# Principais padrões J2EE para a construção de aplicações não distribuídas

## Christian Cleber Masdeval Braz

*Neste tutorial mostraremos:*

- *resumo dos principais padrões de projetos necessários à construção de aplicações J2EE profissionais não distribuídas,*
- *algumas estratégias de projeto que devem ser consideradas e*
- *como a framework Struts e seus derivados provêem suporte à maioria desses padrões e estratégias.*

## 1 - Introdução

A plataforma Java J2EE (Java 2 Enterprise Edition) surgiu com o objetivo de padronizar e simplificar a criação de aplicações empresariais. Para isso, propõe um modelo onde componentes J2EE (páginas JSP, Servlets, EJB's, etc) escritos pelos usuários da plataforma, podem fazer uso de serviços providos por esta, os quais simplificam sua implementação e possibilitam maior foco no negócio [SINGH02].

Um diferencial significativo na arquitetura proposta para a plataforma J2EE foi a iniciativa de enfatizar a utilização de padrões de projetos. Tais padrões trazem inúmeras vantagens na modelagem e implementação de um software:

- possibilidade de projetar soluções mais rapidamente e com qualidade já que os padrões são soluções comprovadamente eficientes para problemas já conhecidos;
- visam principalmente flexibilidade, organização e reaproveitamento de código, o que resulta em maior produtividade, qualidade e facilidade de manutenção das aplicações assim desenvolvidas.

Os principais serviços disponibilizados pela plataforma J2EE destinam-se a suprir as necessidades de aplicações empresariais distribuídas, isto é, aquelas que necessitam da flexibilidade de disponibilizar acesso à sua lógica de negócio e dados para diferentes tipos de dispositivos clientes (navegadores, dispositivos móveis, aplicações desktop, etc) e/ou para outras aplicações residentes na mesma empresa ou fora desta. A Figura 1.1 (extraída de [SINGH02]) ilustra um ambiente J2EE típico.

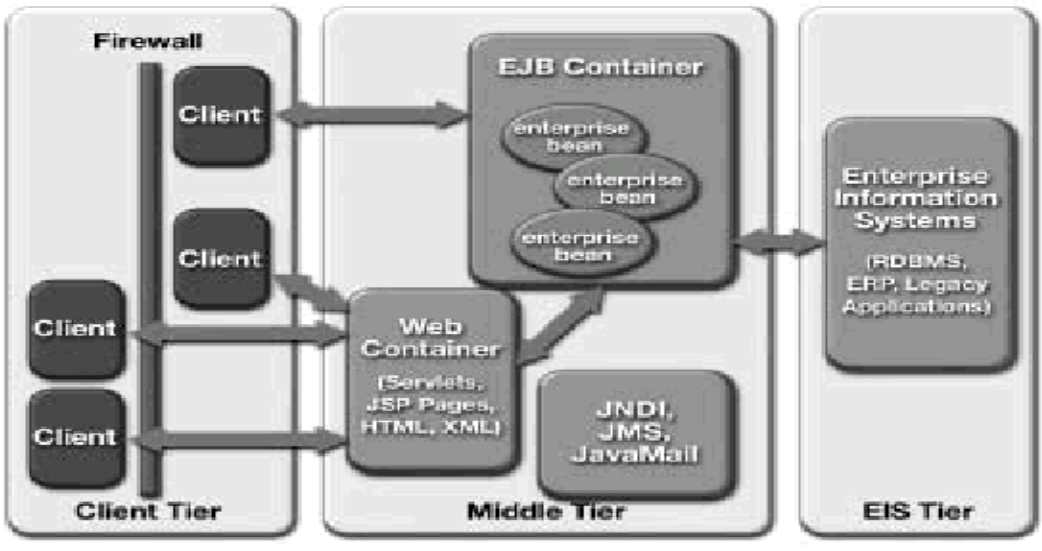


Figura 1.1 : Ambiente J2EE.

Aplicações distribuídas são comumente compostas de uma camada cliente, que implementa a interface com o usuário, uma ou mais camadas intermediárias, que processam a lógica do negócio e provêem serviços à camada cliente, e outra, chamada de *Enterprise Information System* ou EIS, formada por sistemas legados e bancos de dados. A infraestrutura oferecida pela J2EE possibilita que estas camadas, possivelmente localizadas em máquinas diferentes, possam se comunicar remotamente e juntas comporem uma aplicação.

Um componente criado numa aplicação J2EE deve ser instalado no *container* apropriado. Um container é um ambiente de execução padronizado que provê serviços específicos a um componente. Assim, um componente pode esperar que em qualquer plataforma J2EE implementada por qualquer fornecedor estes serviços estejam disponíveis.

Um *web container* destina-se a processar componentes web como servlets, JSP's, HTML's e Java Beans. Estes são suficientes para criar aplicações completas que não necessitam ser acessadas por diferentes tipos de cliente nem tampouco tornar seus dados e lógica distribuídos. Já um *EJB container* destina-se a prover a infraestrutura necessária para a execução de componentes de negócio distribuídos. Um EJB (Enterprise Java Bean) é um componente de software que estende as funcionalidades de um servidor permitindo encapsular lógica de negócio e dados específicos de uma aplicação. Tal componente pode ser acessado de maneiras diferentes, por exemplo através de RMI, CORBA ou SOAP, o que possibilita que este seja utilizado por qualquer tecnologia que provê suporte a um destes padrões de comunicação e que seja localizado virtualmente a partir de qualquer rede TCP/IP.

A plataforma J2EE permite uma arquitetura flexível sendo que tanto o web container quanto o EJB container são opcionais. Alguns cenários possíveis podem ser observados na Figura 1.2.

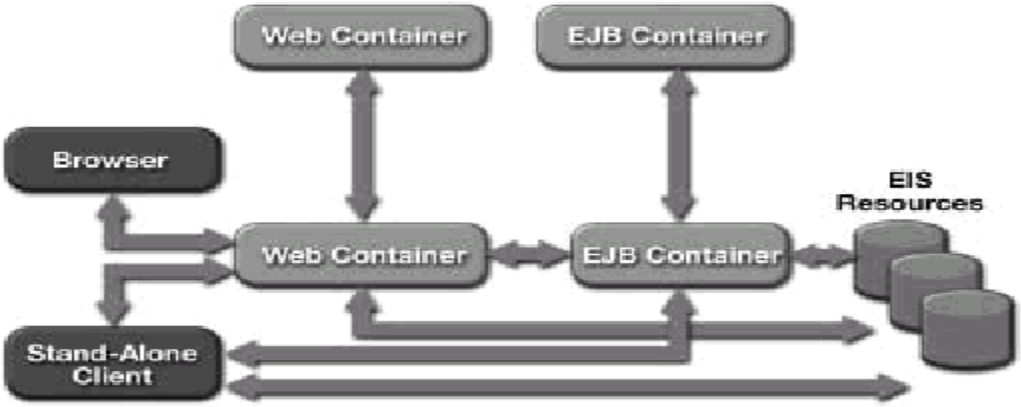


Figura 1.2: Cenários de aplicações J2EE.

O cenário que estaremos considerando neste trabalho é precisamente o ilustrado na Figura 1.3.

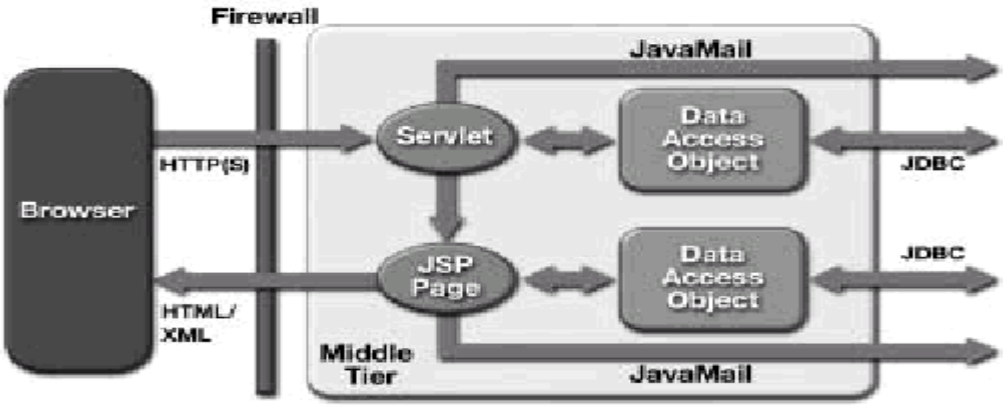


Figura 1.3: Aplicação web J2EE em três camadas.

Como apresentado no livro [J2EETIP], uma aplicação web em três camadas é na maioria das vezes a escolha inicial para uma aplicação J2EE e é como implementar este tipo de aplicação que estaremos discutindo aqui. Nesta configuração, o container web encarrega-se de tratar tanto a lógica de apresentação como a de negócios e veremos qual a melhor forma de organizarmos estas responsabilidades de forma a criarmos aplicações modulares, organizadas, com reaproveitamento de código e mais manuteníveis e extensíveis, ou seja, como criar aplicações profissionais neste cenário.

Neste artigo, discutiremos alguns padrões e estratégias de projeto que consideramos necessários à construção de aplicações J2EE profissionais não distribuídas. Basicamente, consiste num resumo de alguns dos padrões encontrados no livro [ALUR] acrescido de comentários baseados em nossa experiência prática no desenvolvimento destes tipos de aplicações. Muitas referências são feitas à framework Struts [STRUTS] e como esta implementa vários dos padrões e estratégias aqui apresentados.

## 2 – Padrões de Projeto

A correta utilização de arquiteturas e padrões de projeto no desenvolvimento de softwares representa um importante ideal a ser alcançado por qualquer equipe que pretende produzir aplicações computacionais profissionais. Não basta aprender uma tecnologia, é necessário também conseguir projetar soluções com esta tecnologia. A definição de uma arquitetura, por exemplo, permite que tenhamos uma visão completa da aplicação, de quais são seus principais componentes, o objetivo de cada um deles e a maneira como se relacionam a fim de desempenharem suas funções. Quando utilizamos padrões, estamos levando em conta experiências de outros projetos de desenvolvimento, aumentando assim as chances de chegarmos em uma solução correta pois erros passados poderão ser evitados.

Em aplicações sob a plataforma J2EE não é diferente. Em geral estamos tão atolados no processo de compreensão dos serviços da plataforma, de suas APIs (Application Program Interface) e do negócio a ser resolvido que não dedicamos o tempo necessário para aprender a projetar soluções com a tecnologia. Segundo Booch [ALUR] “existe um buraco semântico entre as abstrações e serviços que a plataforma J2EE oferece e a aplicação final que será produzida com esta e os padrões de projeto representam soluções que aparecem repetidamente para preencher este buraco”.

Um padrão provê uma solução para um problema comum baseado em experiências anteriores comprovadamente eficazes. Dispor de um bom conjunto de padrões é como ter um time de especialistas sentado ao seu lado durante o desenvolvimento, aconselhando-lhe com o melhor do seu conhecimento. Boas práticas de projeto são descobertas pela experiência e levam tempo até se tornarem maduras e confiáveis. Um padrão captura essa experiência e serve para comunicar, de forma padronizada, o conhecimento que trazem. Dessa forma, os padrões, além de ajudarem os desenvolvedores e arquitetos a reutilizarem soluções tanto de projeto quanto de implementação, ajudam também a criar um vocabulário comum na equipe, diminuindo assim o esforço de comunicação.

Veremos agora um resumo dos mais relevantes padrões e estratégias de projeto para a construção de aplicações J2EE profissionais não distribuídas.

### 2.1 - Arquitetura Dividida em Camadas

A plataforma J2EE provê uma clara divisão, tanto lógica quanto física, de uma aplicação em camadas. O particionamento de uma aplicação em camadas permite uma maior flexibilidade de escolha da tecnologia apropriada para uma determinada situação. Múltiplas tecnologias podem ser utilizadas para proverem o mesmo serviço possibilitando escolher a que melhor se adeqüa baseado nas características do problema em questão. Por exemplo, como pôde ser observado na Figura 1.1, é possível utilizar para a construção da camada cliente, quando um EJB container está sendo considerado, tanto páginas HTML e JSP como também um cliente desktop.

Como comentado anteriormente, existem situações onde a utilização de EJBs é desnecessária, a aplicação não necessita ser distribuída e apenas um subconjunto dos serviços disponíveis na plataforma atende aos seus requisitos. Dessa forma, nosso foco está na construção de aplicações web em três camadas, sendo a camada dos EJBs desconsiderada. Cada camada e suas responsabilidades são:

- Camada cliente: é responsável por interagir e apresentar os dados aos usuários e por se comunicar com outras camadas da aplicação. Em geral é a única parte da aplicação com a qual o usuário tem contato. A camada cliente comunica-se com outras camadas através de interfaces bem definidas. Projetando desta forma, temos maior facilidade para adicionar novos clientes, contanto que estes estejam em conformidade com a interface para comunicação com as outras

camadas. Em nossas aplicações o único cliente que consideraremos é o navegador web (browser). No caso da camada de Enterprise Java Beans estar presente, passa a ser possível, por exemplo, termos clientes como aplicações desktop escritas em Java e comunicando-se com esta camada via RMI (Remote Method Invocation) ou então aplicações escritas em outras linguagens e comunicando-se com esta via CORBA (Common Object Request Broker Architecture).

- Camada web: é quem recebe e processa as requisições dos usuários através de um web container. É responsável por realizar todo processamento web, como hospedar páginas HTML, executar servlets e formatar páginas JSP a serem apresentadas pelos navegadores. Em nosso caso, é responsável também pelo controle do fluxo da aplicação e por processar toda a lógica do negócio, incluindo o gerenciamento de conexões com banco de dados e conectores para acesso a recursos legados.

- Camada de banco de dados e sistemas de informação legados: tipicamente nesta camada estão incluídos sistemas gerenciadores de banco de dados, sistemas de planejamento de recursos (ou Enterprise Resource Planning (ERP)) e quaisquer sistemas legados. Com o apoio da API JCA (Java Connector Architecture) esta camada provê uma infraestrutura de software que mapeia dados e recursos de aplicações existentes dentro de projetos J2EE de forma padronizada.

A camada responsável pelo processamento dos EJBs é a Enterprise JavaBeans Tier. Esta fornece serviços, implementados por um EJB container, para os EJBs. Alguns deles são: controle de transação, mecanismos de segurança, persistência, dentre outros. Estes serviços disponíveis facilitam a vida do programador, que fica livre da obrigação de ter que tratá-los, podendo se concentrar mais na implementação das funcionalidades inerentes ao seu negócio. Quando esta camada está presente, a lógica de negócio (toda ou parte desta) pode passar a ser implementada nos componentes de negócio distribuídos, ficando a camada web assim mais voltada à exibição dos dados.

Por fim, mantenha em mente que estas são divisões lógicas que servem principalmente para melhor estruturar uma aplicação. Com exceção talvez da camada cliente, não está implícito que cada camada esteja em localizações físicas diferentes. Nada impede por exemplo que o banco de dados esteja localizado no mesmo servidor que hospeda um web container numa arquitetura em três camadas, ou que tanto o web container e o EJB container estejam rodando na mesma máquina virtual Java. O importante é a divisão de responsabilidades que resulta das tecnologias separadas.

### 2.2 - Model-View-Controller

Um dos principais padrões que utilizaremos para construção de nossas aplicações é o padrão MVC ou Modelo-Vista-Controlador (do inglês Model-View-Controller). Este padrão separa três formas distintas de funcionalidades em uma aplicação. O **modelo** representa a estrutura de dados e operações que atuam nestes dados. Em uma aplicação orientada a objetos, constitui as classes de objetos da aplicação que implementam o que quer que a aplicação tenha que fazer. **Visões** implementam exclusivamente a lógica de apresentação dos dados em um formato apropriado para os usuários. A mesma informação pode ser apresentada de maneiras diferentes para grupos de usuários com requisitos diferentes. Um **controlador** traduz ações de usuários (movimento e click de mouse, teclas de atalho, etc) juntamente com os valores de entrada de dados em chamadas à funções específicas no modelo. Além disso, seleciona, baseado nas preferências do usuário e estado do modelo, a vista apropriada. Essencialmente, estas três camadas abstraem: dados e funcionalidades, apresentação e comportamento de uma aplicação.

Os benefícios em usar o padrão MVC são:

- Separação Modelo-Vista: separando o modelo da vista é mais fácil adicionar diferentes apresentações do mesmo dado, além de facilitar também a adição de novos tipos de visão a medida que a tecnologia evolui. Os componentes relativos ao modelo e visão podem ser projetados, desenvolvidos e modificados independentemente (com exceção das suas interfaces), melhorando a manutenibilidade, extensibilidade e testabilidade.

- Separação Controlador-Vista: separar o controlador das visões permite maior flexibilidade para selecionar, em tempo de execução, visões apropriadas baseado no fluxo de trabalho, preferências do usuário ou estado interno do modelo, quando mais de uma estiver disponível. Por exemplo, se o nível de acesso, linguagem ou localização de um usuário determinar uma apresentação diferente, esta decisão é feita na camada de controle que retornará para este usuário a visão correta.

- Separação Controlador-Modelo: separando o controlador do modelo temos como criar mapeamentos configuráveis de ações capturadas pelo controlador para funções no modelo. Estas configurações possibilitam, por exemplo, que uma mesma ação seja executada, para usuários diferentes, por funções diferentes.

Mais uma vez, a divisão de responsabilidades que resulta deste particionamento lógico possibilita uma melhor compreensão e organização dos requisitos da aplicação e facilita a implementação, manutenção e extensão dos mesmos. Nossas aplicações obedecerão a divisão dessas três camadas lógicas da seguinte forma:

- A camada modelo será composta por um conjunto de classes de objetos com toda a implementação da lógica de negócio da aplicação. Os desenvolvedores Java responsáveis por criar estas classes o farão com ferramentas propícias a esta tarefa e pouco ou nenhum contato com os membros responsáveis pelo desenvolvimento das visões.

- A camada de visão será composta por arquivos HTML, JSP, arquivos de imagens, arquivos multimídia, enfim, tudo que englobe o desenvolvimento de páginas web a serem apresentadas aos usuários. Como o objetivo de uma visão é tão somente apresentar os dados que vêm do modelo, **teremos como premissa remover o máximo possível de código Java, isto é *scriptlets*, das páginas JSP**.

- A camada de controle determina o fluxo da apresentação, servindo como uma camada intermediária entre a apresentação e lógica da aplicação. Qualquer solicitação de um usuário é interceptada primeiro pelo controlador que pode, neste momento, realizar qualquer tipo de controle, como verificação de restrições de segurança, gravação de logs, etc. Depois disso dá-se início ao processamento da solicitação, provavelmente com o controle designando para classes auxiliares a execução do comando. Quando a execução termina o controlador escolhe, baseado no resultado e outras configurações, qual página retornar ao usuário. O controle centralizado deste fluxo traz inúmeras vantagens e veremos mais detalhes sobre este no padrão Front Controller.

### 2.3 – Front Controller

Em geral qualquer aplicação, independente de plataforma, é projetada com um determinado fluxo de operação em mente. Esperamos que os usuários interajam com a aplicação seguindo este fluxo, pois cada etapa de uma operação tem um propósito a ser cumprido para que toda ela complete com sucesso. Não podemos, por exemplo, exibir detalhes do perfil de um usuário sem antes este ter sido selecionado.

Diferentemente das aplicações desktop, os programas baseados na web estão sujeitos a interrupções que podem ocorrer no fluxo esperado de operação. O usuário pode pressionar o botão de “Voltar” ou “Recarregar” do navegador, abortar prematuramente uma solicitação em andamento, abrir novas janelas a qualquer momento, resubmeter uma operação, etc. Nada disso ocorre numa aplicação tradicional desktop, onde o programador tem controle muito maior sobre o fluxo de operacionalização do programa. É de responsabilidade da aplicação web tratar estes inconvenientes e garantir que o fluxo e estado adequado da aplicação sejam mantidos.

Uma das maneiras de fazer isso é inserir o código necessário para efetuar esse controle em cada uma das páginas JSP. Esta abordagem tem dois sérios problemas. Primeiro, o código referente à lógica de controle é misturado com o código da lógica de apresentação, dificultando a manutenção destas páginas. Outro problema está relacionado ao fato de que a disseminação da lógica de controle em múltiplas páginas JSP requer também manutenção em cada uma das páginas. Numa aplicação com dezenas ou centenas de páginas, sempre que essa lógica for alterada ter-se-á um grande trabalho para executar uma simples manutenção, já que o código de controle foi reaproveitado numa abordagem copia-cola.

Uma solução mais adequada para implementação da lógica de controle é a utilização de um ponto centralizado para receber e tratar as solicitações dos usuários. Isto é feito adicionando-se um controller ao projeto das aplicações. Um controlador, além de prover os benefícios comentados no contexto do padrão MVC, possibilita a centralização do código referente ao controle do fluxo de execução de uma aplicação reduzindo a quantidade de código Java (scriptlets) embutido nas páginas JSP. Isto promove a reutilização de código e melhora o desenvolvimento e manutenção das páginas JSP, que passam a conter apenas código relativo a apresentação de dados.

Além destes benefícios, a centralização do acesso provida por um controlador ajuda na implementação dos seguintes ítens:

- controle de transações,
- serviços de sistema comuns como acesso a banco de dados e serviços de autenticação e autorização,
- acompanhamento das ações de um usuário através do site,
- etc.

Cabe neste momento comentarmos sobre a framework Struts [STRUTS]. Esta é uma implementação bastante completa e flexível de um front controller e MVC, além de disponibilizar recursos que ajudam na construção da camada de visão e aplicar vários dos padrões e estratégias que discutiremos a seguir.

### 2.4 – View Helper

Um dos principais objetivos que desejamos alcançar no desenvolvimento de nossas aplicações é a clara divisão de responsabilidades. No modelo MVC vimos que a camada de visão (view) é responsável pela apresentação dos dados. Quando lógica de negócio e lógica para formatação dos dados estão inseridas nesta camada, o sistema torna-se menos flexível, reutilizável e suscetível à mudanças. Isto também prejudica a clara separação do trabalho de web designers e programadores.

Como vimos, todo o processamento da lógica de negócio deve estar centralizado na camada modelo do MVC. Isto implica que não haverá scriptlets com regras de negócio em nenhuma parte das páginas JSP, que irão somente apresentar o estado do modelo que chega encapsulado em helpers denominados *value objects* (que nada mais são do que Java Beans que representam uma porção dos dados do modelo num dado instante). Procedendo desta forma, estaremos prezando pelo reaproveitamento de código e manutenibilidade, já que porções de códigos que fazem a mesma coisa foram retirados dos scriptlets e inseridos em classes Java que fazem parte do modelo.

Devemos evitar também a inclusão de código para formatação dos dados nas páginas JSP, tais como código para controle de fluxo ou iteração. Nesses casos a melhor opção é utilizar *custom tag helpers*, que encapsulam estas funcionalidades que, de outra forma, seriam embutidas diretamente no JSP como scripts, reduzindo a modularidade e manutenibilidade. Veja abaixo o trecho de código que mostra a apresentação de dados de um empregado feita com e sem custom tags:

```
<html>
<head><title>Employee List</title></head>
<body>
<%-- Display All employees belonging to a department and earning at most the given salary --%>
<%
// Get the department for which the employees are
// to be listed

String deptidStr = request.getParameter(Constants.REQ_DEPTID);
// Get the max salary constraint
String salaryStr = request.getParameter(Constants.REQ_SALARY);

// validate parameters
// if salary or department not specified, go to
// error page
if ( (deptidStr == null) || (salaryStr == null ) )
{
   request.setAttribute(Constants.ATTR_MESSAGE,"Insufficient query parameters specified" +
      "(Department and Salary)");
   request.getRequestDispatcher("/error.jsp").forward(request, response);
}
//convert to numerics
int deptid = 0;
float salary = 0;

try
{
   deptid = Integer.parseInt(deptidStr);
   salary = Float.parseFloat(salaryStr);
}
catch(NumberFormatException e)
{
   request.setAttribute(Constants.ATTR_MESSAGE, "Invalid Search Values" + "(department id and
         salary )");
   request.getRequestDispatcher("/error.jsp").forward(request, response);
}

// check if they within legal limits
```

```
if ( salary < 0 )
{
   request.setAttribute(Constants.ATTR_MESSAGE,"Invalid Search Values" +"(department id and
      "salary )");
   request.getRequestDispatcher("/error.jsp").forward(request, response);
}
%>

<h3><center> List of employees in department # <%=deptid%> earning at most <%= salary %>. </h3>

<%
Iterator employees = new EmployeeDelegate().getEmployees(deptid);
%>

<table border="1" >
<tr>
   <th> First Name </th>
   <th> Last Name </th>
   <th> Designation </th>
   <th> Employee Id </th>
   <th> Tax Deductibles </th>
   <th> Performance Remarks </th>
   <th> Yearly Salary</th>
</tr>

<%
while ( employees.hasNext() )
{
   EmployeeVO employee = (EmployeeVO) employees.next();
   // display only if search criteria is met
   if ( employee.getYearlySalary() <= salary )
   {
%>

<tr>
<td> <%=employee.getFirstName()%></td>
<td> <%=employee.getLastName()%></td>
<td> <%=employee.getDesignation()%></td>
<td> <%=employee.getId()%></td>
<td> <%=employee.getNoOfDeductibles()%></td>
<td> <%=employee.getPerformanceRemarks()%> </td>
<td> <%=employee.getYearlySalary()%></td>
</tr>

<%
}
   }
%>

</table>
</body>
</html>
```

Agora a versão usando custom tag:

```
<%@ taglib uri="/WEB-INF/corepatternstaglibrary.tld" prefix="corepatterns" %>
<html>
<head><title>Employee List</title></head>
<body>
<corepatterns:employeeAdapter />
<h3><center>List of employees in <corepatterns:department attribute="id"/> department - Using
Custom Tag Helper Strategy </h3>

<table border="1" >
<tr>
   <th> First Name </th>
   <th> Last Name </th>
   <th> Designation </th>
   <th> Employee Id </th>
   <th> Tax Deductibles </th>
   <th> Performance Remarks </th>
   <th> Yearly Salary</th>
</tr>

<corepatterns:employeelist id="employeelist_key">
<tr>
   <td><corepatterns:employee attribute="FirstName"/></td>
```

```
   <td><corepatterns:employee attribute="LastName"/></td>
   <td><corepatterns:employee attribute="Designation"/> </td>
   <td><corepatterns:employee attribute="Id"/></td>
      <td><corepatterns:employee attribute="NoOfDeductibles"/></td>
   <td><corepatterns:employee attribute="PerformanceRemarks"/></td>
   <td><corepatterns:employee attribute="YearlySalary"/></td>
</tr>
</corepatterns:employeelist>
</table>
</body>
</html>
```

A framework struts disponibiliza uma vasta biblioteca de tags que desempenham inúmeras funções que facilitam a inclusão de lógica de formatação em páginas JSP.

### 2.5 - Value Object

Os dados que vêm do modelo são expostos à camada de visão através de DTOs (Data Transfer Objects) que podem ser Value Objects. Um value object é um objeto Java arbitrário que encapsula os valores de retorno dos componentes de negócio, como por exemplo o retorno de um método de um DAO. Geralmente seus atributos são feitos públicos e o construtor recebe cada um de seus valores. O nome das classes Java que representam value objects podem receber um "VO" para identificar que participam deste padrão.

### 2.6 - Data Access Object

A maioria das aplicações J2EE necessitam, num dado momento, acessar algum tipo de informação persistente. Estas informações podem estar nos mais diversos dispositivos de armazenamento, como servidores mainframes, repositórios LDAP (Lightweight Directory Access Protocol), banco de dados relacionais e orientado a objetos, ou mesmo dados provenientes de serviços disponibilizados por sistemas externos, como quando há integração business-to-business (B2B).

Estes dispositivos são acessados por diferentes APIs, muitas vezes proprietárias e específicas. Por exemplo, um banco de dados relacional pode ser acessado pela API JDBC. Esta API possibilita às aplicações emitir comandos SQL que são a maneira padrão para interação com as tabelas contidas nestes bancos. Ainda assim, mesmo que estejamos considerando apenas o ambiente de sistemas gerenciadores de banco de dados relacionais (SGBD), a sintaxe dos comandos SQL pode variar dependendo do SGBD em particular.

Esta diversidade de fontes de dados pode criar uma dependência da aplicação em relação ao código de acesso aos dados, uma vez que os componentes de negócio conterão código específico das APIs e dispositivos de armazenamento sendo utilizados. Isto introduz um alto acoplamento entre a lógica de negócio e a implementação do acesso às fontes de dados, tornando mais difícil migrar a aplicação de uma fonte de dados para outra.

A solução é abstrair e encapsular o acesso a fontes de dados através de Data Access Object (DAO). Um DAO implementa o mecanismo de acesso para se trabalhar com uma fonte de dados específica. Um componente de negócio fica exposto apenas à interface do DAO, que esconde toda a complexidade relativa à interação com a fonte de dados sendo utilizada. Como a interface de um DAO não se altera quando sua implementação precisa ser modificada, este padrão permite alterar a fonte de dados sendo utilizada numa aplicação sem afetar os componentes de negócios que fazem uso deste.

Quando utilizamos DAO é comum utilizarmos também o padrão Abstract Factory. Uma fábrica de objetos tem a função de criar objetos de um determinado tipo e, quando uma existe, devemos utilizá-la para criar determinados objetos ao invés de fazê-lo explicitamente através do comando **new**. Pense no seguinte, uma vez que nossa aplicação está cheia de DAOs, cada um com os métodos apropriados para acessar uma determinada fonte de dados, como fazer para substituir um DAO por outro quando a fonte de dados precisar ser alterada? Bom, uma alternativa é remover todas as classes que representam DAOs na aplicação e substituí-las por classes com o mesmo nome (e é claro mesma interface) porém projetadas para acessar a nova fonte de dados. Uma opção mais profissional e flexível é criar uma classe abstrata que representa uma fábrica de DAOs. Nesta classe estarão definidos os métodos para criação de todos os DAOs que a aplicação precisa. **Quando o dispositivo de armazenamento persistente é passível de mudança, deve-se criar uma fábrica concreta de DAOs, derivada da classe abstrata, específica para cada fonte de dados a ser utilizada**. Assim, é possível, por exemplo, definir uma variável global na aplicação do tipo `DAOFactory` e inicializá-la com a fábrica concreta necessária num dado momento. Quando a fonte de dados precisar ser alterada, basta inicializarmos a

variável com outra fábrica capaz de construir DAOs aptos a interagir com a nova fonte. Em qualquer parte da aplicação onde um DAO está sendo criado, e isto certamente está sendo feito utilizando-se a variável global com a fábrica de DAOs corrente, nada precisará ser modificado. A figura 2.1 apresenta o diagrama de classes deste esquema.

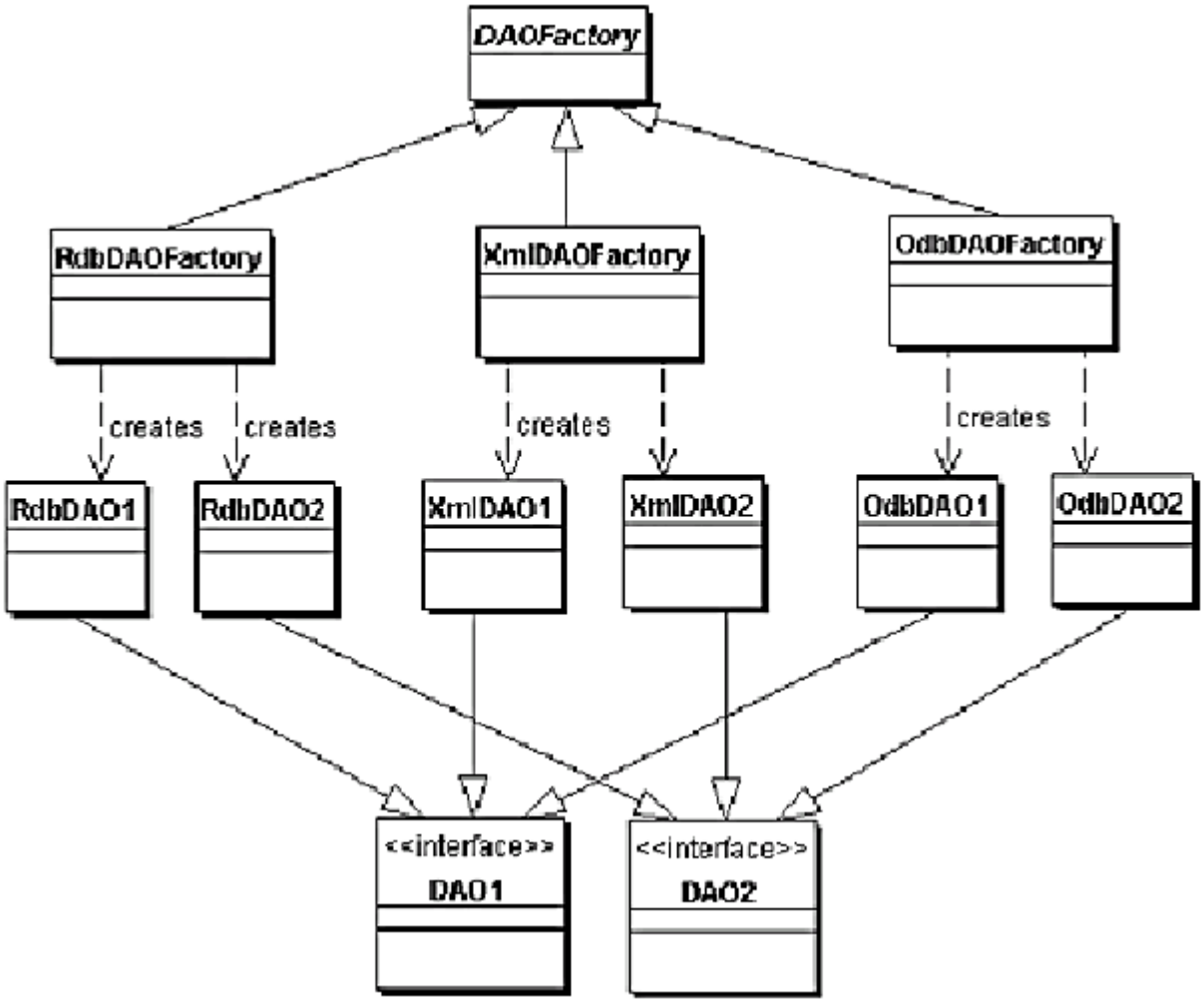


Figura 2.1: Abstract Factory para Data Access Object.

### 2.7 - Business Delegate

A camada cliente pode interagir diretamente com os serviços disponibilizados pela camada responsável por implementar a lógica do negócio. Esta interação direta expõe sobremaneira a API destes serviços de negócio, tornando a camada de apresentação mais vulnerável às mudanças efetuadas nos métodos de negócio. Dito isso, pode-se concluir que é desejável reduzir o acoplamento entre estas duas camadas, escondendo ao máximo detalhes de implementação.

O padrão para resolver este problema é o business delegate que age como uma abstração, no lado cliente, para os métodos de negócio. Um business delegate pode esconder, por exemplo, a complexidade da busca por recursos como EJBs, numa aplicação distribuída, ou data sources, que possibilitam obter conexões a fontes de dados. Sua utilização torna transparente aos usuários dos serviços de negócio detalhes sobre serviços de nome e diretório e a busca nestas estruturas. Outro benefício é o tratamento das exceções que podem ser lançadas pelos serviços de negócio, o qual pode ser efetuado antes que uma delas chegue à camada de apresentação. Assim, caso realmente o problema ocorrido não seja recuperável, uma mensagem mais amigável pode ser produzida e apresentada ao usuário.

### 2.8 - Service Locator

Aplicações J2EE distribuídas interagem com componentes como EJB e JMS os quais provêem variados serviços. Geralmente estes componentes precisam ser localizados através de mecanismos de busca em

serviços de nome e diretório. Aplicações não distribuídas também precisam localizar recursos, tal como data sources afim de estabelecerem conexão com dispositivos de armazenamento.

Para gravar, localizar e criar recursos de forma padronizada, as aplicações J2EE utilizam as facilidades providas pela API JNDI (Java Naming and Directory Interface). Esta API possibilita mapear nome de componentes aos objetos que estes representam. Quando é preciso recuperar um componente, a aplicação deve primeiramente obter um contexto inicial e então prover um nome JNDI previamente registrado para que a busca possa ser efetuada. Recursos precisam ser obtidos repetidamente desta maneira através de uma aplicação, conseqüentemente fazendo com que o código JNDI apareça múltiplas vezes. Isto resulta numa desnecessária duplicação de código e, como a criação de um contexto inicial é dependente de fabricante, introduz também dificuldades de manutenção.

A solução é utilizar um objeto service locator que abstrai todas as chamadas JNDI e esconde as complexidades inerentes à criação de um contexto inicial e busca dos componentes. Este objeto pode ser utilizado em toda parte de uma aplicação que precisa acessar um recurso gerenciado por JNDI, reduzindo a complexidade do código, provendo um ponto único de controle e, em determinados casos, melhorando a performance através de mecanismos de cache.

### 2.9 - Composite View

Uma única página web pode ser composta por múltiplas subvisões, isto é, a página é dividida em porções, cada uma com um conteúdo específico. Por exemplo, uma página pode conter um cabeçalho, um rodapé, uma porção destinada ao menu e outra para apresentar o conteúdo do que foi escolhido num determinado momento. Algumas subvisões, como o cabeçalho e rodapé, são reaproveitadas em várias outras visões. Se páginas com estas características são construídas acrescentando-se o conteúdo de cada porção diretamente na página, a aplicação será mais propícia a erros, devido à duplicação de código, e mudanças de layout serão bem mais difíceis de serem executadas.

A solução para isso é utilizar composite views, ou seja, visões compostas de múltiplas subvisões atômicas. Cada componente independente pode ser incluído dinamicamente e o layout da página passa a ser mais facilmente gerenciável independentemente do conteúdo. Esta abordagem promove a reutilização de porções atômicas da visão e facilita em muito a alteração do layout.

Uma das formas mais simples de se implementar composite views é através de tags JSP, como a ação `<jsp:include page="localURL"/>` ou a diretiva `<\%@include file="localURL"\%>`. A utilização destas tags para gerenciamento do layout e composição das visões é bastante simples de implementar. Porém, esta estratégia não provê a mesma flexibilidade da abordagem com custom tags, pois o layout de cada página permanece embutido dentro da própria página. Ou seja, apesar das tags possibilitarem a inclusão dinâmica de conteúdo e conseqüentemente o reaproveitamento de porções comuns a várias páginas, a estruturação do layout (a divisão da página em tabelas ou frames por exemplo e o posicionamento de cada porção atômica dentro desta divisão) continua contida em cada página.

Existem duas possibilidades para inclusão dinâmica de conteúdo - a inclusão em tempo de compilação ou em tempo de execução. A primeira é obtida usando-se a diretiva `<\%@include file="localURL"\%>` e a última através da ação `<jsp:include page="localURL"/>`. Os pontos positivos e negativos de cada uma destas abordagens são discutidos a seguir:

- Diretiva Include:
  - Todas as páginas incluídas são compiladas juntas criando um único servlet.
  - Compartilhamento de variáveis entre as páginas (já que na verdade geram um único servlet).
  - Melhor performance tendo em vista que não requer o overhead de despachar a solicitação para a página incluída e depois incorporar a resposta no output da página original.
  - A inclusão ocorre em tempo de compilação. Sendo assim, quando uma das páginas incluída é alterada, para ter efeito a alteração é necessário realizar forçosamente uma alteração na página principal e assim garantir que uma nova página com o novo conteúdo seja criada.
  - A página a ser incluída não pode ser especificada dinamicamente.
  - Possibilidade de incluir apenas páginas estáticas ou outra página JSP.

- Ação Include:
  - Cada página incluída é um servlet separado, que deve ser carregado e executado pelo container.

- A troca de informações entre as páginas ocorre através do objeto session ou request.
- Como cada página incluída é um servlet separado, uma alteração em alguma delas não requer uma alteração forçada na página principal para a alteração ter efeito (inclusão em tempo de execução).
- Resulta em menores tamanhos de classe, já que o código correspondente ao arquivo incluído não é repetido nos servlets para todas as páginas JSP que fazem a inclusão.
- Possibilidade de especificar em tempo de execução a página a ser incluída.
- Possibilidade de incluir páginas estáticas, CGI's, servlets ou outra página JSP.

A implementação do padrão composite view através de custom tags possibilita uma flexibilidade superior. Contudo, requer maior esforço de programação e gerenciamento, tendo em vista os inúmeros artefatos que precisam ser criados e configurados no ambiente de execução.

A framework Tiles [TILES], que pode ser utilizada em conjunto com Struts, é uma implementação de composite view que faz uso de custom tags e provê toda a flexibilidade de reaproveitamento e gerenciamento de layout que o padrão propicia. Com Tiles, além do reuso das "partes" é possível também definir um padrão para a disposição destas nas páginas, ou seja, definir o layout que as organizam. Basicamente, define-se uma template com o layout padrão que um conjunto de páginas terá. Este molde representa um esqueleto HTML e especifica onde serão inseridas as partes menores. O código abaixo define uma template JSP para a tela de abertura de um portal:

```
<%@page contentType="text/html"%>
<%@page pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/WEB-INF/struts-tiles.tld" prefix="tiles" %>
<HTML>
<HEAD>
<link rel="stylesheet" href="imagens/stylestruts.css" type="text/css">
<title><tiles:getAsString name="title"/></title>
</HEAD>

<body bgcolor="#ffffff">
<table width="100%" height="100%" border="0">

<tr valign="top">
   <td height="73" colspan="3"><tiles:insert attribute="header"/></td>
</tr>

<tr colspan="3">
   <td width="118" align="center" valign="middle"><tiles:insert attribute='ladoesq'/> </td>
   <td  width="632"  align="right"  valign="top"><div  align="right"><tiles:insert  attribute='body'/>
   </div> </td>
   <td width="209" align="center" valign="top"><tiles:insert attribute='ladodir'/> </td>
</tr>

<tr>
   <td height="20" colspan="3" valign="bottom"> <tiles:insert attribute='footer'/> </td>
</tr>

</table>
</body>
</html>
```

A Figura 2.2 mostra o layout padrão gerado pela template. Supondo que o arquivo com a template chama-se layoutAbertura.jsp, é preciso agora criar uma definição para cada página na aplicação que será construída seguindo o molde padrão. Nesta definição estará especificado o que irá conter cada porção variável, ou seja, o que cada página deve apresentar. O código abaixo é um exemplo de definição.

```
<definition name='portal.layout.abertura' path='/layoutAbertura.jsp'>
  <put name='title' value='Título da Aplicação' />
  <put name='header' value='/header.jsp' />
  <put name='footer' value='/footer.jsp' />
  <put name='ladoesq' value='ladoesq.jsp' />
  <put name='ladodir' value='ladodir.jsp' />
  <put name='body' value=''/>
</definition>
```

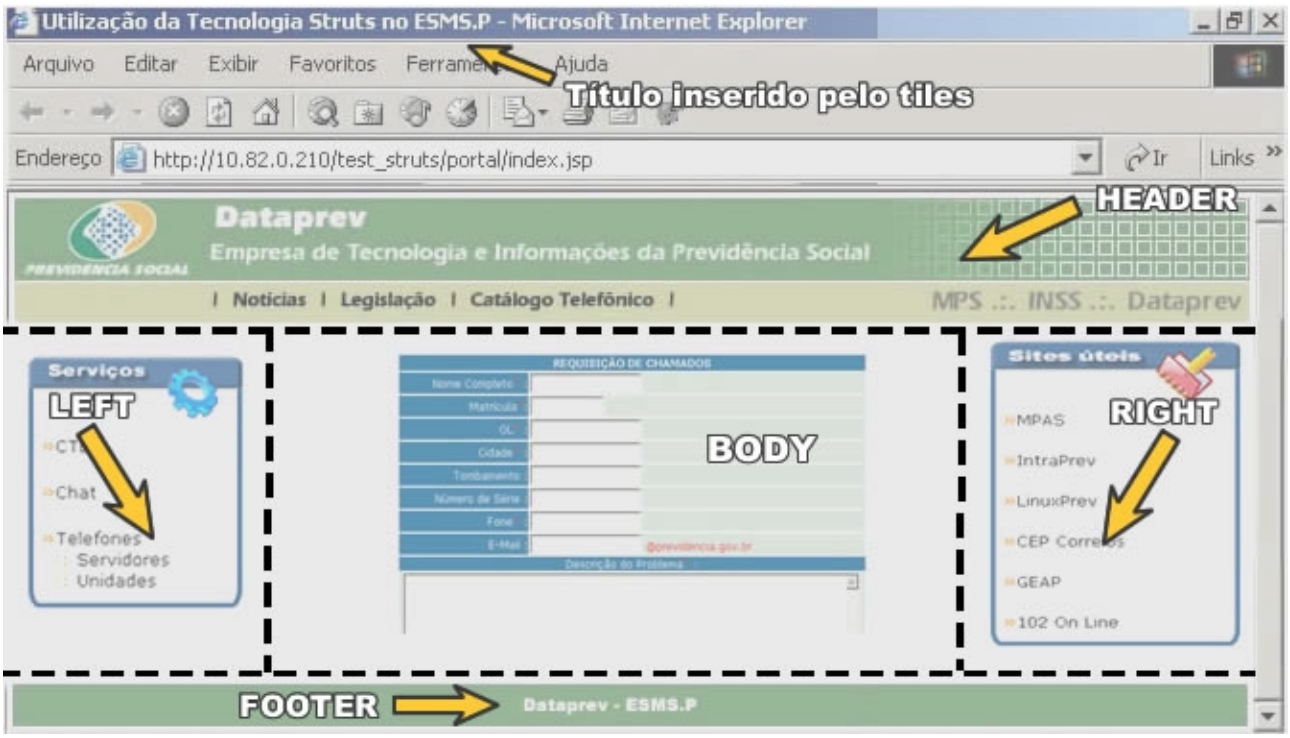


Figura 2.2: Template Tiles.

Pode-se verificar que para quase todas as porções foram especificadas as visões apropriadas. No entanto, para a porção body nada foi definido. Isto por que esta é uma definição padrão para várias telas da aplicação. Ou seja, não existirá uma página criada a partir dela, mas sim de alguma outra que derive da definição padrão. As porções comuns a todas as páginas já foram definidas e apenas a variável foi deixada em branco. Assim, quando for preciso criar uma página específica, pode-se fazer como abaixo,

```
<definition name='portal' extends='portal.layout.abertura'>
  <put name='body' value='/body.jsp' />
</definition>
```

e criar uma nova definição que estenda da primeira e defina o que deve conter o corpo. Por fim, basta criar a página que irá materializar esta definição:

```
<\%@ taglib uri='/WEB-INF/struts-tiles.tld' prefix='tiles'\%>
<html> <body>
  <tiles:insert definition='portal' flush='true'/>
</body></html>
```

Perceba que se for preciso alterar o layout básico das páginas na aplicação é suficiente que se altere apenas o arquivo `layoutAbertura.jsp` que todas as páginas automaticamente passarão a respeitá-lo.

## 3 – Estratégias

Nesta seção discutiremos algumas estratégias para implementação ou utilização de alguns dos padrões apresentados e outras que resolvem problemas ou chamam a atenção para aspectos importantes ainda não abordados.

### 3.1 - Controle de Transações

Uma das limitações das aplicações web e do navegador como interface cliente é a total falta de controle de uma aplicação sobre a navegação. Um usuário pode, por exemplo, submeter um formulário que resulta numa transação que irá debitar uma quantia de sua conta e, depois de receber a confirmação de que a operação ocorreu com sucesso, nada impede que ele clique no botão “Voltar” e resubmeta o mesmo formulário. Para controlar a ordem de acesso dos usuários às páginas e o acesso a determinados recursos deve-se fazer uso de um mecanismo de *token* compartilhado.

O mecanismo funciona da seguinte maneira:

- quando chega a solicitação de um usuário para a execução de algum processamento que requer controle de etapas um token é gerado, gravado na sessão do usuário e encaminhado para a próxima página do fluxo de execução,

- quando ocorrer uma nova submissão deste usuário, o token que chega dele deve ser o mesmo token que o servidor enviou na sua última resposta, confirmando que não é uma resubmissão ou, quando forem diferentes, sugerindo que isto pode ter acontecido,

- o processo de criação e envio de um novo token é repetido para todas as páginas intermediárias do processamento até chegar na página final, onde a transação deve ou não ser confirmada.

Este processo está ilustrado na Figura 3.1.


Figura 3.1: Ciclo de vida de um token.

Esta implementação permite que a todo momento tenha-se o controle quanto à ordem de interação do usuário com as páginas. Quando alguma comparação de token falhar significa que o usuário acessou a página diretamente, possivelmente através de um bookmark, ou que realizou duas vezes a mesma submissão.

A melhor maneira de gerenciar a geração e comparação do token é através de um controlador. A Struts, por exemplo, disponibiliza, embutido na sua implementação de front controller, mecanismos para controle de transações. Ao invés de criar uma classe utilitária separada para encapsular a lógica de geração e comparação de tokens, Struts provê suporte a isso adicionando esta funcionalidade às classes já existentes que fazem parte do seu mecanismo de controle.

### 3.2 - Exposição Desnecessária de Estruturas de Dados

Estruturas de dados da camada de apresentação, tal como objetos da classe `HttpServletRequest`, devem estar confinados apenas a esta. Compartilhar estes detalhes com a camada de negócios (camada modelo do MVC) ou qualquer outra, aumenta o acoplamento entre estas camadas e reduz drasticamente a reusabilidade dos serviços oferecidos por estas. Se a assinatura de um método em um serviço de negócio aceita um parâmetro do tipo `HttpServletRequest` então qualquer outro cliente deste serviço deve encapsular sua requisição num objeto do tipo `HttpServletRequest`. Além disso, a camada de negócios terá que compreender como interagir com estas estruturas de dados
específicas da camada de apresentação, aumentando a complexidade do código.

A solução é modificar os métodos de negócio para aceitarem parâmetros mais genéricos. Uma forma de se conseguir isto é decompondo cada um dos parâmetros contidos em `HttpServletRequest` para argumentos individuais como strings ou inteiros. No entanto, esta não é a melhor abordagem pois ainda mantém um forte acoplamento com a camada de apresentação. Se as informações necessárias para o método de negócio executar forem alteradas, então necessariamente as assinaturas de seus métodos também terão de ser modificadas. Uma alternativa mais flexível é copiar os dados relevantes da camada

de apresentação em uma estrutura de dados mais genérica, como um value object. Neste caso os serviços continuam aceitando este objeto mesmo se os detalhes da sua implementação mudarem.

O exemplo abaixo mostra um objeto fortemente acoplado com o `HttpServletRequest`.

/** The following excerpt shows a domain object that is too tightly coupled with HttpServletRequest **/

```
public class Customer
{

    protected String firstName;
    protected String lastName;

    public Customer ( HttpServletRequest request )
    {
        firstName = request.getParameter("firstname");
        lastName = request.getParameter("lastname ");
    }
}
```

Ao invés de expor o objeto `HttpServletRequest` ao objeto genérico `Customer`, decomponha os dois valores e crie uma classe assim:

```
public class Customer
{

    protected String firstName;
    protected String lastName;

    public Customer (String first, String last )
    {
        firstName = first;
        lastName = last;
    }
}
```

### 3.3 - Restringindo Acesso às Páginas JSP

Freqüentemente é necessário controlar o acesso a determinadas páginas JSP em uma aplicação. Quando existe um controlador, todo o acesso passa por este e o controle de permissões pode ser feito de forma centralizada. Uma opção para garantir maior proteção é tornar as páginas JSP totalmente inacessíveis, exceto para o controlador, movendo-as para baixo do diretório `WEB-INF`.

Por exemplo, considere uma aplicação chamada HelloWorld. Por padrão, nada impede que façamos:

`http://localhost:8080/HelloWorld/secure_page.jsp`

Para restringir acesso direto podemos simplesmente mover o arquivo JSP para um subdiretório de `WEB-INF`. O diretório `WEB-INF` é acessível apenas indiretamente através de redirecionamento, tal qual aqueles que chegam através de um controlador. Desta forma, se a página for acessada diretamente

`http://localhost:8080/HelloWorld/WEB-INF/privateaccess/secure_page.jsp`

o servidor responderá com uma mensagem de “recurso não disponível”.

### 3.4 - Validação

Em geral é desejável que as validações de entrada de dados sejam feitas tanto no cliente como no servidor. Apesar da validação no lado cliente ser menos sofisticada, esta possibilita verificar mais rapidamente, com menor tráfego na rede e sem ocupar recursos do servidor, se as informações

submetidas estão com a formatação esperada. Não é recomendado que a validação esteja apenas no cliente pois esta depende de configurações locais que podem desabilitar a execução de \textit{scripts} pelo navegador.

Tipicamente validações no cliente envolvem embutir código, como JavaScript, nas visões JSP e não serão comentadas aqui em maiores detalhes. Validações no servidor podem ser categorizadas de duas formas: validação centrada em formulários e validação baseada em tipos abstratos.

Na validação centrada em formulários, para cada formulário é criado um método Java de validação. Todo código de validação para verificar, por exemplo, valores nulos ou se um campo é ou não numérico, é escrito nestes métodos que acabam tendo muito código em comum. Neste caso, como não existe um código central para tratar das validações, havendo campos em vários formulários que precisem ser validados da mesma maneira, cada um é manipulado separadamente e de forma redundante em vários lugares da aplicação. Esta estratégia é relativamente fácil de implementar e funciona, porém leva à duplicação de código e a uma conseqüente ineficiência de seu reaproveitamento.

Uma solução mais flexível, reutilizável e fácil de manter consiste em abstrair os tipos dos campos e restrições impostas sobre eles e separar a validação do modelo da lógica da aplicação. Um benefício desta estratégia é que o sistema torna-se mais genérico pois fatora as informações de tipo dos dados que formam o modelo (metadados) e suas restrições para fora da lógica da aplicação. Um exemplo desta implementação é ter um componente que encapsula lógica de validação tais como: decidir se uma string é vazia, se um número esta dentro de um intervalo, se um conjunto de caracteres está formatado de uma determinada maneira e assim por diante. Quando uma aplicação precisa validar diferentes aspectos do modelo em lugares diferentes, não é necessário cada um deles escrever seu próprio código de validação, ao invés disso o mecanismo central de validação é usado.

A framework struts possibilita a implementação dos dois tipos de validação. Uma extensão da struts chamada Validator possibilita uma configuração de validação baseada em arquivos XML. Em um arquivo está descrito uma série de validações comuns e as classes designadas para tratá-las e em outro mapeia-se os campos de formulários HTML com as validações que estes devem sofrer.

## 4 - Conclusão

A construção de aplicações J2EE profissionais requer a correta compreensão e utilização dos recursos disponíveis na plataforma. Para projetar soluções coerentes, flexíveis, manuteníveis, com baixo grau de acoplamento e alta coesão e reaproveitamento de código, é necessário, antes de mais nada, estudar casos de sucesso no uso da tecnologia, casos onde todos os requisitos acima foram alcançados. Visando facilitar este processo, a SUN compilou alguns padrões de projeto que capturam soluções comprovadamente eficazes para problemas geralmente encontrados no projeto de sistemas com a plataforma J2EE. Conhecendo e empregando tais padrões, teremos mais chances de construir aplicações corretas, isto é, aplicações que serão construídas num tempo hábil, atenderão aos requisitos estipulados e serão gerenciáveis no decorrer do seu ciclo de vida.

Neste artigo, procuramos apresentar de forma sucinta alguns dos principais padrões e estratégias evidenciadas pela SUN. Esperamos que com este conhecimento básico o leitor possa ter tido uma idéia de fatores importantes envolvidos no planejamento de projetos com a tecnologia J2EE, evoluindo assim sua capacidade de projetar soluções com esta tecnologia.

## 5 - Bibliografia

[ALUR] Alur, D.; Crupi, J.; Malks, D. Core J2EE Patterns First Edition, Prentice Hall / Sun Microsystems Press, 2002.
[SINGH02] Singh, I.; Stearns, M. J., Enterprise Team. Designing Enterprise Applications with the J2EE Platform, Second Edition, Addison Wesley, 2002.
[J2EETIP] Rick, C.; Inscore, J.; Enterprise Partners. J2EE Technology in Practice: Building Business Applications with the Java 2 Platform, Enterprise Edition, Sun Microsystems Press, 2001.
[STRUTS] Struts http://struts.apache.org/.
[TILES] Tiles http://struts.apache.org/userGuide/dev_tiles.html.

**Christian Cleber Masdeval Braz** (christian.braz@*previdencia.gov.br*) é mestre em Ciência da Computação e Analista em Tecnologia da Informação na Dataprev – Empresa de Tecnologia e Informações da Previdência Social.