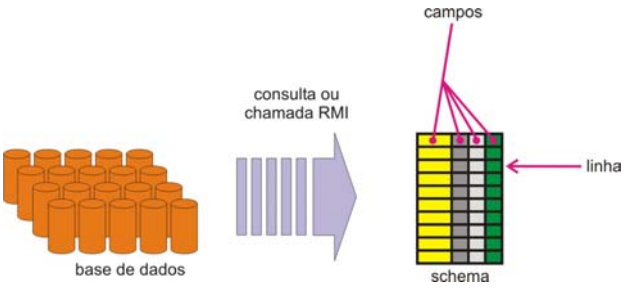

fig. 3 – Gerando schemas

Um schema permite, essencialmente, manipular uma coleção de linhas. É possível iterar através dessa coleção, adicionar filtros de restrição e modificar campos em linhas individuais. É o **schema** quem determina o **tipo** de seus campos. O **schema** é bastante próximo conceitualmente dos **extents** JDO[2], exceto pelo fato de que **schemas** não são intrusivos; isto é, não requerem nada de muito específico da base de dados além de uma maneira de diferenciar as linhas umas das outras.

Resta saber agora como é feita a ligação entre **schemas** e *widgets.* Os *widgets* até poderiam, a princípio, saber como acessar um **schema.** Um problema fundamental, todavia, diz respeito a como manter um controle de quais linhas do **schema** estão em uso em cada *widget.* Resolver esse problema é fundamental para podermos modelar o seguinte caso de uso:

Suponha que tenhamos num *form* hipotético três listas – uma representando clientes, uma representando datas e uma representando pedidos. Gostaríamos de que, toda vez que fossem selecionados um conjunto de clientes e um conjunto de datas, fossem mostrados na lista de pedidos apenas os pedidos feitos pelos clientes selecionados nas datas selecionadas.

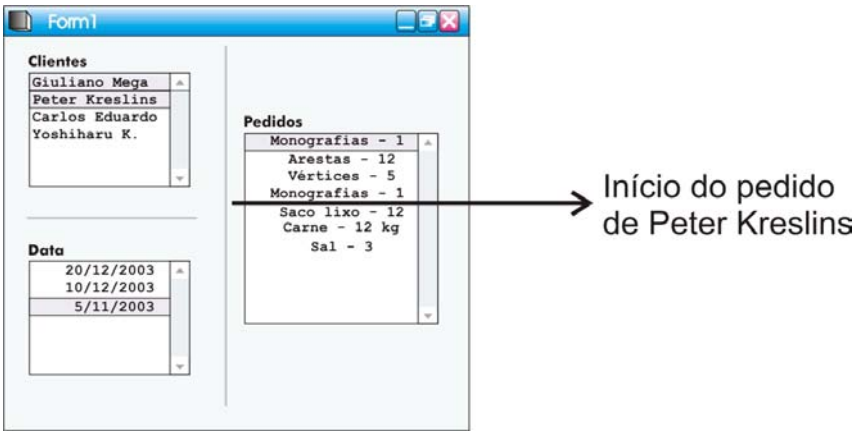

fig. 4 – Um dos exemplos mais problemáticos

Vejamos então o que é que está implícito nesse exemplo. É necessário um mecanismo que, partindo do resultado da seleção em um ou mais *widgets* compostos seja capaz de, por algum critério, filtrar na lista **Pedidos** apenas aqueles pedidos que satisfaçam às restrições requeridas (pedidos cujas datas correspondam àquelas selecionadas na lista **Data** e tenham sido efetuados pelos clientes selecionados na lista **Clientes**).

Cabe aqui fazer um parêntese com relação à problemática na implementação desse mecanismo. Antes de discutir o modelo, é interessante pensar como poderíamos traduzir essas restrições numa estrutura de consulta. No caso do **schema** ser utilizado para acesso a uma base de dados que forneça **SQL**, seria tentador traduzir essa seleção como uma simples *string* de consulta:

```
SELECT produto, quantia
FROM pedido
WHERE ((cliente = "Giuliano Mega) OR (cliente = "Peter Kreslins")) AND
((data = "5/11/2003"))
```

[2] Java Data Objects - framework de persistência para modelos de objetos Java. Para mais informações visite http://www.jdocentral.com

Note, todavia, que o **SQL** permite, no máximo, a concatenação de 32 operadores de restrição (de acordo com a especificação, alguns bancos permitem mais, ou menos). Isso seria bastante desastroso para a nossa implementação, uma vez que esse número pode ser, num exemplo como esse, superado com facilidade. Torna-se necessário portanto fracionar a expressão de consulta, executando sua composição na camada do **schema.** A idéia, grosso modo, é a seguinte:

Se o usuário seleciona as linhas c_1, c_2 e c_3 na **lista 1** e as linhas l_1, l_2 e l_3 na **lista 2,** temos:

$$(c_1 \vee c_2 \vee c_3) \wedge (l_1 \vee l_2 \vee l_3) =$$
$$= (c_1 \wedge l_1) \vee (c_1 \wedge l_2) \vee (c_1 \wedge l_3) \vee (c_2 \wedge l_1) \vee (c_2 \wedge l_2) \vee (c_2 \wedge l_3) \vee (c_3 \wedge l_1) \vee (c_3 \wedge l_2) \vee (c_3 \wedge l_3)$$

As expressões do tipo $(c_i \wedge l_j)$ podem ser traduzidas em uma consulta **SQL** do tipo

```
SELECT ...
FROM ...
WHERE (c = c_i) AND (l = l_j)
```

E os operadores **OR** podem ser obtidos adicionando-se todas as linhas provenientes das consultas ao **schema** que aceita os filtros. A tradução dessas expressões para chamadas RMI é também razoavelmente direta, mas requer a especificação de informações extras que, no caso do SQL, já estão subentendidas.

Quando o usuário desceleciona uma linha em alguma tabela, é necessário determinar quais subexpressões da forma $(c_i \wedge l_j)$ fazem referências às linhas descelecionadas e removê-las da string de expressões. Inúmeras otimizações podem ser feitas no sentido de reduzir a quantidade de vezes que as consultas têm de ser refeitas, temos a forte impressão de que é possível implementar esse mecanismo de forma incremental.

Voltando agora à questão do associar de **schemas** e *widgets*, vemos que é necessário, ainda, manter um controle de quais linhas estão selecionadas em cada *widget,* de tal maneira que alterações nesses **conjuntos de linhas selecionadas** sejam propagadas aos interessados e convertidas em filtros para o **schema** que serve como base para a lista **Pedidos**, por exemplo.

Uma outra questão diz respeito ao seguinte exemplo:

fig. 4 – Elo entre múltiplos widgets

Embora essa relação possa parecer, num primeiro momento, mais complexa do que a discutida anteriormente, note que o único problema aqui diz respeito a mapear uma **linha selecionada** em uma lista num conjunto de **widgets simples**. Isso pode ser feito de maneira bastante direta, desde que se especifique em qual **widget simples** devemos mapear cada um dos campos da linha que corresponde ao item selecionado (que possui uma relação de 1 para 1 com linhas no *schema*).

Note, todavia, que pode surgir uma ambigüidade nesse mapeamento quando houver mais de uma **linha selecionada**. Devemos escolher, dentre as linhas selecionadas, uma **linha atual**. Esse conceito está presente no nosso modelo, onde a linha atual possui a propriedade adicional de que é a única que pode ser modificada.

Na explicação dada até agora, deve ser possível notar um buraco a ser preenchido por uma entidade cuja responsabilidade é manter essas informações relativas à linha selecionada e linha atual. Além disso, essa entidade deve ser responsável por propagar informações relativas a alterações nesses conjuntos de linhas para outras entidades, de tal maneira que as relações que descrevemos acima possam ser concretizadas.

A essa entidade que faz a ponte entre **schema** e **widget** demos o nome de **domain**. Um **domain**, basicamente, cuida de agregar todos os **widgets** que fazem referência a uma dada visão da base de dados (**schema**) e que fazem acesso somente às mesmas linhas; isto é, compartilham um conjunto de linhas atuais e selecionadas. Isso, na prática, significa que não pode haver duas listas no mesmo domain. Embora essa restrição possa parecer um tanto forte, note que a expressividade do modelo não é prejudicada.

O **domain,** portanto, agrega um **schema** (visão da base de dados), zero ou um **widgets compostos** e um número arbitrário de **widgets** simples. As relações **widget-schema** são exemplificadas nessa figura tomada emprestada do cartaz:

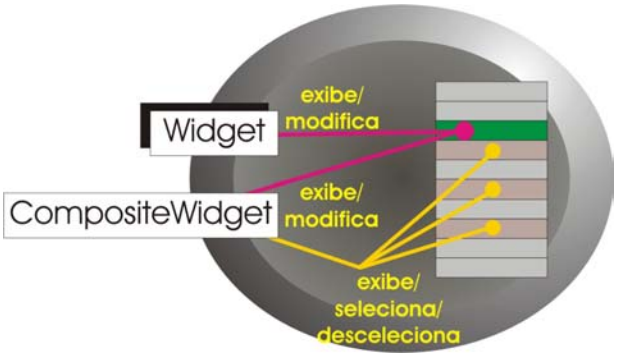

fig. 5 – domain

A estrutura começa agora a ganhar uma certa coesão - é possível identificar quais participantes são responsáveis por quais funções. Particularmente, já é possível observar que os eventos gerados por modificações nos conjuntos de linhas selecionadas devem ser propagados pelos **domains** responsáveis pelos *widgets* onde ocorreram as alterações. É possível notar, também, que esses eventos irão interessar a outros **domains** que deverão, de alguma forma, modificar o **schema** a ele associado. Os **domains** são dotados, portanto, de canais de entrada (*input connectors*) e de canais de saída (*output connectors*). Um canal de saída pode levar eventos a múltiplos canais de entrada, mas um canal de entrada só pode receber eventos de um único canal de saída (por motivos óbvios).

Segue abaixo o diagrama que mostra, em sua totalidade, a natureza das relações entre as diversas entidades do modelo:

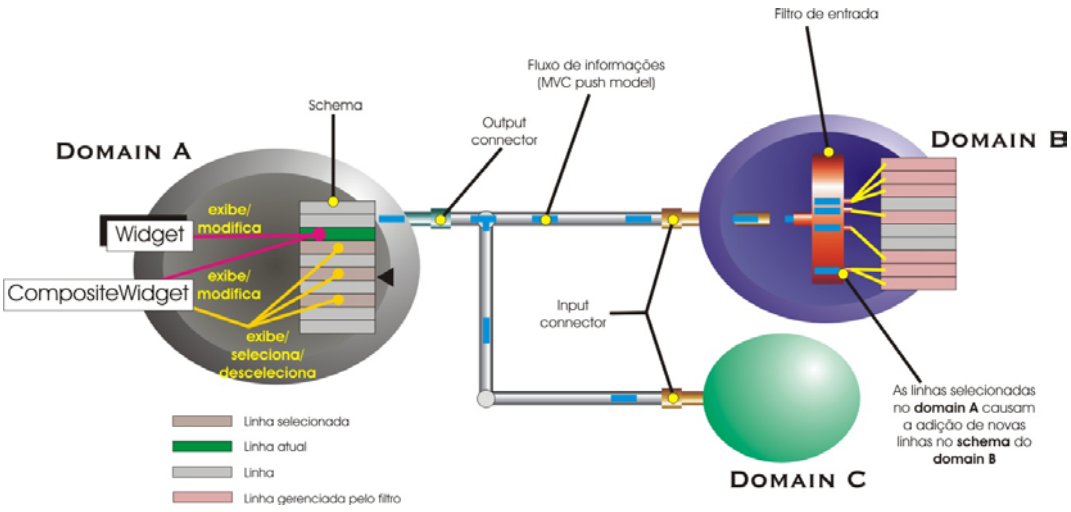

fig 6. – O modelo (quase) completo

Falta agora definir a maneira pela qual os **domains** modificam o **schema** associado. Deve ser claro, neste ponto do texto, que esse mecanismo de modificação envolve os já mencionados **filtros de restrição**, que podem ser associados e dissociados dos **schemas.**

FILTERS (FILTROS DE RESTRIÇÃO)

Como foi dito anteriormente, deve haver uma maneira de se definir os filtros de restrição que alteram o conteúdo de um *schema*. Esses filtros são construídos através de expressões envolvendo *fields* dos *domains* origem e destino.

Um filtro sempre está associado a um conector de entrada que será usado para processar as informações provenientes de outros domain e alterar o conteúdo do schema.

Assim, tomemos como exemplo um domain chamado Clientes e um domain chamado Pedidos. O domain Clientes possui o conjunto de fields {codigo, nome, endereço}. Já o domain Pedidos possui o conjunto {cod_pedido, data, codigo_do_cliente, codigo_do_produto}. Uma expressão de filtro que poderia ser colocada num conector de entrada do domain Pedidos seria a seguinte:

Exemplo 1: `|codigo| = codigo_do_cliente`

O termo entre | (barras) é o *field* que vem do *domain* de origem (neste caso o *domain Clientes*) e o outro termo vem obviamente do *domain Pedidos*. Note que não é possível comparar dois *fields* de um mesmo *domain*.

Os operadores aceitos nesta expressão são os operadores de comparação usuais, como =, >, <, >=, <=, !=. É possível, também, concatenar diferentes expressões contendo comparações entre fields através dos operadores lógicos **AND** e **OR**. Assim, um outro exemplo de expressão, supondo que existissem os *fields* c1 em *Clientes* e p1 em *Pedidos*, seria o seguinte:

```
|codigo| = codigo_do_cliente AND |c1| > p1
```

Entretanto, percebemos que este caso descrito anteriormente não é tão usual quanto à possibilidade de se misturar diferentes *domains* de origem e gerar uma expressão para o *domain* de destino que queremos moldar. Assim, suponha o caso em que tenhamos um terceiro *domain* chamado **Produtos**, com conjunto de *fields* {codigo, nome}. Um relacionamento natural seria filtrar o *domain Pedidos* com um código de cliente e um código de produto. No entanto, um conector de entrada do *domain Pedidos* só pode estar relacionado a um único *domain*. Dessa forma, precisaríamos de um novo conector que ligasse Pedidos a Produtos.

Tendo este conector, bastaria usar a expressão:

Exemplo 2: `|codigo| = codigo_do_produto`

Conseqüentemente teríamos duas expressões envolvendo conectores diferentes que precisam trabalhar em conjunto quando aplicadas sobre o *domain Pedidos*. Para isso, definimos a concatenação de conectores de entrada que é feita através do conceito de **composite filters** (filtros compostos). Os *composite filters* aceitam tanto o operador lógico AND como o OR. Assim, suponha que o exemplo 1 se referisse ao conector de entrada chamado input1 e o exemplo 2 ao conector de entrada, input2. Então teríamos a expressão do *composite filter* como:

```
input1 AND input2
```

Vale notar que na ausência de dados no conector de entrada *input1*, a expressão seria avaliada como:

```
input2
```

E assim, a expressão para *input1* seria ignorada por questões óbvias.

Concluindo, percebemos que os filtros têm papel fundamental na arquitetura, provendo um mecanismo de processamento da comunicação de dados entre os diversos *domains.*

TRANSLATORS

Os tradutores foram desenvolvidos para dois fins específicos:

- prover uma lista de valores que serão associados a algum widget, de modo que o usuário só possa escolher um dos itens dessa lista (ex.: Sim, Não, Talvez)
- prover uma maneira padrão de traduzir um identificador em um valor (ex.: código de produto 00001 no nome de produto Sabonete)

Com esse mecanismo, um certo tipo de relacionamento, muito comum, é facilmente mapeado para o modelo do JAdaptiveForms. Referimo-nos ao relacionamento que ocorre quando um determinado campo de uma tabela é referenciado por um código, cuja associação encontra-se em outra tabela. Um exemplo disso é o código para clientes do sexo masculino e o código para clientes do sexo feminino (como 0 para masculino e 1 para feminino). Esse tipo de relacionamento torna o uso do formulário pouco amigável se obrigarmos o usuário a conhecer tais códigos. Assim, o tradutor pode ser muito útil, pois além de mapear os identificadores de forma transparente, permite que o usuário escolha uma das opções disponíveis em uma lista geralmente exibida através de um ComboBox.

O tradutor pode ou não estar associado a um schema. Quando está associado, ele pode mapear um conjunto de dados deste schema e devolver os valores traduzidos. Assim, se

tivermos uma tabela de cidades em nosso banco de dados, e esta tabela conter um identificador e o nome da cidade, então uma tradução usual seria converter um código no seu nome. Por outro lado, podemos ter tradutores estáticos que não usam um schema e contêm um mapeamento fixo, como é o caso do Sexo. Não há porque colocar os possíveis valores para sexo no banco de dados, pois sabemos que só existem dois tipos: Masculino e Feminino. Dessa forma, podemos entender o tradutor como um HashMap da linguagem Java.

Os tradutores têm um conector de entrada que aceita somente um tipo. Assim, no caso de um tradutor para sexo, o conector aceita somente o identificador do sexo a ser traduzido. Por outro lado, um tradutor pode ter um ou mais conectores de saída. Cabe ao widget que queira usar tal tradutor, escolher de qual conector ele deseja obter um valor. Como um exemplo, poderíamos ter um tradutor que recebe um identificador de cliente e devolver seu nome ou seu sobrenome. O widget teria que especificar qual dos conectores de saída deseja usar.

O conceito de tradutores é usualmente conhecido como list-of-values na literatura. Ele é, como já foi dito, muito próximo de um hashmap da linguagem Java, no entanto possuindo a característica de distribuir listas de possíveis valores para um determinado widget.

FORMULÁRIOS

Formulários ou forms, como são conhecidos na nomenclatura do JAdaptiveForms, são containeres de domains. Na verdade eles podem conter não só domains, mas especificações de tradutores e definições de widgets.

Podemos entender um form como uma janela de exibição e entrada dos dados para o usuário. Cada form vem associado a um template que será explicado mais adiante. Dessa forma, é natural entender que um form deve ser composto por pelo menos um domain, que será responsável pela obtenção e exibição dos dados. Como já abordamos anteriormente, um domain pode se comunicar com outro domain para troca de dados e alteração do schema associado. Isto pode ser útil para criar formulários ricos que apresentam, por exemplo, listas de valores de uma tabela que podem ser selecionados, sofrer um determinado processamento e o resultado ser exibido em uma outra lista.

Em geral, um formulário será exibido como uma janela independente do sistema e terá alguns botões de gravação ou atualização dos dados.

Como veremos mais adiante, um formulário é a base de definição do arquivo de configurações do JAdaptiveForms.

TEMPLATES

A última abstração do JAdaptiveForms a ser abordada neste documento é o conceito de template.

Para definir um formulário, precisamos nos preocupar, além da comunicação e da definição dos widgets, de uma forma de dispor todos esses elementos gráficos de uma forma amigável para o usuário. Neste momento entra o template, que funciona como um roteiro padronizado para que os widgets apareçam no formulário em uma certa ordem ou disposição.

Um template simples poderia ser considerado como aquele que permite ao desenvolvedor do formulário especificar as posições absolutas de cada um dos widgets. Assim,

para cada widget definido na especificação do formulário, o desenvolvedor precisaria explicitar a posição X e Y em coordenadas relativas à janela em que o formulário de encontra.

A realidade é que o conceito de template não teve um amadurecimento suficiente e é o único ponto fraco do modelo. Fraco no sentido de não ter sido explorado para que pudesse ser incluído na arquitetura. No entanto, temos uma noção básica dos serviços que queremos obter dele:

- posicionamento automático dos widgets de acordo com "dicas" do desenvolvedor
- arquitetura robusta como a disponível nos layouts de Java
- navegação facilitada pelo layout automático (a tecla enter salta para o próximo widget, etc)
- itens secundários como cor, tamanho, disposição em colunas ou linhas, etc

Com isso, temos um modelo básico do que é um template, no entanto precisamos de uma maneira inteligente de adequá-lo à arquitetura já consolidada do JAdaptiveForms.

Exemplo de arquivo de configuração (XML)

Nesta seção apresentaremos um exemplo prático do funcionamento do JAdaptiveForms, através de um arquivo de configurações de um formulário.

Abaixo, apresentamos a transcrição do arquivo em formato XML contendo as linhas enumeradas para facilitar a visualização na descrição do exemplo.

```
01 <form name="formClientes" template="template1">
02   <domain name="Clientes" schema="Clientes">
03     <widgets>
04       <widget widget-ref="ListaClientes">
05         <subwidget widget-ref="NomeDoCliente" uses="nome_cliente"/>
06         <subwidget widget-ref="TelefoneRes" uses="tel_res"/>
07       </widget>
08     </widgets>
09     <connectors>
10       <output-definition>
11         <output name="outputCodigo">
12           <fields>
13             <field name="cod_cliente"/>
14           </fields>
15           <connects-to>
16             <connect-to domain="Pedidos"/>
17           </connects-to>
18         </output>
19       </output-definition>
20     </connectors>
21   </domain>
22   <domain name="Pedidos" schema="Pedidos">
23     <widgets>
24       <widget widget-ref="ListaPedidos">
25         <subwidget widget-ref="DataPedido" uses="data_pedido"/>
26         <subwidget widget-ref="ValorPedido" uses="valor_pedido"/>
           <subwidget widget-ref="TipoPedido"
27                    translation="TransTipoPedido"
                     translation-field="outputTipo"
                     list-of-values="true" uses="cod_tipo_pedido"/>
28       </widget>
29     </widgets>
```

```
30      <connectors>
31        <input-definition>
32          <input name="inputCodigoCliente" field="codigo">
33            <filter expression="|cod_cliente| = codigo"/>
34          </input>
35          <input name="inputCodProduto" field="cod_produto">
36            <filter expression="|cod_prod| = cod_produto"/>
37          </input>
38          <composite-filter expression="inputCodProduto AND
                              inputCodigoCliente"/>
39        </input-definition>
40      </connectors>
41    </domain>
42    <translator name="TransTipoPedido" schema="TipoPedidos">
43      <input name="inputCodTipo" field="cod_tipo_pedido"/>
44      <output name="outputTipo" field="nome_tipo"/>
45    </translator>
46    <translator name="TranslatorSexo">
47      <list-of-values>
48        <item caption="Masculino" id="#MASC"/>
49        <item caption="Feminino" id="#FEM"/>
50      </list-of-values>
51    </translator>
52    <widget-definition>
53      <widget name="NomeDoCliente" type="TextBox"
                caption="Nome do Cliente">
54        <property name="read-only" value="false"/>
55      </widget>
56      <widget name="EnderecoCliente" type="TextBox"
                caption="Endereço do Cliente">
57        <property name="read-only" value="false"/>
58      </widget>
59      <widget name="ListaClientes" type="Table"
                caption="Lista de Clientes"/>
60        <property name="multi-select" value="true"/>
61      </widget>
62      <widget name="TelefoneRes" type="TextBox"
                caption="Telefone Residencial"/>
63        <property name="mask" value="(0xx99) 9999-9999"/>
64        <property name="readonly" value="false"/>
65        <property name="preserve-mask" value="true"/>
66      </widget>
67    </widget-definition>
68  </form>
```

Inicialmente, definimos um form, que como já foi explicado anteriormente, representa um conjunto de domains a serem exibidos em uma janela. Isto acontece na linha 01, onde definimos um form de nome formClientes. Um formulário pode trazer consigo uma definição de template, que nada mais é do que um modelo de posicionamento e disposição dos widgets.

Na linha 02, apresentamos a definição de um domain. Como todo objeto do JAdaptiveForms, o domain também recebe um nome, neste caso Clientes.

Um domain pode vir associado a um schema, onde o primeiro provê a comunicação e o segundo define a camada mais baixa na arquitetura, estando ligado diretamente à fonte de dados. Na verdade, o domain faz a comunicação entre os widgets e o schema.

Um domain é definido como contendo os seguintes itens que não são necessariamente obrigatórios: um conjunto de widgets, uma especificação de conectores de entrada e uma especificação de conectores de saída. No exemplo dado, podemos observar nas linhas 10-19, uma especificação de conectores de saída. É interessante notar que há uma definição de conector de saída que exporta a informação de código do cliente para o domain Pedidos. Assim, sempre que uma linha da lista de clientes é selecionada, um evento é desencadeado neste domain, que, como veremos mais adiante, define um conector de entrada para tratar deste evento.

Um conector de saída é constituído por um nome, pelos campos do schema que ele exporta e pelos domains a que é conectado. Note que ele pode ser conectado a mais de um domain e assim, um evento e despachado para todos os seus interessados.

Para finalizar o domain Clientes, resta definir a configuração dos widgets (linhas 03-08). Há uma restrição já explicada anteriormente que impõe a utilização de um único widget de lista por domain. Assim, se o usuário optou por ter um widget de lista no domain, não poderá especificar outros widgets no mesmo nível. Obviamente o widget lista conterá subwidgets convencionais, como TextBoxes, ComboBoxes, etc. No exemplo podemos ver que o widget de lista ListaClientes contém subwidgets NomeDoCliente e TelefoneRes.

Cada widget simples (não sendo um widget lista) é associado a um field do schema (cláusula uses). Essa é a forma do JAdaptiveForms saber qual informação deve ser colocada dentro do widget. Ao longo dessa descrição perceberemos que o widget pode também estar associado a um translator.

Outra característica importante de um widget qualquer é o fato de que ele está associado a uma definição de widget que contém suas propriedades gráficas (cláusula widget-ref). Essas propriedades são definidas na seção widget-definition (linhas 52-67).

Agora definimos outro domain chamado Pedidos, para que possamos exemplificar a idéia de relacionamento com o domain Clientes. A definição de widgets ocorre de maneira similar, agora com o widget de lista ListaPedidos. No entanto, vale notar que o widget TipoPedido trabalha com tradução. Este conceito, já explorado na seção “Translators” define uma maneira de se obter um valor ou conjunto de valores representativos a partir de um identificador (ex.: cidade de São Paulo com identificador 1). No nosso exemplo, o widget TipoPedido utiliza tradução no field cod_tipo_pedido, recebendo como resultado a lista de todos os tipos de pedidos existentes no banco e o tipo selecionado de acordo com o identificador passado para o tradutor. Assim, se o identificador for 2 e tivermos os tipos de pedidos {“Venda”, “Compra”, “Troca”} com identificadores {1,2,3} respectivamente, então o widget TipoPedido terá esses nomes incluídos na lista e o nome Compra selecionado por padrão.

A definição de conectores de entrada é dada nas linhas 31-39. Ali podemos ver que existem dois conectores: inputCodigoCliente e inputCodigoProduto. Pelo primeiro, entram os dados vindos do domain Clientes. Assim, quando um cliente é selecionado, este input é alimentado e a especificação de filtro entra em ação. O filtro define uma expressão em que um campo do domain de origem dos dados (no caso Clientes) é comparado com um campo do domain de destino (no caso Pedidos):

```
|cod_cliente| = código
```

Onde o field entre as barras vem do domain de origem. O outro conector seria utilizado para suportar a filtragem dos pedidos pelo produto associado. Como temos dois conectores de entrada, precisamos de uma especificação para a combinação entre os dois. Para isto existe a linha 38 onde há a definição de um filtro composto (cláusula composite-filter). Assim, podemos observar que há um operador lógico E ligando os dois conectores de entrada. Portanto se há informação de cliente entrando em inputCodigoCliente e há informação de produto entrando em inputCodigoProduto, então o filtro composto selecionará somente as linhas do schema que

contenham ao mesmo tempo o código do cliente e o código do produto selecionados. Há um caso especial que deve lidar com a ausência de informação em um ou ambos os conectores. Neste caso, o filtro composto ignora parte da expressão que não pode ser completada.

O próximo passo é definir os tradutores eventualmente usados na definição de um formulário. O tradutor possui duas responsabilidades bem definidas: uma é prover uma lista de possíveis valores para um widget e a outra é converter um identificador em um valor, como código em nome de produto, por exemplo. Como um novo exemplo, podemos observar o tradutor de sexo (TranslatorSexo). Ele define um mapa de identificadores em nomes, contendo #MASC sendo traduzido em “Masculino” e #FEM em “Feminino”. De uma forma geral, toda informação de relacionamento vinda da base de dados se dá por identificadores e assim, poderíamos ter uma linha contendo {“Peter”, #MASC}. Para que o usuário do formulário não se sinta desconfortável ao olhar para este código nada intuitivo, o tradutor entra em ação. Mas além disso, ele disponibiliza a lista de possíveis valores para Sexo. Assim o usuário simplesmente seleciona um dos possíveis valores dessa lista quando precisar escolher um sexo para uma pessoa.

Agora chegamos ao fim do arquivo de configurações. Nas linhas 52-67 encontramos as definições dos widgets dadas através de propriedades. Para cada widget referenciado nas definições de um domain, devemos ter um bloco com as suas propriedades gráficas. Assim, como um exemplo, podemos tomar o widget NomeDoCliente, que é um TextBox e tem a propriedade read-only definida como falso. Este widget terá sua capacidade de edição dos dados habilitada. Além disso, todo widget pode ter um caption, que nada mais é do que um texto descritivo que é disposto numa área próxima ao widget, dependendo do template utilizado.

JAdaptiveForms: onde se encontra?

Hoje o projeto se encontra em fase de desenvolvimento, mas com o modelo totalmente definido. Como já dissemos anteriormente, os templates ainda requerem um certo esforço para que possam se adequar à arquitetura como um todo.

Iniciamos o desenvolvimento e temos alguns arquivos disponíveis no CVS do projeto no site do SourceForge: **www.sf.net/projects/jadaptiveforms/**

JAdaptiveForms: para onde desej amos ir?

Como é um projeto totalmente voltado para a comunidade, acreditamos que ele possa ser de grande valia. Assim, focaremos nossos esforços na conclusão da implementação do software e acreditamos que em breve teremos uma versão funcional.

Outro ponto interessante é a possibilidade de desenvolvimento de um plug-in para o ambiente de desenvolvimento Eclipse. Nossa proposta inicial contava com esta possibilidade, mas o planejamento inadequado impediu que pudéssemos chegar a tal nível.