USP – UNIVERSIDADE DE SÃO PAULO

Padrões de Arquitetura em Projetos de Sistemas Multicamadas

Cleiton Cabral dos Santos

Supervisor: José Coelho de Pina

5 de dezembro de 2005

Índice

1. Introdução

1.1 Objetivos

Esta monografia descreve a arquitetura e início de implementação do sistema *Call Web*, realizado na **Van Rooy** de junho a novembro de 2005. O foco principal é a arquitetura do sistema, sendo independente de linguagens de programação. Não se trata exatamente de um estágio porque já trabalho há sete anos nesta empresa.

Eu mesmo sugeri o desenvolvimento do *Call Web* como objeto deste estudo e foi uma boa oportunidade para a empresa poder desenvolver um trabalho de caráter acadêmico. O *Call Web* é a evolução de outro sistema da Van Rooy chamado *Call Solution*, desenvolvido em Delphi 7 com uma arquitetura cliente-servidor. Esta arquitetura em duas camadas dificulta a migração para um ambiente como a Internet. Por isso surgiu a necessidade de um sistema que fosse flexível o suficiente para que pudéssemos utilizá-lo com qualquer tipo de interface moderna (*browser* na Internet, Pocket PCs, celulares, etc.) e com qualquer banco de dados (SQL Server, Oracle, Interbase, etc.).

1.2 A empresa

A Van Rooy atua no ramo de *telemarketing* desde 1992. Possui soluções bastante abrangentes neste ramo, como *software* de controle de PABX (*Call Manager*) e gravação de ligações (*Call Corder*). O principal produto, no entanto, é o *Call Solution*, um sistema de CRM (*Customer Relationship Management*).

O termo CRM tem sido muito utilizado para definir sistemas de gerenciamento da relação com o cliente. Estes sistemas fornecem recursos que permitem a uma empresa melhorar a satisfação de seus clientes e aumentar resultados. Isto é obtido a partir de conhecimento aprofundado das suas necessidades. Muitas empresas têm investido muito neste ramo e têm obtido ótimos resultados.

Em relação à equipe técnica, a empresa possui um departamento de desenvolvimento, composto por cinco analistas-programadores, e um departamento de suporte técnico que faz testes continuamente e dá suporte a usuários dos nossos produtos.

1.3 Escopo

O projeto desenvolvido na Van Rooy nestes seis meses abrange diversos assuntos potencialmente interessantes. A própria linguagem de programação e o ambiente que escolhemos para o desenvolvimento (Microsoft Visual Studio 2003 e a linguagem C#) e o *framework* .NET são interessantes e completamente novos para mim e para a empresa. Da mesma forma, o domínio da aplicação – sistemas de CRM – seria um campo vasto para exploração.

No entanto, este estudo será concentrado apenas na arquitetura do sistema por ser um tópico interessante e de razoável complexidade. A literatura a respeito trata arquiteturas de forma geral e muitas vezes é difícil saber quais caminhos escolher. Decidir a arquitetura final do sistema *Call Web* exigiu muita pesquisa e discussão entre os membros da equipe, sendo o tópico mais valioso de todo o projeto.

É importante salientar que todas as escolhas feitas, alternativas descartadas e decisões tomadas são específicas do projeto *Call Web*, mas podem ser aplicadas a diversos outros sistemas corporativos. Na verdade, em vários pontos do projeto os requisitos tiveram soluções conflitantes, nas quais algumas qualidades foram sacrificadas em troca da melhoria de outras.

A maior fonte de inspiração para a estrutura do sistema *Call Web* foi o livro de padrões de arquitetura de Martin Fowler chamado "*Patterns of Enterprise Application Architecture*". Este livro trata de padrões de sistemas corporativos, que normalmente acessam uma base de dados. A maior parte dos padrões referidos nesta monografia são provenientes deste livro.

2. Parte Técnica

2.1 Motivações

Nesta seção, gostaria de mostrar as forças que levaram à decisão de se projetar o novo sistema que é objeto de estudo desta monografia e como ele irá suprir as novas exigências de nossos clientes.

Como mencionado na introdução, o principal produto da Van Rooy atualmente é o *Call Solution*, um sistema desenvolvido em Delphi 7 utilizando uma arquitetura cliente-servidor. Ele é composto por uma aplicação *Windows* que acessa um banco de dados relacional (ver figura 2.1). Atualmente o *Call Solution* está preparado para acessar os seguintes bancos de dados: SQL Server, Oracle, Interbase e Firebird. É um sistema monolítico, composto por um único executável de 22 megabytes.

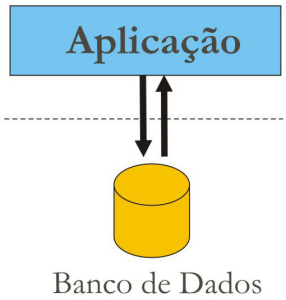

Figura 2.1 – Arquitetura do *Call Solution* (cliente-servidor)

Estas características do *Call Solution* possuem algumas desvantagens:

a) Por ter uma estrutura cliente-servidor, o sistema não é acessível através da Internet e alterá-lo para esta finalidade demandaria muitas horas de alterações.

b) A atualização de novas versões é prejudicada por causa do tamanho do arquivo executável de 22 megabytes.

c) Pequenas alterações implicam em enviar o executável inteiro novamente para o cliente.

d) Como a estrutura é cliente-servidor, todos os terminais que utilizam o *Call Solution* precisam ter conectividade com o banco de dados, exigindo que *software* específico seja instalado nas máquinas dos clientes. Isso dificulta o processo de instalação.

e) A lógica de negócio do sistema está ainda muito acoplada à interface com o usuário, diminuindo o reaproveitamento de código e tornando-o confuso.

f) Quando o sistema precisa acessar um novo tipo de banco de dados que possui sintaxe de comandos não compatível com os anteriores, é necessário refatorar o sistema inteiro para que ele aceite o novo banco. Isto se deve ao fato de cada banco de dados possuir sintaxe diferente para alguns de seus comandos básicos (SELECT, UPDATE e DELETE).

g) Quando novas rotinas de acesso a dados são adicionadas ao sistema, o desenvolvedor precisa lembrar-se de que deve programar de forma diferenciada para todos os bancos aceitos pelo sistema. Além disso, ao alterar rotinas que fazem acesso a dados, ele freqüentemente esquece de fazer as alterações para todos os bancos, o que causa erros somente em tempo de execução.

h) Os desenvolvedores precisam conhecer aspectos diferentes de diversos bancos de dados. Por exemplo, é necessário que o programador saiba que *left joins* em Oracle são feitos usando o símbolo "(+)" e em SQL Server o símbolo "*=".

Percebe-se que muitos problemas estão relacionados à conectividade com o banco de dados e outros à sua própria estrutura monolítica e pouco flexível. Esses inconvenientes atrapalham a manutenção do sistema e pequenas alterações demandam muitas horas de desenvolvimento.

Foi a partir deste cenário que surgiu a idéia de iniciar aos poucos um novo sistema, em uma linguagem de programação mais moderna. Uma arquitetura multicamadas foi a solução apropriada.

2.2 A equipe de desenvolvimento

Apenas eu e o meu colega Marcin fomos designados para o desenvolvimento deste projeto. Ambos já conhecíamos padrões de *design* (GoF, [4]) e conceitos de linguagens orientadas a objetos. No entanto, no início do projeto nenhum de nós dois tinha conhecimento da linguagem C# e do ambiente integrado do Visual Studio 2003.

Todas as etapas do desenvolvimento foram intensamente discutidas em reuniões periódicas. Dedicamos a maior parte do tempo fazendo pesquisas sobre arquitetura de *software* multicamadas, sendo que metade de cada dia foi destinada ao projeto *Call Web*. A outra metade era dedicada aos outros sistemas da Van Rooy, como o *Call Solution*.

O projeto *Call Web* foi deixado inteiramente sob minha responsabilidade. A cada duas semanas eu me reunia com meu chefe para relatar o andamento do projeto e mostrar os resultados obtidos.

2.3 O *Call Web*

O domínio da aplicação *Call Web* é o mesmo do *Call Solution*, ou seja, o ramo de CRM, conforme explicado na seção 1.3, mas com suas funcionalidades totalmente acessíveis através de um *browser*. A intenção é aos poucos migrar as funcionalidades presentes no *Call Solution* para o *Call Web*.

No entanto, reescrever todo um sistema de CRM, e ainda em uma linguagem totalmente nova, não é uma idéia financeiramente viável para a Van Rooy. Por isso, as primeiras funcionalidades serão disponibilizadas para atender algumas necessidades básicas. Por exemplo, um módulo importante do *Call Solution* é a visualização de agendamentos. Seria importante que o cliente pudesse consultar seus agendamentos e respondê-los a partir de qualquer lugar que possua uma conexão com a Internet, não necessariamente dentro da empresa. Consultores que trabalham a maior parte do tempo fora da empresa poderiam checar seus agendamentos pela Internet.

É importante ainda salientar que o *Call Web* está sendo desenvolvido em cima de um banco de dados já existente, acessando a mesma estrutura de banco de dados do *Call Solution*. Isto possibilitará que o *Call Web* seja comercializado inicialmente como um produto complementar. Os clientes que já possuem o *Call Solution* e com ele fazem os seus cadastros e agendamentos, poderão visualizá-los e editá-los na *web*.

2.4 Os padrões

Nas disciplinas sobre *design* orientado a objetos aqui do curso de bacharelado em ciências da computação sempre trabalhamos num "universo" orientado a objetos. No entanto, o próprio Fowler escreve (em [3], capítulo 18): "*Softwares* interessantes raramente vivem isolados. Até o sistema mais puramente orientado a objetos muitas vezes precisa lidar com coisas que não são objetos, tais como tabelas de bancos de dados relacionais, transações CICS e estruturas de dados XML". Os padrões descritos aqui procuram conciliar esses dois "universos".

No nível mais alto de abstração da arquitetura de *software*, Martin Fowler sugere (em [3], capítulo 1) uma divisão em **camadas**. Desta forma, o sistema é divido em partes lógicas distintas, com responsabilidades bem definidas e que se comunicam. Cada camada deve utilizar os recursos disponibilizados pelas camadas subjacentes e nunca deve "conhecer" as camadas superiores.

Existem duas definições para o termo "camada". Uma **camada física** (*tier*) representa uma máquina na qual uma parte da aplicação é executada. No exemplo de uma aplicação cliente-servidor, o banco de dados é normalmente instalado num servidor. Os clientes são normalmente máquinas menos potentes que estão numa rede local e através dela acessam o banco de dados. Neste exemplo, a aplicação está sendo executada em dois "lugares" físicos diferentes: parte dela no servidor e parte na máquina do cliente. Uma **camada lógica** (*layer*) é uma separação conceitual dentro da aplicação. Diversas camadas lógicas podem residir num mesmo programa executável e, portanto, serem executadas numa única camada física (máquina). Sempre que houver referência a camadas neste trabalho, elas serão as camadas lógicas da arquitetura.

As três camadas principais são:

a) Apresentação (PL – *Presentation Layer*)

b) Lógica de Negócio (BLL – *Business Logic Layer*)

c) Acesso a Dados (DAL – *Data Access Layer*)

Fowler usa o termo "fonte de dados" (*data source*) para a camada de acesso a dados e "domínio" (*domain*) para a camada de negócios. Preferi adotar aqui os termos que são mais utilizados atualmente no dialeto de arquitetos de sistema e suas respectivas abreviações em inglês.

A camada de **apresentação** é a responsável pela interface com o usuário. Nela são mostradas informações (*output*) e dados são coletados (*input*) para processamento nas camadas inferiores. As características desta camada dependerão do tipo de interface: *browsers*, PALMs, celulares, SmartPhones, Pocket PCs, etc. (Ver figura 2.2). A idéia é que a camada de apresentação seja o "*view*" do padrão MVC definido no livro *Design Patterns* (ver [4] para mais referências sobre MVC).

Figura 2.2 – Exemplos de tipos de interface com o usuário

A camada de **negócios** é o núcleo do sistema, na qual toda a lógica do domínio específico de cada aplicação está centralizada. É certamente a parte mais complexa de ser projetada e implementada. A sua responsabilidade é processar e validar informações.

Por último, a camada de **dados** é responsável pela persistência das informações. Normalmente será representada principalmente por um banco de dados relacional. Exemplos: Oracle, Interbase, MySQL, SQL Server, etc. (Ver figura 2.3).

Figura 2.3 – Exemplos de pacotes de bancos de dados

Uma estrutura simples em três camadas pode ser vista no diagrama da figura 2.4.

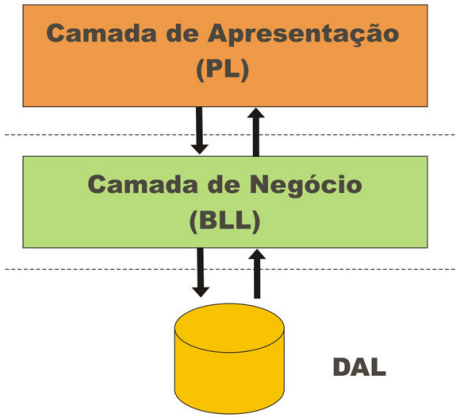

Figura 2.4 – As três camadas principais da aplicação

Normalmente, a camada de dados é composta apenas pelo banco de dados. No entanto, veremos que os padrões que adotamos aqui colocam mais responsabilidades na DAL para aumentar o seu desacoplamento da BLL.

A camada de negócios

O foco principal da nossa atenção neste ponto foi descobrir como trabalhar com a camada de negócios, que é normalmente o coração do sistema. Em [3], Martin Fowler começa a aprofundar-se nas camadas de *software* a partir da camada de negócios (BLL), pois é a camada central. Decidir o padrão da BLL influenciará na estrutura das demais camadas do sistema. Ele considera três padrões principais de arquitetura para a camada de negócios:

a) Transaction Script (110)

b) Table Module (125)

c) Domain Model (116)

O número entre parênteses após o nome é uma convenção de notação de padrões. Ele indica a página do livro no qual o padrão foi escrito.

A escolha de um dos três padrões está diretamente relacionada à complexidade do domínio da aplicação. Para aplicações mais simples, **Transaction Script** ou **Table Module** serão mais apropriados. Escolhemos o padrão **Domain Model** por ser aconselhado para domínios complexos, como é o caso do *Call Web*.

Não entrarei em detalhes sobre os padrões **Transaction Script** ou **Table Module**, pois foge do escopo deste trabalho. Entretanto, a figura 2.5 nos dá uma idéia do quanto modificar o sistema torna-se difícil conforme a lógica de domínio torna-se mais complexa, para cada um dos três padrões possíveis.

O padrão Domain Model

Como se pode perceber observando o gráfico da figura 2.5, o padrão **Domain Model** é recomendável no caso de aplicações com lógica de negócios relativamente complexa. É importante lembrar que muitas aplicações começam simples e tendem a tornarem-se complexas no decorrer de sua vida útil. Portanto, se existe a possibilidade de a aplicação crescer em um curto espaço de tempo, é recomendável a utilização de **Domain Model**. O *Call Web*, apesar de ainda possuir poucas funcionalidades, foi projetado com este padrão por termos como objetivo fazê-lo crescer a ponto de tornar-se um sistema completo de CRM.

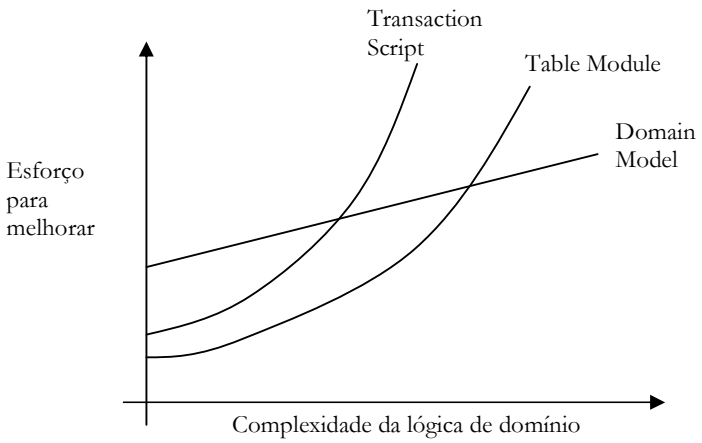

Figura 2.5 – Uma idéia da relação entre os três padrões da camada de negócio (extraído de [3])

Resumidamente, o padrão **Domain Model** prevê uma teia de objetos de negócios que estão interconectados e colaboram na definição do domínio da aplicação. Estes objetos usam herança, estratégias e outros padrões do GoF [4]. Basicamente, esta arquitetura não deve estar preocupada com a representação da camada de dados. Em **Table Module**, por exemplo, os objetos que residem na camada de negócios são praticamente mapeados diretamente de tabelas do banco de dados. A idéia em **Table Module** é que o comportamento está diretamente associado aos dados que o representam. Isto em geral não é verdade em domínios complexos que utilizam, por exemplo, herança. Não existe um equivalente direto em bancos de dados relacionais no que diz respeito à herança em orientação a objetos.

Por isso, o mapeamento dos dados contidos em tabelas no banco de dados é mais complexo no caso de **Domain Model**. Às vezes mais de uma tabela poderá ser mapeada para uma única classe ou uma única tabela poderá ser mapeada para diversas classes no domínio da aplicação.

Para se ter uma idéia, na figura 2.6 está representada a estrutura do padrão **Active Record** (160), que é mais apropriada para uma camada de negócios do tipo **Transaction Script** ou até mesmo uma **Domain Model** não muito complexa. Basicamente o que vemos é que um registro de uma determinada tabela no banco de dados está diretamente mapeado para um objeto na camada BLL.

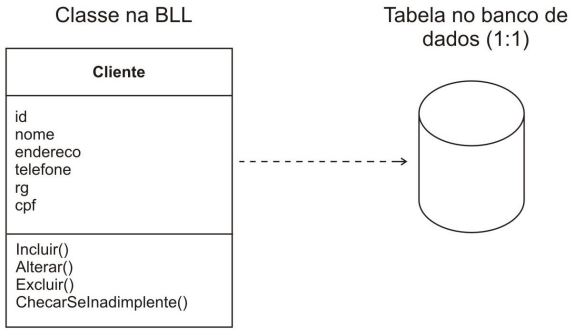

Figura 2.6 – Padrão Active Record (160). Uma classe na BLL representa um registro de uma tabela na DAL.

Active Record é ideal para domínios simples, mas para o caso do *Call Web* este padrão possui alguns problemas:

a) O objeto mapeado na camada BLL está acoplado à representação de uma tabela no banco de dados. Isto significa que a representação dos dados está acoplada à lógica de negócios (BLL e DAL misturadas).

b) Domínios complexos não evoluem facilmente com este padrão. Por exemplo, se no futuro for necessário utilizar herança nestes objetos, será difícil refatorar o código para fazê-la.

c) O objeto "sabe" interagir com o banco de dados, como por exemplo, gravar-se. Isto significa que ele possui comandos SELECT, INSERT e DELETE, operações comumente conhecidas como CRUD (**C**reate, **R**ead, **U**pdate, **D**elete). Portanto, existe um forte acoplamento entre os objetos na BLL e a DAL.

Por estes motivos, descartamos **Active Record** e optamos por **Data Mapper** (165), que é mais apropriado para domínios complexos. O próprio Fowler comenta em [3], capítulo 10: "Se o mapeamento for complexo, **Data Mapper** funciona melhor porque é melhor em desacoplar a estrutura de dados dos objetos do domínio da aplicação. Os objetos do domínio não precisam conhecer o *layout* do banco de dados".

Na figura 2.7 está representada a estrutura deste padrão. Note que agora a classe **Cliente** não possui mais responsabilidade de "se gravar" no banco de dados. Esta responsabilidade foi transferida para uma camada de "mapeadores", pertencente à camada de dados.

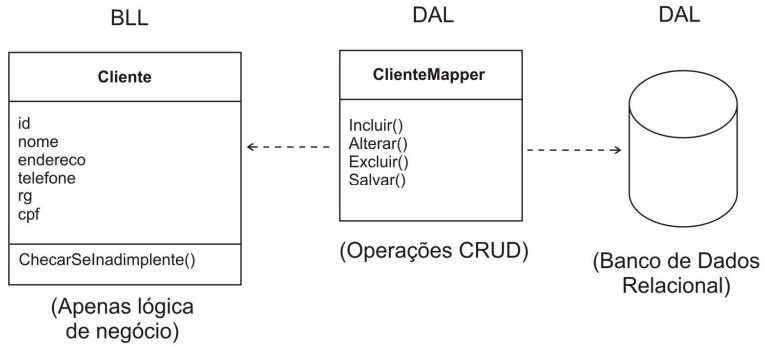

Figura 2.7 – Padrão Data Mapper (165). Total abstração do banco de dados na camada BLL.

Desta forma, se a empresa decidir trocar o banco de dados de Oracle para SQL Server, por exemplo, a única camada a ser alterada será essa nova camada de **mapeadores**, que fazem nada mais do que operações CRUD.

Na figura 2.8 está representada de forma esquemática a arquitetura do sistema após a inclusão da camada de mapeadores na DAL.

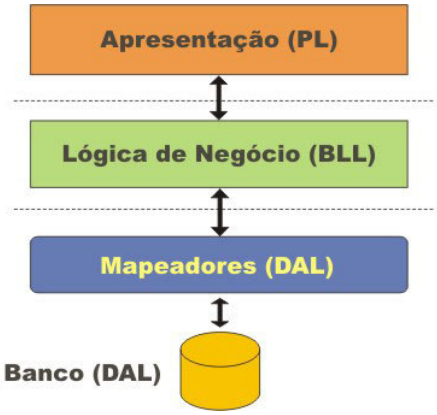

Figura 2.8 – Arquitetura usando Domain Logic (116) e Data Mapper (165).

Não foi ao acaso que chegamos a essa arquitetura. Na verdade o raciocínio que utilizamos na prática foi inverso ao que está sendo demonstrado aqui. O mesmo acontece na descoberta de teoremas em matemática! Têm-se uma intuição, sabe-se de uma verdade e procuramos mais ou menos um caminho reverso para chegar até o resultado. A seguir, falarei sobre *softwares* chamados de "mapeadores objeto-relacionais" e veremos qual o seu papel na arquitetura apresentada até aqui.

2.5 Mapeadores Objeto-Relacionais

Os *softwares* conhecidos como ORM (*Object-Relational Mapping*) estão tornando-se cada vez mais populares. Eles fazem exatamente a parte "chata" da construção da arquitetura mostrada na figura 2.8, ou seja, associar tabelas no banco de dados a classes na camada BLL, numa relação 1:1. Talvez esta seja a única parte automatizável desta arquitetura. A entrada destes *softwares* é um mapeamento, e a saída é código-fonte gerado na linguagem desejada.

Existem ferramentas ORM para diversas plataformas. Para Java, por exemplo, existe o **Hibernate**, um *software open source* (maiores informações em www.hibernate.org). Para a plataforma .NET existe o equivalente **NHibernate** (www.nhibernate.org), também *open source*. Ambos os *softwares* utilizam como entrada um mapeamento através de arquivos XML, ou seja, um arquivo no qual são informados o nome da tabela e dos campos e o respectivo mapeamento para a classe na camada de negócios.

Optamos pelo LLBLGen Pro (ver figura 2.9) por ter uma interface totalmente gráfica e por ser fácil de utilizar. Assim, simplesmente arrastando e soltando objetos é possível fazer os mapeamentos entre tabelas e objetos. Não entrarei nos detalhes de como o LLBL faz os mapeamentos, mas acho importante salientar que estudamos a estrutura interna do código gerado e ele segue diversos padrões que estão descritos no livro do Fowler. Dentre eles destacam-se: **Lazy Load** (200), **Foreign Key Mapping** (236), **Metadata Mapping** (306) e **Query Object** (316).

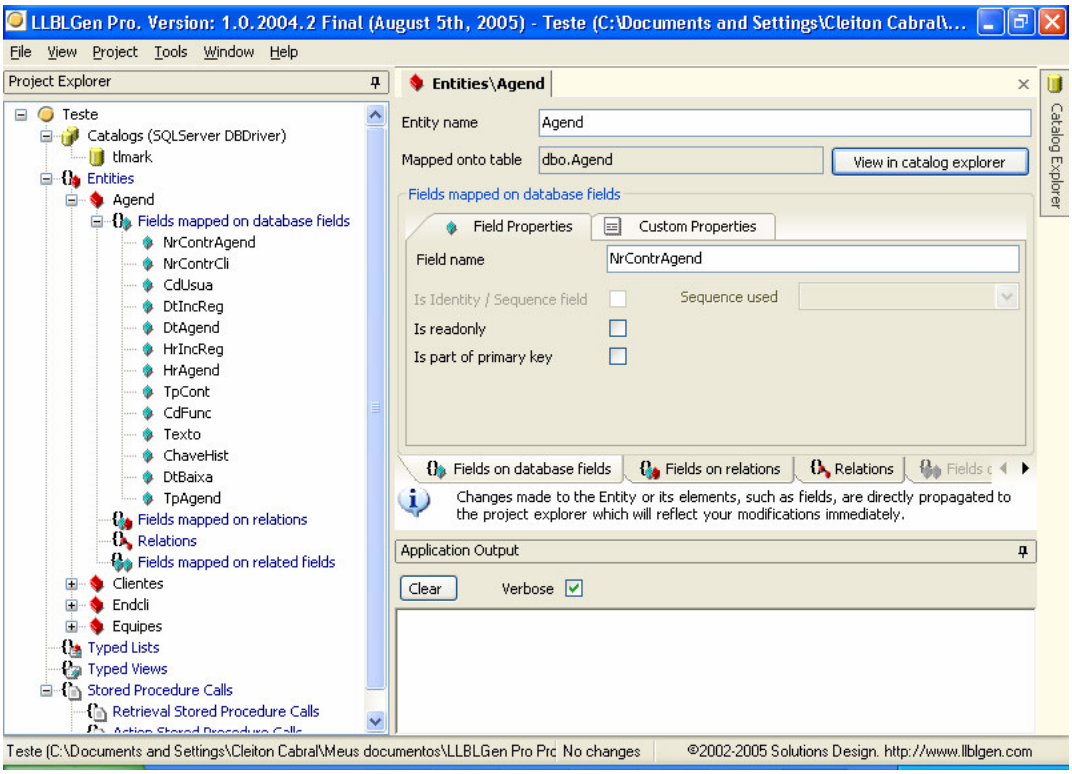

Figura 2.9 – LLBLGen Pro, um *software* mapeador objeto-relacional.

2.6 As classes geradas pelo LLBLGen Pro

O LLBLGen Pro fornece tabelas mapeadas do banco de dados numa relação 1:1. Elas são chamadas de entidades (*entities*). Por exemplo, uma tabela chamada **Clientes** no banco de dados é mapeada para uma classe **ClientesEntity**.

É muito simples utilizar as classes geradas em C#. A figura 2.10 mostra um exemplo no qual um novo cliente é criado. A classe utilizada para esse exemplo é **ClienteEntity**, que representa um registro da tabela **Clientes** no banco de dados.

```
[C#]

// Criar uma nova instância vazia.
ClienteEntity cliente = new ClienteEntity();

// Preencher atributos.
cliente.ID = "00125";
cliente.Nome = "José da Silva";
cliente.Endereco = "Rua das Rosas, 170";
cliente.Cidade = "São Paulo";
cliente.Estado = "SP";
cliente.Telefone = "5789-9912";

// Salvar. Necessário um Adapter para isso.
DataAccessAdapter adapter = new DataAccessAdapter();
adapter.SaveEntity(cliente, true);
```

Figura 2.10 – Criando um novo objeto mapeado e depois gravando-o no banco de dados.

Na figura 2.11 é mostrado o procedimento para recuperar o cliente do banco de dados. Basta passar a chave primária no construtor e depois usar um objeto Adapter para carregá-lo.

```
[C#]

DataAccessAdapter adapter = new DataAccessAdapter();
ClienteEntity cliente = new ClienteEntity("00125");
adapter.FetchEntity(cliente);
```

Figura 2.11 – Criando um objeto mapeado de um registro já existente no banco de dados.

Os relacionamentos entre as tabelas do banco de dados também são todos mapeados. Por exemplo, suponha que precisamos buscar todos os pedidos de um determinado cliente. Se as chaves estrangeiras foram corretamente definidas no banco de dados, o LLBLGen Pro criará automaticamente atributos na classe ClienteEntity para mapeá-los. Na figura 2.12 está um trecho de código que exemplifica a recuperação de pedidos de um cliente.

```
[C#]

// Primeiro, recuperar o cliente
DataAccessAdapter adapter = new DataAccessAdapter();
ClienteEntity cliente = new ClienteEntity("00125");
adapter.FetchEntity(cliente);

// Recuperar pedidos
EntityCollection pedidos = cliente.Pedidos;
adapter.FetchEntityCollection(pedidos, cliente.GetRelationInfoPedidos());
```

Figura 2.12 – Recuperando os pedidos de um cliente. Usa o padrão Foreign Key Mapping (236).

Note que o programador não precisa saber quais são os campos que compõem a chave estrangeira, o código para os relacionamentos é criado automaticamente.

Os objetos criados pelo LLBLGen Pro podem ser ligados diretamente a componentes da interface do usuário. Este recurso é chamado em .NET de *databinding*. Um *array* de "entidades" (chamadas no LLBL de *EntityCollection*), podem ser diretamente ligados (*bound*) a um componente *DataGrid* de uma página *web* em ASP.NET. Portanto, é uma tentação simplesmente usar as classes geradas para popular componentes visuais, ligando-os diretamente. Veremos mais adiante os problemas que isso pode causar e como fizemos para resolver os conflitos.

Note que o mapeamento 1:1 entre tabelas e objetos não é exatamente o padrão descrito por Fowler. **Domain Model** prevê domínios complexos, nos quais muitas vezes uma única classe poderá encapsular mais de uma tabela no banco de dados.

Por este motivo, tivemos que adaptar a arquitetura para acomodar a introdução do mapeador objeto-relacional sem acoplar o domínio da aplicação à estrutura do banco de dados. No caso do *Call Web* o problema é ainda pior porque o banco de dados que vamos utilizar possui problemas na estrutura, devido ao fato de ter sofrido diversas alterações nos últimos anos.

O tópico a seguir trata destas adaptações e foi resultado de longas reuniões e pesquisas em fóruns na Internet para chegarmos a uma conclusão.

2.7 Os padrões *versus* ORM

Após consultar diversos fóruns e observar como as pessoas utilizavam objetos gerados por mapeadores, destacamos 4 formas principais de utilizá-los. São elas:

a) Acrescentar lógica de negócio às classes "*entity*" geradas.

Esta opção, conforme pudemos observar nos fóruns de discussão da Internet, era utilizada por muitos. As classes acabavam tornando-se **Active Records**, ou seja, tabelas mapeadas que contém lógica de negócio. Vimos no capítulo 2.4 que esta alternativa não era aconselhada com **Domain Model**. O maior problema, no entanto, estava no fato de a camada de apresentação ter de lidar diretamente com estas classes geradas. A camada de apresentação estaria "enxergando" a estrutura do banco de dados. Apesar de não estar lidando diretamente com o banco de dados, a camada de apresentação precisaria conhecer os relacionamentos entre as tabelas, etc.

b) Herdar das classes "*entity*" e adicionar funcionalidades às classes herdadas.

Muitos outros usuários também usavam esta alternativa para não lidar diretamente com o código gerado. Consideramos esta a pior alternativa, pois iria inviabilizar a camada de negócios no caso de precisarmos utilizar herança para outras finalidades.

c) Utilizar as classes "*entity*" como DTOs (**Data Transfer Objects** (401))

Muitos usuários de geradores de código dizem utilizar as classes geradas como se fossem **Data Transfer Objects**. Esse padrão define objetos que não possuem nenhuma lógica, são apenas repositórios de dados que trafegam entre as camadas de negócio e de apresentação. A idéia é utilizar as classes geradas para vinculá-las diretamente à camada de apresentação, pois elas já possuem alguns mecanismos automáticos de *databinding*. Isso viola encapsulamento, não abstrai a estrutura do banco de dados e cria um acoplamento entre a PL e a DAL.

d) Encapsular as classes geradas criando uma camada adicional na BLL

Todo o código gerado pelo LLBLGen Pro é isolado numa camada denominada LLBL (*Lower Level Business Logic*). LLBL significa "camada de negócios inferior", para denominar uma camada que está logo abaixo da camada de negócios, sendo apenas uma camada de suporte. Esta camada passa a ser uma abstração do banco de dados como se ele fosse orientado a objetos. Em cima dela vem a BLL propriamente dita, que encapsula a LLBL fornecendo para a PL somente objetos puramente de negócios.

Escolhemos a última alternativa porque ela se encaixa melhor na arquitetura proposta por Fowler. Nos fóruns que consultamos, percebemos que muitos usuários fazem desta forma. Os que são contra esta alternativa usam como argumento principalmente o fato de aparecer uma nova camada na aplicação, o que poderia torná-la mais complexa. Decidimos, no entanto, que a maior complexidade traria em contrapartida mais flexibilidade e maior abstração para os usuários da camada de negócios (BLL) e de apresentação (PL).

Na figura 2.13 está representada a arquitetura final que desenvolvemos. Ela é uma adaptação do padrão **Data Mapper** que leva em consideração a geração de código através de mapeadores objeto-relacionais.

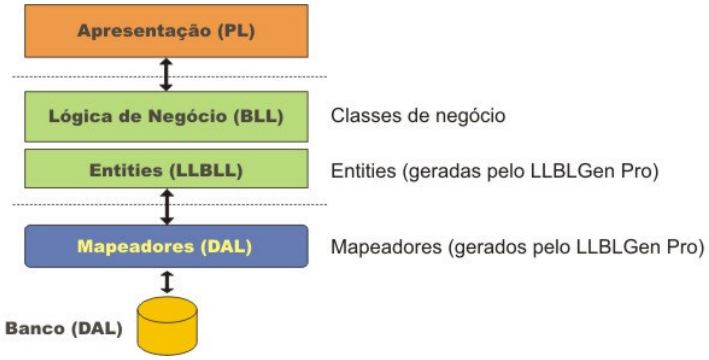

Figura 2.13 – Nossa adaptação para o padrão Data Mapper (165). Isolamento da camada LLBL.

2.8 Resultados Obtidos

A adaptação feita na seção 2.7 foi a parte mais difícil da arquitetura do sistema *Call Web* e a que nós perdemos mais tempo discutindo. Não existem livros atualmente que comentem essa necessidade específica que tivemos, então precisamos pesquisar como as pessoas estavam fazendo uso dos mapeadores e procurar a melhor solução. Os fóruns de discussão do LLBLGen Pro estão repletos de discussões fervorosas sobre como as classes geradas devem ser utilizadas e qual arquitetura adotar. Muitos acham que não é necessária uma solução "puramente orientada a objetos", mesmo porque ela não seria eficiente.

Ainda estamos no começo do desenvolvimento e não posso descrever aqui o impacto em relação ao desempenho do sistema. No entanto, já no início tivemos um ganho considerável em produtividade. O gerador de código não só automatiza os mapeamentos como também abstrai o banco de dados, permitindo que a atenção se volte para a camada de negócios e sua lógica. O programador não tem mais que se preocupar se o banco sendo utilizado é Oracle ou SQL Server, ele trabalha em cima da camada LLBL "vendo" as tabelas em forma de objetos.

Os nossos testes iniciais foram todos feitos em cima do SQL Server. Fizemos alguns testes trocando o banco de dados para Oracle e Interbase e tudo funcionou sem alterarmos uma única linha de código. Tudo o que precisamos fazer foi entrar no LLBLGen Pro, selecionar o banco e os dados, clicar no botão "*Generate*" e tudo já estava pronto para funcionar.

2.9 Atividades realizadas

Durante o mês de **junho** deste ano, eu e o Marcin aprendemos uma linguagem de programação totalmente nova (C#) e o ambiente de desenvolvimento Visual Studio 2003. É claro que o processo de aprendizagem continuou nos meses seguintes, mas toda a parte introdutória foi adquirida neste período.

O Marcin fez um curso de imersão de duas semanas numa empresa de consultoria no qual aprendeu sobre as ferramentas de desenvolvimento e também sobre padrões de projeto na Internet. Estes padrões foram ensinados como sendo reconhecidos no mercado. Como não tínhamos nenhum conhecimento anterior, aquela realmente nos pareceu uma boa solução.

Quanto a mim, aprendi a linguagem principalmente com o livro "*C# .NET Web Developer's Guide*" [1]. Além disso, o meu colega Marcin me passou muitas informações que ele adquiriu no curso de imersão.

Em **julho** começamos a fazer alguns testes de como seria o *Call Web* segundo a arquitetura sugerida pela consultoria. Após desenvolvermos algumas telas, comecei a perceber que aqueles padrões não pareciam muito elegantes. Até aquele momento, no entanto, era apenas uma sensação. Foi então que resolvi começar a ler "*Patterns of Enterprise Application Architecture*", do Fowler [3]. Eu já tinha ouvido falar do livro e da reputação do autor na disciplina "Tópicos Avançados de Programação Orientada a Objetos", mas não o tinha lido.

Agosto e **setembro** foram os meses em que mais aprendi sobre C#, sobre Visual Studio e sobre padrões de arquitetura. Comecei a ler a teoria do livro e notar que os padrões que foram sugeridos como "certos" pela empresa de consultoria, na verdade não funcionariam para nós. Talvez eles até funcionem para sistemas

muito pequenos, mas não para o nosso caso. O meu colega estava relutante em abandonar as idéias que aprendeu no curso, mas ele se interessou pelo livro também e concordou que havia algo de errado no que ele tinha aprendido. Foram dois meses de intensas discussões, reuniões, consultas em fóruns na Internet, etc.

Em **outubro**, depois de termos analisado diversos pontos de vista, chegamos à conclusão de como deveria ser a arquitetura final do sistema *Call Web*. Iniciamos o desenvolvimento.

Novembro foi dedicado inteiramente à implementação das funcionalidades básicas do *Call Web*: consulta e inclusão de agendamentos e cadastro de clientes. Neste período também atualizamos o Visual Studio para a versão 2005, mas ainda não tivemos tempo para explorar todas as novidades da versão 2.0 do .NET. Também neste mês comecei a escrever a monografia.

3. Parte Subjetiva

3.1 Desafios e Frustrações

O maior desafio deste projeto foi aprender tanta coisa em tão pouco tempo. Em apenas seis meses eu aprendi uma linguagem de programação inteiramente nova, o ambiente de desenvolvimento Visual Studio e o mais valioso: projetar a arquitetura de um sistema como o *Call Web*.

Houve momentos, nos meses de agosto e setembro, em que as nossas escolhas eram conflitantes entre si, havia muitas alternativas e eu cheguei a ficar completamente perdido sobre qual delas seria a mais apropriada para o nosso projeto. Gastei diversas horas em fóruns de discussão para tentar captar outros pontos de vista, problemas encontrados e suas respectivas soluções. Ter de pensar e chegar a uma conclusão enriqueceu muito nosso conhecimento sobre o assunto.

Não acho que tenha havido frustrações. Não foram estabelecidos prazos e meu chefe deu total liberdade para o desenvolvimento da arquitetura e estudo das tecnologias. Por parte da empresa, foi depositada total confiança na equipe e no projeto.

3.2 Interação da equipe

A troca de idéias entre o meu colega e eu foi de vital importância. Muitas vezes o que parecia muito certo para mim era rapidamente apontado como problema por ele e vice-versa. Nos finais de semana cada um lia algo sobre os padrões do Fowler e na segunda-feira nos reuníamos para discutir os tópicos relevantes.

Percebi que a linguagem de padrões era muito útil nas nossas conversas. Falávamos sobre DTOs, Row Data Gateways, etc., e isto ajudava bastante nas nossas análises.

3.3 O IME e o Projeto

O IME foi muito importante para a base teórica do projeto *Call Web*. O meu primeiro contato com padrões tinha sido na disciplina "Tópicos Avançados em Programação Orientada a Objetos" (MAC413), que cursei antes de "Programação Orientada a Objetos" (MAC441). Desde então, padrões de *software* têm sido a minha maior área de interesse.

3.4 Disciplinas Relevantes

- **MAC122 (Princípio de Desenvolvimento de Algoritmos)**

  Importante para conhecer as formalidades de desenvolvimento de algoritmos.

- **MAC242 (Laboratório de Programação II)**

  Foi o meu primeiro contato com programação em três camadas na *web* utilizando JSP.

- **MAC332 (Engenharia de Software)**

  Esta disciplina foi muito importante para o conhecimento de técnicas de desenvolvimento de *software*, muitas das quais apliquei no meu ambiente de trabalho.

- **MAC413 (Tópicos Avançados de Programação Orientada a Objetos)**

  Nesta disciplina tive o primeiro contato com padrões de *design* e padrões de arquitetura de *software*. Motivou-me a cursar MAC441.

- **MAC441 (Programação Orientada a Objetos)**

  Considero esta disciplina uma das mais importantes atualmente, devido à grande popularidade de linguagens orientadas a objetos. Aqui vimos os padrões de *design* com mais detalhes.

- **MAC446 (Princípios de Interação Homem-Computador)**

  Esta foi outra matéria que gostei bastante por mostrar uma metodologia de desenvolvimento que eu nunca tinha imaginado antes. Isto mudou muito minha percepção como programador. Antigamente eu costumava achar que o usuário de um sistema era "burro" por não conseguir fazer corretamente uma

determinada tarefa. Hoje eu penso que se o usuário errou, muito provavelmente a interface foi mal feita ou conceitos do sistema foram mal definidos.

3.5 O futuro do Projeto

O projeto *Call Web* continua evoluindo e já existem diversos módulos que estão sendo elaborados para implementação. A intenção é que a partir de abril de 2006, o *Call Web* já seja oferecido como um sistema opcional para que os clientes possam ter acesso às suas informações pela Internet.

Os padrões da camada de apresentação de [3] não foram muito explorados por falta de tempo, mas pretendemos estudar e seguir os padrões que estão definidos no capítulo 14 ("*Web Presentation Patterns*").

Outro assunto que pretendo explorar foi motivado durante o curso da disciplina MAC446 (Princípios de Interação Homem-Computador). Interessei-me bastante sobre desenvolvimento orientado à interação com o usuário. Por isso, pretendo ler o livro "*Designing Web Usability*" de Jakob Nielsen [5], que fala sobre como projetar interfaces voltadas para a *web*.

3.6 Considerações Finais

Uma das maiores lições que tirei deste projeto foi a importância de se questionar mesmo aquelas decisões que parecem ser as melhores possíveis. Contratamos a empresa de consultoria para nos ensinar padrões de desenvolvimento que, segundo eles, eram os melhores e mais bem planejados. No entanto, logo no início notei que havia algo errado. Eu não tinha nenhum conhecimento para refutar o que havia sido ensinado, por isso tive que pesquisar bastante e convencer a equipe de que havia uma solução melhor.

É claro que muitas vezes é difícil mudar metodologias há muito enraizadas numa corporação, mas não é impossível. Se você conseguir fazer isso, será certamente muito respeitado.

4. Bibliografia

[1] A. Turtschi *et al.*, *C# .NET Web Developer's Guide*. Syngress Publishing, Inc, Maryland, USA, 2002.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, *Pattern-Oriented Software Architecture – A System of Patterns*. John Wiley and Sons Ltd, Chichester, UK, 1996.

[3] M. Fowler, *Patterns of Enterprise Software Architecture*. Addison Wesley, USA, 2002.

[4] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, USA, 1994.

[5] J. Nielsen, *Designing Web Usability: The Practice of Simplicity*. New Riders Publishing, USA, 2000.

[6] *Site*: www.msdn.com. *Microsoft Developer's Network*.