

Falando em Camadas

(Walter Itamar Mourão – 10/98)

Nos sistemas cliente/servidor tradicionais (ou duas camadas) temos três opções com relação às regras de negócio: colocamos junto da interface do usuário, junto do banco de dados ou mesclamos as duas opções. Nenhuma dessas opções é 100 % boa.

Regras junto da interface

Esse caso é o mais comum e somos praticamente induzidos a trabalhar assim pelas ferramentas RAD atuais, inclusive o Delphi. A possibilidade de se ligar um componente visual (TDBEdit) diretamente a um campo de uma tabela em um banco de dados (TTable), é excelente no que diz respeito à velocidade de programação e apresentação de resultados. Essa é uma técnica muito boa para prototipação e desenvolvimento de pequenos sistemas, mas traz incomodos enormes quando se trata de sistemas de grande porte.

Entre os maiores problemas que essa forma de programar acarreta estão:

- Dificuldades de manutenção – quando trabalhamos com os eventos de componentes visuais é fácil dispersarmos as regras, tais como associarmos consistência de campo ao evento OnExit do TDBEdit em um momento e ao OnValidate do TField em outro momento. Obviamente isso causa dores de cabeça no momento da manutenção, e compulsão ao suicídio caso a manutenção seja feita por outro programador.
- Problemas de performance (aplicação, banco e rede) – é público e notório que a visão das tabelas de um banco de dados relacional de forma *flat*, mimetizando um banco não relacional causa, por si só, um impacto considerável no banco de dados e na rede como um todo, devido ao grande número de comandos executados e dados retornados. De fato, basta montarmos uma pequena aplicação, com um TDBGrid ligado a uma tabela no InterBase, por exemplo, e observarmos o log gerado pelo SQL Monitor após algumas inserções e edições, é assustador !
- Replicação das regras – todas as aplicações que atualizam os dados de uma determinada tabela terão que replicar as mesmas regras e restrições que se aplicam a esta tabela.
- Dificuldade de distribuição – sempre que uma regra é alterada o programa tem que ser redistribuído, trazendo problemas de controle de versão e distribuição física do aplicativo e seus complementos.

Como exemplo desse último item, imagine um ambiente com 300 usuários. Acabamos de desenvolver uma aplicação que vai ser distribuída para esses 300 usuários (com BDE e etc.). Após uma 2 semanas de instalações descobrimos um erro grave no código que nos obriga a redistribuir tudo. Mais duas semanas e é lançada uma nova versão do BDE ou do transporte nativo do banco que melhora a performance de nossa aplicação consideravelmente... Enfim, podemos cair em situações tão inadmissíveis quanto inevitáveis.

Regras junto dos dados

Ao colocarmos as regras dentro do banco de dados, usando os recursos nativos do banco de dados (triggers, stored procedures e constraints), estamos automaticamente ligando nossa aplicação ao banco de dados que estiver sendo usado, aumentando muito o retrabalho caso a aplicação venha a ser convertida para ser usada com outro banco de dados.

Tipicamente isso não é um problema considerável para a maioria das empresas, uma vez que a mudança de um banco de dados é um operação complexa por si só, e que não pode ocorrer frequentemente. No entanto, praticamente toda software house que desenvolve em ambiente cliente/servidor enfrenta esse problema em maior ou menor grau, dependendo da complexidade e da forma de projetar os programas.

Ainda em relação a essa forma de programar, pesa o estigma das restrições impostas pelas linguagens nativas dos bancos, que geralmente são direcionadas e otimizadas para a manipulação de dados, e não para situações genéricas e algoritmos complexos, obrigando o programador a “costurar” para superar as deficiências, em particular no que se refere à depuração do código e tratamento de exceções.

Como pontos positivos restam as questões relativas ao ganho de performance na manipulação dos dados e a centralização do código que facilita a atualização após a manutenção, e garante a segurança do modelo de dados.

Parte das regras na interface e parte junto dos dados

Este o modelo mais facilmente encontrado atualmente e é uma tendência natural. Quando bem aplicado é um modelo eficaz e pode atender a várias situações e ambientes. No entanto carrega os estigmas da dificuldade de manutenção, dificuldade de distribuição e falta de portabilidade.

3 camadas

Quando falamos em desenvolvimento em 3 camadas, estamos nos referindo a um modelo de programação que prevê a divisão do programa em 3 partes bem definidas e distintas: interface, regras de negócio e banco dos dados.

Nossa primeira preocupação quando estamos desenvolvendo com essa visão deve ser uma preocupação constante com a expressão “bem definidas e distintas”. A divisão e não intromissão de uma camada na outra é a pedra fundamental desse modelo.

Apesar de vários autores usarem outras expressões tais como “n-camadas” e “multi-camadas”, prefiro a expressão 3 camadas, por estarmos nos referindo às camadas lógicas da aplicação. Podemos de fato subdividir as camadas lógicas em n camadas físicas, no entanto considero isso quase sempre irrelevante a nível de projeto de programa.

A interface com o usuário

A primeira regra na construção da interface deve ser a economia e simplicidade de código. A necessidade de manutenção de um programa é diretamente proporcional à sua complexidade, portanto devemos sempre nos preocupar em desenvolver interfaces simples e estáveis, já que essa é a parte do programa que deve ser distribuída para os usuários. Essa camada também é conhecida como camada **cliente**.

Só para lembrar: essa camada não necessita do BDE/SQL Links, facilitando muito o processo de instalação.

Regras de negócio

Essa camada tem a função de **servir** a camada cliente, executando processos em função de suas requisições. A “inteligência” do sistema deve se concentrar nessa camada, sendo que todo e qualquer acesso aos dados deve ser feito por essa camada.

Banco de dados

Dentro da filosofia de desenvolvimento 3 camadas, deve-se utilizar o banco de dados como um repositório, evitando-se a utilização de *triggers* e *stored procedures* com o objetivo de evitar a dispersão do código das regras e aumentar a portabilidade. Alguns projetistas chegam a evitar completamente a utilização de constraints no banco, conseguindo um alto grau de portabilidade.

MIDAS

No ambiente do Delphi, o MIDAS (?) é o que “cola” as partes, possibilitando a comunicação entre elas. Assim como tudo no Delphi, o MIDAS foi feito com o cuidado de encapsular os detalhes, deixando o programador se preocupar com questões mais relevantes para o desenvolvimento da aplicação. O MIDAS está presente no lado cliente (interface) nos componentes de conexão (TDCOMConnection, TsocketConnection, etc), e no lado servidor (regras de negócio) exercendo o papel de mediador entre a aplicação e o meio de transporte dos dados. Uma das grandes vantagens do MIDAS é que o desenvolvimento da aplicação é totalmente independente do transporte, podendo inclusive mudar de transporte sem (ou quase sem) alterações no código fonte.

Transporte dos dados

O transporte dos dados entre as camadas pode ser feito por um entre vários mecanismos, cada um com suas vantagens e desvantagens. O MIDAS pode ser usado com os seguintes mecanismos: COM/DCOM, CORBA, OleEnterprise e Socket.

- ◆ COM/DCOM – O COM (Component Object Model) e o DCOM (Distributed COM) são os mecanismos desenvolvidos pela Microsoft para o ambiente Windows. A diferença entre eles é que o COM é o padrão de comunicação entre processos rodando na mesma máquina, enquanto que o DCOM permite a comunicação entre programas que estão sendo executados em máquinas distintas. De fato o MIDAS faz uso de várias

características do COM, mesmo quando usando outros métodos. O DCOM é nativo no ambientes Windows NT/98, enquanto que alguns arquivos adicionais são necessários para sua utilização no Windows 95.

O COM é base de toda a implementação de ActiveX encontrada no ambiente Windows, e é basicamente o mesmo do OLE 1, presente no Windows 3.1.

- ◆ CORBA – Mais que um mecanismo o CORBA (Common Object Request Broker Architecture) é um padrão desenvolvido pelo (????) para comunicação entre processos. Atualmente é o mecanismo mais maduro e completo, e sua principal implementação é o Visibroker, que acompanha o Delphi nas versões Client/Server e Enterprise. É um mecanismo altamente escalável, indicado para ambientes de médio e grande porte.
- ◆ OleEnterprise – Apesar de ainda ser suportado, aparentemente o OleEnterprise está fadado a não ser usado. Foi desenvolvido pela (???) e é baseado no COM.
- ◆ Socket – A Inprise desenvolveu um mecanismo simples e muito prático ser usado com o Delphi, baseado no Winsock 2.0. Muito indicado para ambientes de pequeno e médio porte.

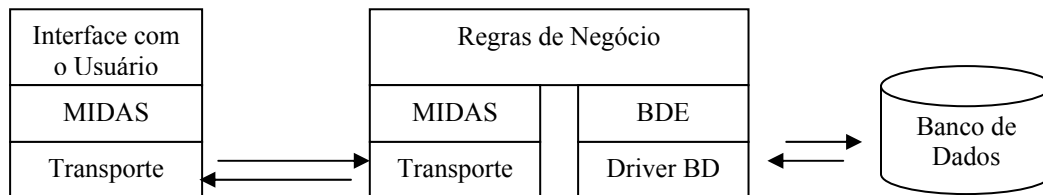


Fig.1.1 – Diagrama Geral da aplicação 3 camadas