Uma Experiência na Utilização de UML, Padrão de Projeto e Servlet

Valter Vieira de Camargo [*]
Gustavo A. Prieto [*]
Rosângela D. Penteado

Universidade Federal de São Carlos – UFSCar
Departamento de Computação
Caixa Postal 676
13565-905 – São Carlos – Sp- Brasil

{valter, prieto, rosangel}@dc.ufscar.br

Resumo

O crescente número de aplicações para Internet, devido às vantagens e facilidades que ela fornece, faz com que pesquisas constantes sejam realizadas. Muitas dessas aplicações utilizam banco de dados relacional. O conflito existente entre o paradigma orientado a objetos e a persistência de dados em um banco relacional, consome tempo e recursos dos projetistas. Uma solução, nesse caso, é a utilização do padrão *Persistence Layer*, uma camada que cuida da interface entre os objetos da aplicação e o banco de dados, protegendo-os de mudanças constantes. Este trabalho apresenta um sistema exemplo especificado em UML e disponibilizado na Internet, sendo sua implementação realizada com a linguagem Java e *servlet*, o padrão Persistence Layer e o banco de dados relacional SyBase.

Palavras-chave: Orientação a Objetos, *Persistence Layer*, Java , *Servlet*, Internet, *SyBase*.

[*] Trabalho realizado com o apoio financeiro da CAPES.

1. Introdução

A preocupação com o desenvolvimento de sistemas é constante para os engenheiros de software. Com o surgimento da Internet acentuou-se a concorrência entre as empresas para a venda de seus produtos por meio dela devido às facilidades que apresenta.

A tecnologia utilizada para a construção de aplicações para Internet está em constante evolução, visando oferecer melhores serviços aos seus clientes, ou seja, aplicações mais rápidas, mais confiáveis e independentes de plataformas. Para a implementação dessas aplicações algumas tecnologias disponíveis são:

1. CGI (*Common Gateway Interface*) é a mais difundida e é suportada pela maioria dos servidores *Web*. O servidor comunica-se com o programa CGI utilizando variáveis de ambiente e fornece as entradas do usuário por meio de parâmetros de linhas de comando. Cantú [15] apresenta as seguintes variantes para CGI: a) WinCGI – que utiliza um arquivo com extensão INI do Windows para a passagem de informações; b) ISAPI/NSAPI DLLs - são bibliotecas ligadas ao *Microsoft Internet Server* e ao *Netscape Internet Server*, respectivamente. Essa tecnologia utiliza um arquivo com extensão DLL que fornece funções específicas chamadas por um servidor.
2. ASP (*Active Server Pages*) são *scripts* interpretados por um servidor Internet, escritos em *VBScript* e executáveis somente no ambiente Microsoft [10].
3. *Applets* são programas Java executados por um *browser* [9].
4. *Servlet* é uma versão *Applet* que é executada pelo servidor e é mais eficiente que CGI.

O interesse em desenvolver sistemas orientados a objetos que sejam utilizados via Internet, é crescente. A construção de páginas HTML deve ser dinâmica pois os dados atualizados devem estar disponíveis a qualquer momento para os usuários.

Paralelamente os padrões de projeto (*design patterns*) [1,2,3], têm auxiliado os projetistas no desenvolvimento de sistemas orientados a objetos. Yoder [1] apresenta o padrão *Persistence Layer* como uma forma otimizada para uma implementação que utilize uma linguagem orientada a objetos e banco de dados relacional, minimizando os conflitos existentes entre esses dois paradigmas.

O enfoque principal deste trabalho é um estudo sobre a viabilidade de aplicações para Internet que utilizem *Java servlet* [4, 6, 12], banco de dados relacional *Sybase* [11], padrão *Persistence Layer* [1] e UML [18], sem ainda, a preocupação com o desempenho do sistema. Um sistema exemplo que trata do atendimento de clientes em uma loja de conveniência, é utilizado. UML é utilizada para a especificação desse sistema. Os diagramas de classes especificam tanto as classes da aplicação bem como as que devem ser persistidas no banco de dados relacional.

Este trabalho está organizado da seguinte forma: na Seção 2 o conceito de padrões de projeto e o padrão *Persistence Layer* são apresentados; na Seção 3 discute-se as características da tecnologia *servlet* bem como o seu comportamento e vantagens. Na Seção 4 o sistema exemplo proposto e a sua solução são apresentados e, finalmente na Seção 5 são feitas as considerações finais.

2. Padrões de Projeto

O uso atual da palavra “Padrão” vem dos trabalhos do arquiteto Christopher Alexander [2] que escreveu vários livros sobre planejamento urbano e arquitetura de construções. Apesar de seus escritos serem sobre construção e arquitetura, os seus princípios podem ser aproveitados para várias disciplinas, incluindo desenvolvimento de software. Na Engenharia de Software, padrões de projeto são utilizados para auxiliar desenvolvedores e projetistas na resolução de problemas recorrentes encontrados em qualquer desenvolvimento de software. Entre as vantagens de se utilizar padrões pode-se citar: aumento de produtividade, uniformidade na estrutura do software, padronização no desenvolvimento de software, aplicação imediata por outros desenvolvedores e redução da complexidade do sistema.

Gamma e outros [3] definem padrões de projeto como “a descrição planejada de objetos e classes interconectados para a resolução de um problema genérico de projeto em um contexto em particular”. Um padrão de projeto nomeia, identifica e abstrai os aspectos chave de uma estrutura de projeto identificando-a para criação de objetos reutilizáveis. Nesse contexto, eles são independentes da tecnologia e da arquitetura, dependendo somente do domínio do problema e do contexto em que o mesmo acontece.

Yoder e outros [1], afirmam que desenvolvedores de sistemas orientados a objetos que utilizam bancos de dados relacionais despendem muito tempo e esforço na implementação de mecanismos que possibilitem tornar persistentes os objetos. Devido a conflitos existentes entre os dois paradigmas, uma solução, para a questão da persistência dos dados da aplicação, é a implementação do padrão *Persistence Layer* [1]. Trata-se de uma camada que cuida da interface entre os objetos e o banco de dados, protegendo tanto a aplicação como o banco de dados de mudanças constantes. Yoder define um conjunto de dez padrões, que podem ser utilizados para a implementação dessa camada de persistência, porém, neste trabalho, são utilizados apenas os padrões descritos a seguir e que interagem entre si como mostra a Figura 1:

Persistence Layer: determina uma camada que provê a gravação (persistência) e recuperação de objetos em um banco de dados relacional.

CRUD (*Create, Read, Update and Delete*): determina as operações para a persistência de objetos: criação, leitura, atualização e eliminação.

***SQL Code Description*:** define o mecanismo de geração das cláusulas SQL que são necessárias para implementar os métodos CRUD para cada objeto a ser persistido.

Table Manager: permite o mapeamento de um objeto para a sua respectiva tabela no banco de dados.

Connection Manager: efetua a conexão do sistema com a base de dados e garante a sua manutenção.

Em [1] são sugeridas três formas para implementação do Padrão *Persistence Layer:*

- Implementar as operações do padrão CRUD nas classes da aplicação que possuem tabelas correspondentes no banco de dados, ou seja, cujos objetos devem ser persistidos;
- Criar classes específicas, na camada de persistência, correspondentes a cada classe da aplicação, que possui tabela equivalente no banco de dados, para implementar as operações do padrão CRUD;

- Criar uma classe *Broker*, na camada de persistência, para implementar métodos que realizam o mapeamento de qualquer tipo de objeto para o banco de dados.

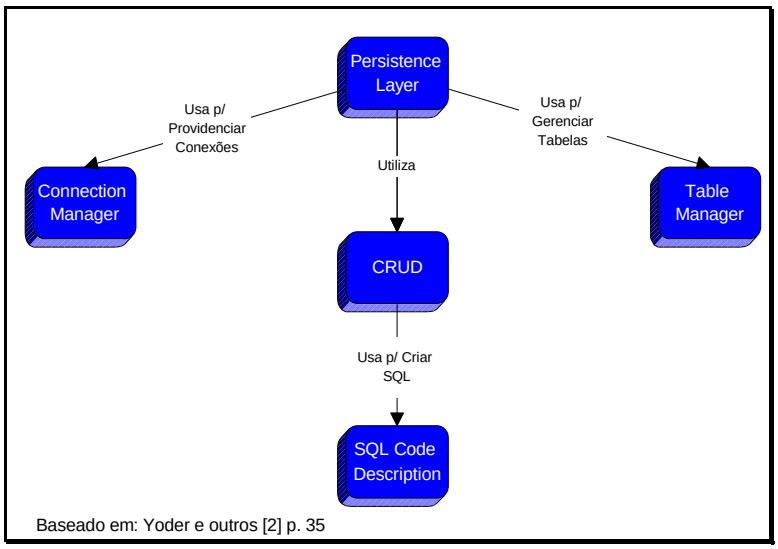


Figura 1 - Diagrama de Interação dos Padrões

Cagnin [16, 17] afirma que a primeira forma é mais simples do que as outras duas e sua utilização acarreta em economia de classes. A segunda forma de implementação necessita de menos esforço para o entendimento do código e para a manutenção, uma vez que a camada de persistência está desacoplada da camada da aplicação, isso entretanto provoca um aumento no número de classes de persistência. A terceira é a mais complexa e exige maior conhecimento do engenheiro de software para sua implementação, porém há redução no número de linhas de código, já que a classe Broker é a responsável pela implementação dos métodos que realizam a persistência dos objetos no banco de dados. Neste trabalho, a primeira forma proposta por Yoder é utilizada.

3. *Servlet*s

A linguagem de programação Java [4] é orientada a objetos, portável, multithread, segura, eficiente, fácil de aprender e permite o acesso a banco de dados via Internet através da utilização de *servlet*s. Em um ambiente cliente/servidor, como é o caso da Internet, as classes Java *servlet* permitem o tratamento de requisições/respostas dos/aos clientes ligados à rede.

Considerando-se que aplicações para a Internet são baseadas no paradigma cliente-servidor, *servlet* [4], é uma versão *applet* do lado do servidor, isto é, um trecho de código Java que é executado por um servidor *Web* e usado para tratar as requisições de clientes, enquanto os *applets* são executados pelo browser do cliente. *Servlet*s são persistentes, independentes de plataforma e incorporam todo conjunto de características avançadas como segurança, facilidade de acesso a banco de dados e integração com *applets* Java. *Servlet* é um meio de estender a funcionalidade do servidor do mesmo modo que um *script* CGI faz, contudo, seu processamento utiliza menos memória melhorando significativamente a performance da aplicação.

Bergesten [6] e Brookwood [7], afirmam que ao utilizar *scripts* CGI, um novo processo é criado a cada nova requisição do cliente, fazendo com que um servidor, que seja precário em recursos, se torne extremamente lento. Já no caso do *servlet*, uma única instância é criada e cada requisição é tratada como uma *Thread* separada. Sempre que se visa um ganho de performance [14], utiliza-se *servlet* em vez de scripts CGI.

A linguagem Java tem uma importante regra [5]: "escreva uma vez, execute em qualquer lugar". Essa regra faz referência à portabilidade da linguagem Java que é conseguida por meio da utilização de uma tecnologia conhecida como Máquina Virtual Java (*Java Virtual Machine* - JVM). Essa tecnologia é um módulo residente na maioria dos *browsers* existentes que tem a função de interpretar classes Java.

A Figura 2 mostra a interação da Máquina Virtual Java com as aplicações Java, sistemas operacionais e hardware. Essa Máquina permite que aplicações Java sejam executadas por meio do hardware em qualquer plataforma de sistema operacional. As aplicações Java fornecem arquivos com extensão ".class" para a Máquina Virtual. Essa por sua vez, interpreta-os gerando arquivos ".exe" que são executados por diversos sistemas operacionais e pelo hardware.

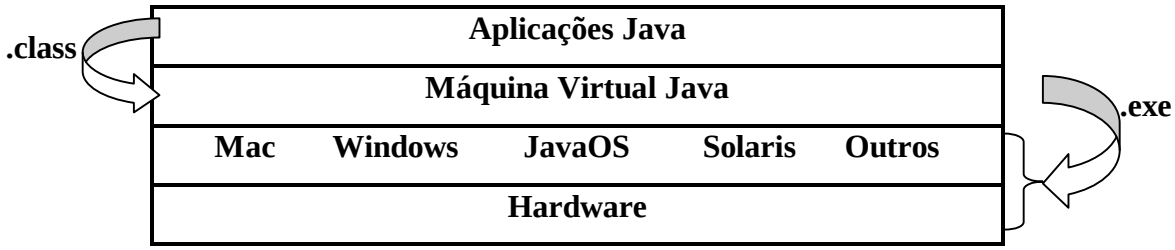


Figura 2 – Máquina Virtual Java

Bergsten [6] afirma que um *servlet* é uma classe Java que necessita ser executada em uma Máquina Virtual Java por meio de um serviço chamado de Máquina *Servlet*, que pode ser nativa do servidor *Web* ou adicionada a esse como um módulo separado. Esse serviço carrega o *servlet* na memória do computador, na primeira vez que ele é requisitado, ou opcionalmente quando a Máquina *Servlet* é inicializada. O *servlet* fica então residente na memória para tratar as múltiplas requisições até que a Máquina *Servlet* seja finalizada.

O paradigma requisição-resposta (*request-response*) é modelado sobre o comportamento do Hyper Text Transfer Protocol (HTTP). Um programa cliente, que pode estar em um *browser Web*, ou algum outro programa que pode fazer conexões através da Internet, tem acesso a um servidor *Web* e pode fazer uma requisição. Essa é, então, processada pela máquina *servlet* retornando uma resposta para o cliente na forma HTTP.

As vantagens do *servlet* [4, 8] sobre os mecanismos comuns de extensão de servidores são: a) maior rapidez que os *scripts* CGI porque usam um modelo de processo diferente, isto é, usam um padrão API que é suportado por muitos servidores *Web*; b) tem todas as vantagens da linguagem Java, incluindo facilidade de desenvolvimento e independência de plataforma e podem acessar um conjunto de APIs disponíveis para a plataforma Java. Um *servlet* implementa a interface `javax.servlet.Servlet`, com três métodos que definem o ciclo de vida do *servlet* [4]:

- `Init()` – executado uma única vez quando o *servlet* é carregado na Máquina *Servlet*.
- `Service()` – executado para tratar uma requisição, pode ser chamado várias vezes até que o *servlet* seja finalizado. Como cada requisição gera uma *thread,* esse método pode ser executado em paralelo para que atenda a todas as requisições.
- `destroy()` – invocado uma única vez quando o *servlet* é descarregado.

Após o *servlet* ter sido carregado pela Máquina *Servlet* ele deve ser inicializado, podendo ler qualquer dado persistente que esteja armazenado, deve inicializar as conexões de banco de dados JDBC e estabelecer referências para outros recursos que despendam um razoável processamento.

A Máquina *Servlet* [4] não precisa manter o *servlet* carregado por um período de tempo específico, nem é dependente do tempo de vida do servidor, ela está livre para usá-lo ou retirá-lo a qualquer momento da memória. Quando a Máquina *Servlet* determina que um *servlet* deve ser destruído (por exemplo, se a Máquina é descarregada ou necessita de recursos), ela deve permitir que ele libere os recursos que estão sendo usados e que salve os dados persistentes. Para isso, a máquina invoca um método chamado `ServletDestroy().`

4. Sistema Exemplo e sua Solução

Para ilustrar o uso de *servlet* e padrões de projeto, um estudo de caso extraído de [13] é utilizado. O sistema controla pedidos de clientes de uma loja de conveniência que efetua vendas através de um *site* na Internet. O cliente pode efetuar o seu cadastro, atualizar os seus dados, consultar os produtos disponíveis e efetuar compras com o seu cartão de crédito. Após o pagamento de uma determinada compra, o gerente avalia o pedido, liberando-o ou não. Os pedidos liberados são enviados ao cliente pelo correio. A aplicação permite ainda que um gerente controle o cadastro de produtos e de clientes, efetuando eliminações e atualizações conforme o necessário.

A notação UML [18] é utilizada para especificar e documentar o sistema exemplo. O Quadro 1 mostra os Use Cases que tratam de Clientes, Compras e Produtos.

Quadro 1 – Lista de Use Cases do Sistema

Nº	Descrição	Evento	Use Case	Resposta
1	ClienteSolicitaCadastro	DadosCliente	CadastrarCliente	Msg01
2	GerenteExcluiCliente	Cpf	ExcluirCliente	Msg02
3	ClienteFazCompra	DadosCompra	RegistrarCompra	Msg03
4	GerenteCadastraProduto	DadosProduto	CadastrarProduto	Msg04
5	ClientePagaCompra	DadosPagamento	PagarCompra	Msg05

A Figura 3 mostra o diagrama parcial de classes, de acordo com as sugeridas em [13]. A compra do cliente consiste de Item de Compra que por sua vez contém um Produto. Por questão de espaço não são exibidos todos os modelos UML, embora tenham sido desenvolvidos pelos autores.

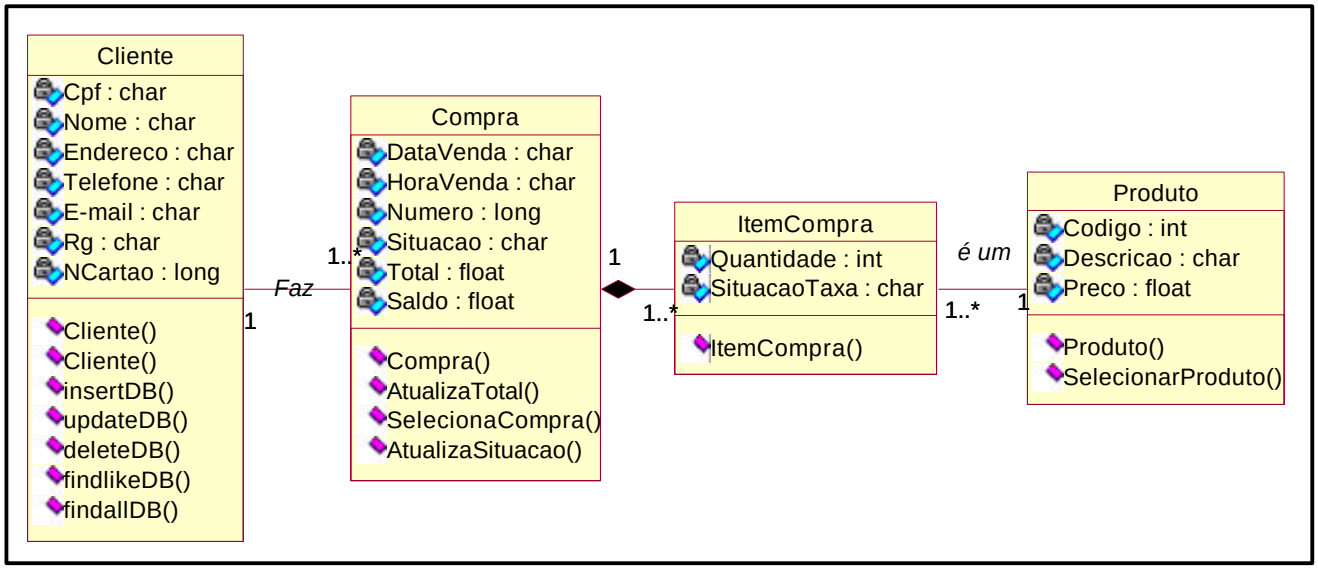


Figura 3 - Diagrama de Classes do Sistema de Loja de Conveniência via Internet

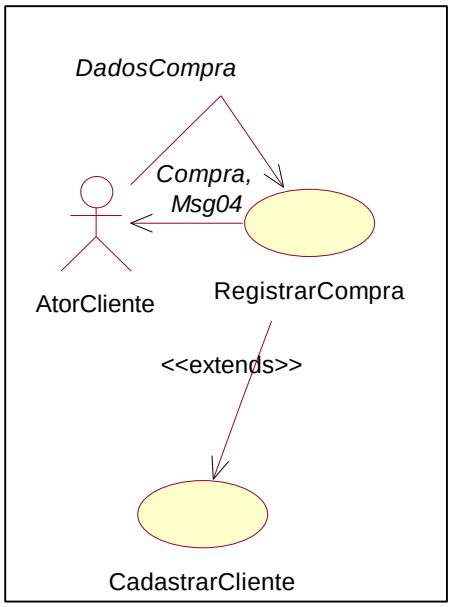
O banco de dados relacional SyBase [11] é utilizado para realizar a persistência dos objetos das classes existentes no sistema. O esquema lógico do banco de dados é mostrado no Quadro 2, e refere-se às classes existentes no Diagrama de Classe mostrado na Figura 3.

Quadro 2 – Esquema Lógico do Banco de Dados do Sistema Implementado

Cliente = (Cpf, Nome, Endereco, Fone, Email, Rg, NroCartao)
Produto = (Codigo, Descricao, Preco)
ItemCompra = (Numero, Codigo, Quantidade)
Compra = (Numero, DataVenda, HoraVenda, Situacao, Total, Saldo, Cpf)

O Use Case RegistrarCompra da Figura 4a e o diagrama de seqüência correspondente, da Figura 4b, ilustram o procedimento realizado por um cliente quando esse efetua uma compra, desde que ele já esteja cadastrado, isto é, seu Cpf já exista no banco de dados.

Para a implementação desse sistema foi utilizada a primeira forma do Padrão de Projeto *Persistence Layer* [1] sugerida por Yoder. Todas as classes da camada de aplicação são persistidas no banco de dados e são herdeiras da classe `PersistentObject`. Essa classe possui apenas as assinaturas dos métodos definidos no padrão CRUD. As classes da camada de aplicação estão associadas à classe `ConnectionManager` que efetua a conexão com o banco de dados. A Figura 5 exibe o diagrama de classe com as camadas de persistência e de aplicação utilizado neste trabalho.



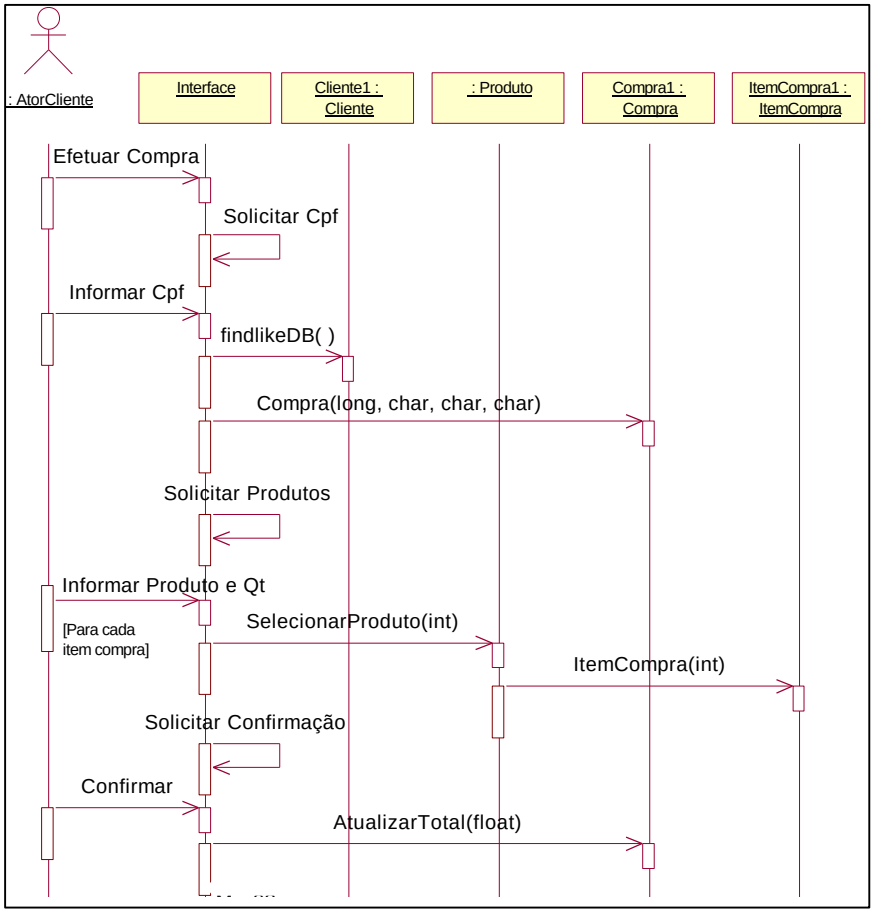


Figura 4a – Exemplo de Use Case **Figura 4b – Exemplo de Diagrama de Seqüência**

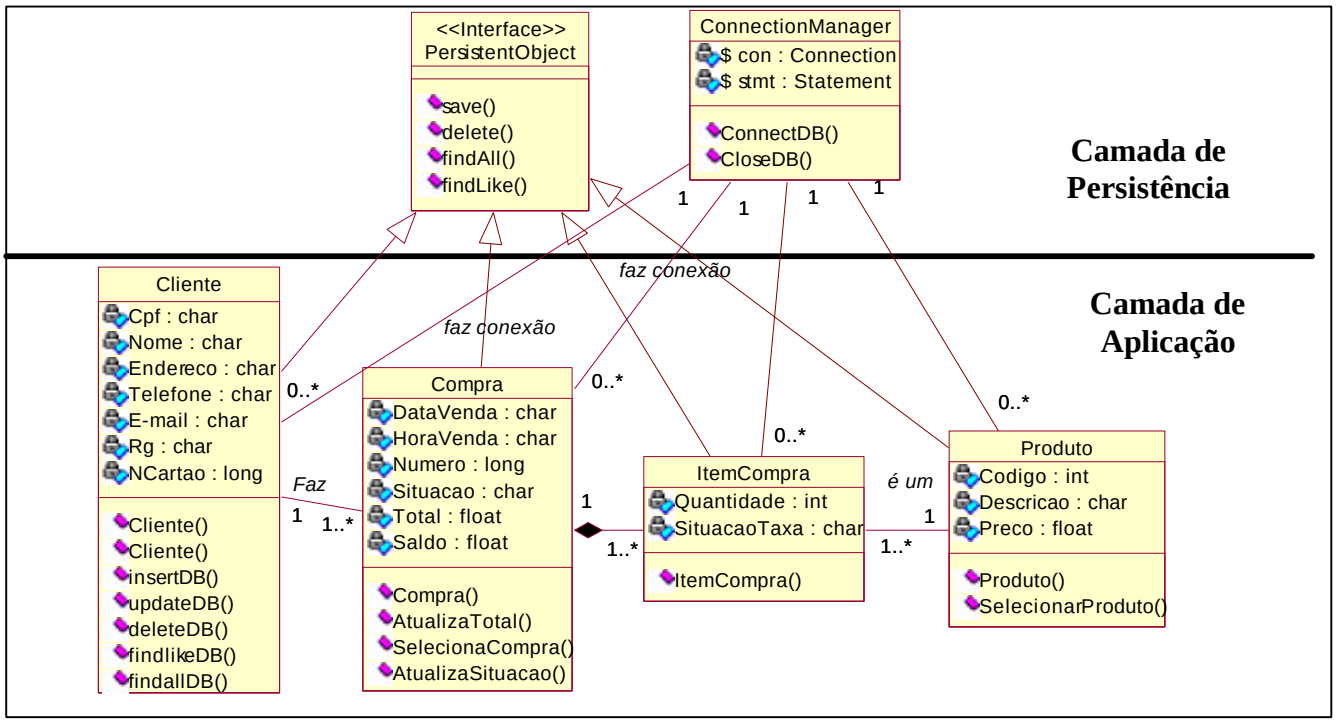


Figura 5 – Classes para o Primeiro Modo de Implementação do *Persistence Layer*.

A Figura 6 apresenta a tela para o cadastro de clientes quando o sistema exemplo é executado. Ao se pressionar o botão "Incluir", o método `POST`, do código HTML, invoca o método `doPost` do *servlet* `srvCadastraCliente.class` a fim de criar uma nova instância para um novo cliente.

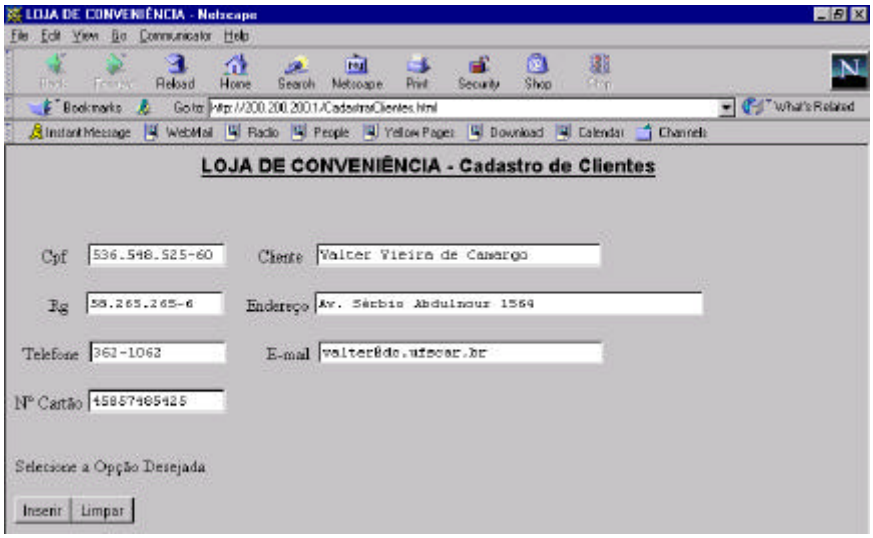


Figura 6- Tela de Cadastramento de Clientes

A invocação do método `doPost` do *servlet* é mostrado, em negrito, na Figura 7, e o corpo desse método é apresentado na Figura 8. As entradas fornecidas pelo cliente são passadas como parâmetro através do método `POST` do código HTML (Figura 7). O *servlet* `srvCadastraCliente` importa diversas classes, entre as quais, a ***`ConnectionManager`***, que possui o método responsável por tratar a conexão com o banco de dados, e a `Cliente`, responsável pela inicialização e armazenagem dos dados desse tipo de objeto.

```
<body bgcolor="#C0C0C0"> <p align="center"><u><font face="Arial"><strong><big>
LOJA DE CONVENIÊNCIA - Cadastro de
Clientes</big></strong></font></u></p>
<form action="/servlet/vq.demos.srvCadastraCliente" method="POST">
   . . .
<input type="submit" value="Inserir"><input type="reset"  value="Limpar"></p>
</form> </body> </html>
```

Figura 7 - Trecho de código da página Html que invoca o *servlet* srvCadastraCliente.

O método `doPost`, pertencente ao *servlet* `srvCadastraCliente`, invoca o método `Save()` que pertence à classe `Cliente`, como é indicado pela letra (a), na Figura 8. Já o corpo do método `Save()` é mostrado na Figura 9 e indicado pela letra (a) . Esse método tem por objetivo inserir (persistir) um novo cliente no banco e isso é feito por meio de dois outros métodos, `findlike()`e `insertDB()`. Quando o método `Save()`é executado ele invoca o método `findlike()`, redefinido na classe `Cliente`, a fim de verificar se já existe um registro no banco relacional. Caso o retorno seja verdadeiro o método `insertDB()`, da classe `Cliente`, é invocado a fim de inserir o novo cliente no banco. Os métodos `findlike()` e `insertDB()` podem ser visualizados na Figura 9 indicados pelas letras (b) e (c), respectivamente. Após o cadastramento do cliente, o *servlet* cria uma página HTML dinâmica e a exibe, isso pode ser visto através do trecho de código na Figura 8, indicado

pela letra (b). A assinatura dos métodos `save()`, `delete()`, `findLike()` e `findAll()` são definidos pela interface `PersistentObject` e o corpo dos mesmos é implementado na classe `Cliente` conforme a Figura 9.

```
public class srvCadastraCliente extends
HttpServlet {
...
public void init(ServletConfig config)
                     throws ServletException
{
   super.init(config);

 // Efetua a conexão com o banco de dados
 ConnectionManager.ConnectDB ();   }

 // Processa a requisição do usuário.
public void doPost(HttpServletRequest
request,
      HttpServletResponse response) throws
      ServletException, IOException {
    Cliente Cliente1;
    ErrorMessage = "Cadastro bem sucedido!";
    ErrorCode = true;
    Cliente1 = new
    Cliente(request.getParameter("Cpf"),

       . . .

    //Persiste o objeto cliente no banco de
dados.
    ErrorCode = Cliente1.save ();}(a)
    // Verifica excessões.
    if (ErrorCode == false) {
      response.sendRedirect

("http://200.200.200.1/ClienteJaExiste.htm");
}
    else { // Cria a página html dinâmica.
      response.setContentType("text/html");
      PrintWriter out = new PrintWriter
     (response.getOutputStream());

 out.println("<html>");
 out.println(
 "<head><title>Servlet1</title></head>");
 out.println("<body>");
 out.println("<P> Cliente: " +
 Cliente1.getCpf () + " </P>");
 out.println("<P> Nome:"+Cliente1.getNome
 () + " </P>";

  . . .

 out.println("<P> Mensagem :"+ErrorMessage
 + " </P>");
 out.println("<P> Número : " + ErrorCode +
    "</P>");
 out.println("<P>" );
 out.println("</body></html>");
 out.close(); }}
```

(b)

Figura 8 – Código do Servlet *srvCadastraCliente*

```
import java.sql.*;
import java.util.Vector;
import PersistentObject;

   public class Cliente implements
PersistentObject {
...
  //Implementação do método findlike, herdado
    da classe PersistentObject
public ResultSet findlike(){   } (b)
   return findlikeDB();  }

//Seleciona um determinado objeto do banco de
dados.
public ResultSet findlikeDB () {
   try {
    String findSQL = "select * from cliente
    where Cpf = '"  + this.Cpf + "' ";
    ResultSet rs =
ConnectionManager.stmt.executeQuery(findSQL);
    return rs;}
   catch (SQLException e)
    {System.out.println(e.getErrorCode ());
    return null;    }  }

// Implementação do método save, herdado da
//classe PersistentObject
public boolean save(){
   ResultSet rs;
   try { rs = findlike();
     if (rs.next ()) {
       return false;      }
     else { return insertDB();  }   }
   catch (SQLException e) {
     System.out.println(e.getMessage());
     return false;    }  }

// Efetua a inserção no Banco de dados
public boolean insertDB() {
   try {
    String insertSQL = "Insert into
  Cliente(Cpf, Nome, Endereco,Fone,EMail,
   Rg, NroCartao) values ('" + this.Cpf +
   "', '" + this.Nome + ..... where
    Cpf = '" + this.Cpf + "' ";
    ConnectionManager.stmt.executeUpdate
    (insertSQL);
    return true;    }
   catch (SQLException e)
    {try
     { if (e.getErrorCode () == -193)
     {System.out.println("Já está
     cadastrado");
    } }
```

(a)

(c)

Figura 9 – Definição da Classe Cliente que implementa a interface *PersistentObject*

5. Considerações Finais

A implementação do sistema exemplo foi realizada anteriormente sem a utilização de Padrões de Projeto. Uma das dificuldades encontradas, durante essa implementação, foi realizar a persistência dos objetos em um banco relacional. A tecnologia CGI, foi utilizada em vez de *servlet*, o que fez com que o tratamento das requisições e respostas se tornasse mais lento à medida que o número de requisições, ao servidor, fosse aumentando.

Após a implementação que utilizou Java *servlet* e padrão *Persistence Layer*, pôde-se observar que há separação entre a funcionalidade do sistema e as tarefas de recuperação e gravação das informações no banco de dados relacional, o que proporciona redução do número de linhas de código da aplicação, permitindo sua melhor organização. Isso facilita a manutenção do sistema e torna o projeto mais inteligível e de fácil compreensão permitindo que atualizações sejam realizadas de forma mais prática e organizada, ampliando assim a manutenibilidade do mesmo.

A forma de implementação do padrão *Persistence Layer* aqui utilizada é a mais simples, rápida e fácil de ser desenvolvida, porém, não usa o padrão *Table Manager*, que gera as cláusulas SQL de forma automática. Para cada classe persistente da aplicação, há necessidade de reescrever os métodos que implementam as cláusulas SQL's. Os métodos de persistência do padrão CRUD são implementados nas classes da aplicação por meio de um pequeno trecho de código específico para a persistência dos objetos no banco de dados. Esse trecho é reutilizado pelas outras classes com pequenas alterações para adaptar as cláusulas SQL's.

A utilização de *servlet*s garante que o sistema seja executado inteiramente no servidor, dando ao projetista total controle do ambiente de desenvolvimento e execução do sistema. Como se trata de uma aplicação em linguagem Java, a sua execução é permitida na maioria dos *browsers* e sistemas operacionais conhecidos, devido a portabilidade dessa linguagem. Finalmente, o *servlet* permite a criação de páginas HTML dinâmicas com um desempenho superior às tecnologias similares.

Com a realização deste trabalho pôde-se observar a integração de diferentes tecnologias e a utilização de UML que proporcionam facilidades quanto ao desenvolvimento de sistemas. Experiências anteriores mostraram que a manutenibilidade de sistemas com o uso de padrões de projeto no processo de reengenharia orientada a objetos de sistemas é facilitada [16] . Assim, a integração de tecnologias, aqui realizada, fornece indícios de que o mesmo ocorre, para o desenvolvimento de sistemas que são disponibilizados na Internet e que devem ser constantemente atualizados e melhorados.

A continuidade deste trabalho se dará com a implementação do padrão *Persistence Layer*, terceira forma, como sugerido por Yoder [1] no mesmo sistema exemplo. A utilização de outros padrões de projeto também será estudada visando a obtenção de facilidades no desenvolvimento de sistemas para Internet em linguagem Java com *servlet*s.

Referências Bibliográfias

[1] **Yoder, J. W.; Johnson, R. E.; Wilson, Q. D.** – Connection Business Objects to Relational Databases. In: Conference on the Pattern Languages of Programs, 5, Monticello-IL, EUA, 1998.

[2] **Patterns and Software:** Essential Concepts and Terminology – URL: http://www.enteract.com/~bradapp/docs/paterns-intro.html.

[3] **Gamma, E**.; **Helm, R.; Johnson, R.; Vlissides, J.** – *Design Patterns* – Elements of Reusable Object Oriented Programming, Finland. 1997.

[4] **Developping Servlets** – URL: http://www.borland.com/techpubs/jbuilder/jbuilder3/distributed/serv_servlets.html

[5] **Secure Computing with Java – now and the future** – URL:http://www.javasoft.com/marketing/collateral/security.html
[6] **Bergsten, H. -** An Introduction to Java Servlets – URL: http://webdevelopersjournal.com/articles/intro_to_servlets.html
[7] **Brockwood, T.** - Java - URL: http://webdevelopersjournal.com/articles/wijava.html
[8] **Bergesten, H. -** Improving Performance with a Connection Pool – URL: http://webdevelopersjournal.com/columns/connection_pool.html
[9] **Jensen, Cary;Anderson, Loy; Stone, Blake**. *Jbuilder Essential*. MacGraw-Hill, 1998.
[10] **Design ASP Files With Visual Basic** – URL: http://msdn.microsoft.com/library/periodic/period98/aspVB.htm
[11] **Sybase -** Sybase Inc. URL: http://www.sybase.com.
[12] **Java** – JDBC Basics. URL: http://www.java.sun.com/docs/books/tutorial/jdbc/basics/index.html.
[13] **Coad, P.; North, D.; Mayfield, M.** - *Object Models: Strategies, Patterns, and Aplicantions*. Yourdon. 1995.
[14] **Holf, M. A.**; **Silva, M. B.**; **Costa, C.M.** – MUX: Uma ferramenta para visualização de programas multithread. In Proceedings da XXV Conferência Latino Americana de Informática (CLEI99). Assunção Paraguai. pp.191-202, 1999
[15] **Cantú, M.** - *Dominando o Delphi 3 - A Bíblia*. Makron Books, 1998.
[16] **Cagnin, M.I.; Penteado, R.D. -** Alternativas de Reengenharia para um sistema implementado em C. In Proceedings da XXV Conferência Latino Americana de Informática (CLEI99). Assunção Paraguai, 1999**.**
[17] **Cagnin, M.I**. – Avaliação das Vantagens Quanto à Facilidade de Manutenção e Expansão de Sistemas Legados Sujeitos a Engenharia Reversa e Segmentação. Dissertação de Mestrado apresentada ao PPG-CC/DC/UFSCar. São Carlos – SP, agosto/1999.
[18] **Fowler, M; Scott, K.** *UML Distilled – Applying the Standard Object Modeling Language*, Addison – Wesley, 1997.