UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

ACELINO GEHLEN DA SILVA

Utilização de *callback* no aprimoramento da interface com o usuário

Trabalho de Conclusão de Curso apresentado como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação

Prof. Dr. Marcelo Soares Pimenta
Orientador

Porto Alegre, novembro de 2010.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitora de Graduação: Profa. Valquiria Link Bassani
Diretor do Instituto de Informática: Prof. Flávio Rech Wagner
Coordenador do CIC: Prof. João César Netto
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

AGRADECIMENTOS

A minha noiva, eterna namorada, eterna amiga, eterna companheira, que nos momentos mais delicados teve paciência para aturar minha falta de tempo, sabedoria para me aconselhar, e com certeza, muito amor para poder abdicar de tantos momentos em que precisei dedicar mais atenção aos estudos;

A minha mãe, que seria nesse momento a mãe mais orgulhosa do mundo e poderia dizer sem falsa modéstia que foi graças aos seus conselhos e seu apoio que cheguei até aqui;

A minha família, que é a base para tudo. Sem ela, nada disso seria possível;

Aos meus amigos, que sempre me incentivaram nessa jornada;

Ao meu orientador, pela qualidade dos ensinamentos durante as disciplinas em que interagimos, pela confiança depositada desde o primeiro momento em que conversamos sobre este trabalho, e pelas valiosas contribuições para que este trabalho fosse realizado;

Aos professores do Instituto de Informática e demais Institutos, que ministraram as disciplinas com grande dedicação;

A todas as pessoas que ao longo dos anos apoiaram todos os momentos que fizeram parte desse curso;

À Universidade Federal do Rio Grande do Sul, por continuar sempre oferecendo um ensino público, gratuito, e de excelente qualidade.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

UFRGS	Universidade Federal do Rio Grande do Sul
IDL	Interface Description Language
COM	Component Object Model
COM+	Component Object Model Plus
MTS	Microsoft Transaction Server
IHC	Interação humano-computador
GoF	Gang of Four
SGBD	Sistema gerenciador de bancos de dados
HTTP	HyperText Transfer Protocol
HTML	HyperText Markup Language
Febrl	Freely Extensible Biomedical Record Linkage
CORBA	Common Object Request Broker Architecture
SOAP	Simple Object Access Protocol
XML	eXtensible Markup Language
PDA	Personal Digital Assistant

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

O objetivo do presente trabalho é encontrar uma forma de prover real *feedback* aos usuários de sistemas interativos, em especial, no tocante a sistemas que realizam a carga de grandes volumes de dados demandando muito tempo de processamento e recursos de máquina para serem concluídos, aprimorando, desse modo, a interação humano-computador.

Além de melhorar a qualidade do *feedback* para o usuário, boas práticas recomendadas pela Engenharia de Software serão utilizadas no sentido de oferecer alta manutenibilidade para o sistema, garantir baixo acoplamento entre as diversas camadas de *software* e, ainda, como bom efeito colateral, diminuir os requisitos de máquina necessários para a execução das tarefas.

O padrão *Observer* (GAMMA, 1995), um dos padrões GoF, juntamente com a utilização do modelo de componentes distribuídos COM+, atendem justamente ao tipo de resposta que pretendemos para o usuário e, também, às recomendações de Engenharia de Software já citadas. Através de sua utilização podemos manter um conjunto de aplicações cliente informados sobre o andamento das tarefas ao mesmo tempo em que preservamos o baixo acoplamento e mantemos o nível de modularização bastante adequado no sentido de proporcionar independência entre as camadas a aumentar a manutenibilidade do sistema.

Todos os passos da solução, desde a apresentação do aplicativo inicial, com seus problemas de interação, os conceitos necessários ao entendimento da solução proposta e de que modo os *callbacks* podem nos auxiliar na busca pelo aprimoramento da interface com o usuário, estão detalhados ao longo do texto.

Palavras-Chave: *feedback*, *callback*, projeto de interface, IHC, padrões de projeto, padrão Observer.

ABSTRACT

The main goal of this study is to investigate a way to provide real feedback to users of interactive systems, in particular with regard to systems that deal with large volumes of data requiring lots of processing time and machine resources to its accomplishment, improving the human-computer interaction.

Besides improving the quality of feedback to the user, good practice recommended in software engineering will be used in order to provide high maintainability to the system, ensure low coupling between the various software layers, and also, as a nice side effect, reduce the computer requirements needed to perform the tasks.

The Observer pattern (Gamma, 1995), one of the GoF patterns, combined with the use of distributed component model COM+, matches exactly the kind of response we want for the end-user and also the recommendations of Software Engineering mentioned above. Through its use we can maintain a suite of client applications informed about the progress of tasks, preserve the low coupling and maintain the level of modularization quite adequate in providing independence between layers to increase the maintainability of the system.

All steps for the solution, since the initial application, with its interaction problems, the concepts needed to understand the proposed solution and how the callbacks can assist us in the quest for improving the user interface, are detailed along the text.

Keywords: feedback, callback, interface design, HCI, design patterns, Observer pattern.

1 INTRODUÇÃO

Com a grande diversidade de plataformas de desenvolvimento e a integração constante dos sistemas de computação, as tecnologias multicamadas têm sido cada vez mais utilizadas de maneira a propiciar essa integração. Surge, nesse contexto, um novo tipo de aplicação cliente, comumente denominada de *Thin Client*, que se encarrega basicamente da camada de apresentação, em oposição ao modelo *Rich Client*, em que tanto a camada de apresentação como várias regras de negócio ficavam "embutidas" em uma única aplicação.

Como resultado da aplicação de boas técnicas de engenharia de software, podemos facilmente separar da camada de apresentação as regras de negócio e o processamento pesado, permitindo, principalmente que sejam utilizadas várias tecnologias diferentes na porção cliente, tais como web, dispositivos móveis, etc., uma vez que, independentemente do cliente utilizado, as regras de negócio serão processadas em outra camada, no SGBD, ou, no nosso caso de estudo, em um servidor de aplicação.

Entretanto, deve-se tomar cuidado ao utilizar este tipo de abordagem, no que diz respeito à interação com o usuário. Aplicações multicamadas são, em geral, *stateless*, e isso tem lá suas vantagens, mas ao mesmo tempo, pode se tornar um problema se a porção cliente solicitar algum processamento que demande muito tempo. Isso pode ferir gravemente a usabilidade do sistema ao causar a sensação de que a aplicação "parou de responder", caso não sejam tomadas precauções no sentido prover algum tipo de *feedback*, isto é, manter o usuário informado sobre o que está acontecendo no servidor enquanto ele espera (PREECE, 2002). Não queremos, entretanto, perder a característica de *stateless*, e muito menos, aumentar o acoplamento entre a porção servidora e a porção cliente.

Nesse contexto, a utilização de funções de *callback* se torna útil para a implementação dessa interação. Através delas, podemos estabelecer um canal de comunicação entre a aplicação cliente e a aplicação servidora, no sentido do servidor para o cliente, de modo que o servidor possa informar ao cliente o andamento da tarefa que estiver sendo executada, sem que haja acoplamento, sem perder a característica de *stateless* e mantendo o processamento mais pesado no servidor, preservando o conceito de *Thin Clients*, e permitindo, ainda, que o mesmo canal seja utilizado em mais de um tipo de plataforma cliente.

No capítulo 2, introduziremos breves explicações sobre alguns fundamentos necessários para a compreensão do problema e da solução proposta. Em especial, abordaremos a maneira como a ausência de *feedback* adequado pode comprometer uma ferramenta que tenha um núcleo funcional excelente.

No capítulo 3, descreveremos brevemente uma versão inicial do sistema em que o problema ocorre, levantando os pontos em que a comunicação é falha e fazendo uma análise dos pontos positivos e negativos dessa ferramenta.

No capítulo 4 será então apresentada uma proposta de solução para o problema da interação com o usuário, as escolhas feitas no sentido de prover feedback adequado, e de que modo as alterações na implementação satisfazem aos requisitos de usabilidade pretendidos com esse estudo.

Por fim, no capítulo 5, faremos uma breve análise dos ganhos obtidos com o desenvolvimento da solução e também de como a tecnologia escolhida para a implementação atende a requisitos de modularidade e baixo acoplamento.

.

2 AMBIENTE E FERRAMENTAS , ARQUITETURAS DE SISTEMAS DE INFORMAÇÃO, FEEDBACK, PADRÕES: fundamentos para a compreensão da solução

Conforme já mencionado, nosso problema trata da manipulação de grandes volumes de dados. Essa manipulação dar-se-á pela troca informações através de arquivos. Esses arquivos são, em geral, arquivos de texto, baseados em um leiaute, que descreve como as informações estão organizadas dentro do arquivo.

Apesar de aplicativos *web* estarem ganhando cada vez mais espaço, ainda é muito comum encontrarmos implementações de sistemas de informações seguindo a arquitetura cliente-servidor, em que um aplicativo rodando em um computador pessoal assume a função de *cliente* e fica responsável por todo o processamento e também por comunicar-se com algum SGBD, que responde como *servidor*.

Nessa arquitetura, ou as regras ficam armazenadas no SGBD, ou então no aplicativo, que acaba sendo um executável único responsável por todas as camadas de aplicação: apresentação, regras de negócios, e camada de persistência. A essas camadas damos o nome de camadas lógicas (*layers*). Quando são disponibilizadas em computadores distintos, dizemos que elas estão implementadas em diferentes camadas físicas (*tiers*).

Existe hoje uma solução desenvolvida que faz uso da arquitetura cliente-servidor e aplica a modularização em *layers*. Essa solução, apesar de estar bem escrita em termos de engenharia de software, não se mostra satisfatória no momento em que não permite, de maneira simples e sem acoplamento entre as camadas, uma boa comunicação com o usuário. A comunicação se dá basicamente através de uma janela de diálogo solicitando que o usuário aguarde pela conclusão e mostrando um indicador de progresso, não necessariamente real, do andamento da tarefa. Essa solução também é limitada no que diz respeito ao fato de outro usuário não poder conectar-se em outro computador e ser notificado sobre o andamento da tarefa. Essa solução será apresentada apenas a título de ilustração, para demonstrar o ganho efetivo na implementação da solução proposta, na questão de interação com o usuário, quando comparamos as duas. Os demais benefícios obtidos são em nível mais interno e dizem respeito à questão da manutenibilidade do *software*.

Para as implementações, tanto no nível lógico quanto físico, podemos – e devemos - utilizar padrões de projeto (*design patterns*). Como já foi dito, faremos uso aqui do padrão *Observer* (GAMMA, 1995), que trata de comunicar um conjunto de observadores a um modelo observável. Esse é um padrão comportamental e será construído sobre uma plataforma COM+.

Todos os conceitos que serão abordados ao longo do texto aqui exposto serão descritos nas seções seguintes, de modo a garantir o entendimento da solução proposta

e, eventualmente, proporcionar uma fonte de informação para futuras implementações em que se opte por utilizar as mesmas tecnologias.

2.1 Ambiente e ferramentas utilizadas

O ambiente utilizado para o desenvolvimento e testes envolveu o uso de dois computadores, conforme descrito abaixo:

- Servidor: desktop com processador Intel Pentium D, de 3.00 GHz, dois núcleos operacionais, 2GB de memória RAM, rodando com Windows XP Professional, 32 bits;
- Cliente: laptop com processador Intel Pentium Core 2 Duo, de 2.40 GHz, dois núcleos operacionais, 2GB de memória RAM, rodando com Windows Vista Business, 32 bits.

Além disso, elencamos outros itens de infraestrutura relevantes para o desenvolvimento da solução proposta:

- rede disponível com capacidade de transmissão de 100 Mbps;
- SGBD SQL Server® 2005 Express Edition;
- Ambiente de programação Delphi 7 Enterprise®;
- software para geração dos conjuntos de dados de teste: Febrl 0.4.1 (Freely Extensible Biomedical Record Linkage);
- gerenciador de máquinas virtuais Oracle Virtual Box;
- Sistema operacional para uso do ambiente Febrl: Linux Ubuntu 9.10, através de máquina virtual.

2.2 Arquivos baseados em leiaute

Arquivos baseados em leiaute são simples e muito úteis para permitir a interoperabilidade entre sistemas em que um não conhece a implementação do outro. Frequentemente esses arquivos estão em formato de texto tabulado ou de valores separados por vírgula, com especificação de um leiaute para seu entendimento. Através deles, estabelece-se um protocolo que será utilizado por ambas as partes envolvidas para a troca de informações. Isso fornece um meio rápido e fácil para integração de sistemas. No entanto, esse tipo de arquivo sempre necessita de algum tipo de processamento antes que as informações nele contidas possam ser utilizadas de fato no âmbito de cada sistema.

2.3 Arquitetura cliente-servidor

No que diz respeito à arquitetura utilizada para implementação de soluções para importação de dados baseada em arquivos texto, é muito comum encontrarmos aplicativos desenvolvidos sobre plataforma *cliente-servidor*. No nosso contexto, o *cliente* visto como um aplicativo único e o *servidor* como o SGBD. Esse tipo de aplicação tornou-se bastante popular quando os sistemas começaram a migrar dos grandes computadores (*mainframes*) para computadores pessoais (PC's).

Na maioria das aplicações desse tipo, tanto a camada de apresentação como de regras de negócio e de persistência são implementadas em um único aplicativo monolítico totalmente auto contido. Isso pode facilitar um pouco o desenvolvimento, mas ao mesmo tempo traz alguns problemas bastantes sérios:

- memória: um aplicativo totalmente auto contido irá demandar muita memória para ser executado;
- processamento: no momento em que as regras de negócios são processadas na camada cliente, é necessário que haja à disposição dos usuários, computadores com alta capacidade de processamento;
- manutenção: embora seja possível separar as responsabilidades de uma aplicação em camadas lógicas, essa tarefa se torna mais difícil em um aplicativo monolítico, uma vez que, como o programador tem acesso a todas as unidades do aplicativo, pode passar despercebido um acoplamento que dificultará uma eventual expansão da solução ou a simples agregação de novos módulos.

Em ambientes corporativos, a combinação dos fatores ora enumerados pode significar um aumento de custos muito além do que seria necessário para solucionar o problema, pois acaba exigindo a existência de um parque tecnológico com grandes capacidades de processamento, bastante memória disponível, e uma equipe de desenvolvimento com conhecimento muito aprofundado sobre todas as camadas do aplicativo.

2.4 Aplicações em camadas

Falamos em camadas e as contextualizamos como método para separar as responsabilidades de uma aplicação. Quando nos referimos a camadas, é necessário esclarecer que há duas abordagens no tocante a seu significado. Uma delas é quando queremos representar camadas lógicas (*layers*) e a outra é quando queremos representar camadas físicas (*tiers*).

Várias tecnologias estão disponíveis para implementação de aplicações em camadas. Entre elas, podemos citar CORBA, Web-services, COM+, etc. Todas elas atendem à demanda cada vez mais crescente por aplicativos com capacidade de se comunicar uns com os outros. Web-services estão se tornando particularmente populares hoje em dia, dada a facilidade de implementação em diversas linguagens e por trafegar sob protocolo HTTP, de fácil gerenciamento.

2.4.1 n-Layer

Em uma aplicação monolítica, se ela for bem escrita, pode-se implementar facilmente abstrações para tratar das camadas lógicas. Em aplicações GUI, podemos pensar em uma camada de apresentação, responsável por apresentar os elementos gráficos e pela interação com o usuário; uma camada de negócios, responsável por processar os dados recebidos da camada de apresentação; e uma camada de persistência, responsável pela comunicação da camada de negócios com o SGBD, persistindo os dados no banco de dados ou obtendo dados para que sejam enviados à camada de negócios e então apresentados ao usuário através da camada de apresentação. Nesse caso, a separação das responsabilidades se dá apenas no nível lógico (*layers*). No caso mais comum, temos as três camadas ora descritas, mas podemos adotar outros modelos

em que o número de camadas é maior, no objetivo de modularizar ainda mais as responsabilidades de cada uma delas. Nesse caso, dizemos que a aplicação é *n-Layer*, isto é, *n-camadas*.

2.4.2 n-Tier

Quando queremos que nossos aplicativos atendam melhor aos requisitos de manutenibilidade, isto é, que sejam de fácil manutenção, podemos optar por outro modelo de desenvolvimento em camadas, chamado de *MultiTier*. No modelo *MultiTier* temos as camadas lógicas separadas em camadas físicas, isto é, em módulos independentes mas com mecanismos explícitos para garantir a interoperabilidade entre eles. Esse modelo é adotado principalmente na *web*, em que os dados obtidos das páginas HTML são enviados para serem processados em um servidor de aplicação, que é basicamente um computador servidor com boa capacidade de processamento, e no qual estão instalados os aplicativos ou módulos que respondem pelas camadas de negócios e de persistência. Neste caso, dizemos que a aplicação está operando no modelo *MultiTier*, também chamado de *n-Tier*, isto é *n-camadas*, sendo que o navegador *web* se encarrega apenas da camada de apresentação (cliente), e um ou mais servidores de aplicação se encarregam das outras camadas envolvidas na implementação.

2.5 *Feedback*

Um dos grandes problemas ao se escolher um método de interação com o usuário no sentido de lhe prover feedback é justamente se este método será adequado para essa comunicação. É preciso coletar dados sobre o tempo de execução das tarefas e então realizar a escolha correta (NIELSEN, 1993).

Para o desenvolvimento desse trabalho, foram levados em conta os três números básicos que regem o aconselhamento para as escolhas de interação:

- 0.1s: é considerado como resposta imediata e nenhum tratamento especial é necessário além de apresentar a resposta para o usuário;
- 1.0s: é o limite para que o usuário mantenha a atenção no sistema, embora ele já perca a noção de estar operando diretamente sobre os dados;
- 10s: é o limite para que o usuário mantenha a atenção em algum diálogo de interação.

Para tempos de resposta acima dos descritos anteriormente, é aconselhado prover *feedback* contínuo, na forma de um indicador de progresso (MYERS, 1985).

Indicadores de progresso tem três vantagens principais:

- asseguram que o sistema não parou de responder;
- indicam aproximadamente quanto tempo o usuário ainda terá de esperar; e
- proveem um elemento de interação que o usuário pode observar, tornando a espera menos dolorosa.

2.6 O padrão Observer

O padrão *Observer* é um dos padrões de comportamento elencado como padrões GoF (Gang of Four, ou Gangue dos Quatro) em (GAMMA, 1995). Consiste na definição de uma relação de dependência entre objetos, de um-para-muitos, de tal forma que, quando um objeto sofre uma modificação no seu estado, os outros objetos relacionados são notificados.

O objeto que sofre a modificação no estado é chamado de *subject*, e reflete mais comumente uma classe de negócios do domínio da aplicação. Cada objeto que deve ser notificado é chamado de *observer*, e geralmente está associado a uma aplicação cliente, mais especificamente a uma camada de apresentação.

Uma possibilidade de diagrama ilustrativo das relações envolvidas no padrão *Observer* é mostrada na Figura 2.1.

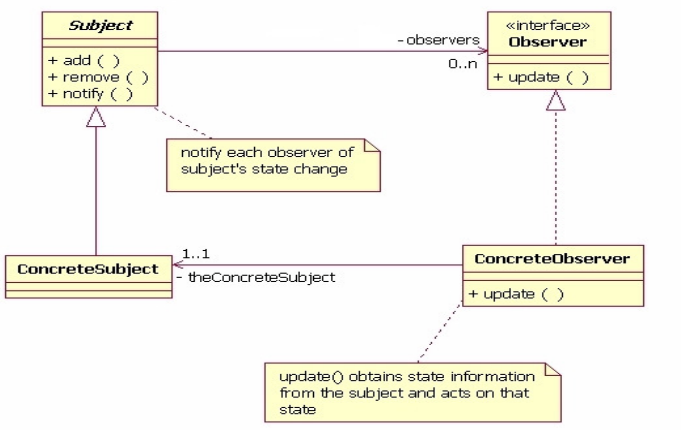

Figura 2.1: Diagrama ilustrativo do padrão *Observer*

Obtemos como benefícios na utilização do padrão *Observer* o fato de manter o baixo acoplamento entre o *subject* e seus observadores, além de permitir um ou mais observadores. Como desvantagens, pode tornar mais complicado um projeto quando notificações codificadas especificamente para aquele fim forem suficientes, além de eventualmente causar vazamento de memória no caso de os observadores não serem desanexados de seus *subjects* (KERIEVSKI, 2008).

2.7 COM+

COM+ (lê-se "COM Plus") é uma evolução do Microsoft Component Object Model (COM) e Microsoft Transaction Server (MTS), que desenvolve e amplia as aplicações escritas usando COM, MTS e outras tecnologias baseadas em componentes. COM+ torna as aplicações mais escaláveis, fornecendo escalonamento de processos e de objetos, e ativação *just-in-time*, i.e., o aplicativo é instanciado apenas no momento em que é necessário. É um modelo de objetos que não depende da linguagem de programação escolhida e fornece uma maneira padrão para que a interoperabilidade entre sistemas possa ser implementada. Essa interoperabilidade é obtida pela utilização de uma linguagem padronizada para a descrição dos objetos, que é a IDL – *Interface Description Language*. Falamos em descrição de objetos mas eles são descritos através das definições de suas interfaces, que são contratos entre a aplicação que provê os objetos e/ou serviços e as aplicações que os utilizam. (SILVA et al., 2003)

Através dessa tecnologia, nosso aplicativo "servidor" passa a ser um "componente" ou um conjunto de "componentes", que pode ser visto como um serviço à disposição no sistema operacional. A comunicação com as camadas de apresentação e de persistência se dá através de interfaces, que são contratos conhecidos por todas as camadas e que definem que funções estão disponíveis para processamento, sem, entretanto, explicitar como essas funções estão implementadas, preservando a noção de encapsulamento e garantindo, assim, a independência entre as camadas (baixo acoplamento).

A definição de um serviço COM+, isto é, suas interfaces e os métodos que serão expostos para utilização pelos clientes, é feita através de uma *Type Library*, que é como uma biblioteca de tipos onde estão as descrições das classes e interfaces que o serviço irá publicar.

3 CARACTERIZAÇÃO DO PROBLEMA: em busca de real feedback nas interfaces de sistemas de informação

A motivação para este trabalho surge de uma necessidade bastante frequente atualmente, de todo tipo de organização, seja privada ou pública, de pequeno, médio ou grande porte: troca de informações através de arquivos. Quando falamos em troca de informações, estamos obviamente nos referindo a sistemas de informações.

Além disso, todo o trabalho será desenvolvido levando-se em conta um sistema já existente, o qual lida com grandes volumes de dados. Esses dados são utilizados na troca de informações bancárias entre a UFRGS e o Banco do Brasil. O sistema roda hoje para atender principalmente à Pró-Reitoria de Planejamento e Administração no sentido de prover informações para as diversas unidades da UFRGS.

Dada essa característica restritiva, e como os dados reais são de caráter sigiloso, utilizaremos para os testes outro conjunto de dados, gerado a partir do sistema Febrl. Esse sistema, entre outras funcionalidades, nos fornece conjuntos de dados grandes o suficiente para que possamos realizar testes de apropriação de dados em grande escala.

Quando o volume de informações é pequeno, não há problemas em se utilizar técnicas tradicionais de interação com o usuário, como a alteração do cursor do *mouse* ou a exibição de uma mensagem solicitando que o usuário aguarde até que a tarefa seja concluída.

À medida que cresce o volume de informações, o tempo para conclusão das tarefas de transformação do arquivo texto em informação formatada de acordo com as regras de negócio e as classes específicas de cada sistema, e também de apropriação dessas informações em algum banco de dados pode se tornar algo a ser fortemente considerado como um fator de desconforto para o usuário, visto que, se a tarefa demorar demais a ser concluída, o usuário não terá certeza se as tarefas ainda estão sendo processadas ou se o sistema simplesmente parou de responder.

Além disso, em sistemas de informações típicos, há muitos usuários acessando as mesmas funções do sistema e necessitando de informações sobre as tarefas que estão em execução e seu estado atual. Surge uma nova necessidade: que cada usuário que venha a se conectar no sistema receba informações sobre o que está acontecendo.

Desse modo, nosso problema consiste em desenvolver uma estratégia de apropriação de grandes volumes de dados que permita aos diversos utilizadores do sistema obter informações sobre o andamento das tarefas mesmo quando conectados a computadores diferentes, em momentos diferentes. Não queremos, entretanto, que essa comunicação seja de responsabilidade da aplicação cliente, mas sim, que a aplicação servidora seja capaz de notificar a seus clientes a cada vez que ocorre uma mudança no estado atual da tarefa.

Uma solução proposta para esse tipo de interação está associada à utilização de padrões de projeto, mais especificamente na utilização do padrão *Observer* (GAMMA, 1995).

3.1 Situação atual

O aplicativo que demonstra a situação atual está implementado seguindo o modelo cliente-servidor, tendo sido desenvolvido em Delphi 5 Enterprise, e acessando, na época, um SGBD Sybase SQL Server. Posteriormente, a plataforma de dados foi migrada para Microsoft SQL Server. Isso não gerou grande impacto para adaptação, visto que foi necessário basicamente modificar as bibliotecas de acesso ao SGBD.

Esse aplicativo está construído como um único executável auto-contido que agrega todas as camadas da nossa aplicação: apresentação, lógica de negócios, e persistência.

Um diagrama ilustrativo da estrutura de funcionamento pode ser visto na figura 3.1, e as explicações sobre cada camada e suas funções poderão ser encontradas nas seções que se seguem.

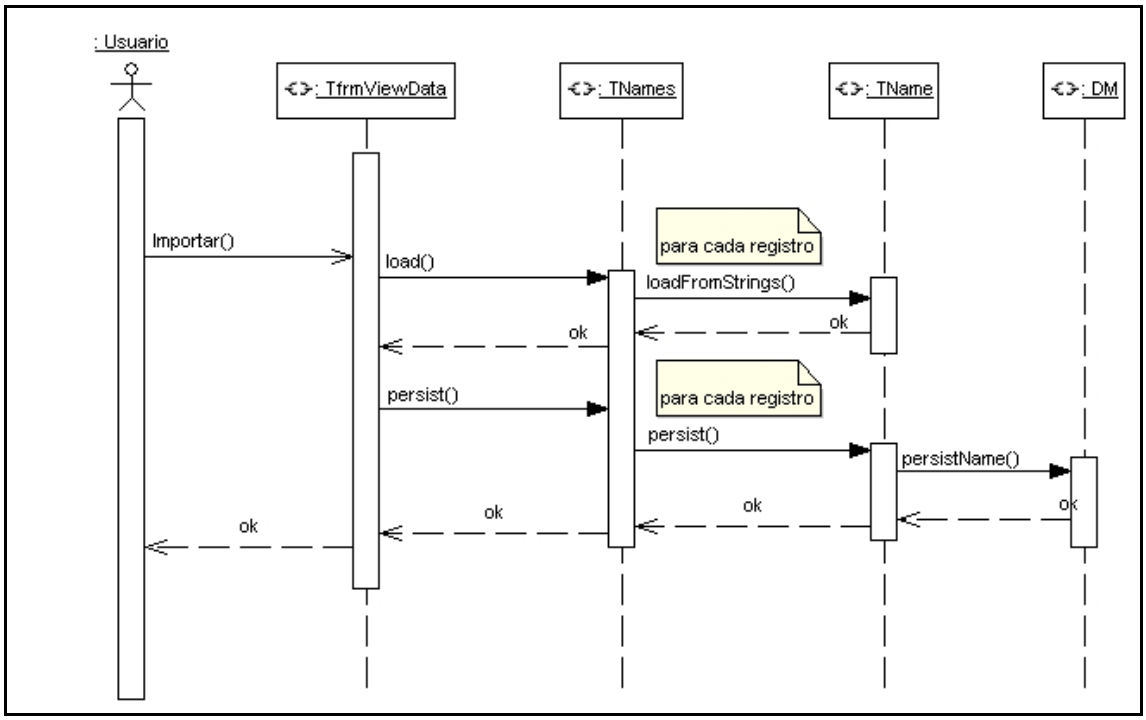

Figura 3.1: Diagrama de sequência da solução atual

A estrutura de dados aqui apresentada foi adaptada do modelo original, de modo a permitir a utilização dos conjuntos de dados obtidos a partir do sistema Febrl.

3.1.1 O Monólito

Iniciaremos esta seção pela explicação sobre alguns termos utilizados quando da definição da camada de persistência. Para essa finalidade, o Delphi nos coloca à disposição um repositório de componentes que farão o acesso a dados. Esse repositório recebe o nome de *data module* e sua declaração e implementação estão em uma *unit*,

que é uma unidade de programa no Delphi, representado a nossa camada lógica de persistência.

No *data module* serão colocados todos os objetos ligados à persistência, a saber:

- TADOConnection: componente que estabelece uma conexão com o SGBD;
- TADOQuery: componente utilizado para submeter uma instrução SQL para o SGBD.

No *data module* está definido apenas um método:

- persistName: encarregado de persistir as informações de um objeto da nossa classe de negócios.

Estabelecida a camada de persistência, precisamos estabelecer a camada de regras de negócios. Nesse aplicativo inicial, nossa classe de negócios recebe o nome de *TName* e está constituída de apenas quatro campos, a saber:

- rec_id: identificador do registro;
- given_name: primeiro nome;
- surname: sobrenome;
- state: estado da federação.

Além disso, estão definidos basicamente dois métodos:

- loadFromStrings: recebe um array de valores (*StringList*) e preenche os campos da classe conforme cada caso;
- persist: chama o método de persistência definido na respectiva camada.

De maneira a preservar a atribuição de responsabilidades de cada classe, foi criada uma outra classe, *TNames*, que é uma lista de *TName*. Essa nova classe funciona como um contêiner e mantém uma lista de objetos da classe *TName*, para posterior persistência. Essa classe também têm dois métodos:

- load: recebe um nome de arquivo, carrega seu conteúdo em um array de valores (StringList) e o percorre, instanciando objetos da classe *TName* e adicionando-os na lista;
- persist: para cada objeto na lista, chama iterativamente seu procedimento persist.

Essas duas classes respondem pela camada lógica de negócios, e estão definidas em uma *unit* do Delphi.

Para a camada de apresentação foi utilizada uma unidade de apresentação do Delphi, chamada de TForm, que nos fornece uma superfície sobre a qual podemos projetar nossa interface gráfica. Nessa camada será realizada a interação do usuário com o sistema, em que será feita a escolha do arquivo a ser importado e onde serão apresentados os resultados do processamento.

O modo como essas três camadas lógicas se relacionam pode ser visto na Figura 3.2.

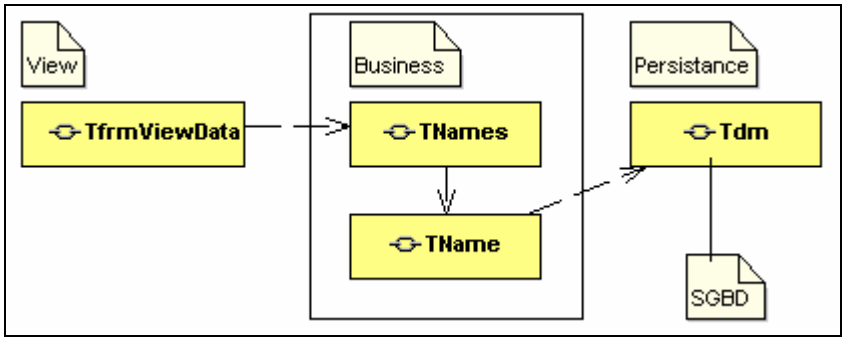

Figura 3.2: Arquitetura do sistema atual em camadas lógicas

3.1.2 Testes

Para os testes, foram utilizados conjuntos de dados obtidos do Febrl. Os conjuntos de testes foram estabelecidos com 1000, 2500, 5000, e 10000 registros. Para se obter uma idéia de qual seria o maior problema a ser tratado, foram feitas medições nos tempos de carga de dados (*load*) e de persistência (*persist*). Estes tempos são apresentados na Tabela 3.1.

Tabela 3.1: Tempos de carga e persistência – versão atual

Nº de registros	carga	Persistência
1000	32	1639
2500	85	4571
5000	203	8187
10000	371	16351

Todos os tempos são dados em milissegundos (ms), calculados sobre uma média de quinze execuções, de modo a simular diversos momentos de processamento. Pode-se observar que os tempos para carga de dados são de ordem bastante baixa, enquanto que o tempo para persistência já se mostra bastante elevado, mesmo para um número pequeno de registros.

3.1.3 Considerações

Para a interação com o usuário, durante o tempo de carga dos dados, pode ser adotada uma estratégia bastante comum, como a simples alteração do cursor do mouse, indicando ao usuário que algum processamento está sendo feito. Entretanto, já prevendo maiores volumes de dados, foi adotada outra solução que é a de apresentar uma janela de diálogo informando sobre o andamento da tarefa.

A mesma estratégia foi adotada durante a persistência dos dados, para que o usuário não tenha a impressão de que o sistema parou de responder. Não obstante, essa estratégia, apesar de amplamente utilizada em diversos sistemas, está longe de ser a mais adequada. Para uma tarefa que potencialmente demandará vários minutos para ser executada, não é possível partir do princípio de que o usuário tenha de ficar em frente ao computador esperando pelo seu término. Além disso, queremos que o usuário possa se conectar em outro computador, ou em outro momento, e acompanhe o andamento da tarefa, sem ter de se preocupar com o fato de que o sistema estaria praticamente

monopolizando o uso do sistema operacional de modo a realizar a comunicação entre cliente e servidor.

Uma solução para este problema, e objetivo principal desse estudo, será apresentada a seguir.

4 PERCIPIENT IMPORTER: provendo *feedback*

A solução proposta visa a atender principalmente à necessidade de prover *feedback* ao usuário, i.e., mantê-lo informado sobre o andamento das tarefas que estão sendo executadas em um servidor de aplicação e permitir que vários clientes possam se conectar simultaneamente para obter essas informações. Além disso, visa a refatoração de partes do código, transferindo as camadas de regras de negócios e de persistência para outra camada física, através da utilização do padrão *Observer*. Desse modo, obtemos um aplicativo cliente enxuto (*Thin Client*) e mais fácil de manter, uma vez que a maior parte das alterações de um sistema se dá no nível das regras de negócios, e não na camada de apresentação. Diminuímos, assim, o impacto das modificações do sistema, pois não precisaremos distribuir uma nova versão do aplicativo a cada alteração nas regras de negócios.

Antes de partirmos para a solução proposta, entretanto, foi descrita, na seção 3.1, uma implementação que segue o modelo cliente-servidor tradicional, na forma de um aplicativo monolítico, encarregado das camadas de apresentação, de negócios, e de persistência. Um aplicativo seguindo esses moldes foi a motivação para o desenvolvimento deste trabalho e a solução será introduzida na seção 4.2.

Essa solução será construída utilizando o padrão *Observer* (GAMMA, 1995) e desenvolvida sobre uma plataforma COM+, utilizando os serviços de componentes distribuídos da Microsoft e persistindo os dados em SGBD Microsoft SQL Server.

Como ferramenta de desenvolvimento será utilizado o Delphi 7 Enterprise, da antiga Borland, atualmente provido pela Embarcadero.

Além disso, faz-se necessário introduzir algumas informações sobre a linguagem escolhida e a definição de alguns termos os quais será necessário conhecer para melhor entendimento das explicações sobre a solução.

4.1 Por que Delphi?

Apesar de haver alguma controvérsia sobre se esta é ou não a melhor ferramenta para desenvolvimento direcionado a ambiente Windows, acredita-se que essa discussão seja mais passional do que racional. Atualmente, a maioria das linguagens que seguem o paradigma de orientação a objetos, como é o caso de Delphi, apresentam recursos e classes de componentes muito semelhantes. Desse modo, nos parece que outras ferramentas poderiam ter sido escolhidas sem prejuízo para a proposta desse estudo.

Foi escolhido o Delphi por ser a ferramenta de uso corrente na UFRGS para sistemas desenvolvidos no modelo cliente-servidor e, nesse contexto, podemos encarar esse fato como sendo uma restrição para o desenvolvimento da solução. Além disso, contamos com boa desenvoltura com a linguagem subjacente – Delphi é uma evolução

de Pascal – e também razoável conhecimento do paradigma sobre o qual está a ferramenta está fundamentada - orientação a objetos.

4.2 Solução baseada no padrão Observer

Como mencionado anteriormente, o aplicativo no modelo em que hoje está desenvolvido não atende aos requisitos de interação com o usuário, no momento em que não permite que ele obtenha informações sobre o andamento da tarefa em momentos diferentes e em computadores diferentes. Além disso, por ser um aplicativo monolítico instalado no computador do cliente, demanda muitos recursos de máquina, o que pode impactar bastante nos custos em se tratando de grandes ambientes corporativos.

Abordaremos essa questão através da modularização mais efetiva da aplicação, dessa vez não apenas em camadas lógicas (*layers*), mas também em camadas físicas (*tiers*), como pode ser visto na Figura 4.1 em que o quadro da esquerda ilustra a camada física que representa a aplicação cliente e o quadro da direita ilustra a camada física que representa a aplicação servidora, com as camadas lógicas de negócios e persistência.

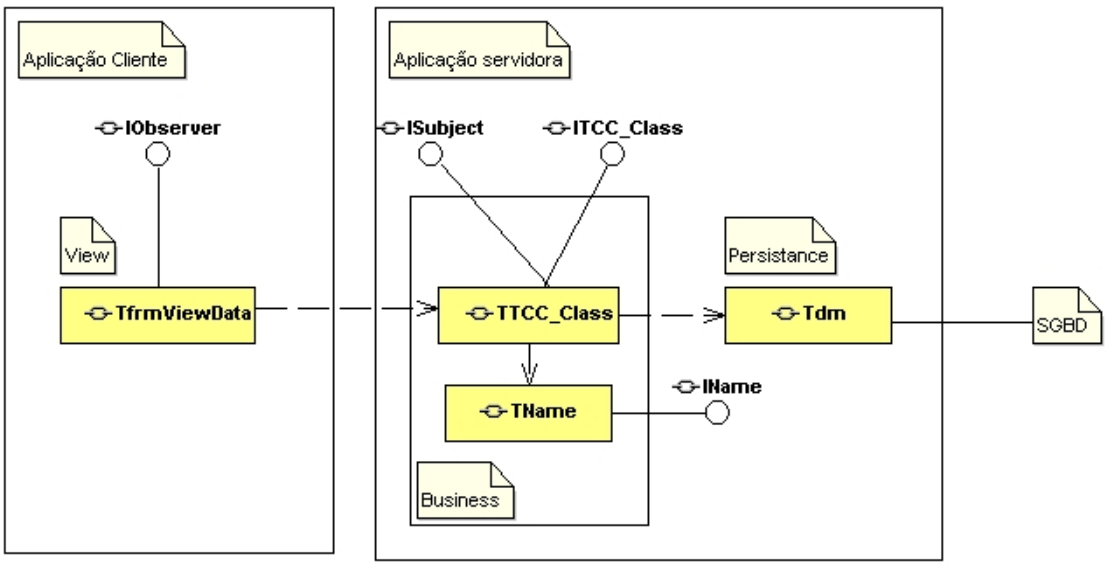

Figura 4.1: Percipient Importer: camadas lógicas e físicas

Para atingir esse objetivo, será desenvolvido um conjunto de aplicações utilizando o padrão *Observer*, um dos padrões de projeto conhecidos como Padrões GoF (*Gang of Four*). Esse padrão está definido em GAMMA, 1995. Através do uso desse padrão, é possível fazer com que uma aplicação servidora (*subject*) mantenha atualizadas as aplicações que estarão agindo como clientes (*observers*), provendo ao usuário informações sobre o progresso das tarefas solicitadas, preservando, no entanto, o baixo acoplamento entre essas duas camadas.

O padrão *Observer* será utilizado para a comunicação entre a aplicação servidora e seus clientes. Esse padrão é do tipo comportamental, e permite que uma aplicação servidora, que passa a ser chamada de *subject*, se comunique com seus clientes, que passam a ser chamados de *observers*, e os notifique quando alguma modificação acontece na classe de negócios que está sendo observada. Desse modo, o processamento pesado e as regras de negócios ficam em uma camada separada do aplicativo cliente ao mesmo tempo em que permite que a comunicação seja bidirecional, i.e., tanto no

sentido cliente → servidor como no sentido servidor → cliente, apesar do servidor (*subject*) não conhecer nenhum detalhe sobre seus clientes.

4.2.1 Definição das interfaces

Iniciaremos o desenvolvimento pela definição das interfaces, tidas aqui como um elemento das linguagens orientadas a objetos que funciona mais ou menos como um contrato entre as partes que as utilizarão, não revelando, no entanto, detalhes sobre sua implementação, preservando fortemente o encapsulamento.

Para implementar o padrão Observer, serão definidas, então, duas interfaces básicas:

- ISubject: conterá as definições dos métodos responsáveis por adicionar (*attach*) ou remover (*detach*) observadores de uma lista de observadores, além de método para notificação (*notify*) desses observadores. Como queremos, também, informação a respeito do estado do processamento, incluiremos aqui métodos para essa fim (*get/set state*);
- IObserver: conterá a definição do método que é responsável pela atualização (*update*) das informações na camada de apresentação.

Essas interfaces, bem como as demais interfaces necessárias para o desenvolvimento, serão definidas em uma *Type Library*, utilizando, para isso o editor disponibilizado pelo Delphi.

4.2.2 A *Type Library*

Uma *Type Library* é um arquivo no qual estarão as definições das interfaces utilizadas para o desenvolvimento tanto da aplicação servidora como da aplicação cliente, bem como, definições das classes de automação que serão disponibilizadas posteriormente utilizando o modelo de componentes COM+.

É descrita utilizando uma linguagem padrão de descrição de interfaces conhecida por IDL – Interface Description Language. Através da utilização dessa linguagem é que se alcança a interoperabilidade entre diversas linguagens de programação que potencialmente podem ser utilizadas para desenvolvimento de aplicações que serão consumidoras dos serviços expostos pelas interfaces do nosso servidor COM+ disponibilizado através dos serviços de componentes do sistema operacional (SILVA, 2003).

No Delphi, especificamente, a descrição de uma Type Library pode alternativamente ser feita utilizando a linguagem Pascal ou o ambiente gráfico. A própria ferramenta se encarrega de realizar a conversão das definições para IDL, de modo que possa ser efetivamente utilizada com outras linguagens.

A Figura 4.2 mostra o editor de Type Library disponibilizado pelo Delphi.

O símbolo  representa a definição de uma interface, e o símbolo  representa a definição de uma classe de automação que implementa a respectiva interface.

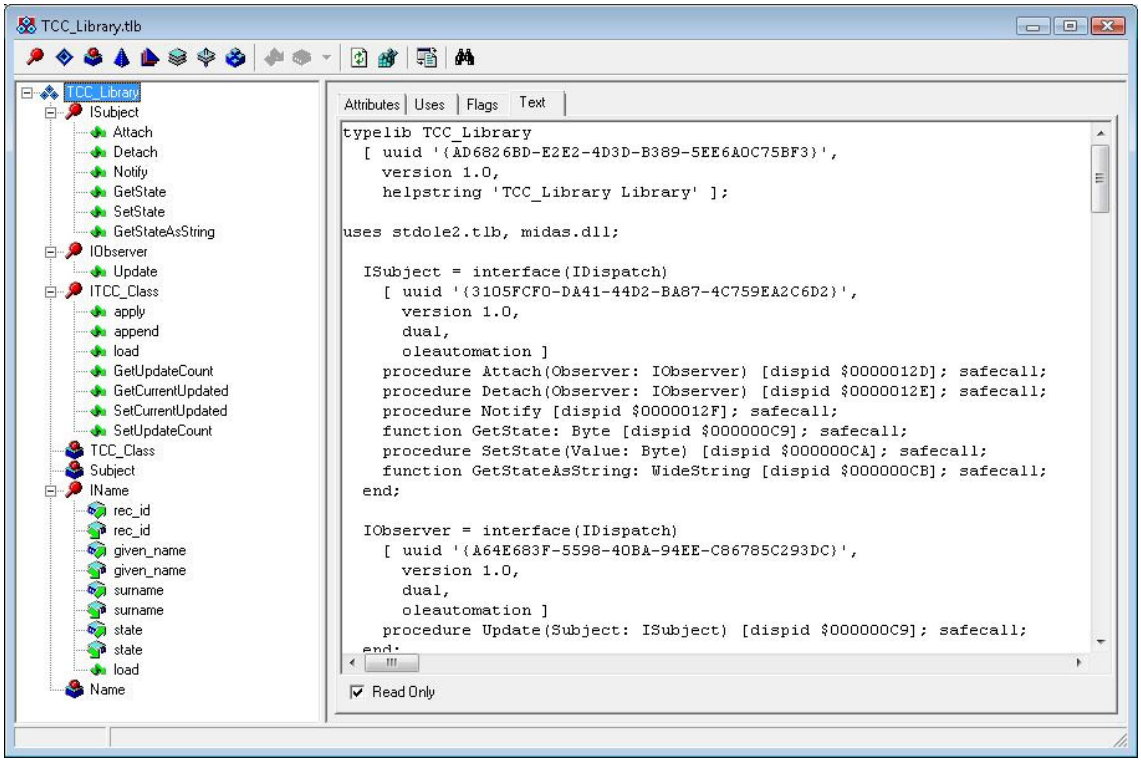

Figura 4.2: Editor de *Type Library* do Delphi

4.2.3 A aplicação servidora

Observando a Figura 4.2, percebe-se que não há uma classe de automação que implemente a interface *IObserver*. Isso é intencional, uma vez que essa interface será implementada pela nossa aplicação cliente, como veremos adiante.

No momento, nos interessam as definições da aplicação servidora. Esta será uma biblioteca que passará a conter a camada de negócios e a camada de persistência, e será registrada e disponibilizada como um serviço de componente do sistema operacional.

Nossa antiga classe de negócios *TName* implementa agora uma interface *IName* e passa, então, a ser definida na camada remota como uma classe de automação, tirando essa responsabilidade do aplicativo cliente. Sua instanciação na camada cliente ainda é possível, já que sua interface é exportada através do serviço de componentes.

O *data module*, responsável pela persistência dos dados não sofre alterações, visto que sua função já estava bem definida no escopo do sistema.

Surge uma nova classe de automação, *TTCC_Class*, que implementa uma interface *ITCC_Class* e também a interface *ISubject*, e passará a responder como o canal de comunicação entre a camada de apresentação e a camada de persistência. Ou seja, TTCC_Class será, na realidade, o nosso "observável" ou *subject*. Essa classe substitui a classe *TNames* da implementação anterior para permitir a utilização de um *TClientDataset*, componente do Delphi que realizará a manipulação de uma lista de objetos, mapeados então para uma lista de registros, facilitando sobremaneira a comunicação com a camada de persistência. Na criação de *TTCC_Class*, definimos que o modelo de instanciação será *multi instance*, o que significa que haverá apenas um objeto dessa classe atendendo às requisições dos vários clientes (RODRIGUES, 2002).

Uma característica interessante no Delphi é que ele permite que uma interface seja implementada por delegação, isto é, os métodos definidos na interface não precisam

necessariamente estar todos implementados na própria classe. Dessa maneira, quando dizemos que nossa *TTCC_Class* implementa a porção *subject* da nossa aplicação, já que implementa a interface *ISubject*, na realidade existe uma propriedade nessa classe que delega a implementação dos métodos da interface *ISubject* para uma classe *TSubject*. Com isto, melhoramos a reutilização de código e evitamos ter de reescrever os métodos de *ISubject* em cada classe que a implemente. Além disso, a interface *ISubject* não contém os métodos e propriedades que nos interessam em termos de informações sobre o modelo, ou a nossa classe de negócios. Essas definições estão todas em *ITCC_Class*, e serão potencialmente diferentes para outras classes de negócios que venham a implementar a interface *ISubject*.

O método *persist* da implementação de *TNames* é escrito agora nessa nova classe e ligeiramente modificado para permitir a chamada aos métodos de persistência através da utilização dos eventos providos pelo componente *TClientDataset*. Passamos a chama-lo de *apply*, porque o nome do método disponível no componente é *ApplyUpdates*. Através dessa chamada, os registros são enviados da classe de negócios para a camada de persistência. À medida que cada registro é salvo na base de dados, é gerado um evento *AfterUpdateRecord*. Através desse evento, atualizamos o nosso *subject* com a informação sobre a quantidade de registros processados. Quando o *subject* é atualizado, ele aciona o método *Notify* definido na interface *ISubject*, notificando, desse modo, todos os observadores que estiverem anexados no momento. Definimos, dessa forma, nosso *callback*, que em termos do padrão *Observer* é o nosso método de notificação, responsável por "avisar" a todos os observadores que alguma informação relevante para o modelo sofreu uma alteração que merece ser repassada para a camada cliente e então para o usuário do sistema.

4.2.4 A aplicação cliente

Uma vez definidas a *Type Library* e a aplicação servidora, passamos à construção da aplicação cliente. Essa aplicação será a porção *observer* do nosso modelo, pois implementará a interface *IObserver*.

Como agora não temos mais as regras de negócios nem a camada de persistência na aplicação cliente, esta se encarrega apenas da camada de apresentação, o que resulta em um aplicativo menor, mais leve, que exige muito menos recursos de processamento no computador do cliente, uma vez que deslocamos todo o processamento pesado para a aplicação servidora. Esse tipo de aplicativo é conhecido como *Thin Client*.

Nosso cliente agora não conhece mais os detalhes de implementação da nossa classe de negócios e, para esse caso específico de importação de dados, não é necessário instanciá-la na aplicação cliente, popular uma lista e depois realizar a persistência. Tudo isso será feito pela a aplicação servidora.

Temos basicamente os mesmos métodos da solução original, com a diferença de que ao iniciar, a aplicação instancia um objeto de automação, no caso, o *subject* que é nossa *TTCC_Class* e faz uma chamada ao método *Attach* de *ISubject*, passando uma referência de sua própria interface para o método. Desse modo, a aplicação fica registrada na lista de observadores de *TTCC_Class*.

Como nossa camada de apresentação implementa a interface *IObserver*, é necessário escrever o método *Update*, que é o método chamado pelo *subject* quando são feitas as notificações aos observadores. Observe que esse método é apenas chamado pelo *subject*, que não conhece sua implementação. O código implementado no método

Update é que será responsável por decidir quais as informações da nossa classe de negócios e de que forma elas serão apresentadas para o usuário.

Nosso objetivo é manter o usuário informado do andamento da tarefa de importação dos dados, bem como o estado atual da nossa aplicação servidora, então nos interessam o número de registros já processados, o total de registros, tempos de execução das tarefas, e se a aplicação servidora está ou não disponível.

Optamos por utilizar agora, para a interação com o usuário, uma barra de progresso e também exibir algumas informações na área de notificação. Já no início da aplicação, a aplicação servidora é consultada e seu estado atual é exibido para o usuário. Se novas conexões ocorrerem durante o andamento da tarefa, o novo cliente (*observer*) será notificado e exibirá as informações para o usuário através do método *Update*.

Para o caso específico e em caráter ilustrativo, a opção foi de fazer a notificação a cada 100 registros processados, a fim de evitar sobrecarga de processamento na aplicação cliente. Em um ambiente de execução real, o ideal seria fazer com que os observadores fossem notificados baseados em um temporizador, por exemplo, a cada segundo, pois o número de registros pode não ser uma boa base em um ambiente de muita concorrência, em que o servidor pode estar muito ocupado para atender às requisições.

Um diagrama ilustrativo das sequências de interações entre as camadas cliente e servidora pode ser visto na Figura 4.3.

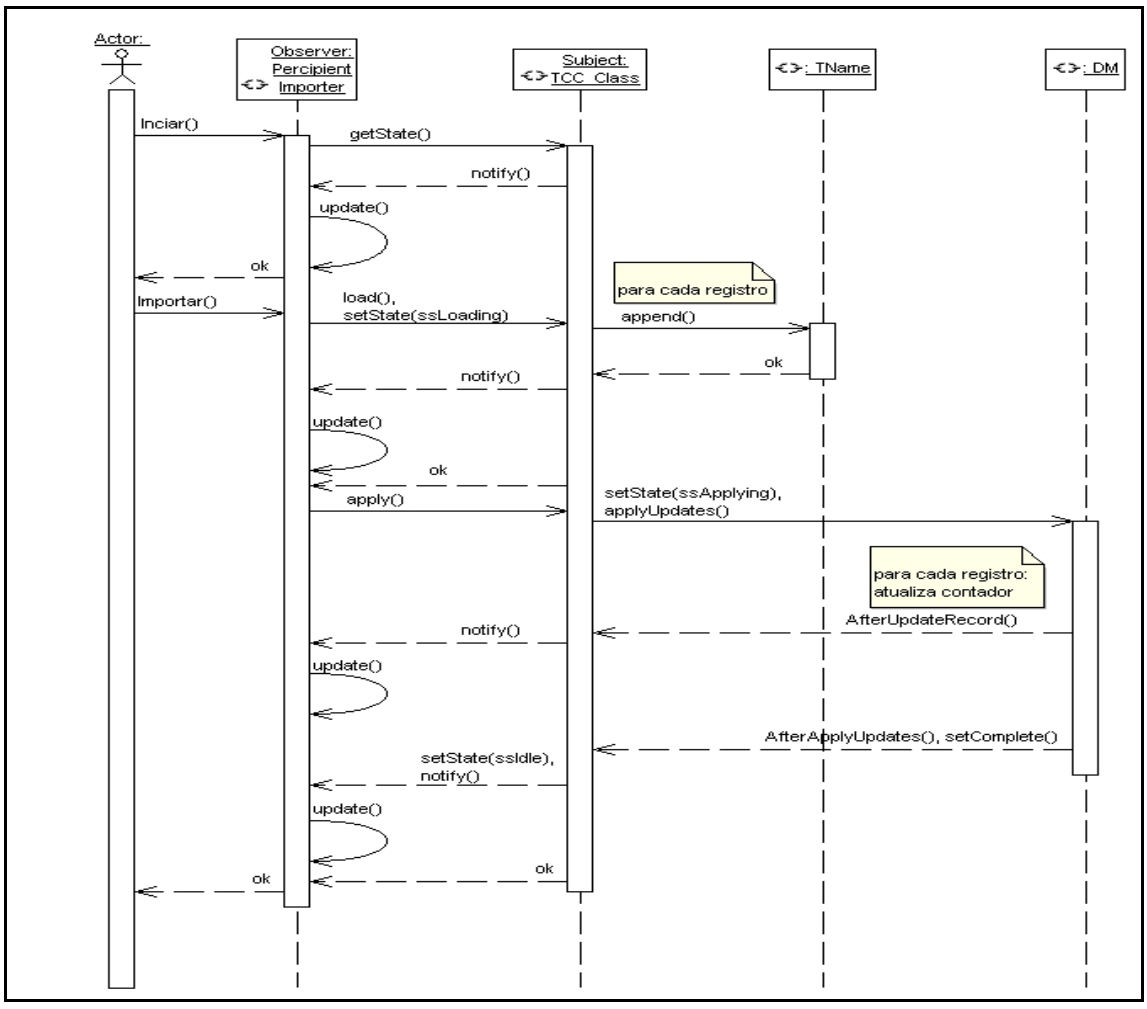

Figura 4.3: Diagrama de sequência – Percipient Importer

Pode ser visto na Figura 4.3 que, diferentemente da implementação anterior apresentada na seção 3.1, temos agora uma estrutura de eventos para contemplar as notificações dos vários observadores que estiverem anexados ao *subject* que estiver sendo observado. Esse tipo de notificação não era possível na solução anterior, visto que não tínhamos como consultar o "estado" atual do processamento da tarefa.

Na realidade, ainda não fazemos isso. Em vez de o aplicativo cliente consultar o estado atual da tarefa é o *subject* que tem registrados os diversos observadores e utiliza o método *notify()* para prover esse feedback para o usuário. Quando o observador é notificado, dispara então a execução do *callback update()*, definido na interface IObserver e implementado em nosso Percipient Importer, responsável por atualizar as informações para o usuário, ou seja, atualizar a barra de progresso e o texto da área de notificação.

O processamento pesado agora é realizado pelo nosso *subject*, o que nos leva a obter o tipo de aplicação cliente que almejávamos desde o início: um *Thin Client*, que exigirá muito menos recursos de máquina para ser executado e potencialmente pode ser escrito em qualquer linguagem de programação que tenha suporte ao modelo de componentes distribuídos COM+.

4.2.5 Testes

Após construída a solução, foram realizados os mesmos testes e aferidas as medidas de tempo como na solução anterior. Observamos na Tabela 4.1 que o tempo de carga inicial sofreu uma elevação. Isso deve-se ao fato de que agora o conteúdo do arquivo baseado em leiaute é transmitido através da rede diretamente para o servidor, o que acaba por gerar um tráfego que antes não existia, uma vez que toda a carga e transformação do texto era feita no aplicativo cliente. Conclui-se aqui que o fator determinante para esse tempo é o tráfego de rede e não a capacidade de processamento da estação cliente.

Essa aparente perda, no entanto, resultou em um grande ganho no momento da persistência. Como os dados já estão transformados e a persistência de informações agora é feita em lote, através da utilização de classes de objetos específicas do Delphi para esse fim quando trabalhando em soluções de processamento distribuído, os tempos aferidos foram, em muito, inferiores aos tempos observados na solução anterior.

Vale lembrar que, na solução inicial, a camada responsável pela persistência estava implementada em um computador diferente daquele em que o SGBD estava instalado. Isso é a prática comum no modelo cliente-servidor. Na solução que utiliza o padrão *Observer*, o servidor de aplicação está instalado no mesmo computador em que temos o SGBD, e esse fator é determinante para o grande ganho nos tempos aferidos para a persistência.

No cômputo do tempo total, conseguimos, para o pior caso, em torno de 23% (vinte e três por cento) de ganho, e para o melhor caso, mais de 50% (cinqüenta por cento), o que apesar de não estar entre os objetivos primários desse estudo, sempre é uma boa notícia a se dar para o usuário do sistema.

A tabela 4.1 mostra os tempos de carga e persistência para os conjuntos de dados utilizados. Novamente foi utilizada a média de quinze execuções.

Tabela 4.1: Tempos de carga e persistência – versão nova

nº de registros	carga	persistência
1000	232	1257
2500	345	2914
5000	478	4863
10000	631	7425

4.2.6 Considerações

Para observar se o quesito de interação seria satisfeito no caso de vários usuários estarem conectados simultaneamente, foi feita uma simulação de uso por diversas aplicações cliente obtendo informações da aplicação servidora. Cada aplicação cliente, como proposto no nosso modelo, é um *observer* e está registrada na lista de observadores de nosso *subject*. Vale lembrar que a tarefa é iniciada por apenas um deles e todos os outros que já estiverem conectados ou que venham a se conectar receberão as notificações sobre o estado atual do *subject*. Aqui mostramos, na Figura 4.4, apenas quatro observadores. Entretanto, durante a fase de realização de testes, chegamos em torno de vinte observadores, mostrando-se a plataforma COM+ bastante estável no tocante à escalabilidade da solução.

Um exemplo de execução com diversos observadores sendo notificados simultaneamente pode ser visto na Figura 4.4.

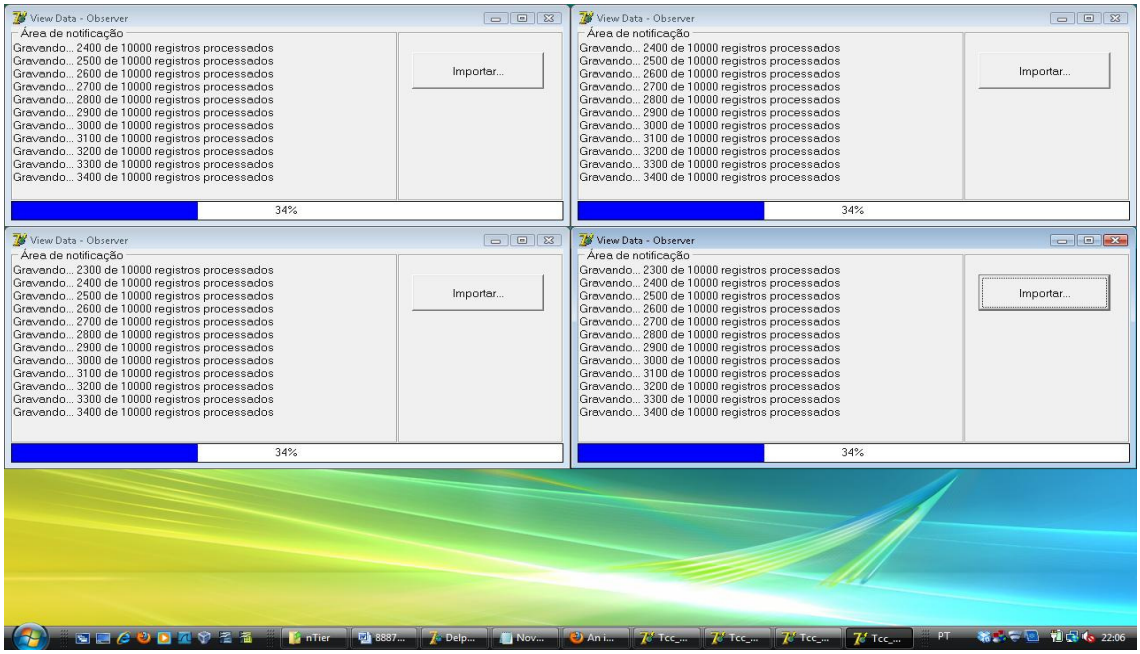

Figura 4.4: Exemplo de execução com diversos observadores

Para a operação do sistema, quando o cliente desenvolvido seguir a implementação para plataforma Windows, qualquer computador que suporte a instalação do Windows 98 será suficiente para a tarefa. Isso significa que a solução pode ser utilizada mesmo quando o parque tecnológico disponível não conta com grandes recursos de hardware, o que, para grande parte dos casos, tanto em empresas públicas como privadas, é uma realidade.

5 CONSIDERAÇÕES FINAIS

Com a finalização desse estudo, podemos concluir que a solução proposta e implementada com a utilização do padrão *Observer* - que define um *subject*, o qual passa a ser o nosso canal de comunicação e responsável por acionar o nosso *callback* de modo a notificar uma lista de *observers* - vem ao encontro das expectativas de interação com o usuário e está de acordo com o que tínhamos em mente no que diz respeito à modularidade e baixo acoplamento descritos na parte inicial deste trabalho.

Atendemos principalmente ao quesito especificado em uma das heurísticas de Nielsen, em seu artigo de 1993, que recomenda que o usuário deva ser sempre informado sobre o estado atual do processamento das tarefas (*feedback*). O maior ganho obtido foi na possibilidade de vários clientes poderem conectar-se simultaneamente e obterem informações atualizadas sobre o andamento da tarefa, mesmo em vários computadores diferentes.

Como efeito secundário, mas nem por isso menos importante, na busca pela melhoria da interação com o usuário, o sistema passou por uma reestruturação de modo que agora está implementado em camadas físicas, separando a camada de apresentação das camadas de negócios e de persistência. Essa estrutura é bastante recomendada em engenharia de software, em especial em sistemas orientados a objetos, uma vez que se deseja maximizar o reuso de artefatos de software, minimizando impactos nos custos de desenvolvimento e manutenção de sistemas.

Os testes efetuados nas fases inicial e final do projeto demonstram também, apesar de que isso não estava nos nossos objetivos, que a estratégia de separação em camadas físicas, transferindo o processamento mais pesado para uma camada remota, proporcionou ganhos na velocidade da execução, em especial no tocante à persistência, pois o cliente não precisa mais efetuar uma transmissão para o servidor a cada registro que deve ser salvo na base de dados, já que tanto o servidor de aplicação como o SGBD estão instalados no mesmo computador.

Ainda baseado nos testes percebemos que, embora limitados a 10000 registros, o modelo apresenta boa escalabilidade para volumes maiores de dados, visto que o modelo de componentes distribuídos da Microsoft, o COM+, é bastante estável e não precisamos nos preocupar especificamente com o escalonamento da tarefa já que o próprio sistema operacional se encarrega disso.

Em caso de crescimento demasiado no volume de dados ou na necessidade de conexões simultâneas, o Delphi apresenta alguns componentes especializados que facilitam a programação no sentido de permitir balanceamento da carga entre diversos servidores, de modo que, se um deles parar de responder, outro assume as tarefas, provendo, dessa forma, um mecanismo de tolerância a falhas quando necessário, ou simplesmente de distribuição de processamento para o caso de aumento na demanda.

Na questão da portabilidade, a opção por utilizar COM+ deu-se justamente pelo fato de que as aplicações clientes podem ser escritas em qualquer linguagem que suporte o modelo de componentes distribuídos. E é de nosso conhecimento que várias delas, como Java, C#, PHP, C++, Delphi, Lazarus, Visual Basic, ASP, entre outras, suportam esse modelo de instanciação. Logo, um mesmo aplicativo servidor desenvolvido sobre a plataforma COM+ pode atender requisições de vários aplicativos clientes, escritos em diversas linguagens diferentes disponíveis atualmente.

Não se pretende aqui dizer que a tecnologia COM+ é um mundo perfeito. Em especial, deve-se levar em conta que é uma tecnologia proprietária da Microsoft, e restrita ao ambiente Windows. Para trabalhos futuros, podemos considerar fortemente a implementação desse mesmo tipo de solução através da construção de um Web Service, usufruindo das vantagens com a facilidade de gerenciamento que o protocolo http oferece e também do tráfego utilizando SOAP, que basicamente troca mensagens em formato XML. Isso possibilitaria atingir mais facilmente o grupo de usuários de dispositivos móveis, como PDA's, *smartphones*, ou mesmo celulares mais avançados.

Um detalhe na implementação utilizando o Delphi, e isso restrito à versão que foi utilizada (Delphi 7), diz respeito à utilização de variáveis globais, que são altamente desaconselhadas em boas práticas de programação. Entretanto, como a linguagem ainda não disponibilizava variáveis de classe, mais seguras, esse recurso teve de ser utilizado para implementar, por exemplo, a lista de observadores de um determinado *subject*. Nas versões mais novas do Delphi essa deficiência já foi resolvida e é possível utilizar perfeitamente as variáveis de classe, preservando a segurança no acesso aos dados.

REFERÊNCIAS BIBLIOGRÁFICAS

GAMMA, Erich et alli. **Design Patterns: Elements of reusable Object-oriented Software /** Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Addison-Wesley, 1995.

KERIEVSKY, Joshua. **Refatoração para padrões**. Porto Alegre. Bookman, 2008.

MYERS, B. A. (1985). **The importance of percent-done progress indicators for computer-human interfaces**. *Proc. ACM CHI'85 Conf.* (San Francisco, CA, 14-18 April), 11-17.

NIELSEN, Jakob. http://www.useit.com. Acessado em novembro/2010. 1993.

PREECE, Jennifer. **Interaction design: beyond human-computer interaction**. Jennifer Preece, Yvonne Rogers, Helen Sharp. 2nd. ed. John Wiley & Sons, Inc. 2002.

RODRIGUES, Anderson Haertel. **Sistemas Multicamadas com Delphi: DataSnap e dbExpress** – Conceitos, Implementação e Macetes. Anderson Haertel Rodrigues. Primeira edição. Santa Catarina. VisualBooks Editora, 2002.

SILVA, Alex de Araujo et alli. **Metodologia e Projeto de Software Orientado a Objetos: modelando, projetando e desenvolvendo sistemas com UML e componentes distribuídos /** Alex de Araujo Silva, Carlos Francisco Gomide e Fabio Petrillo. Primeira edição. São Paulo. Editora Érica, 2003.