UFSM

Trabalho de Graduação

Estudo Comparativo entre Frameworks Java para Construção de Aplicações Web

Marcos Vinícius Bittencourt de Souza

Curso de Ciência da Computação

Santa Maria, RS, Brasil

2004

Estudo Comparativo entre Frameworks Java para Construção de Aplicações Web

por

Marcos Vinícius Bittencourt de Souza

Trabalho de Graduação apresentado ao Curso de Ciência da Computação – Bacharelado, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de
Bacharel em Ciência da Computação.

Curso de Ciência da Computação

Trabalho de Graduação n° 191

Santa Maria, RS, Brasil

2004

Universidade Federal de Santa Maria
Centro de Tecnologia
Curso de Ciência da Computação

A Comissão Examinadora, abaixo assinada, aprova o Trabalho de Graduação

Estudo Comparativo entre Frameworks Java para Construção de Aplicações Web

elaborado por

Marcos Vinícius Bittencourt de Souza

como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Comissão Examinadora:

Msc. João Carlos Damasceno Lima
(Orientador)

Dra. Iara Augustin

Dra. Andrea Schwertner Charão

Msc. Rodrigo Giacomini Moro

Santa Maria, 17 de dezembro de 2004.

"É melhor tentar e falhar, que preocupar-se e ver a vida passar; é melhor tentar, ainda que em vão, que sentar-se fazendo nada até o final.
Eu prefiro na chuva caminhar, que em dias tristes em casa me esconder. Prefiro ser feliz, embora louco, que em conformidade viver."

(Martin Luther King)

Agradecimentos

Agradeço à minha família pelo apoio e compreensão concedidos ao longo do desenvolvimento de todo o trabalho que, nos momentos mais difíceis, sempre se mostraram compreensíveis através de palavras e carinhos. Agradeço à minha namorada por compreender a minha ausência em certas ocasiões devido à execução desse trabalho, além de me ajudar nos momentos mais difíceis da vida acadêmica.

Agradeço ao CPD por permitir a realização do estudo com a utilização da sua estrutura. Agradeço à equipe de desenvolvimento *web* do CPD pela ajuda prestada durante o desenvolvimento do trabalho. Agradeço ao meu orientador pelo apoio e incentivo prestados.

A todos os meus colegas de turma com quem compartilhei meus problemas e alegrias durante estes quatro anos de vivência acadêmica.

A todos os professores que transmitiram os seus conhecimentos, auxiliando na minha vida profissional. Aos que, de certa forma, não conseguiram transmitir com clareza os seus objetivos, agradeço, pois sempre se consegue aproveitar alguma idéia.

Agradeço, ainda, aos que aqui não foram mencionados, mas que colaboraram, de forma direta ou indireta, para que esse projeto fosse concluído com sucesso.

Sumário

Lista de Tabelas **vi**

Lista de Figuras **vii**

Resumo **i**

1 INTRODUÇÃO **1**

2 REVISÃO DE LITERATURA **3**

2.1 PADRÕES DE PROJETO 3
2.2 O PADRÃO MVC 4
2.2.1 CAMADA MODELO 5
2.2.2 CAMADA VISÃO 5
2.2.3 CAMADA CONTROLADORA 6
2.3 FRAMEWORKS 6
2.3.1 DEFINIÇÃO 6
2.3.2 CONSTRUÇÃO 7
2.3.3 CLASSIFICAÇÃO 8
2.3.4 VANTAGENS E DESVANTAGENS DO USO DE *FRAMEWORKS* 9
2.4 ESCOLHA DOS *FRAMEWORKS* 11
2.5 *FRAMEWORK* SOFIA 11
2.5.1 CARACTERÍSTICAS 12
2.6 *FRAMEWORK* JAKARTA STRUTS 14
2.6.1 CARACTERÍSTICAS 15
2.7 FRAMEWORK SPRING 17
2.7.1 CARACTERÍSTICAS 17

3 DOCUMENTO DE INFORMAÇÕES CADASTRAIS - DIC 20

4 COMPARAÇÃO DOS *FRAMEWORKS* 24
4.1 CRITÉRIOS DA COMPARAÇÃO 24
4.2 CAMADA MODELO 25
4.2.1 SOFIA 25
4.2.2 STRUTS 26
4.2.3 SPRING 28
4.3 CAMADA VISÃO 29
4.3.1 SOFIA 30
4.3.2 STRUTS 32
4.3.3 SPRING 33
4.4 CAMADA CONTROLE 34
4.4.1 SOFIA 34
4.4.2 STRUTS 36
4.4.3 SPRING 36
4.5 VALIDAÇÃO DE DADOS 38
4.5.1 SOFIA 39
4.5.2 STRUTS 40
4.5.3 SPRING 40
4.6 MÉTRICAS DO CASO DE USO 41
4.7 PADRÕES UTILIZADOS NO CASO DE USO 44
4.7.1 SOFIA 44
4.7.1.1 MVC 44
4.7.1.2 DAO 45
4.7.2 STRUTS 45
4.7.2.1 MVC 45
4.7.2.2 DAO 46
4.7.3 SPRING 46
4.7.3.1 MVC 46
4.7.3.2 DAO 48

5 CONCLUSÃO 49

Anexo A GLOSSÁRIO 55

Lista de Tabelas

4.1 Tabela com as métricas do caso de uso para *framework*. 42
4.2 Tabela final de comparação entre os *frameworks*. 44

Lista de Figuras

2.1 Modelo MVC original 4
2.2 Modelo MVC *web* 4
2.3 Composição de um *framework* horizontal, Sauvê [8] 9
2.4 Composição de um *framework* vertical, Sauvê [8] 10
2.5 Assistente para a criação da camada modelo 14
2.6 Arquivo de configuração do fluxo da aplicação. 16
2.7 Módulos do Spring Framework - Johnson [15] 18

3.1 Primeira parte do formulário do DIC. 21
3.2 Segunda parte do formulário do DIC. 22
3.3 Terceira parte do formulário do DIC. 23

4.1 Trecho de arquivo de um mapeamento de objeto. 27
4.2 Configuração do gerenciamento de transações do *framework* Spring. . 29
4.3 MacroMedia DreamWeaver integrado ao *framework* SOFIA. 30
4.4 Mapeamento das requisições para as classes da aplicação. 37
4.5 Configuração das validações utilizando o *framework* Commons Validator. 39

RESUMO

Trabalho de Graduação
Ciência da Computação
Universidade Federal de Santa Maria

Estudo Comparativo entre Frameworks Java para Construção de Aplicações Web

Autor: Marcos Vinícius Bittencourt de Souza
Orientador: Msc. João Carlos Damasceno Lima
Data e Local da Defesa: Santa Maria, 17 de dezembro de 2004.

O desenvolvimento de aplicações freqüentemente utiliza o apoio de *frameworks*. De fato, conforme o grau de complexidade das aplicações aumenta, torna-se inviável a codificação a partir do zero. Faz-se uso, então, de *frameworks* que auxiliam o desenvolvedor. Devido à grande diversidade de soluções passíveis de serem utilizadas, surge a necessidade de se conhecer qual a melhor escolha para o domínio de aplicações web desenvolvidas pelo Centro de Processamento de Dados da Universidade Federal de Santa Maria (CPD/UFSM). O presente trabalho provê uma comparação, a fim de auxiliar no momento da decisão sobre qual *framework* deva ser utilizado, entre os *frameworks* mais conhecidos pelos desenvolvedores de aplicações *web*: SOFIA, Apache Struts e Spring. Com o objetivo de avaliar a arquitetura e os padrões empregados por cada um, foi desenvolvida a aplicação: Documento de Informações Cadastrais (DIC). Essa aplicação representa a versão eletrônica para um processo de cadastro econômico anteriormente em uso pela prefeitura municipal de Campinas (SP). Nesse cadastro, o contribuinte informa os dados referentes às atividades econômicas desenvolvidas, bem como seus dados pessoais. Os resultados da avaliação mostram que o *framework* mais indicado, devido a recursos não presentes nos outros *frameworks*, é o Spring. Porém a curto prazo o Spring não deve ser adotado devido à estrutura presente já construída com Struts.

Capítulo 1

INTRODUÇÃO

Freqüentemente, os desenvolvedores de aplicações se deparam com situações em que os problemas começam a se repetir em diversas partes do sistema. Para resolvê-los, criam-se rotinas que vão sendo replicados por todo o sistema, o que acaba tornando o código facilmente suscetível a erros e demasiadamente replicado.

Conforme a complexidade do sistema aumenta, começam a surgir dificuldades para o gerenciamento de todos os arquivos e objetos envolvidos. Faz-se necessário o uso de soluções semi-prontas, como *frameworks*, para agilizar e tornar mais rápido o desenvolvimento de projetos.

Os *frameworks* são soluções que, na maioria dos casos, seguem padrões de projeto bem definidos. Tais padrões permitem que sejam re-utilizadas soluções para problemas que outros desenvolvedores já enfrentaram. Dessa forma, os *frameworks* tornam-se recursos altamente confiáveis.

Atualmente, existem diversos *frameworks* passíveis de serem utilizados na implementação de aplicações *web*. Devido à essa grande disponibilidade de opções, é preciso que se conheça qual o *framework* mais adequado a um dado domínio de aplicações.

Para a realização do estudo comparativo, escolheu-se os *frameworks* SOFIA, Struts e Spring, por serem largamente utilizados por desenvolvedores do mundo todo. O estudo objetiva comparar as características das três soluções e definir qual deles se adapta melhor ao tipo de aplicação desenvolvida pelo CPD da Universidade Federal de Santa Maria.

Para que os aspectos fossem comparados, fez-se necessário a implementação de uma aplicação *web*. A aplicação consiste no cadastro econômico de contribuintes, o Documento de Informações Cadastrais - DIC. A escolha por tal aplicação deveu-se

por ela representar o perfil das aplicações desenvolvidas no CPD. Essas aplicações costumam ser interativas com o usuário, possuir uma grande comunicação com um banco de dados e serem voltadas para a utilização através de um navegador *web*.

Este trabalho está assim organizado: o capítulo 2 apresenta alguns conceitos importantes para o entendimento do trabalho, o motivo da seleção dos *frameworks* e uma visão geral sobre as características de cada *framework* utilizado no estudo. O capítulo 3 apresenta a aplicação que permitiu a comparação dos *frameworks* escolhidos.

Logo após, no capítulo 4, são explanados os critérios utilizados para a comparação entre os *frameworks* e a comparação entre eles. Por último, no capítulo 5 são apresentados os resultados do trabalho realizado.

Capítulo 2

REVISÃO DE LITERATURA

2.1 PADRÕES DE PROJETO

Um padrão de projeto descreve e provê uma solução para um problema freqüente, sendo genérico e re-usável. São criados a partir de problemas comuns enfrentados no desenvolvimento de projetos de *software* [4].

A criação de componentes reutilizáveis é umas das técnicas mais exploradas em Engenharia de *Software*. O uso de componentes diminui o tempo de produção de um *software* e a taxa de erros de codificação de um projeto, já que disponibiliza uma solução para um problema que é freqüentemente enfrentado e aplica padrões para tais soluções.

O primeiro registro sobre padrões de projeto foi publicado em 1995 por Gamma [1]. Depois dessa publicação, os desenvolvedores procuram adotar a algum padrão especificado pelos autores visando aproveitar suas características.

Um padrão pode ser entendido como a abstração de detalhes sobre a implementação de um *software*. Ao se adicionar uma camada de abstração, pode-se esconder detalhes específicos sobre como um problema foi resolvido, facilitando o entendimento e o relacionamento entre as camadas.

Quando uma empresa ou instituição cria seus padrões, ela deve catalogá-los para que toda a equipe tenha acesso. A equipe, ao ser designada a desenvolver um sistema, deve primeiro consultar esse catálogo a fim de que os padrões sejam seguidos ao máximo. Com a aplicação dos padrões nos projetos, é esperada a redução do trabalho repetitivo e o aumento da qualidade do sistema.

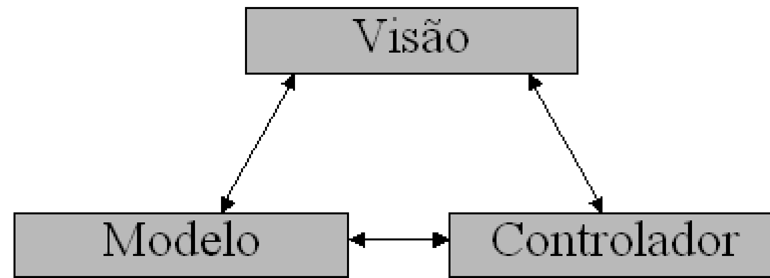

Figura 2.1: Modelo MVC original

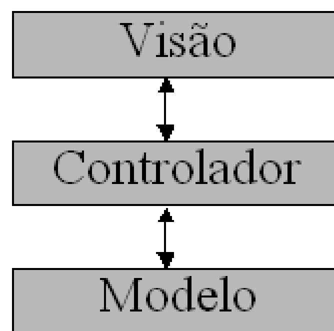

Figura 2.2: Modelo MVC *web*

2.2 O PADRÃO MVC

O padrão mais recomendado para aplicações interativas é o MVC (*Model-View-Controller* - Modelo-Visão-Controlador) [4]. Aplicando-se esse padrão nos projetos de *software*, ganha-se uma separação entre as camadas conforme as suas responsabilidades, o que traz uma maior manutenibilidade ao sistema, pois se uma alteração for necessária, basta apenas modificar a camada específica.

Nas aplicações *web*, o padrão MVC sofre algumas alterações em relação ao padrão original descrito por Gamma [1]. No padrão original, a camada visual é notificada quando modificações na camada modelo acontecem. A figura 2.1 demonstra o modelo original do padrão MVC. Pode-se notar que todas as camadas podem se comunicar entre si, o que não acontece nas aplicações *web* devido ao desacoplamento entre o cliente e o servidor. O padrão MVC modificado para a *web* é mostrado na figura 2.2, ilustrando a comunicação entre as camadas.

A separação das camadas permite o aumento da flexibilidade e re-uso de có-

digo. Sem essa separação, as funcionalidades podem ficar mescladas, o que acarreta um maior esforço para as eventuais manutenções, pois as responsabilidades podem ficar difundidas entre as camadas. O padrão MVC se adapta perfeitamente para a construção de aplicações *web*, pois separa as camadas da aplicação conforme as suas funcionalidades.

2.2.1 CAMADA MODELO

A camada Modelo representa o estado da aplicação. É responsável por fazer a interface da aplicação com a fonte dos dados, muito freqüentemente um banco de dados. Quando existe a necessidade de se guardar o estado da aplicação, é através dessa camada que as informações manipuladas pelo sistema podem ser armazenadas na base de dados.

Devido à maioria dos BDs serem relacionais, em projetos de softwares Orientados a Objetos, faz-se necessário um mapeamento entre as tabelas do BD e os objetos da aplicação. A fim de tornar esses dois paradigmas compatíveis, desenvolveram ferramentas de Mapeamento Objeto Relacional (ORM - *Object Relational Mapping*).

Através do uso de ferramentas de ORM na camada Modelo, pode-se abstrair o conceito de tabelas do banco de dados e trabalhar apenas com objetos. Atualmente, existem diversas ferramentas disponíveis para esse fim, permitindo que a aplicação permaneça orientada a objetos. Normalmente, esse objeto que acessa os dados segue o padrão DAO (*Data Access Object* - Objeto de Acesso aos Dados).

Os objetos da camada modelo são responsáveis por manipular todas as informações que possam ser gravadas ou recuperadas da fonte de dados. Possíveis mudanças na fonte dos dados, implicarão em um menor impacto sobre a aplicação, visto que apenas o objeto envolvido com essa fonte de dados precisa ser modificado, ficando evidenciadas as vantagens da adoção de tal padrão.

2.2.2 CAMADA VISÃO

A camada Visão é composta pela GUI (*Graphical User Interface* - Interface Gráfica de Usuário) e tem a responsabilidade de apresentar a aplicação ao usuário. Tal camada pode ser implementada de diversas formas, desde voltadas para a internet até voltada para o *desktop*.

Nas aplicações voltadas para a *web*, normalmente a camada visual é construída através de páginas HTML. A utilização de HTML, somente, não permite que o conteúdo das páginas seja dinâmico, fazendo-se uso de linguagens que permitam tal funcionalidade.

Para a geração de conteúdo dinâmico, pode-se utilizar as linguagens JSP, ASP, PHP, etc. Tais linguagens permitem que o conteúdo seja construído no servidor, muitas vezes utilizando consultas ao banco, e logo após o resultado do processamento seja enviado ao cliente.

2.2.3 CAMADA CONTROLADORA

A camada Controladora tem o papel de receber os dados digitados pelo usuário e definir o fluxo da aplicação. Assim que recebe os dados, ela executa a validação e passa para a camada de modelo os objetos necessários.

Logo após o processamento, o controlador redireciona as informações para a camada visão, possivelmente com os resultados do processamento, de modo que o usuário possa novamente seguir utilizando o *software*. Em sistemas *web*, essa camada costuma ser implementada através de *Servlets*.

Nas aplicações *web*, freqüentemente, opta-se por definir o fluxo da aplicação em um arquivo XML. Dessa forma, a camada controladora não sofre alterações caso, por exemplo, uma visão mude de nome, e as classes não precisam ser re-compiladas.

2.3 FRAMEWORKS

2.3.1 DEFINIÇÃO

Um *framework* consiste em um conjunto de classes que se relacionam e representam uma solução incompleta. "Um *framework* é o esqueleto de uma aplicação que pode ser customizado por um desenvolvedor da aplicação", de acordo com [7].

Um *framework* deve servir de base para a implementação da aplicação. Os objetos devem ser estendidos e implementados de acordo com o domínio de problema particular, Assis [5]. "*Frameworks* fornecem um mecanismo para obter a re-utilização e são bem apropriados para domínios onde várias aplicações similares são construídas várias vezes, partindo-se apenas de idéias", Fiorini [6]. Os *frameworks* se concentram em resolver problemas que surgem da construção de aplicações com requisitos em comum.

A idéia principal de um *framework* é separar as partes que são modificadas das que permanecem as mesmas, Eckel [2]. Partindo-se desse princípio, pode-se separar o *framework* em partes que ficarão estáticas, desempenhando as suas funções sem modificações, das partes que poderão/deverão ser modificadas para prover uma funcionalidade especial. A parte imutável é comum ao domínio ao qual o *framework* serve e não sofre alterações quando é aplicada em sistemas. Já a parte flexível (*hot-spots*) não representa uma solução completa, pois permite que o desenvolvedor a customize para adequá-la a sua aplicação.

Numa abordagem orientada a objetos, as classes abstratas possuem as soluções comuns. Ao se derivar essas classes, é possível que se aproveite os métodos já definidos e se introduza novos para prover o funcionamento específico desejado, permitindo um alto grau de re-utilização de código.

2.3.2 CONSTRUÇÃO

A criação de um *framework* surge da observação do uso da mesma solução para os problemas que se repetem na tarefa da construção de aplicações. Com o passar do tempo de desenvolvimento, o programador passa a notar que repetidas vezes faz uso das mesmas funções ao longo da codificação dos *softwares*. A partir disso, começam a ser criados objetos que mais tarde poderão fazer parte de um *framework*.

A construção de um *framework* envolve três passos: análise das funcionalidades comuns, definição dos *hot-spots* e projeto do *framework*, Fiorini [6]. O primeiro passo é analisar as aplicações que já foram e que estão sendo desenvolvidas. Partindo dessa análise, pode-se identificar quais características comuns se prolongam pelos softwares e prover a construção de componentes re-usáveis. Aplicando-se os conceitos de Programação Orientada a Objetos, criam-se classes abstratas que servirão de estrutura para as aplicações.

A segunda etapa é definir os *hot-spots*. Depois de conhecer os pontos que permanecerão fixos, é necessário que identificar as partes que variam de acordo com o contexto em que vão ser utilizadas e que necessitam de adaptações para a sua utilização.

A última fase da construção do *framework* é o projeto. Nessa fase, unem-se as partes identificadas nas fases anteriores procurando aplicar os padrões de projeto para garantir que o produto final seja reusável e alcance o objetivo que está disposto a cumprir. Ainda nessa fase, são feitas adaptações para corrigir eventuais defeitos

e para remover componentes não usados, além da adaptação de componentes para prover novas funcionalidades. Tais observações surgem da utilização do *framework* em diversos projetos.

Um *framework* deve ser uma solução re-usável, estável e possuir uma boa documentação, Johnson [3]. A documentação deve ser feita com o intuito de facilitar a aplicação do *framework* em outros projetos. Caso essa documentação não esteja presente, corre-se o risco de o *framework* nunca ser usado. Johnson [3] define os aspectos que a documentação deve possuir:

- propósito: Define para que serve e quais os problemas o *framework* soluciona;
- utilização: Descreve como construir uma aplicação utilizando os componentes já prontos;
- detalhes: Define como os objetos participantes se relacionam.

2.3.3 CLASSIFICAÇÃO

Os *frameworks* podem ser divididos em três categorias de acordo com a sua utilização. Sauvê [8] descreve como os *frameworks* podem ser divididos conforme o domínio onde são utilizados.

1. Middleware: provê a integração entre sistemas, normalmente utilizado em sistemas distribuídos. Um exemplo é o ORB (*Object Request Broker*).
2. Suporte: visa facilitar a construção da infra-estrutura dos sistemas e não resolvem o problema como um todo. São conhecidos como *frameworks* horizontais. O presente trabalho estuda *frameworks* dessa categoria. A figura 2.3 ilustra a composição de um *framework* horizontal. Na figura, pode-se notar que há um alto nível de generalidade nos objetos constituintes do *framework*. Pode-se notar, ainda, que existe uma pequena parte da aplicação já disponível e, devido a isso, uma grande parte da aplicação deve ser implementada.
3. Aplicação: voltado para aplicações, tem a capacidade de construção de aplicações destinadas ao usuário final. Conhecidos por *frameworks* verticais. São exemplos dessa categoria: IBM® São Francisco, *framework* para jogos e monitoração de risco financeiro. A figura 2.4 ilustra a composição de um *framework* vertical. Na figura, pode-se observar que existe uma grande parte da aplicação

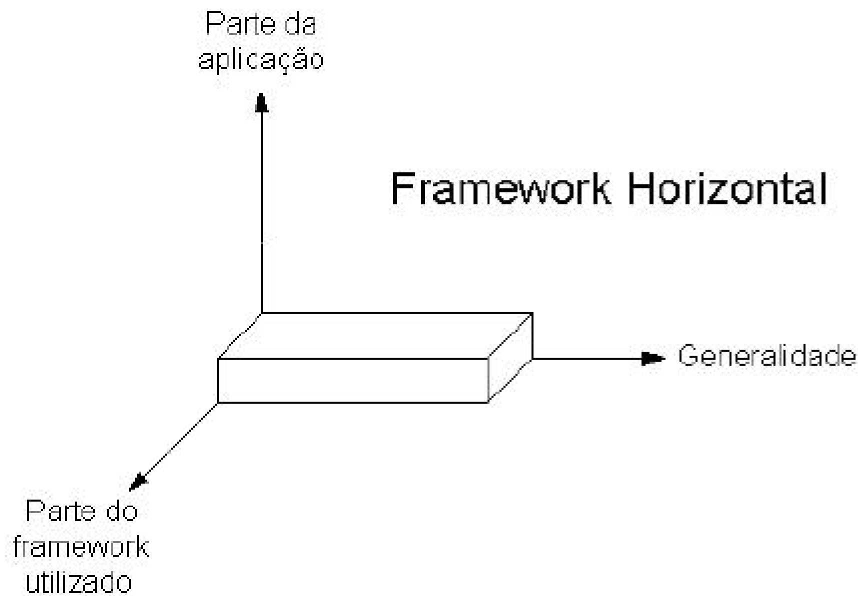

Figura 2.3: Composição de um *framework* horizontal, Sauvê [8]

já implementada, bem como existe uma grande parte do *framework* utilizado. No entanto, fica evidente que a generalidade do *framework* fica reduzida.

2.3.4 VANTAGENS E DESVANTAGENS DO USO DE *FRAMEWORKS*

Sabe-se que o preço do produto final é proporcional ao custo despendido com ele, por isso uma análise mal feita sobre quais recursos serão empregados no projeto poderá acarretar um alto preço de venda. O uso de um *framework* em projetos de *software* traz benefícios e custos que devem ser analisados no momento de se escolher quais ferramentas serão utilizadas na aplicação a ser construída.

As vantagens do uso de *frameworks* nos projetos, de acordo com Assis [5] e Sauvê [8] são:

- baixo tempo de codificação: devido à estrutura semi-pronta, muitas funcionalidades necessárias já estão disponíveis;

- uso de soluções bem testadas por outras pessoas: conforme o uso de *framework* aumenta, estes passam a adquirir maturidade ao se descobrirem erros e adicionar novas funcionalidades;

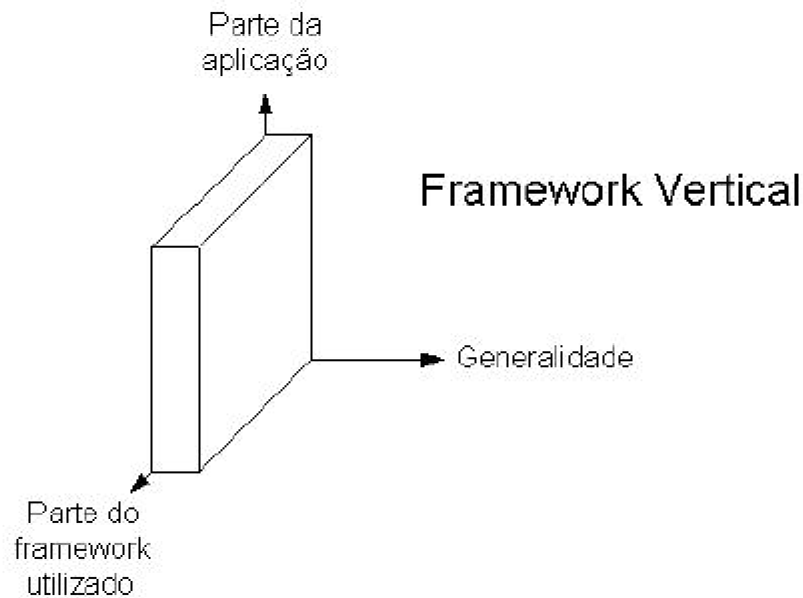

Figura 2.4: Composição de um *framework* vertical, Sauvê [8]

- desenvolvedores se preocupam em implementar o que é necessário: não é preciso que se codifique todo o *software*, pois se utiliza os componente que já estão prontos;

- menor probabilidade de erros nos códigos: com uso de *frameworks*, menos linhas de código são escritas pelos programadores, diminuindo, assim, a possibilidade de erros comuns.

Por outro lado, existem desvantagens provindas do uso de *frameworks*. De acordo com Assis [5] e Sauvê [8] essas desvantagens são:

- *frameworks* requerem pessoas especializadas: para a utilização de um *framework*, deve-se possuir uma equipe com conhecimentos e para isso, é necessário que se façam treinamentos, demandando tempo e aumentando o prazo final para o produto;

- depuração dos programas mais difícil: se o fabricante do *framework* não disponibilizar os códigos-fonte, ficará difícil de se encontrar possíveis erros, visto que estes podem estar contidos nos objetos do *framework*;

- mudança do foco de desenvolvimento: os desenvolvedores têm que assimilar

idéias que, na maioria das vezes, foram propostas por pessoas que não fazem parte da sua equipe de trabalho;

- implementação em linguagem específica: como os *frameworks* são desenvolvidos em linguagens de programação específicas, perde-se portabilidade em relação às linguagens que podem ser usadas em conjunto com o *framework*. Tal restrição obriga os desenvolvedores a utilizar a mesma linguagem empregada pela solução adotada.

2.4 ESCOLHA DOS *FRAMEWORKS*

Para a realização do estudo comparativo, foi necessária uma pré-seleção dos *frameworks* a serem analisados. A escolha pelos *frameworks* SOFIA, Struts e Spring deveu-se pela popularidade que eles possuem junto à comunidade de programadores J2EE^{TM}.

Escolheu-se o *framework* SOFIA versão 2.2, devido a alta integração com ferramentas de suporte ao desenvolvimento. A partir dessa integração, a equipe do CPD poderia aumentar a sua produtividade no desenvolvimento de aplicações, fazendo-se necessário um estudo aprofundado sobre tal *framework*.

O *framework* Struts versão 1.1 foi escolhido por ser considerado um padrão utilizado por desenvolvedores do mundo todo. Devido à tamanha popularidade, uma comparação desse *framework* com os outros tem bases bem sólidas, deixando-o como um ponto de referência.

A terceira escolha, o *framework* Spring versão 1.0.2, é devido a este ser um *framework* que disponibilizado recentemente e por assimilar conceitos novos. Tais conceitos, como Programação Orientada a Aspectos (AOP) e Injeção de Dependência, começam a ser utilizados pelos desenvolvedores. Apesar disso, é crescente o seu uso nos projetos J2EE^{TM}atualmente, pois insere conceitos que ainda não estão presentes em outros *frameworks*.

2.5 *FRAMEWORK* SOFIA

SOFIA (*Salmon Open Framework for Internet Applications*) é um *framework* J2EE^{TM}para construção de aplicações *web*. Desenvolvido pela Salmon LLC, que é uma companhia especializada em desenvolvimento de aplicações Java e *web*, teve a

sua primeira versão disponibilizada em maio de 2002 quando ainda possuía o nome de JADE.

O *framework* SOFIA foi desenvolvido pelos membros da própria companhia com o intuito de permitir que as aplicações fossem construídas mais rapidamente, Salmon [10]. A equipe buscava a integração com outras ferramentas, pois as soluções disponíveis na época foram sendo descontinuadas. A partir disso, os desenvolvedores do *framework* construíram uma solução de rápida e fácil utilização, mas que trouxesse recursos avançados para atender aos mais variados propósitos.

A equipe se norteou por conceitos utilizados em ferramentas do tipo RAD (*Rapid Application Development* - Desenvolvimento Rápido de Aplicações) da época, entre eles a Programação Orientada a Eventos, Salmon [9]. Nesse estilo de programação, os componentes, nesse caso os controladores, são notificados sobre os acontecimentos e mudanças nas outras camadas.

No princípio, a companhia não iria disponibilizar o *framework* para a distribuição, fazendo dele uma ferramenta interna da empresa. No entanto, quando os membros da equipe pesquisaram as outras soluções que estavam surgindo, viram que o que haviam construído não estava muito aquém e resolveram disponibilizar para que os desenvolvedores do mundo inteiro pudessem usá-lo e testá-lo.

O SOFIA possui seu código-fonte aberto (*open source*), é grátis e protegido por dois tipos de licença. Se a versão não for registrada, será regida pela licença GNU GPL (*General Public License* GNU - Licença Pública Geral GNU), na qual diz que o projeto que o usar deve ser disponibilizado com o código-fonte aberto também. Caso ocorra o registro, que não implica em custos, o *software* estará sob a Salmon Software License e poderá ser distribuído com o código-fonte fechado.

2.5.1 CARACTERÍSTICAS

O SOFIA é baseado na arquitetura MVC e os sistemas que o utilizam possuem uma separação sutil entre as camadas. O SOFIA pertence à categoria dos *frameworks* de suporte (horizontais) e é implementado na linguagem de programação Java, o que requer que as aplicações que o utilizam também sejam desenvolvidas na mesma linguagem. Para a construção das camadas, é possível que se utilizem assistentes ou ferramentas que se integram ao *framework*.

Dentre as suas vantagens, está a sua vasta taglib[1] que já vem em conjunto com

[1] Biblioteca de tags JSP que auxiliam no desenvolvimento de sistemas baseados na *web*, Leme[12]

esse *framework*. Através dessa *taglib*, pode-se criar componentes visuais como barras ou árvores de navegação de uma forma que facilita o uso desses recursos visuais, que poderiam levar muito mais tempo de desenvolvimento caso esse recurso não estivesse disponível.

Outra vantagem é a integração com diversas ferramentas do mercado, o que permite que os desenvolvedores possam utilizar os aplicativos que já estão habituados. Para a construção da camada visual, o *framework* pode ser integrado ao Macromedia® DreamWeaver® [20], que é uma ferramenta do tipo WYSIWYG[2]. Com essa integração, são disponibilizados diversos componentes, com funções desde a construção de formulários até a consulta à base de dados.

Para a construção das outras duas camadas - controlador e modelo -, é possível integrar o *framework* a IDEs (*Integrated Development Environment* - Ambiente de Desenvolvimento Integrado), tais como o IntelliJIDEA®[18] e Eclipse®[21], conforme descrito por Salmon [11]. Com isso, é possível que se criem os controladores das páginas através de assistentes que facilitam o processo. Da mesma maneira, pode-se construir os objetos da camada modelo visualizando as tabelas disponíveis no banco de dados de forma *on-line*. A figura 2.5 ilustra a criação dos objetos com a utilização de assistentes. Na figura, pode-se perceber que o assistente provê as tabelas do BD a serem selecionadas para compor o objeto da camada Modelo.

Uma importante característica do SOFIA é que os componentes visuais podem ser acessados através do controlador da página. O controlador de uma página pode acessar, por exemplo, campos de texto, botões e outros, programaticamente, permitindo que alguns apareçam somente em determinadas ocasiões, Salmon [9]. Essa funcionalidade também pode ser conseguida com outros frameworks, porém implica em uma maior lógica nas páginas JSP.

Devido a essa facilidade de acessar os componentes visuais, diretamente na camada controladora as duas camadas tornam-se muito acopladas. Se ocorrer uma mudança dos elementos da página JSP, deverá haver, também, uma mudança no seu controlador, a fim de que os novos componentes sejam adicionados. Tal característica torna-se uma desvantagem no uso do *framework* SOFIA, pois necessita que a classe associada à página modificada seja re-compilada.

Para executar as aplicações desenvolvidas com tal *framework*, é preciso que

[2] *"What You See Is What You Get"*. Os componentes podem ser arrastados para prover as funcionalidades desejadas

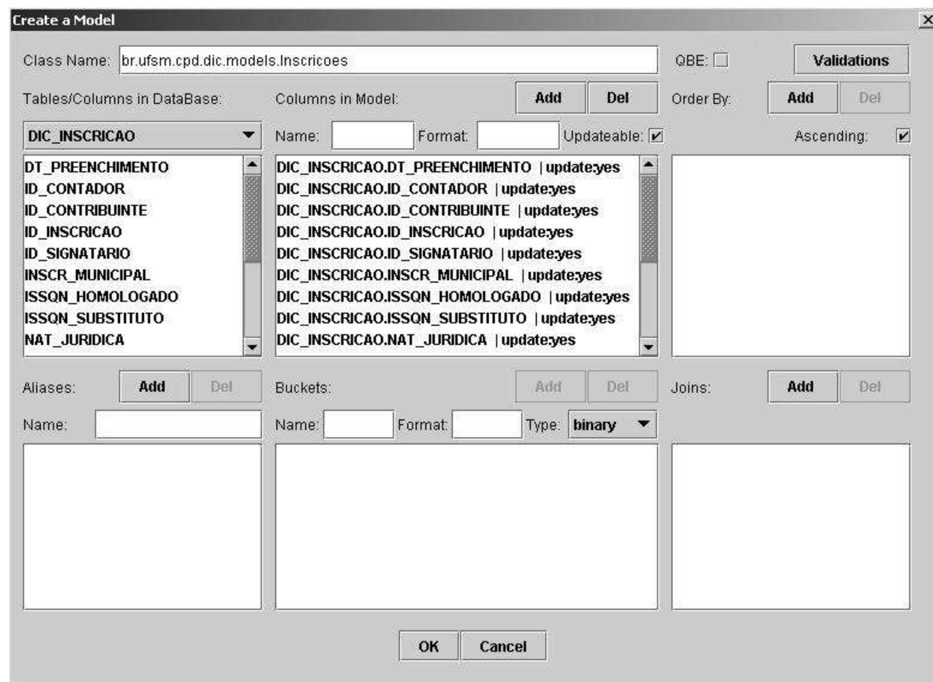

Figura 2.5: Assistente para a criação da camada modelo

se instale o servidor *web* Tomcat[24]. Esse servidor é de fácil uso e instalação, não necessitando muito tempo para configurações. Além disso, é largamente conhecido e utilizado pela comunidade de desenvolvedores, possuindo uma boa documentação.

Essa característica torna-se proveitosa e restritiva, pois acaba deixando a aplicação dependente do servidor ao mesmo tempo. Com isso, não é possível a migração para outros servidores de aplicação, perdendo-se a portabilidade entre plataformas de servidores, pois os objetos do *framework* são dependentes da estrutura construída pelo servidor Tomcat.

2.6 *FRAMEWORK* JAKARTA STRUTS

Criado por Craig R. McClanahan e cedido para o projeto Jakarta da Apache Software Foundation (ASF) em 2000, o Jakarta Struts Framework é um projeto destinado a facilitar o desenvolvimento de aplicações *web*, Cavaness [13]. Atualmente, está na versão 1.2.4 e é um dos projetos da ASF mais conhecidos e desenvolvidos por pessoas do mundo inteiro.

Craig esteve envolvido em diversos grupos que atuaram nas especificações de

Servlets e JSP, desenvolvendo, também, boa parte do código do servidor Tomcat, Cavaness [13]. Mais tarde, se juntou ao grupo que já desenvolvia o *framework* visando aumentar a sua qualidade e diminuir o tempo de construção do mesmo.

A distribuição do *framework* Struts está regida pelos termos da licença Apache 2.0. O *software* que for desenvolvido sob tal licença pode modificar o código-fonte, conforme for necessário, desde que na distribuição haja uma cópia da licença e que explicite que ocorreram mudanças.

Toda a equipe que trabalha no desenvolvimento do *framework* é composta por voluntários não remunerados, Struts [14]. Os desenvolvedores interessados em contribuir, devem assinar as listas de e-mail (*mailing list*) da comunidade, a fim de observar quais discussões estão ocorrendo, saber quais erros foram descobertos, etc, para depois sugerirem alterações.

2.6.1 CARACTERÍSTICAS

O Struts é baseado na arquitetura MVC, por isso permite a separação entre as camadas da aplicação, sendo perfeitamente adequado às aplicações *web*. É um *framework* pertencente à categoria de suporte e é implementado utilizando a linguagem Java. Como é largamente utilizado por desenvolvedores, possui uma grande quantidade de livros publicados e tutoriais disponíveis na internet, o que acaba facilitando o seu aprendizado e uso.

A sua principal característica é prover uma camada de controle flexível baseada em padrões de tecnologia já bem estabelecidos, Struts [14]. Entre as tecnologias usadas pelo *framework*, estão: Servlets, JavaBeans e XML (*eXtensible Markup Language* - Linguagem de Marcação Extensível).

O Struts provê a sua própria camada de controle, bastando que as classes da aplicação estendam a classe `org.apache.struts.action.Action` para que elas executem o processamento desejado. Para a configuração do fluxo da aplicação, é deve-se construir um arquivo (*struts-config*) no formato XML definindo quais ações devem ser mapeadas para os objetos responsáveis. Essa facilidade permite que todo o fluxo do sistema permaneça separado do código-fonte.

A figura 2.6 ilustra um arquivo de configuração do fluxo da aplicação. Pode-se perceber que as ações são mapeadas para as classes responsáveis por fazer o processamento necessário, bem como, as visões que podem ser mostradas após o processamento de tal classe. A camada de visão pode ser desenvolvida a fim de que

```
<action path="/login" type="br.ufsm.cpd.dic.actions.LoginAction"
        name="loginForm" input="/index.jsp" validate="true">
    <forward name="feliz" path="form1.jsp" />
</action>
<action path="/primeiraParte" type="br.ufsm.cpd.dic.actions.PrimParteAction"
        name="form1" input="/form1.jsp" validate="true">
    <forward name="localizaImovel" path="localizar_imovel.jsp" />
    <forward name="felizAviso" path="aviso.jsp" />
    <forward name="felizF" path="form2.jsp" />
    <forward name="felizJ" path="form2_jur.jsp" />
</action>
<action path="/segundaParte" type="br.ufsm.cpd.dic.actions.SegParteAction"
        name="form2" input="/form2.jsp" validate="true">
    <forward name="volta" path="form1.jsp" />
    <forward name="localizaAtividades" path="localizar_ativ_economica.jsp" />
    <forward name="felizFisica" path="form4.jsp" />
    <forward name="feliz" path="form3.jsp" />
</action>
<action path="/terceiraParte" type="br.ufsm.cpd.dic.actions.TercParteAction"
        name="form3" input="/form3.jsp" validate="true">
    <forward name="volta" path="form2.jsp" />
    <forward name="feliz" path="aviso.jsp" />
</action>
<action path="/quartaParte" type="br.ufsm.cpd.dic.actions.QuarParteAction"
        name="form4" input="/form4.jsp" validate="true">
    <forward name="voltaFisica" path="form2.jsp" />
    <forward name="volta" path="form3.jsp" />
    <forward name="feliz" path="concluir.jsp" />
</action>
```

Figura 2.6: Arquivo de configuração do fluxo da aplicação.

se torne totalmente internacionalizável. Todas as páginas podem ser construídas de modo a permitir que elas assumam o idioma do país do visitante. Tal funcionalidade é provida por tags disponibilizadas juntamente com o *framework*.

Para auxiliar na criação das páginas JSP, existe a taglib que faz parte da distribuição do *framework*. As tags que compõem essa taglib possuem funções que permitem desde a criação de formulários até a renderização de erros provenientes da camada de controle.

Para a camada do modelo pode-se escolher entre várias tecnologias, dentre as quais: JDBC[3], EJB[4]) ou ferramentas ORM. A escolha por qual tecnologia utilizar depende da arquitetura que o desenvolvedor possui previamente. A construção dessa camada se mostra bastante flexível, pois permite que padrões já bastante conhecidos possam ser utilizados, o que traz um alto grau de portabilidade entre os servidores de aplicação.

[3] *Java Data Base Connectivity*. Tecnologia que provê conectividade entre diversos Gerenciadores de Bancos de Dados [26]

[4] *Enterprise JavaBeans*. Componente executado no servidor que simplifica o desenvolvimento de aplicações distribuídas [16].

2.7 FRAMEWORK SPRING

Desenvolvido por Rod Johnson, Jüergen Höeller e sua equipe, o Spring Framework tem o intuito de ser uma solução rápida e leve, para a construção de aplicações J2EE™. A sua arquitetura é baseada no modelo MVC, similarmente aos frameworks já citados, de acordo com Johnson [15]. Após o sucesso desse *framework*, os inventores se reuniram e fundaram a companhia Interface21 em Londres a fim de prestar suporte pago aos usuários.

A distribuição do *software* pode ser adquirida de forma gratuita com o seu código-fonte incluído. A licença que rege esse *framework* é a Apache 2.0, que permite que o seu código-fonte seja modificado, desde que as alterações estejam explícitas para a comunidade.

Para diminuir os custos de manutenção, os desenvolvedores orientam os usuários do *framework* para que o *software* seja orientado à interfaces. Com essa abordagem, não se constroem classes concretas diretamente, e sim interfaces que serão implementadas. Dessa forma, abstraem-se os detalhes da construção dos métodos e facilita a manutenção da aplicação.

2.7.1 CARACTERÍSTICAS

Os desenvolvedores desse *framework* acreditam que as aplicações não devem depender ou depender o menos possível do Spring. Com base nessa idéia, foi criado um *framework* totalmente modular de forma que apenas as partes que a aplicação realmente precise tenham que ser dependentes.

Devido à estrutura ser dividida em módulos, existe um bom suporte para a programação. A partir dessa estrutura, o desenvolvedor pode aproveitar uma grande quantidade de funcionalidades que já estão disponíveis, o que acaba diminuindo o tempo de codificação e, com isso, o custo geral do sistema.

Entre os principais módulos que compõem o Spring estão: Spring AOP, Spring ORM, Spring Web e Spring Web MVC. O módulo AOP (*Aspects Oriented Programming* - Programação Orientada a Aspectos) permite que a aplicação possa utilizar os conceitos da programação orientada a aspectos de forma totalmente integrada ao *framework*, Johnson [15]. Uma das vantagens de se usar esse módulo é que o sistema pode definir métodos interceptadores para que se alcance as funcionalidades desejadas sem que o código-fonte da aplicação seja alterado, Johnson [15]. A figura

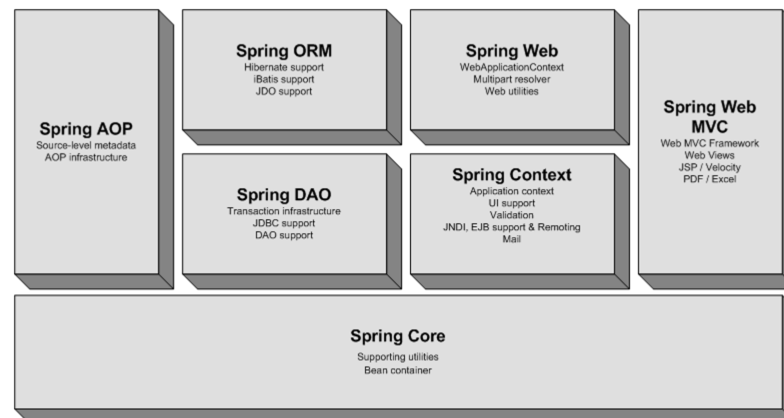

Figura 2.7: Módulos do Spring Framework - Johnson [15]

2.7 ilustra a arquitetura de módulos que compõe o *framework* Spring.

O módulo ORM permite que a aplicação utilize objetos de acesso aos dados na camada Modelo. Através desse módulo, a aplicação pode se integrar com ferramentas ORM, tais como Hibernate [19] e iBATIS [22], mesmo que não utilizem os outros módulos do *framework*. Através desse módulo, pode-se usar as vantagens dessas ferramentas com as vantagens do Spring, tais como gerenciamento de transações declarativo.

O gerenciamento de transações declarativo permite que os métodos transacionais possam ser declarados em um arquivo de configuração, separadamente do código-fonte da aplicação. Utilizando-se essa abordagem, o código da aplicação permanece independente do mecanismo de gerenciamento de transações, facilitando a manutenção do código-fonte.

O módulo MVC permite que a aplicação seja desenvolvida totalmente baseada no modelo MVC. É possível que se construam objetos para a camada de controle para suportarem formulários *web* para entrada dos dados e validações do próprio *framework*. Além disso, possui objetos que permitem que se formulários na forma de assistentes.

O Spring, ainda, permite que se use o conceito de Inversão de Controle (IoC) ou Injeção de Dependência, termo mais correto segundo Johnson [15]. Tal conceito quer dizer que o servidor de aplicações, juntamente com os objetos do *framework*, pode fazer o preenchimento dos objetos na hora em que são criados. Dessa forma, o programador fica dispensado de programar tais tarefas.

Com o emprego de conceitos novos, como o AOP, o custo de treinamento torna-se elevado. Esses conceitos demandam pessoas que já tenham um conhecimento aprofundado na plataforma Java, o que acaba aumentado os custos do desenvolvimento de aplicações que o utilizam. No entanto, pode-se achar diversos tutoriais e artigos na internet que abordam suas características, vantagens e desvantagens.

Capítulo 3

DOCUMENTO DE INFORMAÇÕES CADASTRAIS - DIC

Este capítulo visa explicar a aplicação desenvolvida para a comparação entre os *frameworks*. A versão eletrônica do Documento de Informações Cadastrais (DIC) foi desenvolvida para a prefeitura municipal de Campinas (SP) com o intuito de facilitar o processo manual que antes existia.

Toda pessoa que deseja desenvolver uma atividade econômica no município de Campinas, deve preencher o DIC. Assim que o contribuinte especifica seus dados, um membro da prefeitura os analisa e, caso esses sejam válidos, atribui um número de inscrição municipal ao contribuinte. Essa inscrição municipal permite que a pessoa exerça a sua profissão de forma legal e contribui para a prefeitura identificar os seus contribuintes.

Antes da implantação do sistema eletrônico, todos os contribuintes do município deveriam se deslocar até a sede da prefeitura e preencher o formulário em papel. A tarefa de preenchimento tornava-se repetitiva e demorada uma vez que, para serem feitas alterações no documento, era preciso que o contribuinte preenchesse novamente todos os seus dados, especificando as alterações feitas.

O formulário eletrônico foi dividido em três partes: dados pessoais e endereço, atividade desenvolvida e observações gerais. Na primeira parte, como mostra a figura 3.1, são requisitados os seguintes dados: nome, CPF, endereço e se o contribuinte já possuiu ISSQN[1]. Para o preenchimento do endereço, pode-se executar pesquisas no BD para localizar o imóvel ao qual o contribuinte reside. Essas pesquisas podem

[1]Imposto Sobre Serviços de Qualquer Natureza

Figura 3.1: Primeira parte do formulário do DIC.

ser feitas com o nome do logradouro, a matrícula ou, ainda, o código estruturado do imóvel[2]. A informação fornecida pelo usuário é confrontada com uma listagem de endereços da prefeitura possui que irá gerar os resultados encontrados.

Para preencher a segunda parte do formulário, ilustrada na figura 3.2, é necessário que seja identificado o código da atividade exercida pelo contribuinte e a data de início do exercício. Na forma antiga, a localização da atividade era feita manualmente, através de listagens de papel. No atual sistema, o usuário pode consultar uma listagem das atividades econômicas disponíveis, visualizando o nome da atividade e o código respectivo. Caso a profissão exercida pelo contribuinte for de natureza autônoma, este, ainda, deverá especificar a entidade de classe ao qual ele está ligado, caso a profissão exija.

A terceira parte é destinada às observações. É nessa parte que o contribuinte

[2] Código geográfico do imóvel localizado no mapa do município.

Figura 3.2: Segunda parte do formulário do DIC.

deverá preencher observações que julgar importantes para os analisadores da prefeitura. A última parte do formulário é ilustrada na figura 3.3, onde se pode notar o campo para o preenchimento das observações.

A camada visual foi desenhada por um profissional da área de *webdesign* a fim de se respeitar a identidade visual existente da prefeitura. A ajuda de tal profissional é importante para que a disposição da informação (*layout*) para o usuário possa ser feita da melhor forma possível.

A base de dados utilizada na aplicação provém de informações cadastradas através do SIM - Sistema de Informações Municipais - previamente implantado na prefeitura. Através desse sistema, a prefeitura pôde digitalizar os cadastros que possuía, criando uma vasta fonte de dados confiável.

Toda a informação coletada pelo DIC é armazenada em uma base de dados temporária. Após o preenchimento do usuário e da análise dos membros da prefeitura, o SIM atualiza o seu cadastro com o dados do DIC para complementar o cadastro econômico do contribuinte.

Figura 3.3: Terceira parte do formulário do DIC.

Capítulo 4

COMPARAÇÃO DOS *FRAMEWORKS*

4.1 CRITÉRIOS DA COMPARAÇÃO

A fim de que o estudo pudesse ser realizado, foram estabelecidos critérios para a execução da comparação entre os *frameworks*. Os critérios adotados foram: comparação da implementação do MVC, validação dos dados recebidos pela camada visual e a implementação dos objetos de acesso aos dados.

O estudo comparativo possuiu duas fases: fase teórica e fase prática. Na fase teórica, estudou-se os tipos de licenças, as documentações e as principais características que cada framework possui. Para a realização da parte prática, foi necessário o desenvolvimento de uma aplicação para que a arquitetura, a validação dos dados e a implementação dos Objetos de Acesso aos Dados de cada *framework* pudessem ser melhor estudados.

O perfil das aplicações *web*, em geral, exige que o padrão MVC seja adotado, a fim de que as vantagens descritas na seção 2.2 possam ser alcançadas. Dessa forma, torna-se importante uma comparação entre as metodologias adotadas por cada *framework* para a construção das camadas do padrão MVC.

A forma com que a validação dos dados digitados pelo usuário é executada é de fundamental importância para que a aplicação torne-se confiável. Caso a validação de um determinado *framework* não seja corretamente projetada, podem acontecer casos onde os dados da camada Visão tornem a aplicação instável ou até a leve a comportamentos danosos ao sistema. Com isso, fica evidente que uma comparação desse mecanismo entre os *frameworks* torna-se necessária.

Através da implementação dos objetos de acesso aos dados obtida com a ado-

ção de cada *framework*, é possível que se saiba quais fontes de dados a aplicação suportará, bem como os impactos na mudança dessa fonte de dados. Dessa forma, uma comparação entre as formas com que cada *framework* possibilita a construção de tal camada torna-se importante.

4.2 CAMADA MODELO

4.2.1 SOFIA

O *framework* SOFIA possui suporte próprio para a construção da camada Modelo. Dessa forma, o desenvolvedor não precisa utilizar outros mecanismos para que a aplicação acesse a base de dados.

Cada objeto que precisa ser persistido na base de dados deve possuir os métodos responsáveis por pesquisas e gravações, o que acaba replicando trechos de código que poderiam ser reaproveitados entre os objetos. Devido a essa abordagem, a camada Modelo acaba tornando-se acoplada ao SOFIA e não possibilita que ela seja portada para outros *frameworks*, caso seja necessário. Tal característica deve ser levada em consideração no momento da escolha sobre qual *framework* a aplicação utilizará.

No entanto, para a criação da camada Modelo, o SOFIA provê um assistente quando integrado com os IDEs Eclipse® ou IntellijIDEA®. Esse assistente permite que sejam visualizadas as tabelas do BD que poderão compor o objeto, facilitando a tarefa de construção das classes. Através do assistente, pode-se definir as ligações necessárias entre as tabelas conforme critérios desejados, o que dispensa a criação de visões sobre a base de dados.

Como o SOFIA possui a sua própria implementação da camada Modelo, a aplicação somente poderá utilizar os BDs que ele suportar: Sybase®, SQLAnywhere®, MSSQLServer®, IBM® DB2®, Oracle® e MySQL®. Devido a esse baixo suporte, uma possível troca de BD pode tornar-se impossível.

Para a construção de consultas ao BD, o SOFIA não provê mecanismos muito eficientes e confiáveis. As cláusulas SQL necessárias para a realização de consultas devem ser especificadas junto ao código-fonte das classes, fazendo com que possíveis alterações na base de dados forcem uma recompilação das classes que executam consultas.

A construção das cláusulas SQL dá-se através da concatenação das *Strings*

fornecidas para formar as pesquisas necessárias. Com isso, os comandos SQL resultantes, se não previamente tratados, podem conter códigos maliciosos para ataques de SQL *Injection*[1]. Tais ataques podem corromper o BD e permitir que pessoas desautorizadas acessem dados restritos.

Para prevenir esses ataques, há que se fazer uma verificação dos dados digitados pelo usuário. Com isso, cada consulta às tabelas, a partir de dados fornecidos pelo usuário, tem que ser cuidadosamente verificada para que sejam identificados comandos ilegais, o que aumenta o processamento.

4.2.2 STRUTS

O *framework* Struts possibilita que a camada Modelo seja altamente flexível, podendo-se utilizar diversas tecnologias disponíveis. O padrão utilizado, normalmente, para a camada de Modelo é o padrão DAO, disponibilizando as vantagens citadas na seção 2.2.

Entre as tecnologias passíveis de serem empregadas na camada Modelo das aplicações que utilizam o Struts, está o Hibernate. O Hibernate é uma ferramenta de Mapeamento Objeto-Relacional que permite que a aplicação não mais utilize o conceito de tabelas de dados e sim objetos que as representam. Além disso, a aplicação torna-se mais flexível em relação ao BD utilizado.

O Hibernate suporta aproximadamente 20 tipos de BDs diferentes. Com o emprego do Hibernate na camada Modelo, ganha-se flexibilidade, pois pode-se migrar entre diversos BDs sem que a aplicação tenha que ser modificada, devido a estrutura de dialetos que ele utiliza. Tal estrutura permite que apenas se modifique o dialeto de acordo com o BD utilizado pela aplicação.

Algumas modificações nas tabelas do BD como, por exemplo, a troca de nome de uma coluna, não implicam em recompilações dos objetos da aplicação. Os objetos que utilizam dados das tabelas possuem arquivos de configuração no padrão XML que mapeiam as suas propriedades às colunas do BD. Se em algum momento do ciclo de vida da aplicação for necessário que se altere o BD, basta que seja alterado apenas o arquivo de configuração e as classes permanecem com o funcionamento normal.

As cláusulas SQL necessárias para consultas ao BD são automaticamente cons-

[1] Espécie de ataque no qual são inseridos comandos SQL através dos dados de entrada da aplicação.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
<hibernate-mapping package="br.ufsm.cpd.dic.beans" >
    <class name="InscricaoBean" table="DIC_INSCRICAO">
        <id name="idInscricao" column="ID_INSCRICAO" unsaved-value="0">
            <generator class="sequence">
                <param name="sequence">ID_DIC_INSCRICAO</param>
            </generator>
        </id>
        <property name="inscricaoMunicipal" column="INSCR_MUNICIPAL"/>
        <property name="senha" column="SENHA"/>
        <property name="observacoes" column="OBSERVACOES" />
        <property name="dtPreenchimento" column="DT_PREENCHIMENTO" />
        <property name="issqn" column="POSSUIU_ISSQN"/>
        <property name="natJuridica" column="NAT_JURIDICA" />
        <property name="status" column="STATUS" />
    </class>
</hibernate-mapping>
```

Figura 4.1: Trecho de arquivo de um mapeamento de objeto.

truídas pelo Hibernate. Com a geração automática das consultas, o sistema torna-se mais confiável e seguro, pois previne possíveis erros dos programadores. Outras consultas mais rebuscadas podem ser escritas no arquivo de configuração do Hibernate, separando-as do código-fonte da aplicação e, com isso, facilitando a manutenção do sistema.

A figura 4.1 ilustra um exemplo de arquivo de configuração do Hibernate. Na figura, pode-se notar o mapeamento da propriedade `observacoes` do objeto `br.ufsm.dic.beans.InscricaoBean` com a coluna do BD `OBSERVACOES` através dos atributos *property* e *column* do arquivo de configuração. Cabe ressaltar que, embora a aplicação possa utilizar o Hibernate ou qualquer outro mecanismo ORM na camada Modelo, o Struts não provê nenhum suporte específico para essas ferramentas. As configurações necessárias para que essas ferramentas possam funcionar corretamente, devem ser executadas por objetos criados pelo desenvolvedor em conjunto com o servidor de aplicações. Com isso, o *framework* permanece independente da solução adotada para a camada Modelo, porém requer mais esforço do programador da aplicação.

4.2.3 SPRING

A camada Modelo do *framework* Spring trabalha de maneira análoga à camada Modelo do Struts. Normalmente, adota-se o padrão DAO para acesso à fonte de dados, conferindo uma melhor manutenção à aplicação.

A fim de facilitar a manipulação dos dados do BD, pode-se utilizar ferramentas de mapeamento objeto-relacional. Assim como o Struts, é possível que a aplicação trabalhe com a base de dados em um formato orientado a objetos, bastando que se façam os mapeamentos necessários dos objetos às tabelas do BD.

O Spring fornece pacotes que facilitam a integração do *framework* com o Hibernate. Ao se desenvolver com o Hibernate, pode-se estender a classe `org.springframework.orm.hibernate.support.HibernateDaoSupport` do Spring, pois ela oferece suporte às rotinas mais comumente utilizadas, tais como acesso à sessão do Hibernate (`getSession`) e acesso ao objeto que executa consultas (`getHibernateTemplate`).

Além do suporte ao Hibernate na camada Modelo, pode-se ainda utilizar iBATIS [22] ou JDO [23] para acesso aos dados. Esses mecanismos funcionam de maneira semelhante ao Hibernate, abstraindo a maneira como os dados são armazenados, porém são desenvolvidos por equipes diferentes, sendo que o JDO é implementado pela Sun® [25], criadora da linguagem Java.

O Spring conta, ainda, com o suporte a transações na camada Modelo. Dessa forma, todos os conceitos de transações são respeitados, fazendo com que os dados sejam confiáveis e o BD não contenha informações inconsistentes.

O gerenciamento de transações ocorre em conjunto com a AOP, definindo-se quais métodos do objeto devem ser gerenciados. No momento em que aplicação solicita um método gerenciado, o *framework* intercepta essa requisição e sinaliza o início de uma transação.

O método requisitado executa o seu processamento normalmente e, quando o retorno acontece, a transação é concluída. Se um processamento anormal acontece como, por exemplo, uma exceção, o gerenciador a detecta e desfaz todas as alterações que já haviam sido feitas por esse método.

A configuração do gerenciador de transações pode ser escrita nos arquivos de configuração do contexto da aplicação, de modo declarativo, seguindo o padrão XML. A grande vantagem dessa abordagem é que a lógica do gerenciamento de transações permanece separada da lógica de acesso aos dados. Com isso, permite-se

```
<!-- ======================== Transacao ======================== -->
<!-- Gerenciador de transacoes -->
<bean id="transactionManager"
    class="org.springframework.orm.hibernate.HibernateTransactionManager">
    <property name="sessionFactory"><ref local="sessionFactory"/></property>
</bean>

<!-- Proxy transacional para o Bean de acesso aos dados -->
<bean id="bdBean"
    class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager"><ref local="transactionManager"/></property>
    <property name="target"><ref local="bdBeanTarget"/></property>
    <property name="transactionAttributes">
        <props>
            <prop key="load*">PROPAGATION_REQUIRED,readOnly</prop>
            <prop key="save*">PROPAGATION_REQUIRED</prop>
        </props>
    </property>
</bean>
<!-- ======================== /Transacao ======================== -->

<!-- ======================== DAOs ======================== -->
<!-- Objeto que faz acesso aos dados (DAO) -->
<bean id="bdBeanTarget" class="br.ufsm.cpd.dic.beans.BdBean">
    <property name="sessionFactory"><ref local="sessionFactory"/></property>
</bean>
<!-- ======================== /DAOs ======================== -->
```

Figura 4.2: Configuração do gerenciamento de transações do *framework* Spring.

que outros mecanismos sejam implementados sem que os objetos da camada Modelo sejam modificados.

A figura 4.2, demonstra a configuração do gerenciamento de transações através do arquivo de configuração do *framework*. Na figura, pode-se perceber que estão sendo gerenciados todos os métodos que começam por *load* e *save* do objeto `br.ufsm.cpd.dic.beans.BdBean`.

4.3 CAMADA VISÃO

A camada visual é a parte em que o usuário final irá interagir com o sistema. Essa camada deve ser amigável ao usuário para permitir que a sua utilização se dê de maneira fácil e intuitiva, tornando o seu desenvolvimento uma tarefa muito importante.

A forma mais comum dos *frameworks web* proverem o suporte para a camada visual é através de *tags* JSP. Tais *tags* são interpretadas pelo servidor de aplicações e, no momento em que a página é requisitada pelo cliente, o servidor executa os

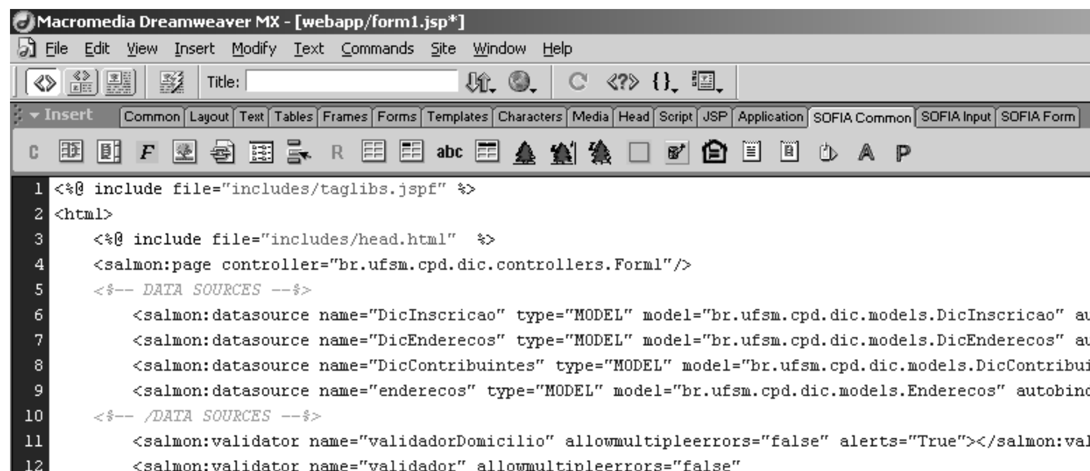

Figura 4.3: MacroMedia DreamWeaver integrado ao *framework* SOFIA.

métodos das classes que as representam e o resultado é enviado ao cliente.

Nas aplicações *web*, geralmente, a parte visual assume o formato de páginas JSP. Para compor as páginas JSP, faz-se necessário que os *frameworks* possuam componentes na forma de *tags* para que essa camada possua todas as funcionalidades necessárias e permita que o seu desenvolvimento torne-se menos dispendioso.

4.3.1 SOFIA

O *framework* SOFIA caracteriza-se por permitir um rápido desenvolvimento da camada Visão. É possível que o SOFIA seja integrado, de maneira fácil, com o aplicativo MacroMedia® DreamWeaver® [20], que é uma ferramenta do tipo WYSIWYG especialmente voltada para designers gráficos.

Com o intuito de facilitar ao máximo a construção das páginas JSP da aplicação, os seus desenvolvedores propiciaram uma *taglib* com mais de 60 *tags*. Essa *taglib* fica disponível para ser acessada através da barra de ferramentas do *software* citado, permitindo que pessoas que não possuem um conhecimento elevado em programação possam construir as páginas.

A figura 4.3 demonstra o aplicativo MacroMedia DreamWeaver integrado ao *framework* SOFIA. Pode-se perceber que na barra de ferramentas são disponibilizadas diversas *tags* a serem utilizadas na construção de páginas JSP.

Entre as *tags* do *framework*, pode-se citar: *DataTable*, *Calendar* e *NavigationBar*. A *tag DataTable* permite que uma tabela do BD seja automaticamente ligada

a uma tabela visual dentro da página. Dessa maneira, os dados presentes no BD são mostrados automaticamente, podendo, inclusive, mostrar alterações recentemente feitas na base de dados.

Além de apresentar os dados das tabelas do banco de dados, a *tag DataTable* permite que a listagem possua paginação. Através desse recurso, o conteúdo a ser mostrado pode se estender por diversas páginas, permitindo que apenas uma porção dos dados seja visualizada sempre. Tal mecanismo facilita a navegação, pois permite que se acessem poucos ítens por vez, diminuindo o tempo de espera do usuário.

A *tag Calendar* permite que um calendário seja criado de forma dinâmica. Em lugares da aplicação onde é necessária a especificação de datas, pode-se auxiliar o usuário através da utilização dessa tag, pois no momento que ela é renderizada, é mostrado um calendário semelhante aos utilizados diariamente pelas pessoas. Para a utilização dessa *tag*, deve-se especificar apenas o ano que deve ser apresentado e algumas configurações de fontes do texto.

A *tag NavigationBar* permite a construção de uma barra de navegação do site. Tal funcionalidade auxilia a navegação dos visitantes através do site, pois pode-se acessar todo o conteúdo de maneira rápida e prática. Para que as páginas possuam esse recurso, basta especificar, através de links, o seu conteúdo, além de permitir que imagens sejam colocadas juntamente.

Além das *tags* para a construção da camada visual, pode-se contar ainda com a capacidade de personalização das páginas pelo *framework*. Pode-se definir que, no instante que um usuário específico acessar o site, as páginas assumirão um aspecto visual diferenciado, através de temas[2] além de permitir que somente alguns componentes fiquem visíveis. A diferenciação dos componentes das páginas pode ser útil para restringir que somente usuários com certos privilégios possam acessar ítens administrativos.

Um dos aspectos negativos do *framework* refere-se à falta de internacionalização. A internacionalização serve para que o conteúdo das páginas seja apresentado conforme o idioma de origem do visitante. Com a utilização do *framework* SOFIA, o conteúdo das páginas terá que ficar disponível em somente um idioma, o que representa uma desvantagem em relação aos outros *frameworks*.

Outra maneira de oferecer suporte à camada visual, além de páginas JSPs, é a utilização de componentes Java Swing. Se alguma parte da aplicação já havia

[2] Estilos de cores aplicados sobre partes do sistema, também chamados skins

sido implementada para um *software* do tipo Desktop, esses componentes podem ser utilizados na aplicação *web*. Para que ocorra tal integração, basta que as classes sofram algumas modificações e estendam as classes necessárias do *framework*.

4.3.2 STRUTS

O *framework* Struts caracteriza-se por possuir uma diversificada e padronizada *taglib*. Desde a sua criação, várias *tags* foram sendo adicionadas, a fim de permitir que se consiga desenvolver aplicações totalmente baseadas nas suas *tags*.

Os desenvolvedores do *framework* Struts preocuparam-se com todas as possíveis rotinas que seriam necessárias para a construção das páginas. Construção de formulários, campos de entrada de texto, visualização de imagens, acesso a JavaBeans[3] e estruturas lógicas de controle integram a *taglib* do Struts.

Para a utilização de elementos HTML nas páginas, o Struts fornece aproximadamente 30 *tags*. Dentre elas, destacam-se: *OptionsCollection*, *Errors* e *Messages*. A *tag OptionsCollection* permite que seja construído um elemento HTML do tipo *select* no qual se pede que o usuário selecione uma opção entre várias provindas de uma coleção especificada pelo desenvolvedor. Para a sua utilização, é necessário, apenas, que se configure a coleção que deverá ser renderizada.

A *tag Errors* permite que sejam visualizadas mensagens de erros. Ao se preencher um formulário, alguns erros podem ser verificados e o desenvolvedor do sistema pode diagnosticar tais anomalias, enviando mensagens de erro a serem mostradas ao usuário. A *tag Messages* trabalha de maneira semelhante, pois permite que mensagens de quaisquer propósitos, não somente de erros, possam ser mostradas aos usuários.

Além de fornecer as *tags* para elementos HTML, o Struts ainda possui *tags* para acesso a JavaBeans. As propriedades dos objetos podem ser acessadas de maneira simples, bastando especificar a propriedade a ser acessada e o *bean* que a possui. Dessa maneira, pode-se tornar as páginas dinâmicas ao se mostrar o conteúdo de objetos que estejam no escopo das páginas.

A *taglib* do Struts, ainda, fornece *tags* para estruturas de controle. Essas *tags* possibilitam iterações por coleções, controle de condições, verificação de parâmetros e redirecionamento das páginas. As *tags* para estruturas de controle permitem que

[3]"Componente de software portável e independente de plataforma escrito em Java", segundo DeSoto[17]. Possui métodos para acesso e configuração de suas suas propriedades

qualquer processamento possa ser executado sem que para isso seja preciso a inserção de scriptlets[4] nos arquivos JSP, facilitando o trabalho em equipes multidisciplinares.

Para o uso de listagens de dados, costuma-se integrar o Struts com a *taglib DisplayTag*, visto que o *framework* não possui componentes próprios para tal recurso. Essa *taglib* provê *tags* que possuem padrões de apresentação de alto nível, permitindo que, entre outras funcionalidades, uma lista de dados a ser exibida ao usuário possa ser paginada ou ordenada, facilitando a navegação.

O *framework* Struts possui suporte a internacionalização através de *tags* próprias associadas a arquivos de propriedades. Esses arquivos são mais comumente chamados de *Resource Bundles* e permitem que se configure as messages e os textos das páginas a serem apresentados para os usuários.

Para a utilização da internacionalização, é preciso, apenas, que se identifique a mensagem e o arquivo que possui o texto a ser apresentado. É preciso que se construa um arquivo com as mensagens traduzidas para cada idioma suportado. No momento que o browser envia o idioma do visitante para o servidor, o *framework* o detecta e opta por qual arquivo utilizar.

Além de JSP, o Struts permite que a apresentação da aplicação seja construída com XSLT (*eXtensible Stylesheet Language Transformation*), na qual as páginas são construídas seguindo um padrão XML. No instante que a página é requisitada pelo browser, o servidor executa uma transformação para aplicar os estilos visuais configurados.

4.3.3 SPRING

O *framework* Spring não possui uma *taglib* muito extensa, uma vez que aconselha que sejam utilizadas a JSTL[5] provida pela Sun. Dessa forma, o Spring não oferece suporte completo ao desenvolvimento das páginas JSPs quando se utiliza apenas as suas *tags*.

Entre as *tags* que possui, pode-se citar: *Message* e *Bind*. A *tag Message* funciona de maneira análoga a *tag* Messages do *framework* Struts. Basta que se especifique a mensagem a ser mostrada ao usuário e o arquivo que a contém para que ela seja renderizada na página HTML.

A *tag Bind* é utilizada para vincular um componente HTML a uma propriedade

[4]Trechos de código Java inseridos nas páginas HTML através das *tags* <% e %>.
[5]Biblioteca de tags JSP padrão.

de um JavaBean. Essa *tag* torna-se bastante útil para a renderização de valores que estejam armazenados nos objetos.

Para a paginação de listas de dados, o Spring possui seu próprio mecanismo. Na camada Controladora, deve-se especificar qual a lista de dados deve ser mostrada e colocá-la na sessão do usuário, especificando-se o número de ítens por página. No instante em que os dados são mostrados ao usuário, é feita a paginação da lista.

Ao se utilizar os mecanismos do Spring para paginação, perde-se a portabilidade entre outros *frameworks*. No entanto, o Spring provê suporte à *taglib* DisplayTag, já discutida na subseção 4.3.2.

O suporte à internacionalização acontece de maneira idêntica ao *framework* Struts. O desenvolvedor precisa construir um arquivo para cada idioma suportado com as mensagens a serem apresentadas ao usuário.

No momento em que o browser do usuário envia uma requisição para uma página, o *framework* detecta o seu idioma e define qual será o arquivo de recursos a ser utilizado. Nos arquivos JSP, basta que se utilize a *tag* Message para que as mensagens sejam renderizadas.

O Spring permite a utilização de várias tecnologias para a camada visual, como XSLT e documentos Excel ou PDF. Para o uso de XSLT, as páginas devem possuir um padrão XML com a definição dos estilos a serem utilizados. Quando o arquivo é requisitado, o servidor aplica os estilos necessários e envia para o cliente.

Para a integração com Excel, o controlador deverá estender a classe `org.springframework.web.servlet.view.document.AbstractExcelView` e implementar o método `buildExcelDocument` executando o processamento necessário para a construção do arquivo. Já para a construção de arquivos PDF, a classe controladora deverá estender a classe `org.springframework.web.servlet.view.document.AbstractPdfView`, necessitando implementar o método `buildPdfDocument` para que o documento seja construído corretamente.

4.4 CAMADA CONTROLE

4.4.1 SOFIA

As classes de controle das aplicações que utilizam o SOFIA, devem estender a classe `com.salmonllc.jsp.JspController` e implementar a interface `com.salmonllc.html.events.PageListener` específicas do *framework*. Toda essa con-

figuração é necessária, pois as classes controladoras assumirão o comportamento de *listeners*[6], [9].

No instante em que uma página é requisitada pelo cliente, um evento é gerado pelo *framework*. Esse evento é encaminhado para o controlador responsável onde deve ser tratado e feito o processamento necessário.

O controlador possui métodos específicos para cada tipo de evento que pode ser acionado. Os eventos podem ser gerados na criação da classe controladora, quando uma requisição à página é efetuada e quando algum botão da página é pressionado. Para cada evento, deve ser executado o processamento que a aplicação necessita, forçando o programador a prever todo o comportamento possível que a página poderá sofrer.

Cada página JSP deve possuir a sua classe controladora correspondente. Para cada página do sistema, terá que ser criada uma classe que será responsável pelos eventos por ela gerados, o que acaba aumentando consideravelmente o número de classes a ser gerenciado no projeto. Essa característica acaba deixando as camadas Visão e Controladora demasiadamente acopladas, dificultando a manutenção da aplicação.

Uma vantagem do SOFIA em relação aos outros *frameworks* é que os componentes visuais podem ser acessados diretamente nas classes controladoras. Componentes tais como, campos de formulários, figuras e botões, podem ser gerenciados programaticamente a fim de que eles possam ser habilitados ou desabilitados.

Através desse funcionamento, as páginas podem ser criadas de forma dinâmica e personalizada conforme o usuário que acessa o sistema. Dessa maneira, podem-se criar perfis para cada pessoa que utiliza o sistema, permitindo que sejam renderizadas apenas as funcionalidades que o usuário está habilitado a usar. No entanto, essa característica torna a camada visual altamente acoplada à camada de controle.

O acoplamento entre a camada Visão e a camada Controladora, torna-se uma desvantagem na utilização do *framework* SOFIA. Caso o desenvolvedor necessite modificar algum componente na página JSP, uma alteração na classe controladora respectiva será necessária, fazendo com que uma re-compilação das classes modificas aconteça, aumentando o custo de manutenção geral da aplicação.

[6]Objetos que ficam "escutando"determinados eventos. No momento em que um evento específico acontece, o *listener* responsável é acionado.

4.4.2 STRUTS

A camada Controladora do *framework* Struts baseia-se em padrões bem definidos e estáveis, como *Servlets* e XML. Os controladores da aplicação que utiliza o Struts devem estender a classe `org.apache.struts.action.Action` provida pelo *framework*.

O método `execute` da classe `org.apache.struts.action.Action` é acionado automaticamente pelo *framework*. É nesse método que deve ser adicionado todo o processamento específico da aplicação, tais como o preenchimento de objetos, validação de dados, comunicação com a camada Modelo, etc.

A classe ainda disponibiliza o método `getResources` que auxilia no processamento, permitindo que sejam acessados os recursos da aplicação. Dessa forma, pode-se acessar o arquivo com as mensagens específicas de cada idioma.

Toda a configuração sobre o mapeamento das requisições é especificada em um arquivo de configuração XML da aplicação. Nesse arquivo, devem ser definidos todos os formulários que o sistema possui, bem como o mapeamento das requisições para as classes.

Para cada mapeamento, é necessário que seja definida a classe responsável, bem como o caminho de origem e os caminhos de destino a serem seguidos após a classe executar o processamento. Com isso, o controle do fluxo do sistema, fica totalmente configurável através de um arquivo XML, permitindo que eventuais mudanças não necessitem a re-compilação das classes.

A figura 4.4 ilustra o mapeamento entre as requisições e os objetos responsáveis. Na figura, pode-se visualizar o mapeamento da requisição `login.do` para a classe `br.ufsm.cpd.dic.actions.LoginAction`. Após o processamento da classe, o fluxo pode ser redirecionado para a página *form1.jsp* se o processamento ocorrer normalmente ou para a página *index.jsp* caso ocorra algum erro.

4.4.3 SPRING

A camada Controladora do *framework* Spring é baseada em padrões já bastante estáveis e utilizados por outros *frameworks*, tais como *Servlets* e XML. Para a implementação da aplicação, os objetos da camada Controladora devem estender os objetos providos pelo *framework*.

O Spring provê objetos específicos os mais variados tipos de aplicações. Dentre

```
<action path="/login" type="br.ufsm.cpd.dic.actions.LoginAction"
        name="loginForm" input="/index.jsp" validate="true">
    <forward name="feliz" path="form1.jsp" />
</action>
<action path="/primeiraParte" type="br.ufsm.cpd.dic.actions.PrimParteAction"
        name="form1" input="/form1.jsp" validate="true">
    <forward name="localizaImovel" path="localizar_imovel.jsp" />
    <forward name="felizAviso" path="aviso.jsp" />
    <forward name="felizF" path="form2.jsp" />
    <forward name="felizJ" path="form2_jur.jsp" />
</action>
<action path="/segundaParte" type="br.ufsm.cpd.dic.actions.SegParteAction"
        name="form2" input="/form2.jsp" validate="true">
    <forward name="volta" path="form1.jsp" />
    <forward name="localizaAtividades" path="localizar_ativ_economica.jsp" />
    <forward name="felizFisica" path="form4.jsp" />
    <forward name="feliz" path="form3.jsp" />
</action>
<action path="/terceiraParte" type="br.ufsm.cpd.dic.actions.TercParteAction"
        name="form3" input="/form3.jsp" validate="true">
    <forward name="volta" path="form2.jsp" />
    <forward name="feliz" path="aviso.jsp" />
</action>
<action path="/quartaParte" type="br.ufsm.cpd.dic.actions.QuarParteAction"
        name="form4" input="/form4.jsp" validate="true">
    <forward name="voltaFisica" path="form2.jsp" />
    <forward name="volta" path="form3.jsp" />
    <forward name="feliz" path="concluir.jsp" />
</action>
```

Figura 4.4: Mapeamento das requisições para as classes da aplicação.

eles, pode-se destacar: `org.springframework.web.servlet.mvc.AbstractController` e `org.springframework.web.servlet.mvc.AbstractWizardFormController`.

A classe `org.springframework.web.servlet.mvc.AbstractController` possui métodos comuns a todos os tipos de controladores do *framework*. Caso a camada Controladora da aplicação estenda essa classe, terá que implementar o método `handleRequest` e poderá contar com o método `setSynchronizeOnSession` para que possa ocorrer o controle de concorrências entre as requisições.

Caso a aplicação necessite de formulários do tipo assistentes (*wizards*), é possível que os controladores estendam a classe `org.springframework.web.servlet.mvc.AbstractWizardFormController`. Essa classe permite que sejam configuradas quais páginas farão parte do assistente, bem como possibilita ao usuário a navegação entre todas as partes do formulário.

Cabe ressaltar que todas as configurações necessárias aos controladores podem ser especificadas via arquivos XML da aplicação. Em conjunto com o mecanismo de IoC, o *framework* Spring detecta as propriedades expressas no arquivo de configuração e configura as classes que as recebem. Através desse mecanismo, não é preciso alterar as classes cada vez que uma propriedade deve ser re-configurada, não sendo

preciso uma re-compilação de todo o sistema.

4.5 VALIDAÇÃO DE DADOS

A tarefa de validação dos dados digitados pelos usuários, é de extrema importância para tornar as aplicações confiáveis. Dessa maneira, ela deve ser executada de acordo com padrões bem especificados.

Se o usuário digitar valores inválidos e a aplicação aceitá-los, um comportamento anômalo pode ocorrer. A camada de Controle, em conjunto com a camada Modelo, é responsável por essa tarefa, fazendo-se necessário que ela possua um processamento específico.

Os processamentos mais comuns para a verificação dos dados incluem: validação de endereço de e-mail, validação de CEP, verificação de campos não preenchidos, etc. A fim de facilitar o desenvolvimento, os *frameworks*, normalmente, fornecem processamentos para tais tarefas.

Entre as soluções adotadas para a validação dos dados, encontra-se o Commons Validator. Esse *framework* é um projeto desenvolvido pela fundação Apache e é largamente utilizado nas aplicações em geral, encontrando-se, atualmente, na versão 1.1.3.

Com o uso de tal *framework*, o desenvolvedor não necessita construir toda a lógica de verificação dos dados. Ao se utilizar o Commons Validator, deve-se definir um arquivo contendo todos os formulários a serem verificados e quais as restrições para cada campo do formulário. Com uso de arquivos de configuração, torna-se desnecessária a alteração dos códigos-fonte da aplicação ao se definir as propriedades a serem verificadas.

O *framework* possui métodos estáveis e bem documentados que podem ser utilizados para a maioria das validações necessárias. Portanto, a aplicação torna-se mais confiável quando o utiliza, visto que foi desenvolvido e testado por diversas pessoas.

A figura 4.5 ilustra um exemplo de arquivo de configuração do *framework* Commons Validator. Na figura, pode-se notar a restrição dos campos *issqn*, *cpf* e *nome* como sendo obrigatórios no preenchimento do formulário *form1*. Dessa forma, a aplicação somente receberá o formulário com todos esses campos devidamente preenchidos.

```
<form-validation>
   <global>
      <validator name="required"
            classname="org.apache.struts.validator.FieldChecks"
               method="validateRequired"
         methodParams="java.lang.Object,
                       org.apache.commons.validator.ValidatorAction,
                       org.apache.commons.validator.Field,
                       org.apache.struts.action.ActionErrors,
                       javax.servlet.http.HttpServletRequest"
                  msg="errors.required">
      </validator>
   </global>
    <formset>
        <form name="form1">
            <field property="issqn" depends="required"/>
            <field property="cpf" depends="required"/>
            <field property="nome" depends="required"/>
        </form>
    </formset>
</form-validation>
```

Figura 4.5: Configuração das validações utilizando o *framework* Commons Validator.

4.5.1 SOFIA

A tarefa de validação de dados da aplicação que emprega o SOFIA em seu projeto, pode ocorrer de duas formas: validação na camada Controladora ou validação na camada Modelo. Se o desenvolvedor optar por validar os dados na camada Controladora, terá que replicar o processamento em todas as páginas ou re-colocar esses métodos em uma classe pai e executá-los em todos os lugares necessários.

Com a replicação dos códigos de validação de dados, uma possível modificação na aplicação acarretará a mudança de diversos arquivos. Conforme a complexidade do *software*, a quantidade de páginas que permitem a digitação de dados por parte do usuário pode ser extremamente grande, o que levará o desenvolvedor a verificar todos os arquivos envolvidos.

Pode-se, ainda, configurar as páginas para que a validação seja executada diretamente no navegador do cliente, via javascript. Com esse comportamento, evita-se que os dados inválidos sejam enviados ao servidor e torna a tarefa mais rápida. No entanto, mesmo que os dados sejam validados no cliente, eles devem ser verificados novamente no servidor, após o envio de todas informações, por motivos de segurança.

Caso a validação seja executada na camada Modelo, irá ocorrer menos repli-

cação de códigos. No entanto, ocorre uma ruptura no padrão MVC, pois a camada Modelo não deve ter possuir mecanismos para enviar mensagens à camada visual.

Uma vez que a verificação de dados do SOFIA ocorre de acordo com padrões próprios, o *framework* Commons Validator não é suportado. Dessa forma, uma possível migração da aplicação para outro *framework*, exigirá que o desenvolvedor reconstrua todo o mecanismo de validação dos dados.

4.5.2 STRUTS

As aplicações construídas com o Struts devem declarar os formulários utilizados para preenchimento de dados em um arquivo de configuração. Os formulários devem estender classes que o Struts possui, para que métodos pré-definidos sejam executados no momento de envio dos dados.

Quando o usuário envia os dados para o servidor, esse invoca o método de validação do formulário. É nesse método de validação que o programador deve especificar as restrições para os dados, tais como campos obrigatórios ou validações de números, por exemplo. Se forem detectados erros, o método os retorna à camada visual, caso contrário, o processamento prossegue normalmente.

As mensagens de erros podem ser configuradas de acordo com o idioma do usuário do sistema. As mensagens são configuradas seguindo o padrão dos *resource bundles*, definido-se um arquivo para cada idioma suportado pela aplicação. Através disso, a aplicação torna-se portável quanto aos idiomas.

O Struts provê suporte ao uso do *framework* Commons Validator através de *plug-ins*. É necessário que seja especificado no arquivo de configuração do Struts que a aplicação utilizará tal *framework*, para que ele seja acionado na ocasião do envio de dados. Com isso, não é preciso que o desenvolvedor implemente toda a lógica de validação dos dados, além de promover portabilidade à aplicação, visto que diversos outros *frameworks* suportam o Commons Validator.

4.5.3 SPRING

A validação dos dados dos formulários das aplicações que utilizam o Spring é baseada em interfaces. Deve-se construir classes que implementam a interface `org.springframework.validation.Validator` do próprio *framework*, pois ela possui o método `validate` que é executado no momento do envio dos dados para o servidor, fazendo a validação necessária dos dados.

A utilização de classes de validação separadas das classes de formulários permite um maior reaproveitamento. Dessa forma, pode-se usar uma mesma classe para executar a validação de vários formulários, reduzindo a replicação de trechos de códigos.

Se diversos formulários da aplicação necessitarem de validações comuns, pode-se utilizar as classes construídas para a validação. Logo, haverá uma menor replicação dos códigos de verificação e existirá uma quantia menor de classes a serem gerenciadas. Com tal abordagem, pode-se prevenir possíveis erros de programadores.

A fim de prover um suporte à internacionalização, as mensagens de erros podem ser construídas conforme o idioma do usuário. As mensagens devem ser configuradas como *resource bundles* da aplicação e, no momento em que elas são enviadas para o cliente, o servidor as localiza nos arquivos de mensagens do idioma corrente e constrói a mensagem a ser visualizada.

4.6 MÉTRICAS DO CASO DE USO

As métricas de *software* são medidas efetuadas a partir de processamentos específicos, que visam identificar o status de um programa. Através das medidas adquiridas, pode-se conhecer a complexidade de um sistema e até mesmo a qualidade do mesmo.

As medidas recolhidas devem ser analisadas pelo gerente de desenvolvimento a fim de que ele tome conhecimento sobre o sistema como um todo. As medidas resultantes fornecem um importante mecanismo de tomada de decisão uma vez que elas podem revelar se um programa está demasiadamente complexo ou muito acoplado.

Dentre as métricas, pode-se utilizar o número de classes de um projeto. Através dessa medida, é possível identificar e prever o impacto de uma alteração estrutural que o projeto possa sofrer.

Uma medida importante a ser realizada diz respeito à quantidade de classes de acesso ao banco. Quanto maior a quantidade de classes de acesso ao banco, maior será o impacto em possíveis migrações de BDs, caso os objetos interajam diretamente com ele.

A quantidade de classes controladoras de uma aplicação ilustra as classes que recebem dados da camada visual e interagem com a camada de acesso ao BD. Dessa forma, normalmente, esse número é diretamente proporcional à complexidade da

Métrica	**SOFIA**	**Struts**	**Spring**
Número de classes	21	30	29
Classes de acesso o banco	8	1	1
Classes controladoras	8	9	7
Classes de formulários	0	8	6
Classes de validação	0	0	5
Classes auxiliares	5	12	10

Tabela 4.1: Tabela com as métricas do caso de uso para *framework*.

aplicação.

A tabela 4.6 ilustra as métricas adquiridas para a aplicação construída com cada *framework* estudado. Através dessas medidas, é possível conhecer a complexidade do projeto para cada *framework*.

Através dos dados das tabela, pode-se concluir que a aplicação construída com o *framework* SOFIA possui menor quantidade de classes. Essa medida, no entanto, reflete um alto acoplamento entre as classes da aplicação, o que aumenta o esforço necessário para a manutenção do sistema.

Ainda, na tabela, pode-se notar que a aplicação desenvolvida com o SOFIA, possui 8 classes que manipulam os dados do BD. Essa medida é importante, pois caso seja necessário modificar o BD, muitos objetos da aplicação terão que ser reajustados, dificultando a manutenção da aplicação.

Conforme o *framework* utilizado, é necessário que exista uma classe para cada formulário que a aplicação possua. No entanto, no caso do *framework* SOFIA, essa situação não é necessária, visto que ele une os elementos do formulário HTML aos atributos da classe controladora. Embora essa característica do SOFIA necessite menos classes a serem criadas no projeto, ela acaba criando um forte acoplamento entre a camada visual e a camada Controladora, dificultando a manutenção.

No caso do *framework* Struts, é possível que os formulários sejam especificados somente no arquivo de configuração do *framework*. Dessa forma reduz-se a quantidade de classes geradas e aumenta-se a flexibilidade da aplicação, permitindo que os campos que o formulário não precisem ser especificados em classes da aplicação.

A validação dos dados de entrada, como já foi visto, é uma importante tarefa do sistema, a fim de torná-lo confiável. No caso do SOFIA, as validações são expressas nas próprias páginas JSP, o que não necessita a criação de classes para tal fim.

Já no caso do Struts, as validações são executadas nas mesmas classes dos

formulários ou expressas em conjunto com o *framework* Commons Validator. No caso do Spring, é necessário que se utilizem classes para a execução das validações na situação em que o *framework* Commons Validator não é utilizado.

Para que as funcionalidades da aplicação sejam alcançadas, muitas vezes, torna-se necessário que se criem classes auxiliares. Na aplicação desenvolvida com os três *frameworks*, foi preciso que certas tarefas, como manipulação de datas e acesso a recursos da aplicação, fossem especificadas em objetos auxiliares das camadas, aumentando o número total de classes. Embora a quantidade de classes aumente, acaba-se ganhando a re-usabilidade de componentes.

A tabela 4.6 ilustra o resultado obtido da realização do presente trabalho. Na tabela, pode-se visualizar os tipos de licença que cada *framework* possui, auxiliando a tarefa de escolha sobre qual a melhor adoção nos projetos de *software*. Ainda, é possível verificar o suporte ao *framework* de validação Commons Validator, que facilita a tarefa de validação dos dados de entrada.

Outro dado importante mostrado na tabela é o suporte à ferramentas ORM. Com uso de tais ferramentas, a aplicação torna-se mais flexível em relação à origem dos dados. Freqüentemente, a aplicação deve ser modificada para a suportar novas funcionalidades ou para se corrigirem erro. Um baixo acoplamento entre as camadas da aplicação facilita essa tarefa, sendo mostrado, na tabela, o grau de acoplamento referente aos *frameworks* estudados.

Para que a camada de interface com o usuário possa ser desenvolvida mais facilmente, é preciso que o *framework* ofereça suporte para tal tarefa. Na tabela, pode-se notar que os *frameworks* SOFIA e Struts possuem uma ampla *taglib*, permitindo a construção de componentes bastante complexos com as tags que os integram. No entanto, o Spring possui uma *taglib* reduzida, mas essa desvantagem pode ser reduzida com o uso das JSTL.

O suporte à internacionalização, é um aspecto bastante importante dos *frameworks*, visto que aplicação deve ser portável para diversos idiomas. Dessa forma, na tabela, fica explicitado que o SOFIA não oferece esse suporte, já os *frameworks* Struts e Spring permite tal funcionamento.

	SOFIA	**Struts**	**Spring**
Tipo de Licença	GNU GPL ou Salmon Software License	Apache 2.0	Apache 2.0
Suporte ao Commons Validator	Não	Sim	Sim
Suporte à ferramentas ORM	Não	Sim	Sim
Acoplamento entre camadas	Alto	Baixo	Baixo
Taglib	Grande	Média	Pequena
Internacionalização	Não	Sim	Sim

Tabela 4.2: Tabela final de comparação entre os *frameworks*.

4.7 PADRÕES UTILIZADOS NO CASO DE USO

A arquitetura utilizada em um projeto é de extrema importância, visto que reflete quais padrões de projeto foram utilizados para a sua implementação. Conforme os padrões empregados, pode-se conhecer os impactos que a aplicação sofrerá se possíveis modificações forem necessárias.

4.7.1 SOFIA

4.7.1.1 MVC

A aplicação desenvolvida com o SOFIA foi implementada procurando ser seguido, ao máximo, o padrão MVC. No entanto, devido ao framework estabelecer um forte acoplamento entre as camadas, as responsabilidades puderam ser corretamente isoladas.

Os objetos da camada Modelo utilizados na aplicação foram construídos utilizando-se o assistente do *framework* que se integra com o IntellijIDEA® [18]. Essa abordagem permitiu que as classes fossem construídas de maneira rápida e prática, visualizando-se as tabelas disponíveis no BD de forma *on-line*.

A comunicação das classes da camada Modelo com o BD se dá através de JDBC [26]. Dessa forma, o *framework* fica totalmente responsável pela abertura e controle de conexões, o que pode não ser uma boa alternativa pois não permite que a aplicação possua outros mecanismos para tais tarefas. Outra maneira de construir os objetos, é criá-los a partir EJBs que o desenvolvedor já possua, o que facilita o desenvolvimento [9].

A camada visual foi totalmente construída em JSP com as *tags* que compõem o SOFIA. Com isso, constatou-se que é possível que toda a parte de interação com o usuário pode ser desenvolvida com os componentes do próprio *framework*.

Devido à alta quantidade de *tags* disponíveis no *framework*, o desenvolvedor não precisa utilizar outros recursos para que a camada visual seja implementada. Além disso, devido à capacidade do *framework* se integrar com aplicativos, foi utilizado o programa Macromedia® DreamWeaver® para que as páginas fossem construídas, verificando-se um desenvolvimento de alta qualidade nos códigos correspondentes gerados.

A camada de controle foi previamente construída pelos assistentes que integram o *framework*. A partir da geração dos controladores, as classes foram sendo modificadas para que os seus métodos tivessem o comportamento necessário.

Além disso, foi criada a classe abstrata `br.ufsm.cpd.dic.controllers.-BaseController` com métodos em comum entre os controladores. Dessa forma, os controladores estendem essa classes abstrata e utilizam os métodos que ela possui, como, por exemplo, o método `constroiMenu` que constrói um menu de visualização com a coleção de objetos passada como parâmetro. Com o emprego de classes abstratas, diminui-se a replicação de códigos, facilitando a manutenção das classes.

4.7.1.2 DAO

Os objetos utilizados na aplicação para a comunicação com o BD, não seguem o padrão DAO. Visto que as classes foram construídas a partir dos padrões que o próprio *framework* possui, elas não implementam as regras definidas para o padrão DAO.

Uma vez que cada objeto que representa uma tabela no BD deve possuir seus próprios métodos para persistência e pesquisa, o padrão DAO não é seguido. Com essa restrição, a aplicação acaba permitindo trechos replicados de códigos para a consistência dos dados. A não utilização de tal padrão, traz as implicações negativas discutidas na seção 2.2.

4.7.2 STRUTS

4.7.2.1 MVC

Na aplicação desenvolvida com o *framework* Struts, o padrão MVC foi totalmente seguido. Dessa maneira, é notável a separação entre as camadas.

A camada Modelo da aplicação foi desenvolvida com a utilização do *framework* Hibernate. As classes foram construídas mapeando-se as suas propriedades para as colunas das tabelas do BD. A utilização do *framework* Hibernate pela aplicação, provê as características citadas na seção 4.2.2.

A interface com o usuário foi totalmente desenvolvida com páginas JSP. Para que os requisitos fosse alcançados, foram utilizadas as *tags* que compõem a *taglib* do *framework*, não sendo necessária a inclusão de código-fonte Java nas páginas JSP.

Ficou evidenciado que as aplicações podem ser totalmente construídas utilizando-se somente a sua *taglib*. Tal característica torna-se importante, pois deixa o sistema sem dependências de outros mecanismos, facilitando o controle dos recursos utilizados.

A camada Controladora foi desenvolvida com classes que estendem a classe Action do Struts. Tal exigência do *framework* acaba tornando-se restritiva pois não permite que outros mecanismos sejam utilizados. A fim de agrupar tarefas comuns entre os controladores, foi criada uma classe abstrata, permitindo que os métodos sejam utilizados por todas as classes que a estendem.

4.7.2.2 DAO

A camada Modelo da aplicação desenvolvida, segue o padrão DAO. Com isso, apenas um objeto tem a responsabilidade de se comunicar com o banco de dados e, através dele, a camada controle acessa a base dos dados.

A classe de acesso aos dados foi implementada em conjunto com o framework Hibernate. Com isso, o impacto proveniente de mudanças no BD não é refletido nos objetos, aumentando a flexibilidade do sistema.

As pesquisas sobre os dados são automaticamente geradas pelo Hibernate em tempo de execução. Dessa forma, o código-fonte do objeto não precisa conter cláusulas SQL para acesso aos dados trazendo praticidade de confiabilidade às pesquisas geradas. No caso das pesquisas mais rebuscadas, deve-se especificá-las no arquivo de configuração do objeto.

4.7.3 SPRING

4.7.3.1 MVC

A aplicação desenvolvida com o *framework* Spring segue, de maneira rigorosa, o padrão MVC. Dessa forma, toda a estrutura foi desenvolvida para que as camadas

permanecessem isoladas ao máximo, garantindo uma melhor manutenção ao sistema.

A camada Modelo foi desenvolvida em conjunto com o *framework* de mapeamento objeto-relacional Hibernate. Fez-se essa opção devido à facilidade de configuração dos objetos em relação às tabelas do BD, aproveitando os mapeamentos já produzidos para a aplicação desenvolvida com o Struts.

Com o intuito de aproveitar, ao máximo, as vantagens providas pelo Spring, foi utilizado o mecanismo de gerenciamento de transações. Dessa forma, no arquivo de configuração do *framework*, foram explicitados quais os métodos deveriam ser gerenciados, permitindo que um objeto do *framework*, seguindo o padrão Proxy, pudesse interceptar as requisições para a classe gerenciada. O padrão Proxy permite que o acesso a um objeto seja controlado, permitindo que a criação desse objeto seja manipulada ou que algum processamento aconteça antes que o acesso, propriamente dito, aconteça [1].

A camada Visão da aplicação foi construída com páginas JSP. Para que as funcionalidades necessárias fossem conseguidas, foi necessária a utilização da *taglib* provida pela Sun, a JSTL, visto que a *taglib* do *framework* não dispõe de muitas opções para a construção visual. No entanto, a *tag* Errors, fornecida pelo *framework*, mostrou-se bastante útil para a renderização dos erros da camada Controladora.

Fica claro, com isso, que a aplicação desenvolvida com o Spring torna-se dependente dos componentes da Sun. Essa característica deve ser levada em consideração para implementação de um sistema, pois cabe ao desenvolvedor do projeto gerenciar todas as dependências do sistema.

A camada Controladora da aplicação foi desenvolvida estendendo-se as classes que o *framework* provê. As classes controladoras estendem uma classe do *framework* que fornece métodos para o controle de submissão de formulários, bem como a requisição de dados a serem mostrados nas páginas.

O fluxo da aplicação foi totalmente descrito no arquivo de configuração da aplicação, da mesma forma que o Struts. Para cada mapeamento das ações, é descrito o formulário a ser utilizado, a classe controladora e visões a serem utilizadas no caso de erros ou sucessos no processamento. Com essa abordagem, a aplicação torna-se flexível, pois não especifica o fluxo diretamente no código-fonte das classes.

Para a validação dos dados digitados pelo usuário, foram utilizadas classes, `br.ufsm.cpd.dic.validation.FormValidator` entre outras, que implementam a interface Validator do Spring. Objetivou-se, com isso, uma maior re-utilização do

mecanismo de validação, pois a mesma classe de validação pode servir à vários formulários.

4.7.3.2 DAO

A camada Modelo da aplicação utilizou o padrão DAO a fim de obter todas as vantagens fornecidas por tal padrão. Para executar as tarefas de acesso ao banco, a camada de controle deve apenas interagir com uma classe, auxiliando na tarefa de manutenção.

A classe DAO (`br.ufsm.cpd.dic.beans.BdBean`) da aplicação foi implementada em conjunto com o Hibernate, estendendo-se a classe `org.springframework.orm.hibernate.support.HibernateDaoSupport`, e o gerenciamento de transações do Spring. O gerenciamento de transações fornece mecanismos eficientes para que os dados utilizados na aplicação sejam confiáveis e seguros.

Foi necessária a criação da interface `br.ufsm.cpd.dic.beans.Dic`, implementada pela classe `br.ufsm.cpd.dic.beans.BdBean` para que o funcionamento do gerenciador de transações ocorresse. Dessa forma, caso seja necessário suportar outra fonte de dados, basta que a nova classe implemente essa interface e o gerenciador de transações continua a funcionar corretamente.

Deve-se mencionar quais os métodos da classe que devem ser gerenciados e quais os atributos que a sua transação deverá possuir. É possível definir ao gerenciador de transações que alguns métodos sejam somente-leitura (*read only*) e outros tenham que refletir totalmente as modificações dos objetos no BD. Com tais configurações, o *framework* pode aplicar alguns ajustes de performance no acesso ao métodos e se certificar que as alterações realmente sejam transferidas ao BD.

O mecanismo de gerenciamento de transações do Spring funciona em conjunto com o suporte à programação orientada a aspectos do *framework*. Dessa forma, o gerenciador se comporta como um objeto interceptador de requisições do objeto gerenciado, da mesma maneira que um Proxy.

Cada vez que a camada Controladora executa um método transacional do objeto DAO, o gerenciador intercepta a requisição e sinaliza o início da transação. Logo após, o método é executado normalmente e, se nenhum erro ocorrer, o método retorna e o gerenciador sinaliza a transação como terminada. Caso alguma exceção aconteça, o gerenciador a identifica e desfaz todas as alterações que já haviam sido refletidas no BD.

Capítulo 5

CONCLUSÃO

A utilização de *frameworks* em projetos de *software* é uma alternativa bastante comum atualmente. Dessa forma, o desenvolvedor concentra-se em implementar as funcionalidades da aplicação procurando reutilizar os componentes que os *frameworks* oferecem.

No domínio das aplicações *web*, são inúmeros os *frameworks* passíveis de serem utilizados. Conforme o perfil da aplicação a ser desenvolvida, deve ser realizada a escolha concentrando-se nas vantagens e nos impactos causados pela sua adoção.

Este trabalho visa estudar os *frameworks* SOFIA, Struts e Spring, a fim de se conhecer qual a melhor opção para os sistemas desenvolvidos no Centro de Processamento de Dados da Universidade Federal de Santa Maria (CPD/UFSM). Essa seleção deveu-se pelas vantagens que cada um pode proporcionar aos projetos que os empregam, além de serem *frameworks* bastante utilizados por desenvolvedores do mundo inteiro.

A fim de comparar os aspectos de cada *framework*, foi desenvolvida uma aplicação, a versão eletrônica do Documento de Informações Cadastrais (DIC). Essa aplicação possui as características dos sistemas desenvolvidos no CPD, tornando-se possível a realização do estudo comparativo.

Após a realização do estudo comparativo, obteve-se resultados que podem auxiliar na tomada de decisões quanto ao *framework* a ser adotado nos projetos de *software* do CPD. Com isso, espera-se que os gerentes de projetos possam utilizá-los para as decisões dos futuros projetos.

O alto acoplamento da camada visual à camada Controladora, observado no caso do SOFIA, torna difícil a manutenção do sistema. Sempre que a página JSP incorporar um novo elemento, a camada Controladora deve ser modificada, necessi-

tando alterações nas classes o que aumenta o impacto sobre a aplicação, tornando-se um ponto negativo do *framework*.

A alta quantidade de documentação que possui o *framework* Struts e o seguimento estrito dos padrões MVC e DAO são aspectos positivos na avaliação deste *framework*. Sabendo-se que diversos tutoriais e livros podem ser encontrados a seu respeito, muitas dificuldades encontradas no desenvolvimento podem ser esclarecidas com a utilização de tal material. A separação entre as camadas permite que a manutenção para cada uma delas ocorra de forma independente, configurando uma vantagem do Struts.

A camada Modelo do Struts segue o padrão DAO, permitindo que se obtenha todas as vantagens que tal padrão oferece. Além disso, é possível que sejam utilizadas ferramentas ORM para auxiliar o desenvolvimento, refletindo a capacidade de integração do *framework*.

O *framework* Spring permite que a aplicação siga os padrões MVC e DAO. As camadas da aplicação permanecem separadas, separando as responsabilidade de cada uma, facilitando a manutenção do sistema. Da mesma forma, com a utilização do Spring, o padrão DAO pode-se ser seguido, além de o *framework* oferecer o suporte à transações para esse objeto.

A inclusão de novos conceitos (IoC e AOP), torna o Spring o *framework* mais robusto, dentre os estudados. No entanto, deve-se levar em consideração que treinamentos devem ser ministrados para que a equipe assimile os tais conceitos e os ponha em prática nos programas desenvolvidos. Embora ele seja o *framework* com mais recursos, o que o torna melhor, a curto prazo, a sua implantação não se torna uma alternativa viável.

Conclui-se com isso, que o Struts é a solução mais adequada, atualmente, a ser utilizada nos sistemas desenvolvidos pelo CPD. Devido à alta experiência da equipe e devido à diversos mecanismos já terem sido construídos baseados no Struts, deve-se seguir utilizando-o nos projetos, porém, a longo prazo, precisa-se pensar sobre a adoção do Spring como nova ferramenta.

Uma possível falha no presente estudo pode ser derivada da comparação dos *frameworks* em somente uma aplicação. Com isso, algumas métricas obtidas podem apresentar variação não proporcional à complexidade do sistema.

Como trabalho futuro, pode-se desenvolver um sistema que reconfigure as aplicações atuais, desenvolvidas com Struts, para o *framework* Spring. Caso o *framework*

Spring venha a ser adotado como nova ferramenta, é possível que algumas aplicações atuais tenham que ser reformuladas para adotar os mecanismos do Spring, necessitando um sistema conversor.

Referências Bibliográficas

[1] GAMMA, E.; et. al. **Design Patterns. Elements of Reusable Object-Oriented Software.** Professional Computing Series. Massachusetts, USA : Addison-Wesley, 1995.

[2] ECKEL, B. **Thinking in Patterns. Problem-Solving Techniques using Java. Revision 0.9.** 0.9 ed. President, USA : MindView, 2003.

[3] JOHNSON, R. E. Documenting frameworks using Patterns. In: CONFERENCE ON OBJECT-ORIENTED PROGRAMMING SYSTEMS AND APPLICATIONS - OOPSLA, 1992, Vancouver, USA. **Proceedings**. ACM Press, 1992. v.27, p.63-76.

[4] SINGH, I. et al. **Designing Enterprise Applications with the J2EE Platform. Second Edition.** 2nd ed. New Jersey, USA : Addison-Wesley, 2002.

[5] ASSIS, S. R.; SUZANO, R. Framework: Conceitos e Aplicações. **CienteFico**, Salvador, v.2, Jun./Dez. 2003.

[6] FIORINI, S. T. **Arquitetura para Reutilização de Processos de Software.** 2001. Tese (Doutorado em Informática) - Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro.

[7] FAYAD, W.E.; SCHMIDT, D,C.; JOHNSON, R.E. **Building Application Frameworks:** object-oriented foundations of framework design. New York, USA : John Wiley & Sons, 1999.

[8] SAUVÊ, J. P. **Frameworks.** Notas de Aula. Paraíba : Universidade Federal da Paraíba, 2004. Disponível em: <http://www.dsc.ufcg.edu.br/ jacques/cursos/map/html/map1.htm>. Acesso em: set. 2004.

[9] SALMON LLC. **The Salmon Open Framework for Internet Applications User Guide.** Revision 8. New York, USA : 2003.

[10] SALMON LLC. **SOFIA Philosophy.** Disponível em: <http://www.salmonllc.com/website/Jsp/vanity/SofiaPhilosophy.jsp>. Acesso em: out. 2004.

[11] SALMON LLC. **Case Study: The Salmon Open Framework for Internet Applications (SOFIA).** Disponível em: <http://www.salmonllc.com/website/Jsp/vanity/bin/CaseStudy.pdf>. Acesso em: out. 2004.

[12] LEME, F. A.; SANTOS, M. N. **Introdução às Bibliotecas JSTL.** Sessão Técnica. São Paulo : JustJava, 2003.

[13] CAVANESS, C. **Programming Jakarta Struts.** 2nd ed. California, USA : O'Reilly Media, 2004.

[14] APACHE STRUTS. **The Apache Struts Web Application Framework.** Disponível em: <http://struts.apache.org/index.html>. Acesso em: set. 2004.

[15] JOHNSON, R.; et. al. **Spring. Java/J2EE™Application Framework. Reference Documentation.** Version 1.0.2. Disponível em: <http://aleron.dl.sourceforge.net/sourceforge/springframework/spring-framework-1.0.2.zip>. Acesso em: out. 2004.

[16] SUNDSTED, T. **Server-Side Development for Business Advantage: Enterprise JavaBeans Technology**. Disponível em: <http://java.sun.com/features/1999/12/ejb.html>. Acesso em: nov. 2004.

[17] DeSOTO, A. **Using the Beans Development Kit 1.0**. California, USA : JavaSoft, 1997. Disponível em: <http://java.sun.com/products/javabeans/docs/Tutorial-Sep97.pdf>. Acesso em: set. 2004.

[18] IntellijIDEA. In: IntellijIDEA. The Most Intelligent Java IDE. Disponível em: <http://www.jetbrains.com/idea>. Acesso em: out. 2004.

[19] Hibernate. In: Hibernate. Disponível em: <http://www.hibernate.org>. Acesso em: nov. 2004.

[20] MacroMedia DreamWeaver. In: MacroMedia - DreamWeaver. Disponível em: <http://www.macromedia.com/software/dreamweaver/>. Acesso em: set. 2004

[21] Eclipse. In: Eclipse.org Main Page. Disponível em: <http://www.eclipse.org>. Acesso em: nov. 2004

[22] iBATIS. In: iBATIS.com. Disponível em: <http://www.ibatis.com>. Acesso em: nov. 2004.

[23] JDO. In: Java Data Objects (JDO). Disponível em: <http://java.sun.com/products/jdo/index.jsp>. Acesso em: dez. 2004.

[24] Apache Jakarta Tomcat. In: The Jakarta Site - Apache Jakarta Tomcat. Disponível em: <http://jakarta.apache.org/tomcat/index.html>. Acesso em: set. 2004.

[25] Sun Microsystems. In: Sun Microsystems. Disponível em: <http://www.sun.com>. Acesso em: out. 2004.

[26] Sun Microsystems. In: JDBC Technology. Disponível em: <http://java.sun.com/products/jdbc/>. Acesso em: nov. 2004.

Anexo A

GLOSSÁRIO

MVC: Model View Controller. Padrão de arquitetura utilizado em aplicações interativas. Procura separar as camadas do software conforme as suas responsabilidades.

DIC: Documento de Informações Cadastrais. Documento de cadastro econômico sobre os contribuintes do município de Campinas (SP).

ORM: *Object Relational Mapping*. Mapeamento Objeto-Relacional. Metodologia utilizada para se trabalhar com o conceito de objetos ao invés de tabelas de BD.

DAO: *Data Access Object*. Objeto de Acesso aos Dados. Objeto encarregado de interagir com a fonte dos dados.

GUI: *Graphical User Interface*. Interface Gráfica do Usuário. Parte da aplicação interativa com o usuário. É nela que são apresentados os resultados dos processamentos.

HTML: *HyperText Markup Language*. Linguagem de marcação de texto. Preocupa-se em organizar a disposição do texto.

JSP: *Java Server Page*. Tecnologia que utiliza a linguagem Java para construção de páginas *web* com conteúdos dinâmicos.

ASP: *Active Server Page*. Linguagem programação orientada à construção de sistemas *web*.

PHP: *Hypertext Preprocessor.* Linguagem de script de propósito geral especialmente utilizada para desenvolvimento *web*.

XML: *eXtensible Markup Language.* Linguagem Extensível de Marcação. Linguagem de marcação de texto utilizada para interagir com estruturas de dados.

J2EE: *Java 2 Enterprise Edition.* Plataforma Java utilizada para desenvolvimento e gerenciamento de aplicações multi-camadas e distribuídas.

AOP: *Aspects Oriented Programming.* Programação Orientada a Aspectos. Metodologia de programação que define aspectos a serem monitorados por outros objetos. Esses aspectos podem ser classes, métodos ou variáveis.

RAD: *Rapid Application Development.* Desenvolvimento rápido de aplicações.

GPL: *General Public License.* Licença Pública Geral. Licença para distribuição de *software* que permite cópia, modificação e redistribuição.

IDE: *Integrated Development Environment.* Ambiente de Desenvolvimento Integrado. Ambiente de desenvolvimento que provê, entre outros recursos, compiladores e analisadores de código.

ASF: *Apache Software Foundation.* Fundação Apache Software. Organização que provê suporte à projetos de *softwares* de código aberto (*open source*).

EJB: *Enterprise JavaBean.* Arquitetura de componentes para sistemas cliente-servidor multi-camadas desenvolvido pela Sun Microsystems®.

JDBC: *Java Data Base Connectivity.* Tecnologia que permite que as aplicações Java sejam desenvolvidas independentemente do banco de dados que utiliza.

IoC: *Inversion of Control.* Inversão de Controle ou Injeção de Dependência. Conceito onde o *framework* cria os objetos e os preenche com outros objetos especificados. Dessa maneira, o programador não precisa criar métodos especiais para essas tarefas.

SIM: Sistema de Informações Municipais. Sistema desenvolvido pelo CPD/UFSM que informatiza as principais tarefas de uma prefeitura.

SQL: *Structured Query Language.* Linguagem Estruturada de Pesquisa. Linguagem de programação padrão para acesso aos dados de um banco de dados.

BD: Banco de Dados. Repositório de dados.

WYSIWYG: *"What You See Is What You Get"*. Interface com o usuário na qual os componentes disponibilizados podem ser arrastados para a sua utilização.

XSLT: *eXtensible Stylesheet Language Transformation.* Linguagem utilizada para transformar documentos XML em outros documentos XML.

JSTL: *Java Standard Tag Library.* Biblioteca Padrão de Tags Java. Biblioteca de tags desenvolvida pela Sun Microsystems® para se tornar padrão entre as aplicações.

PDF: *Portable Document Format.* Formato de Documento Portável. Formato de arquivo criado pela Adobe® para prover uma forma de armazenar e publicar documentos.