UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

COORDENAÇÃO DO CURSO SUPERIOR DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

CURSO SUPERIOR DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

VICTOR SCHNEPPER LACERDA

REFATORAÇÃO DO APLICATIVO GERENCIADOR DE MENUS DINÂMICOS DO SÍTIO ARCABOMK

TRABALHO DE DIPLOMAÇÃO

PONTA GROSSA

2012

VICTOR SCHNEPPER LACERDA

REFATORAÇÃO DO APLICATIVO GERENCIADOR DE MENUS DINÂMICOS DO SÍTIO ARCABOMK

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas, COADS, da Universidade Tecnológica Federal do Paraná.

Orientadora: Prof.ª Dr.ª Simone Nasser Matos

PONTA GROSSA

2012

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Ponta Grossa

Diretoria de Graduação e Educação Profissional

TERMO DE APROVAÇÃO

REFATORAÇÃO DO APLICATIVO GERENCIADOR DE MENUS DINÂMICOS DO SÍTIO ARCABOMK
por

VICTOR SCHNEPPER LACERDA

Este Trabalho de Conclusão de Curso (TCC) foi apresentado em 14 de maio de 2012 como requisito parcial para a obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas. O candidato foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

_____
Profª Drª Simone Nasser Matos
Orientadora

_____
Profª. Msc. Simone de Almeida
Membro titular

_____
Profª. Msc. Helyane B. Borges
Responsável pelos Trabalhos de Conclusão de Curso

_____
Profª. Msc. Helyane B. Borges
Membro titular

_____
Profª. Msc. Simone de Almeida
Coordenadora do Curso
UTFPR - Câmpus Ponta Grossa

- O Termo de Aprovação assinado encontra-se na Coordenação do Curso -

RESUMO

LACERDA, Victor Schnepper. **Refatoração do Aplicativo Gerenciador de Menus Dinâmicos do Sítio ArcaboMK**. 2012. 101f. Trabalho de Conclusão de Curso – Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas, Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2012.

O aplicativo Manipulador *ArcaboMK* é uma ferramenta desenvolvida pelo Grupo de Pesquisa em Engenharia de Software e é responsável pela organização e geração dos menus dinâmicos do sítio *Arcabomk*. Este sítio contém o referencial teórico usado no desenvolvimento de um framework de formação de preço de venda. Este trabalho estudou algumas metodologias e técnicas de refatoração e as adaptou para criação das atividades usadas no processo de refatoração do Manipulador *Arcabomk*, gerando uma estrutura que facilita a sua manutenibilidade. A refatoração foi realizada para o framework *Struts* e sua validação foi obtida por meio do uso de métricas de qualidade e desempenho confrontando a versão antiga do manipulador com a nova.

Palavras-chave: Refatoração de Software. Ferramenta de gerenciamento de menus dinâmicos. Framework *Struts*.

ABSTRACT

LACERDA, Victor Schnepper. **Refactoring the Application Manager Menus Dynamic Site Arcabomk**. 2012. 101f. Trabalho de Conclusão de Curso – Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas, Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2012.

The application Manipulador *ArcaboMK* is a tool developed by the Research Group on Software Engineering and is responsible for organizing and generating dynamic menus for the Arcabomk site. This site contains the theoretical framework used to develop a framework for sale pricing. This paper deals with some methods and techniques of refactoring and adaptation for creating the activities used in the process of refactoring Manipulador *Arcabomk*, creating a structure that facilitates its maintainability. The refactoring was performed for the Struts framework and its validation was obtained through the use of quality metrics and performance facing the old version with the new.

Keywords: Refactoring software. Management tool for dynamic menus. Struts Framework.

LISTA DE ILUSTRAÇÕES

LISTA DE QUADROS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

O Grupo de Pesquisa em Engenharia de Software (GPES, 2011), Universidade Tecnológica Federal do Paraná (UTFPR) Câmpus Ponta Grossa, está desenvolvendo um framework para formação de preço de venda denominado de *FrameMK*.

O material de pesquisa usado na criação deste framework está disponível em um sítio web chamado *ArcaboMK* (ARCABOMK, 2011). Diversos trabalhos foram implantados neste sítio garantindo sua acessibilidade em páginas estáticas.

Em 2011 foi implantado o projeto de desenvolvimento de uma ferramenta de gerenciamento de menus para que a página atendesse o critério de acessibilidade *3.2.4 Navegação Consistente* do documento WCAG 2.0 (WCAG 2.0, 2008), e com isto tornar a incorporação de conteúdos no menu organizada e dinâmica. O nome da ferramenta criada foi o *Manipulador ArcaboMK* (LACERDA, 2011).

Após sua implantação e comparação com outras ferramentas de gerenciamento de conteúdo web, ficou constatado que esta oferecia uma das principais funcionalidades de uma ferramenta de gerenciamento de conteúdo, a geração e controle de menus.

Este gerenciador de conteúdo foi construído sem a aplicação de um padrão e constatou-se a dificuldade de inserção de novas funcionalidades. A solução apresentada para tal problema foi o uso da refatoração de software.

Neste trabalho se estudou algumas metodologias e técnicas de refatoração de software usadas na criação de uma metodologia adaptada para a refatoração do Manipulador *ArcaboMK*. Aplicou-se o framework de *Struts* (HUSTED 2004) para obter um projeto em conformidade com o padrão MVC (HOLMES, 2006).

Para verificar se o processo de refatoração foi realizado com sucesso, analisaram-se as características de qualidade do manipulador antigo e a nova versão por meio do *plugin Metrics* (2012). As características analisadas foram: número de classe, número de filhos, número de métodos sobrepostos, entre outros.

1.1 OBJETIVOS

O objetivo geral e os específicos estão descritos nas subseções 1.1.1 e 1.1.2, respectivamente.

1.1.1 Objetivo Geral

Refatorar o aplicativo *Manipulador ArcaboMK* utilizando uma metodologia de refatoração e o framework de aplicação *Struts* de modo a criar uma versão que facilite seu processo de manutenção.

1.1.2 Objetivo Específico

Os objetivos específicos são:

- Analisar o aplicativo *Manipulador ArcaboMK* em sua primeira versão.
- Adaptar as metodologias de refatoração na criação de uma metodologia que facilite o processo de refatoração do manipulador.
- Compreender o funcionamento do framework de aplicação *Struts.*
- Avaliar a versão antiga e nova do *Manipulador ArcaboMK* em relação a qualidade, desempenho e manutenibilidade.

1.2 PROBLEMA

Atualmente o aplicativo *Manipulador ArcaboMK* possui uma estrutura deficiente e não padronizada para a adição de novas funcionalidades o que dificulta seu processo de manutenção. Por este motivo, foram estudadas metodologias para aplicação da refatoração, um framework que separa o projeto dentro da arquitetura *Model View Controller* (MVC), uma avaliação que compara a versão antiga do manipulador com a nova de modo a obter o resultado referente a questão de qualidade, desempenho e manutenibilidade.

1.3 ORGANIZAÇÃO DO TRABALHO

Este trabalho está dividido em sete capítulos. O capítulo 2 apresenta algumas definições e conceitos sobre a refatoração. O capítulo 3 relata alguns métodos de aplicação da refatoração.

O capítulo 4 descreve algumas características, aplicação, uso e detalhes do desenvolvimento do aplicativo *Manipulador ArcaboMK* em sua primeira versão.

O capítulo 5 apresenta a metodologia proposta para realização do processo de refatoração, possuindo como base as metodologias apresentadas no capítulo 3.

O capítulo 6 narra a aplicação da metodologia proposta, bem como os resultados obtidos com a refatoração do aplicativo. Por fim, o último capítulo apresenta a conclusão e os possíveis trabalhos futuros que podem ser realizados a partir desta pesquisa.

2 REFATORAÇÃO DE SOFTWARE

Este capítulo apresenta alguns conceitos relacionados a refatoração de software. A Seção 2.1 relata as definições sobre o processo de refatoração. A Seção 2.2 descreve as vantagens e dificuldades da refatoração. A seção 2.3 apresenta alguns exemplos de técnicas de refatoração.

2.1 DEFINIÇÕES

Refatoração de software é a transformação do software sem modificar seu comportamento (KERIEVSKY, 2008, p. 35) ou é “uma alteração feita na estrutura interna do software para torná-lo mais fácil de ser entendido e menos custoso de ser modificado sem alterar seu comportamento observável” (FOWLER, 2004, p. 52).

Essa transformação no software tem por objetivo a melhoria do código fonte, possuindo como principais focos: remoção de código duplicado, simplificação de lógica condicional e a clarificação de código que não está claro (KERIEVSKY, 2008, p. 35-37).

Refatorações podem seguir dois caminhos: padrão ou contrárias (KERIEVSKY, 2008, p.57-58). A refatoração rumo a um padrão é a aplicação de um padrão de projeto a uma aplicação, enquanto a refatoração contrária a um padrão é a otimização do código utilizando lógica pura.

Independente do método escolhido, toda refatoração visa melhorar o código de um projeto, assim é importante ressaltar a etapa de testes que se segue após uma refatoração, como maneira de verificar se o código foi melhorado e continua com suas funcionalidades iniciais.

Com base nessas afirmações, a refatoração do projeto *Manipulador ArcaboMK* a princípio seguirá a linha de refatoração para aplicação de um padrão.

2.2 VANTAGENS E DIFICULDADES

Como apresentado na seção anterior, existem dois tipos de refatoração, a saber, a refatoração rumo a um padrão e a refatoração contra um padrão. Os dois tipos podem apresentar desvantagens, como por exemplo, na refatoração rumo a

um padrão, a aplicação do padrão no código pode tornar o desempenho de processamento de código mais lento que uma solução simples de código não padronizado.

Em oposição a isso, na refatoração contra um padrão, códigos não padronizados podem ser confusos e complicados para entendimento.

Segundo Kerievski (2008, p.36-37) as motivações mais comuns para a refatoração são as seguintes:

- Tornar mais fácil a adição de código novo;
- Melhorar o projeto de código existente;
- Obter um melhor entendimento do código;
- Tornar a programação menos irritante.

Essas motivações são detalhadas nas próximas seções.

2.2.1 Tornar mais fácil a adição de código novo

A princípio todo projeto possui uma estrutura que acomoda as funcionalidades existentes no sistema. Para adicionar uma nova funcionalidade ao sistema dois caminhos podem ser escolhidos: a aplicação da nova funcionalidade sem preocupação com a estrutura de projeto atual e a aplicação da funcionalidade levando em conta a estrutura de projeto.

A primeira é útil quando a necessidade de aplicação da funcionalidade é urgente e precisa ser implementada rapidamente, enquanto a segunda é beneficiar no sentido de preparar a estrutura de projeto para comportar a nova funcionalidade sem comprometer o código nem o desempenho do sistema.

Muitas vezes é necessário o desenvolvimento acelerado o que geralmente compromete os padrões, porém sempre é possível refatorar o código após o prazo de entrega no qual a nova funcionalidade precisa ser entregue.

2.2.2 Melhorar o código existente

A partir da melhoria contínua do código, a aplicação de uma nova funcionalidade pode ser facilitada de maneira mais eficaz que um projeto com código confuso.

Através da melhoria, o desempenho e a clareza do código podem diminuir o tempo de aplicação de novas funcionalidades, além de proporcionar maior conhecimento para o programador em relação ao projeto.

2.2.3 Obter um melhor entendimento do código

O código fonte pode apresentar muitas características em relação ao seu desenvolvedor, o que torna seu entendimento fácil. Entretanto, o mesmo nem sempre ocorre com um desenvolvedor que não tenha participado na elaboração do projeto.

A refatoração e melhoria do código é um problema que pode ser solucionado através do estudo do código necessário para efetuar a refatoração.

Analisando o presente estudo sobre refatoração, padrões de projeto podem tornar o desempenho e o desenvolvimento mais lentos, todavia a aplicação de um padrão facilita o entendimento do código. Afinal, a função do padrão é escrever a informação de um modo que todos possam compreendê-lo.

2.2.4 Tornar a programação menos irritante

Códigos complexos e não padronizados precisam ser analisados minuciosamente para a aplicação de novas funcionalidades. Neste ponto, a refatoração entra com a prática de modularizar e padronizar os códigos de forma a diminuir a carga de análise e entendimento do código de uma maneira que torne a programação mais fácil e prazerosa.

2.3 TÉCNICAS DE REFATORAÇÃO

Tomando por ponto de partida as melhorias citadas na seção anterior, refatora-se para melhorar o código e diminuir os erros.

Atualmente, a técnica mais utilizada para detectar partes do programa que necessitam de refatoração é a identificação dos *bad smells*. Segundo Fowler (2004, p. 70), os *bad smells* são estruturas dentro do código que sugerem e possibilitam o

uso de refatoração. Exemplos práticos são: o código duplicado, métodos longos, classes grandes, lista de parâmetros longa, entre outros.

Para os diversos tipos de falha existentes, existem determinadas técnicas que podem ser aplicadas. Por exemplo, o problema de código duplicado mais simples que se pode ter é quando uma mesma expressão está em dois métodos pertencentes a mesma classe. Neste caso, uma técnica de refatoração que pode ser utilizada é *Extrair Método*.

Um outro exemplo é quando se tem a mesma expressão em duas subclasses irmãs, pode-se utilizar a técnica de *Extrair Método* para separar a expressão e *Subir Método na Hierarquia*, para movê-lo para a classe mãe.

Existem diversas técnicas que podem ser utilizadas para refatorar. Nas próximas subseções são descritas algumas destas técnicas, porém outras podem ser encontradas em (FOWLER, 2004).

2.3.1 Extrair Método

Quando se identifica que um trecho de código é duplicado, o primeiro passo é transformá-lo em um método cujo nome explique seu propósito. Se o código que quiser extrair for muito simples, como uma única mensagem ou chamada de função, deve-se extraí-lo se o nome do novo método revelar a intenção do código de uma maneira melhor. Caso um nome mais significativo não seja encontrado, o método não deve ser extraído.

Após extrair o método e passar os parâmetros corretos, se necessários, substituí-se a duplicação de código pela chamada do novo método.

2.3.2 Internalizar Método

Ocorre quando o corpo de um método é tão claro quanto o seu nome. Neste caso, remove-se seu conteúdo para o corpo que faz a chamada do método. Porém, deve-se verificar se o método não é polimórfico, pois se remover um método que seja sobrecarregado por ouras subclasses, um erro é gerado.

Após tomar este cuidado, deve-se encontrar todas as chamadas do método e substituí-la pelo corpo do método.

2.3.3 Substituir Variável Temporária por Consulta

Utiliza-se esta técnica quando uma variável temporária armazena o resultado de uma expressão.

Quando uma variável local a um método armazena o resultado de uma expressão para efetuar uma verificação dentro do método, como por exemplo, *soma=a+b* e precisa-se desse valor fora do método, substitui-se a variável por um método que retorne o valor da expressão. Para recuperar o valor em qualquer parte da classe é necessário apenas fazer uma chamada ao método.

2.3.4 Substituir Algoritmo

Quando um trecho de código de regras de negócio está utilizando um algoritmo confuso se pode tentar elaborar um algoritmo alternativo que seja mais simples e compacto. Se os testes e retornos do algoritmo alternativo não mudarem o comportamento do antigo algoritmo, substitui-se o corpo pelo novo algoritmo.

O próximo capítulo descreve algumas metodologias usadas no processo de refatoração.

3 MÉTODOS DE REFATORAÇÃO

Este capítulo apresenta algumas abordagens para a aplicação e execução de refatoração de software. A Seção 3.1 relata as definições e conceitos do Método utilizado por Mens e Tourwé (2004). A Seção 3.2 descreve o método elaborado por Rapeli (2006). A Seção 3.3 apresenta uma breve comparação entre as metodologias apresentadas.

3.1 METODOLOGIA DE MENS

Segundo Mens e Tourwé (2004), o processo de refatoração possui as seguintes etapas:

- Identificar onde o software será refatorado.
- Determinar quais refatorações devem ser aplicadas nos lugares identificados.
- Garantia da preservação de comportamento da aplicação.
- Aplicar a refatoração.
- Avaliar o efeito da refatoração em características de qualidade do software.
- Manter a consistência do código refatorado e outros artefatos do software (documentação, especificação de requerimentos, entre outras).

Estas etapas citadas anteriormente são descritas com mais detalhes nas subseções seguintes.

3.1.1 Identificar onde o software será refatorado

A primeira decisão que precisa ser feita é determinar o nível de abstração apropriado para aplicar a refatoração. O nível de abstração pode ser: baixo nível em que se refatora apenas o código do programa, alto nível é usado para refatorar a documentação e requisitos de software para depois modificar o código.

Segundo Kataoka (2011), através da ferramenta *daikon* é possível identificar onde a refatoração pode ser aplicada por uma detecção automática de variações de programa que comportam determinada técnica de refatoração citado na seção 2.3.

3.1.2 Determinar quais refatorações devem ser aplicadas nos lugares identificados

Após levantar os trechos de código que podem ser refatorados, é preciso identificar como fazer e qual técnica utilizar para a refatoração. Existem várias maneiras e métricas para se determinar qual técnica utilizar em uma inconsistência encontrada.

Vários autores utilizam maneiras diferenciadas para determinar as inconsistências no código fonte e qual refatoração utilizar. Fowler (2004) informalmente ligou os *bad smells* a refatorações, Mens e Tourwé (2004) utilizam uma abordagem semi-automática baseada em programação para detectar esse *bad smelss* e propor a refatoração adequada para removê-los (MENS,TOURWÉ, 2004). Emden e Moonen (2002) combinam a detecção de inconsistências com um mecanismo de visualização em Java.

Todos os autores focam no uso de relações para propor o método de refatoração a ser utilizado. O conceito chave utilizado é a métrica de coesão baseada na distância para medir o grau em que métodos e variáveis precisam estar juntos.

O uso de métricas baseadas em orientação a objetos é provavelmente a maneira mais adequada para determinar inconsistências e decidir onde aplicar refatorações.

3.1.3 Garantia da preservação de comportamento da aplicação

Por definição, uma refatoração não deve alterar o comportamento do software. Infelizmente, um conceito preciso de comportamento raramente é fornecido, ou pode ser ineficiente para ser verificado na prática.

A definição original de preservação de comportamento, afirma que, para o mesmo conjunto de valores de entrada, o conjunto resultante de valores de saída deve ser o mesmo antes e depois da refatoração.

3.1.4 Aplicar a refatoração

Após identificados as inconsistências no código e deduzidos os métodos que serão empregados para solucioná-las, esta etapa se concentra na aplicação das técnicas de refatoração no código fonte.

3.1.5 Avaliar o efeito da refatoração em características de qualidade do software

Para qualquer parte de um software se pode especificar atributos externos de qualidade tais como: robustez, extensibilidade, reutilização e desempenho.

Refatorações podem ser classificadas de acordo com qual desses atributos de qualidade ela afeta. Isso nos permite melhorar a qualidade de software por meio da aplicação refatoração certa. Para conseguir isso, cada refatoração deve ser analisada de acordo com o seu propósito e efeito.

Algumas refatorações podem remover redundância de código, outras aumentar o nível de abstração, melhorar reuso, e assim por diante. Este efeito pode ser estimado até certo ponto, expressando as refatorações em termos de qualidade interna de atributos que são afetados tais como: tamanho, acoplamento, complexidade e coesão.

Uma característica importante de qualidade de software que pode ser afetada por uma refatoração é o desempenho. Conforme citado na seção 2.2, uma desvantagem da padronização intensiva e aumento do nível de abstração de trechos do código fonte podem tornar o desempenho muito maior que soluções empregadas fora de um padrão.

Para medir ou estimar o impacto de uma refatoração em características de qualidade, muitas técnicas podem ser utilizadas, inclusive o uso de ferramentas automatizadas para esta medição.

3.1.6 Manter a consistência do código refatorado e outros artefatos de software (documentos, especificação de requerimentos, entre outros)

Geralmente, o desenvolvimento de software envolve uma ampla gama de artefatos de software, tais como: especificações de requisitos, arquiteturas de

software, modelos de projeto, código fonte, documentação, *suites* de teste, e assim por diante. Por exemplo, ao refatorar o código-fonte tem-se que garantir que os correspondentes testes de unidade permanecem consistentes. Da mesma forma, se diferentes tipos de modelos de projeto UML são reformulados, os outros têm que ser mantidos consistentes.

Para manter essa consistência, Mens e Tourwé, (2004) propõem o uso da técnica de propagação da mudança para lidar com inconsistências entre diferentes artefatos de software. Esta técnica manipula o fenômeno que, quando uma parte de um software é alterado, partes dependentes deste também precisam ser mudadas.

3.2 METODOLOGIA LEIDE RACHEL

Segundo Rapeli (2006), as etapas necessárias para refatorar um software são:

- Entender o sistema.
- Refatorar o código fonte utilizando padrões de projeto.
- Verificar sistema após refatoração.

Essas etapas são descritas detalhadamente nas subseções seguintes.

3.2.1 Entender o sistema

Dentro desta etapa existem três passos a serem feitos: entender a funcionalidade do sistema, recuperar modelo de classes do sistema existente e identificar padrões de projeto no código fonte do sistema.

O primeiro passo consiste em compreender a funcionalidade do sistema por meio de sua execução e armazenamento das entradas e saídas.

O segundo passo propõe a recuperação do diagrama de classes a partir do código fonte do projeto para reconhecimento de possíveis padrões de projeto. Se o diagrama de classes do projeto já existir, deve-se avaliá-lo quanto à real apresentação de informação em relação ao código fonte. Caso contrário, ele deve ser construído a partir do código fonte por meio do uso de ferramentas automatizadas ou manualmente.

A partir da informação obtida pela execução dos passos anteriores, é possível identificar determinados padrões de projeto a partir do código fonte e assim identificar a solução desejada.

3.2.2 Refatorar o código fonte utilizando padrões de projeto

Além da refatoração para um padrão de projeto, esta etapa também se caracteriza pela atualização dos artefatos de software simultaneamente a refatoração.

Após alguma parte do software sofrer qualquer tipo de mudança, o próximo passo será a atualização dos artefatos relacionados à parte afetada.

A apresentação do processo de refatoração é feita segundo as categorias dos padrões: de criação, estrutural e comportamental, propostas por Gamma et al. (1995).

Padrões de projeto foram feitos para solucionar um determinado tipo de problema que o código pode ter. Tomando isto por base, deve-se encontrar o problema e então implementar o padrão de projeto que o solucione.

3.2.3 Verificar sistema após refatoração

O objetivo principal desta etapa é verificar a funcionalidade do sistema após a refatoração e garantir que não houve nenhuma mudança de comportamento. Para isso recomenda-se a utilização das mesmas interações feitas no Passo 1 da primeira etapa (entender a funcionalidade do sistema) para analisar se as saídas após a refatoração permanecem inalteradas.

Caso existam alterações no comportamento do sistema, é possível que o padrão de projeto não foi implementado corretamente e o processo deve ser reiniciado.

3.3 COMPARAÇÃO ENTRE AS METODOLOGIAS

Tendo por base as etapas das metodologias apresentadas nas seções 3.1 e 3.2, é possível fazer uma breve comparação entre elas.

A principal diferença entre as abordagens é encontrada em suas fases iniciais. Na metodologia Mens, o nível de abstração é mais flexível permitindo ao programador definir a sua vontade o nível de refatoração, enquanto na metodologia de Rapeli (2006) este enfoque é obrigatoriamente uma abstração de alto nível, visto que uma etapa básica é gerar o diagrama de classes para que sejam identificados os possíveis padrões de projeto existentes ou que possam ser aplicados.

Sobre a aplicação da refatoração, pode existir uma diferença entre as metodologias dependendo do nível de abstração escolhido. Na metodologia de Mens, se for determinado um baixo nível de abstração é possível refatorar sem fazer grandes alterações de estrutura nas classes, enquanto na metodologia de Rapeli (2006) essas mudanças sempre são escolhidas com base na análise do diagrama de classes.

O próximo capítulo descreve o *Manipular ArcaboMK* que é o aplicativo utilizado neste trabalho para realizar o processo de refatoração.

4 MANIPULADOR ARCABOMK

Este capítulo apresenta alguns conceitos relacionados a concepção do software *Manipulador ArcaboMK*, bem como suas principais características e utilização. A Seção 4.1 relata os objetivos do aplicativo. A Seção 4.2 descreve a análise de sistemas similares. A seção 4.3 descreve a arquitetura de construção. A seção 4.4 relata alguns exemplos de utilização do manipulador. A seção 4.5 descreve as vantagens e desvantagens do manipulador.

4.1 OBJETIVOS DO APLICATIVO

O aplicativo manipulador foi desenvolvido para a manutenção de conteúdos do sítio *ArcaboMK* (ARCABOMK, 2011). Este sítio é responsável por apresentar o material didático do *FrameMK* – Um framework formação de preço de venda, e a grande característica deste sítio é a acessibilidade.

Segundo Lee (2001), o poder da web está em sua universalidade. O acesso feito por qualquer pessoa, independentemente da sua incapacidade, é um aspecto essencial.

O número de usuários de internet vem aumentando a cada ano, atraindo a atenção de todos os tipos de pessoas, inclusive as portadoras de necessidades especiais, sejam cognitivas quanto motoras. Estas pessoas necessitam de um auxílio maior na operabilidade em páginas web.

Mas, a acessibilidade web não se limita somente as pessoas com deficiências. Ela envolve também pessoas específicas, que por algum motivo, sentem dificuldade em navegar em páginas da web.

A partir destes conceitos, foram estudados os documentos de acessibilidade publicados pela W3C (I *Consorcium*) de 1999 e 2008 (WCAG 1.0, 1999; WCAG 2.0, 2008).

Com base nesses documentos, foram escolhidos certos critérios de acessibilidade que foram implantados no sítio *ArcaboMK*, o qual o permitiu alcançar o nível de acessibilidade AAA (WCAG 2.0, 2008).

Desta maneira, o aplicativo *Manipulador ArcaboMK* tem por objetivo cadastrar os conteúdos, de modo a construir uma estrutura lógica para a geração de

menus dinamicamente, tendo o foco em manter a acessibilidade atual do sítio *ArcaboMK* (LACERDA, 2011).

4.2 ANÁLISE DE SISTEMAS SIMILARES

A análise de sistemas similares teve por base os seguintes aspectos dos softwares:

- Gerenciamento de conteúdo web
- Software de licença gratuita

Existem muitas plataformas de gerenciamento de conteúdo dinâmico, como por exemplo, o *Wordpress* (WORDPRESS, 2011). Este representa uma plataforma com foco na estética, nos Padrões Web e na usabilidade, além de ser um software livre. Todavia, nele os novos conteúdos são adicionados através de *posts*, que geralmente utilizam uma barra em ordem cronológica por postagem.

Outro gerenciador de conteúdo dinâmico é a ferramenta *Joomla* (JOOMLA, 2011) que permite adicionar vários tipos de conteúdos de maneira simples, entretanto, não existe um mecanismo que valide o código HTML adicionado.

Estas ferramentas são fáceis de usar e permitem um controle completo de conteúdo, tanto em questão de organização de menus como em conteúdo da página. Porém, seu funcionamento não se adapta a forma de apresentação do conteúdo do sítio *ArcaboMK*, tanto pelo sistema de funcionamento das ferramentas mas também pelo fato de não gerarem páginas em conformidade com os critérios da W3C.

Ainda não existe também nenhum mecanismo que verifique se o conteúdo adicionado se encontra acessível dentro dos padrões de conformidade exigidos pela W3C.

No presente estágio de desenvolvimento do projeto, este mecanismo de verificação de validade do código adicionado conforme os padrões da W3C de acessibilidade, não chegou ainda a ser implementado, sendo este um dos propósitos para a refatoração de software proposta por este trabalho, pois facilitará a aplicação de novos módulos uma aplicação bem estruturada e padronizada.

4.3 ARQUITETURA DE CONSTRUÇÃO

Para o desenvolvimento do aplicativo, foi escolhido um padrão em camadas baseado no padrão *MVC* (FOWLER, 2006) e a linguagem *JSP* (JÚNIOR, 2003) utilizando *Beans* (JUNIOR, 2003).

O padrão *MVC* organiza aplicações em três módulos separados: um módulo conhecido como modelo – contendo os dados e as regras de negócio, o segundo módulo consiste na visão - responsável por apresentar os dados ao usuário (interface), e o terceiro módulo é o controlador - que direciona as requisições do usuário e o fluxo de dados (HOLMES, 2006).

Um dos benefícios da utilização deste padrão é a separação em módulos, facilitando a manutenção e evitando a duplicação de código. As camadas criadas para o Manipulador ArcaboMK estão separadas conforme ilustra a figura 1 por meio de um diagrama de pacotes, com seus respectivos componentes.

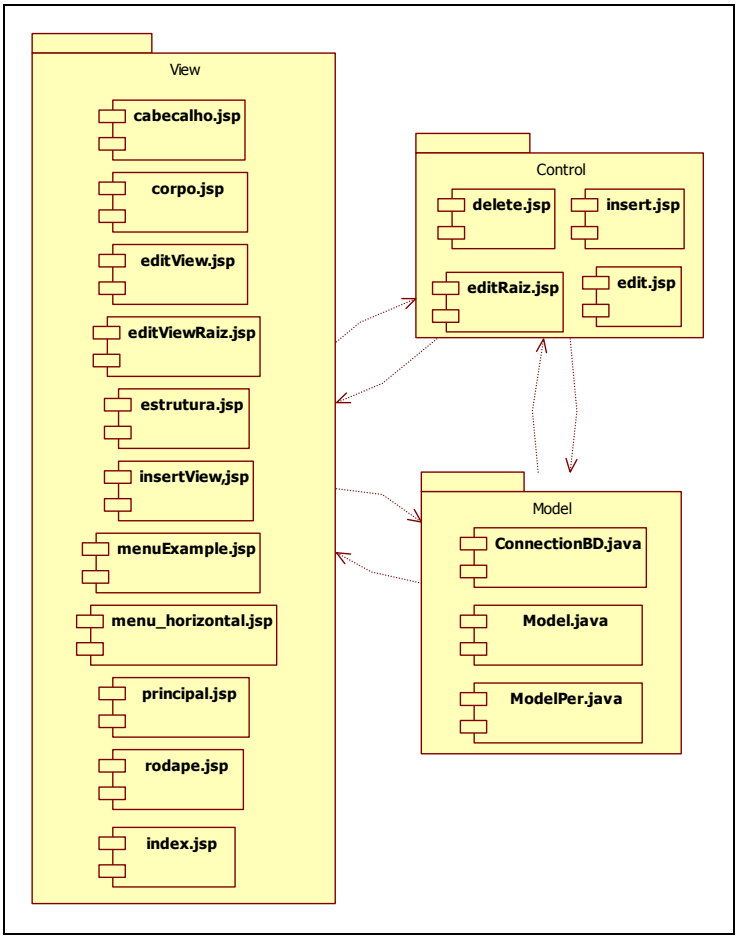

Figura 1 – Estrutura do projeto Manipulador Arcabomk

Fonte: Autoria própria

Três classes foram construídas no projeto, a classe *ConnectionBD* que é responsável pela conexão com o banco, a *Model* que é implementação do diagrama apresentado na figura 3 e a *ModelPer*, responsável por criar um objeto da classe *ConnectionBD* para que os métodos necessários a consulta, inserção, edição e exclusão do itens da tabela tenham acesso a tabela, como ilustra a figura 2.

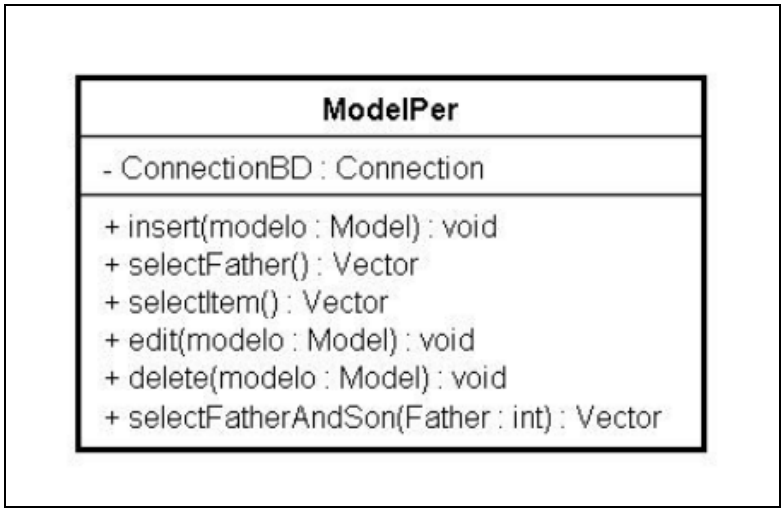

Figura 2 – Classe ModelPer

Fonte: Autoria própria

Como é possível perceber na figura 1, todas essas classes pertencem ao pacote *Model*.

Para banco de dados foi utilizada apenas uma tabela auto-relacional, responsável por construir a ordem lógica da estrutura necessária para os menus. Essa estrutura esta ilustrada na figura 3.

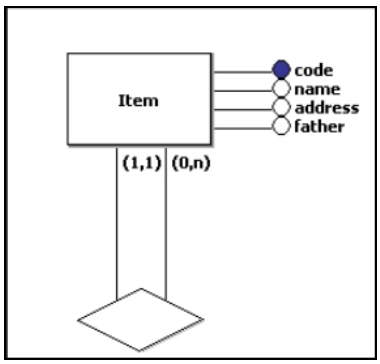

Figura 3 – Modelo entidade-relacionamento da tabela Item

Fonte: Autoria própria

Neste projeto, foram utilizadas duas maneiras diferentes para que os *beans* pudessem ser criados nas páginas *jsp*, apresentadas nas figuras 4 e 5.

```
<%@page import="java.util.Vector, Model.*" %>
```

Figura 4 – Acesso aos beans – Exemplo 01

Fonte: Autoria própria

```
<jsp:useBean id="modelo" class="Model.Model" scope="session"/>
```

Figura 5 – Acesso aos beans – Exemplo 02

Fonte: Autoria própria

As duas maneiras utilizam *tags* personalizadas, a primeira importa pacotes existentes no projeto ou de bibliotecas *Java*, sendo que para se criar um *bean* basta somente utilizar o método construtor da classe.

Na segunda é utilizada *tag jsp <jsp:useBean />*, essa ação tenta localizar o *bean* por intermédio dos atributos *id* e *scope*. Caso o *bean* não seja localizado, ela criará um novo *bean* instanciando a classe correspondente, tomando como referência o atributo *class*. Com essa *tag* também é possível fazer referência a um *bean* que tenha sido criado em outra página *JSP* (JÚNIOR, 2003).

Existem páginas *JSP* que fazem a ponte entre a visão e o modelo quando é realizada uma operação de inserção, edição ou exclusão. Essas páginas são responsáveis por capturar os valores do formulário da camada de visão e mandá-los para a classe *ModelPer* que manipulará o banco de acordo com a operação.

A figura 6 apresenta o código de um método da classe *ModelPer*, de uma operação de inserção, enquanto a figura 7 ilustra o código da página *JSP* correspondente a essa operação na camada de controle.

```java
public class ModelPer {

    ConnectionBD connection;
    public ModelPer()
    {

    }

    public void insert(Model modelo){
        connection = new ConnectionBD();
        int father = modelo.getFather();
        String name = modelo.getName();
        String address = modelo.getAddress();
       String sql = "insert into item(pai,nome,endereco) values("+father+",'"+name+"','"+address+"')";

        PreparedStatement pstm = null;
        try {
            pstm = connection.connect().prepareStatement(sql);
            pstm.executeQuery();
            connection.disconnect();
        } catch (SQLException e) {
              e.printStackTrace();
        }
    }
```

Figura 6 – Código do método de inserção da classe ModelPer

Fonte: Autoria própria

```jsp
<jsp:useBean id="modelo" class="Model.Model" scope="session"/>
<%

modelo.setName(request.getParameter("txtNome"));
modelo.setAddress(request.getParameter("txtEndereco"));
modelo.setFather(Integer.parseInt(request.getParameter("txtPai")));

%>
<jsp:useBean id="conexao" class="Model.ModelPer" scope="session"/>
<%
conexao.insert(modelo);
%>
<h1>O registro foi inserido com sucesso</h1>
<a href="../View/index.jsp">Voltar para a página inicial</a>
```

Figura 7 – Código da página insert.jsp

Fonte: Autoria própria

Na primeira linha se cria um *bean* da classe *Model*, em seguida são recuperados os valores do formulário da visão correspondentes ao campo da classe *Model* através do trecho de código *request.getParameter("Nome do componente ou do parâmetro setado anteriormente")*.

Recuperando os valores inseridos no formulário pelo usuário, é criado um *bean* da classe *ModelPer* para ser utilizado o método *insert* que recebe como parâmetro o objeto em que foram estabelecidos os valores.

As outras operações (edição e exclusão) correspondentes às outras páginas *JSP* presentes na camada de controle, serão explicadas posteriormente por conterem regras de aplicação mais detalhadas.

Sobre as páginas *JSP* da interface, o *index.jsp* é a primeira página do sistema e encontra-se fora do diretório *View* para atender o padrão de projeto usado pelo *netbeans 6.8* (NETBEANS, 2011). Esta página é responsável pela estrutura da página, como por exemplo, o cabeçalho do menu horizontal e o rodapé, como apresenta a figura 8.

O código da página principal é responsável apenas pela tabela presente na figura 8, para os outros componentes foi reaproveitado o código do sítio *ArcaboMK*, utilizando o mecanismo inclusão de páginas apresentado no relatório de Ribas e Lacerda (2010). Dessa maneira, estes componentes são carregados uma única vez pela página *index.jsp* que apenas incluí as páginas do manipulador.

Figura 8 – Visualização da página index.jsp

Fonte: Autoria própria

Como a página *principal.jsp* é responsável apenas pela tabela apresentada na figura 8, ela acessa diretamente a classe *ModelPer* da camada de modelo utilizando métodos que realizam consultas no banco, como ilustra a figura 9.

```
<%@page import="java.util.Vector, Model.*" %>


<%

ModelPer conexao = new ModelPer();
Vector modelo = new Vector();
modelo = conexao.selectItem();
request.getSession().setAttribute("vector", modelo);
%>
```

Figura 9 – Exemplo de código de acesso à classe da camada de modelo

Fonte: Autoria própria

Depois de criado o *bean* da classe *ModelPer*, também é instanciado um objeto do tipo *Vector* que receberá todos os índices da tabela através do método *selectItem*, que realiza uma consulta na tabela. A linha *request.getSession().setAtribute("nome do atributo",nome do objeto que será enviado);* estabelece o objeto do tipo *Vector* em um atributo de sessão, que poderá ser recuperado em qualquer página *JSP*.

O método *selectItem* é similar ao apresentado na figura 7, mudando apenas o código *SQL*. Para mostrar na página cada linha da tabela que se encontra como um objeto do tipo *Model* dentro do objeto *Vector*, é necessário um laço de repetição como ilustra a figura 10.

```
<%
            for (int i = 0; i < modelo.size(); i++) {
                Model m = new Model();

                m = (Model) modelo.elementAt(i);

%>
<tr>
    <td align="center">
        <% out.println(m.getCode());%>
    </td>

    <td align="center">
        <% out.println(m.getName());%>
    </td>

    <td align="center">
        <% out.println(m.getAddress());%>
    </td>

    <td align="center">
        <% out.println(m.getFather());%>
    </td>

    <td align="center">

    <td align="center">

        <input type="button" value="Editar" onClick="javascript:editar(<%
    </td>
    <td align="center">
        <input type="button" value="Deletar" onclick="javascript:deletar(
    </td>
    <% }%>
```

Figura 10 – Código da página index.jsp

Fonte: Autoria própria

Dessa maneira, é mostrada em cada linha da tabela da interface a linha correspondente no banco. Na página *index.jsp* e na página *principal.jsp* existem *links* para a outras páginas de interface da camada visão, como os formulários para a inserção e a edição de registros. A figura 11 ilustra a tela de inserção do sistema.

Figura 11 – Visualização da tela de inserção

Fonte: Autoria própria

A página *insertView.jsp* da camada de visão, responsável pelo formulário, faz um acesso ao banco porque o cadastro necessita que sejam listados os nomes de cada linha da tabela item, que são as opções da caixa de seleção, para que ao cadastrar um novo assunto se escolha um item pai para o mesmo. No caso do item pertencer à raiz do sítio, já existe um dado pré-cadastrado no banco que é a raiz.

O código para a criação do *bean* é o mesmo mostrado para a página *index.jsp*, em que são criados um *bean* da classe *ModelPer* e um objeto do tipo *Vector* que receberá os itens, porém dessa vez é utilizado o método *selectFather();* como ilustra a figura 12.

```
<%

ModelPer conexao = new ModelPer();
Vector modelo = new Vector();
modelo = conexao.selectFather();
%>
```

Figura 12 – Código da página insertView.jsp que utiliza o método selectFather()

Fonte: Autoria própria

Este método difere do método *selectItem();* porque realiza uma busca retornando apenas as colunas: código e nome. Da mesma maneira que os itens

foram listados na página *index.jsp,* os dois atributos referentes às duas colunas citadas anteriormente serão listados no meio da tag *HTML <option value="valor da opção selecionada">"texto da caixa de seleção"</option>*, como ilustra a figura 13.

```
<select id="sel_pai" name="txtPai" aria-labelledby="pai-ariaLabel">
    <%
    for(int i=0; i < modelo.size(); i++){
      Model m = new Model();
      m = (Model) modelo.elementAt(i);

    %>
        <option value=<% out.println(m.getCode()); %> >
        <% out.println(m.getName());  }  %></option>
</select>
```

Figura 13 – Código da listagem das colunas código e nome para a caixa de seleção
Fonte: Autoria própria

Como na página *principal.jsp*, o laço percorre o objeto *Vector*, que em cada passo é atribuído a um objeto do tipo *Model*, porém, agora são as propriedades da *tag HTML <option>* é que recebem a saída.

Preenchidos os dados é chamada a página *insert.jsp* da camada de controle, como já explicado na figura 7.

A estrutura do formulário para edição é basicamente a mesma, a diferença é que ao abri-lo os dados do item selecionado precisam estar preenchidos no formulário para sua modificação, como ilustra a figura 14.

Figura 14 – Visualização da página editView.jsp
Fonte: Autoria própria

Cada linha possui uma chamada para a página *editView.jsp*, pois quando a página é invocada é enviado com ela um parâmetro, que é o índice do objeto *Vector* na página *principal.jsp*. Deste modo, os dados são enviados e recuperados como ilustram as figuras 15,16 e 17.

```
<input type="button" value="Editar" onClick="javascript:editar(<%out.println(i);%>)"/>
```

Figura 15 – Chamada da função editar em javascript

Fonte: Autoria própria

```
<SCRIPT LANGUAGE="JavaScript" TYPE="text/javascript">
    function editar(codigo) {
        if(codigo==0){
            location.href="./index.jsp?link=View/editViewRaiz&codigo="+codigo;}
        else{
            location.href="./index.jsp?link=View/editView&codigo="+codigo;}
    }
```

Figura 16 – Envio do índice como parâmetro através da chamada da página editView.jsp

Fonte: Autoria própria

```
<%

int index = Integer.parseInt(request.getParameter("link"));
Vector vector = new Vector();
vector = (Vector)request.getSession().getAttribute("vector");

Model linha = (Model)vector.elementAt(index);
request.getSession().setAttribute("index", linha);

            String nomePai=null;
            for(int i=0; i < vector.size(); i++){
              Model l = new Model();
              l = (Model) vector.elementAt(i);
              if(l.getCode()==linha.getFather()){
                  nomePai=l.getName();
                  }
              }
            %>
```

Figura 17 – Código de recuperação dos dados na página editView.jsp

Fonte: Autoria própria

Na figura 15 o código é responsável pela chamada da função *javascript editar(código)*.O código da figura 16, primeiro é verificado se o item selecionado

para a edição é a raiz, caso seja, realiza-se a chamada a página *editViewRaiz.jsp* ao invés da página *editView*.

A página *editViewRaiz* funciona de maneira similar a página *editView* e será explicada posteriormente. A parte do código *?link="....."* envia o índice enviando como parâmetro junto com a chamada da página.

Na página *editView.jsp* é recuperado o objeto *Vector* gerado para a página *index.jsp* que foi estabelecido conforme ilustrou a figura 9.

Como a lista possuí índice não é preciso percorrer o *Vector* para obter o objeto desejado como mostra a 4º linha do código da figura 17.

Na próxima linha é estabelecido para esse objeto em outro atributo de sessão para recuperar a coluna do código na página *Edit.jsp* da camada de controle.

Sabendo que a coluna "pai" do banco de dados guarda apenas o código do item pai, é necessário fazer uma busca no *Vector* comparando os códigos, para então identificar o nome do item pai, como mostrado no laço da figura 17.

Com os dados recuperados, os mesmos são colocados no formulário de maneira semelhante ao que foi mostrado na figura 13, porém sem o laço de repetição.

Apenas a caixa de seleção deste formulário envolve uma lógica mais complexa, pois como se recupera o nome e o código do pai do item selecionado é necessário que ele venha preenchido nesse campo e que todos os outros também sejam listados para uma eventual mudança por parte do usuário. Essa lógica está ilustrada no código da figura 18.

```
<select id="sel_pai" name="txtPai" aria-labelledby="pai-ariaLabel"  >
    <option value= <% out.println(linha.getFather()); %> >
        <% out.println(nomePai);    %></option>

    <%

    for(int i=0; i < vector.size(); i++){
      Model m = new Model();
      m = (Model) vector.elementAt(i);
      if((m.getCode()!=linha.getFather()&&(m.getCode()!=linha.getCode()))){
    %>
        <option value= <% out.println(m.getCode()); %> >
        <% out.println(m.getName());  }) %></option>
</select>
```

Figura 18 – Código da caixa de seleção da página insertView.jsp

Fonte: Autoria própria

O funcionamento da listagem de itens para edição é similar ao apresentado no formulário de inserção, a diferença é que o código e o nome são estabelecidos como valor inicial da caixa de seleção, apresentados pelas três primeiras linhas do código da figura 18.

Em seguida, é utilizado um laço para listar os itens restantes da tabela, entretanto é necessário tomar o cuidado de não repetir o item que é o pai do elemento selecionado para a edição e também o próprio elemento, relembrando que eles são recuperados do *Vector* gerado na página *principal.jsp* e por isso possui estes dois itens que não podem ser listados nesta caixa.

Desse modo, todos os dados do item estão recuperados e a validação necessária para manter a integridade dos dados no banco estão prontos. Ao clicar no botão enviar é chamada a página *edit.jsp* da camada de controle. Esta página tem o funcionamento igual à página *insert.jsp*, sendo diferente em alguns aspectos ilustrados no código da figura 19.

```
<%
Model index = new Model();
index= (Model)request.getSession().getAttribute("index");

Model modelo = new Model();

modelo.setCode(index.getCode());
modelo.setName(request.getParameter("txtNome"));
modelo.setAddress(request.getParameter("txtEndereco"));
modelo.setFather(Integer.parseInt(request.getParameter("txtPai")));

%>

<%
ModelPer conexao= new ModelPer();
conexao.edit(modelo);
%>
<h1>O registro foi editado com sucesso</h1>
<a href="../View/index.jsp">Voltar para a página inicial</a>
```

Figura 19 – Código da página edit.jsp

Fonte: Autoria própria

Inicialmente é recuperado o atributo setado na página *editView.jsp*, como mostrado na figura 17. Logo após, os dados são setados em um *bean* do tipo *Model*

e então é utilizado o método *edit(Model modelo)* da classe *ModelPer* que executa uma *SQL* de edição no banco e envia os dados para o banco.

Na camada de visão e de controle existem também mais duas páginas, que são responsáveis pela edição do item Raiz do sítio, pois convencionou-se que é inviável editar o nome e o pai deste item que já está cadastrado como o primeiro registro do banco de dados.

Desta maneira, no método *editar()* em *javascript* da página *principal.jsp* é chamada a página *editViewRaiz.jsp*. Esta página contém um formulário diferenciado, o qual só permite a edição do endereço da raiz do sítio, como ilustra a figura 20.

Figura 20 – Visualização da página editViewRaiz.jsp

Fonte: Autoria própria

Essa página funciona do mesmo modo que a página de edição, porém mais simples porque é necessário mostrar na tela o endereço.

Quando a página é submetida pelo usuário se realiza uma chamada a página *editRaiz.jsp* da camada de controle que faz o mesmo procedimento da página *Edit.jsp*, pois também utiliza o método de edição da classe *ModelPer*, porém é necessária a criação desta página porque o formulário só possui um campo de texto. Então, faz-se a requisição para a página *edit.jsp*, a linha *request.getParameter("nome do componente HTML")* exibirá um erro, porque não existe o componente neste formulário.

Para eliminar um item se utiliza o mesmo procedimento aplicado na edição, em que é enviado o índice do item para a página de controle e na página

principal.jsp é chamada a função *javascript deletar(codigo)* ilustrada pelas figuras 21 e 22.

```
<input type="button" value="Deletar" onclick="javascript:deletar(<%out.println(i);%>);"/>
```

Figura 21 – Chamada da função deletar(codigo) em javascript

Fonte: Autoria própria

```
function deletar(codigo){
    decisao = confirm("Tem certeza que deseja excluir?");
    if (decisao){
        location.href="./index.jsp?link=Control/delete&delete="+codigo;
    }
}
```

Figura 22 – Função deletar(código) em javascript

Fonte: Autoria própria

Na segunda linha da figura 22, é chamada uma caixa de diálogo com o usuário, ilustrada pela figura 23.

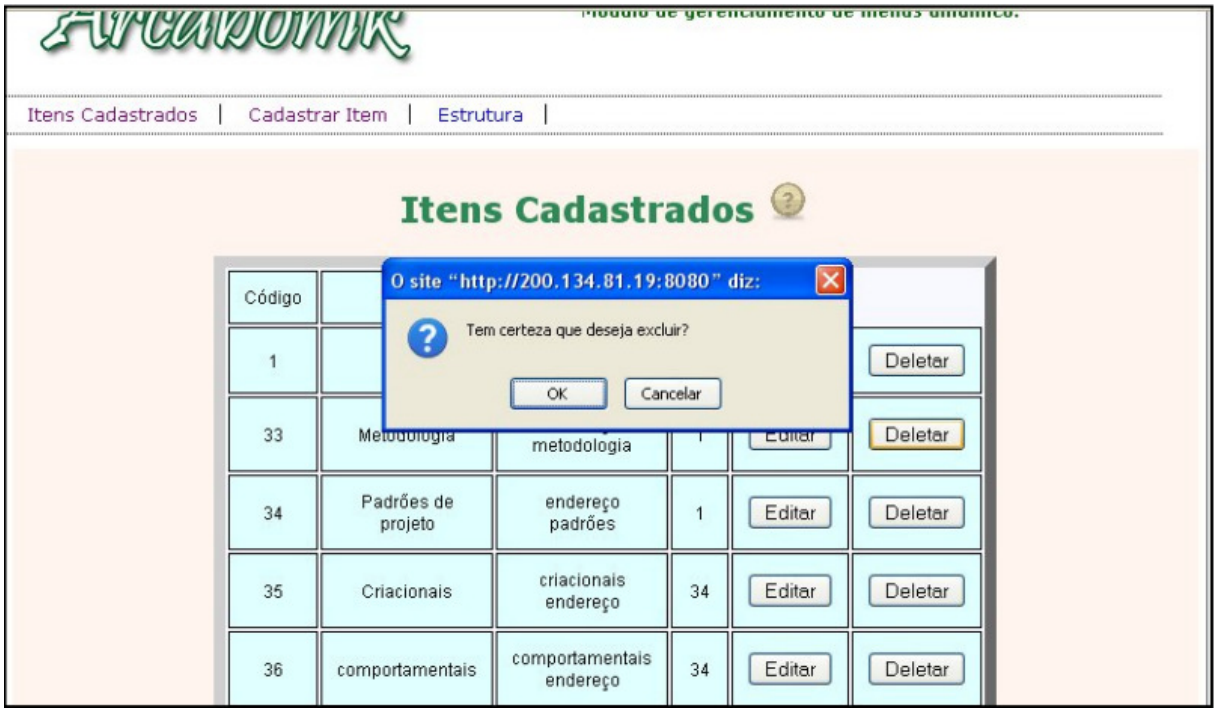

Figura 23 – Caixa de diálogo para a exclusão de item

Fonte: Autoria própria

Se a resposta fornecida pelo usuário for positiva, é chamada a página *delete.jsp* da camada de controle e é enviado o índice do item desejado. Caso contrário, a página atual é mantida.

A página *delete.jsp* funciona de maneira similar a página *edit,jsp* porém a confirmação da exclusão é o retorno do método *delete()* da classe *ModelPer*, como ilustram as figuras 24 e 25.

```
<%@page import="java.util.Vector,Model.*" %>
<%@page import="java.sql.SQLException" %>


<%
int index = Integer.parseInt(request.getParameter("delete"));
Vector vector = new Vector();
vector = (Vector)request.getSession().getAttribute("vector");
Model m = new Model();
m= (Model)vector.elementAt(index);



ModelPer conexao= new ModelPer();


%><h1><%
    out.println(conexao.delete(m));
%>
</h1>
```

Figura 24 – Código da página delete.jsp

Fonte: Autoria própria

```
public String delete(Model modelo) {
    connection = new ConnectionBD();
    int code = modelo.getCode();

    String sql = "delete from item where codigo=" + code;

    PreparedStatement pstm = null;
    try {
        pstm = connection.connect().prepareStatement(sql);
        pstm.executeUpdate();
        connection.disconnect();
    } catch (SQLException e) {
        return e.getMessage();
    }
    return "O registro foi deletado com sucesso";
}
```

Figura 25 – Código do método delete() da classe ModelPer

Fonte: Autoria própria

Na figura 24 observa-se que a página mostra na tela o retorno do método por meio da linha *out.println(conexão.delete(m))*. Caso não ocorra nenhuma exceção, na figura 25 será retornada a mensagem *"O registro foi deletado com sucesso"*, como ilustra a figura 26.

Figura 26 – Visualização da página delete.jsp caso não ocorra nenhuma exceção
Fonte: Autoria própria

Foi utilizado esse sistema porque existem duas validações no banco de dados para a exclusão de um item e caso uma delas ocorrer é gerada uma exceção. Uma dessas validações é a verificação do item para que ele não seja o item raiz, porque para que o cadastro no sistema funcione é necessário que exista pelo menos um registro no banco.

Essa validação foi feita por meio de uma *trigger* ilustrada nas figuras 27 e 28.

```
CREATE TRIGGER undelete
  BEFORE DELETE
  ON item
  FOR EACH ROW
  EXECUTE PROCEDURE undelete();
```

Figura 27 – Trigger de verificação
Fonte: Autoria própria

```
CREATE OR REPLACE FUNCTION undelete()
  RETURNS trigger AS
$BODY$begin
if(old.codigo=1) then
raise exception 'A raiz não pode ser deletada.';
end if;
return old;
end;$BODY$
  LANGUAGE plpgsql VOLATILE
  COST 100;
ALTER FUNCTION undelete() OWNER TO postgres;
```

Figura 28 – Função chamada na trigger de verificação

Fonte: Autoria própria

Na figura 27 é criada a *trigger* após exclusão para a tabela item e é chamada a função *undelete()*, ilustrada na figura 28. Nela é comparado se o código do registro que está sendo excluído é igual a 1. Caso seja, é gerada uma exceção com a mensagem *"A raiz não pode ser deletada"*. Na página *delete.jsp* a visualização deste erro está ilustrada na figura 29.

Figura 29 – Mensagem da página delete.jsp caso o item para a exclusão seja a raiz

Fonte: Autoria própria

Outra validação no banco de dados é que, ao excluir um registro que seja pai de outros filhos, estes também deverão ser excluídos. Para isso foi utilizada a *SQL* ilustrada pela figura 30.

```
CREATE TABLE item
(
  codigo serial NOT NULL,
  pai integer,
  nome text,
  endereco text,
  CONSTRAINT codigo PRIMARY KEY (codigo),
  CONSTRAINT pai FOREIGN KEY (pai)
      REFERENCES item (codigo) MATCH SIMPLE
      ON UPDATE CASCADE ON DELETE CASCADE
)
```

Figura 30 – DDL de criação da tabela item

Fonte: Autoria própria

O campo código é do tipo serial, pois gera automaticamente uma *Sequence(generator)* para o auto incremento do campo.

A parte do código responsável pela validação citada anteriormente é *"MATCH SIMPLE ON UPDATE CASCADE ON DELETE CASCADE"*. Esta funcionalidade parte da ideia de que quando um item é excluído, todos os registros que possuam o valor do código(chave primária) no campo pai(chave estrangeira) também deve ser excluído, ou seja, é um mecanismo de exclusão em cascata(MANUAIS POSTGREE, 2011).

Para exemplificar a utilização do sistema foi implementado também a geração de menus, que busca no banco os itens filhos do item atual e monta o menu dinamicamente a partir dos dados já cadastrados, como exemplifica a figura 31.

Figura 31 – Exemplo de geração de menus

Fonte: Autoria própria

Ao clicar em um item será chamada sua página de conteúdo, o que permite a navegação de anterior e posterior entre os assuntos.

O método responsável por essa geração de menu é *selectFatherAndSon(codigoPai)*, ilustrado na figura 32.

```
    public Vector selectFatherAndSon(int codigoPai) {
        String sql;
        PreparedStatement preparedStatement = null;
        connection = new ConnectionBD();
        ResultSet resultSet;
        Vector vetorModelo = new Vector();
        int contador =1;
        vetorModelo.add(0,null);
        try {
            sql = "select codigo, nome, pai from item where codigo<>1 and pai="+codigoPai+" " +
                    "or codigo="+codigoPai;
            preparedStatement = connection.connect().prepareStatement(sql);
            resultSet = preparedStatement.executeQuery();
            while (resultSet.next()) {
                Model model = new Model();
                model.setCode(resultSet.getInt("codigo"));
                model.setName(resultSet.getString("nome"));
                model.setFather(resultSet.getInt("pai"));
                if (resultSet.getInt("codigo")==codigoPai){
                    vetorModelo.set(0,model);
                } else {
                    vetorModelo.add(contador,model);
                    contador = contador +1;
                }
            }
            connection.disconnect();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return vetorModelo;
    }
}
```

Figura 32 – Método da classe ModelPer responsável pela geração de menus

Fonte: Autoria própria

O método funciona da seguinte maneira: recebe um parâmetro da página *menuExample.jsp* com o código do item clicado e realiza uma busca no banco do próprio item e de seus filhos.

Essa busca é colocada em um objeto do tipo *Vector*, que por convenção sempre receberá o pai na primeira posição e os filhos nas restantes, representado na decisão da figura 32.

Este *Vector* é retornado para a mesma página *menuExample.jsp*, onde é listado nos *links* do menu, para que os valores do código sejam novamente

passados como parâmetro caso um *link* seja clicado. O código da página *menuExample.jsp* está ilustrado na figura 33.

```
<%
            int codigoPai = 1;
            if (request.getParameter("codigoPai")!=null)
                codigoPai=Integer.parseInt(request.getParameter("codigoPai"));
            ModelPer conexao = new ModelPer();
            Vector modeloVetor = new Vector();
            modeloVetor = conexao.selectPaieFilho(codigoPai);
%>
<h1>Exemplo de Estrutura de menu</h1>

            <ul>
                <li><a href="./index.jsp?link=View/menuExample&codigoPai=
                       <%out.print(((Model) modeloVetor.elementAt(0)).getFather());%>">
                        <%out.print(((Model) modeloVetor.elementAt(0)).getName());%>
                    </a>
                </li>
                <%       //remover o pai deixando só os filhos no vetor
                            modeloVetor.remove(0);
                %>

                <ul>
                    <%
                                while (!modeloVetor.isEmpty()) {%>
                    <li>
                        <a href="./index.jsp?link=View/menuExample&codigoPai=
                           <%out.print(((Model) modeloVetor.elementAt(0)).getCode());%>">
                            <%out.print(((Model) modeloVetor.elementAt(0)).getName());%>
                        </a>
                    </li>
                    <%
                    //remove o vetor já printado e reinicia o loop até não ter mais filhos
                    modeloVetor.remove(0);
```

Figura 33 – Código da página menuExample.jsp

Fonte: Autoria própria

Inicialmente é feita uma verificação se o parâmetro da chamada da página *menuExample.jsp* não é nulo. Caso seja verdadeiro, o código do item atual é enviado para o método de busca. Quando a página é chamada a primeira vez o padrão de valor do parâmetro é sempre a raiz.

Depois de buscar os dados no banco, são listados o item atual que foi enviado como parâmetro para a busca, e em seguida todos os seus filhos, que ao serem clicados repetem o processo de navegação.

4.4 EXEMPLOS DE UTILIZAÇÃO

O uso do aplicativo consiste nas seguintes operações:

- Cadastro de itens
- Edição de itens
- Exclusão de Itens
- Visualização dos menus gerados

4.4.1 Cadastro de itens

A partir da tela de cadastro apresentada na figura 11, são inseridos o nome do conteúdo que aparecerá no menu, o endereço do arquivo da página, e o campo pai, criando assim a estrutura de menu conforme o esquema mostrado pela figura 34.

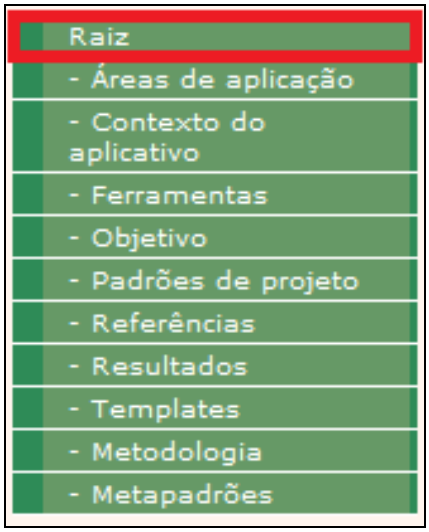

Figura 34 – Estrutura de menu

Fonte: Autoria própria

O item destacado é o item pai, enquanto os itens abaixo são os itens filhos do conteúdo destacado.

4.4.1 Edição de itens

Utilizando o mesmo formulário da tela de cadastro apresentada na figura 14, são preenchidos os campos com os dados do item escolhido para edição, funciona de forma similar ao cadastro.

4.4.1 Exclusão de itens

Utilizando a tela principal, é possível apagar qualquer item cadastrado. Porém, ao excluir um item que seja *pai* de outro, consequentemente todos os subitens de conteúdo serão apagados também.

Isto é realizado a nível lógico, pois os arquivos das páginas dos conteúdos ainda existirão dentro do projeto.

4.4.1 Visualização dos menus gerados

Funcionalidade implementada para oferecer informação visual ao usuário sobre a estrutura menus. Gera os menus da mesma forma que os métodos implementados no sítio *ArcaboMK*.

4.5 VANTAGENS E DESVANTAGENS

No antigo sítio *ArcaboMK*, por se tratar de um sítio estático, para adicionar um novo conteúdo era necessário realizar mudanças nos códigos fontes, e no caso quando um assunto continha um subconteúdo, criavam-se menus dentro das páginas para manter uma ordem de navegação não visual, a exemplo de navegação através do teclado.

Existem várias plataformas e ferramentas *online* que permitem gerenciar os conteúdos de maneira fácil, um exemplo simples delas são os *blogs* como, por exemplo, o *Wordpress* (WORDPRESS, 2011) e a ferramenta *Joomla!* (JOOMLA, 2011) que são gratuitas. Porém, foi priorizado o desenvolvimento desta ferramenta descrita no presente trabalho, que será parte funcional de um gerenciador de

conteúdo com foco em acessibilidade, pois não existe nenhum gerenciador de conteúdo web que vise a melhor acessibilidade da página (WCAG 2.0, 2008).

A solução encontrada para estes problemas foi o desenvolvimento de um aplicativo que permite o cadastro dos conteúdos e que o sítio do *ArcaboMK* realizasse uma busca no banco de dados para gerar o menu. Desta maneira, a adição de conteúdos foi facilitada, pois ao clicar em um subitem um novo menu é gerado.

Dentro deste aspecto, o software possui a desvantagem de não oferecer um gerenciamento completo como estas ferramentas CMS (CMS, 2011), porém sua vantagem se encontra no aspecto de que seu desenvolvimento tem como objetivo gerenciar a página mantendo e aplicando os critérios de acessibilidade, funcionalidade ainda não apresentada pelas outras ferramentas.

Tendo em vista a dimensão de arquitetura e desenvolvimento dessa aplicação, a refatoração do aplicativo *Manipulador ArcaboMK* é necessário para tornar mais fácil o desenvolvimento de novas funcionalidades para a construção do gerenciador de conteúdo.

Portanto, a seguir será descrito o processo de refatoração usado neste trabalho para realizar a refatoração no Manipulador *ArcaboMK.*

5 METODOLOGIA DE REFATORAÇÃO PROPOSTA

Este capítulo apresenta a metodologia proposta e seu objetivo para realizar a refatoração do *Manipulador ArcaboMK*. A Seção 5.1 relata o objetivo da refatoração. A Seção 5.2 descreve a metodologia proposta para a refatoração.

5.1 OBJETIVO DA REFATORAÇÃO

Como citado no capítulo anterior, o aplicativo *Manipulador Arcabomk* é uma parte funcional de um software de gerenciamento de conteúdo web.

Devido ao desenvolvimento ágil empregado em sua construção, os padrões e cuidados com a estrutura do software não foram elaborados de maneira a comportar a adição de novas funcionalidades, processo constantemente usado para o desenvolvimento do gerenciador de conteúdo.

Assim, a refatoração do aplicativo se torna necessária, para padronizar o código e a sua estrutura, de maneira que o desenvolvimento do gerenciador de conteúdo possa ser incrementado e mantido sobre uma base sólida.

5.2 METODOLOGIA PROPOSTA

Com base nos dois métodos apresentados Mens (2004) e Rapeli (2006), elaboraram-se as etapas para a refatoração do aplicativo *Manipulador ArcaboMK* ilustradas na figura 35.

As etapas *Compreender funcionalidade do sistema* e *Avaliar diagrama de classes* foram retiradas do método proposto por RAPELI (2006), enquanto as etapas *Avaliar características de qualidade e desempenho* foi elaborada a partir da metodologia proposta por Mens e Tourwé (2004). As etapas restantes possuem características e processos similares para as duas metodologias.

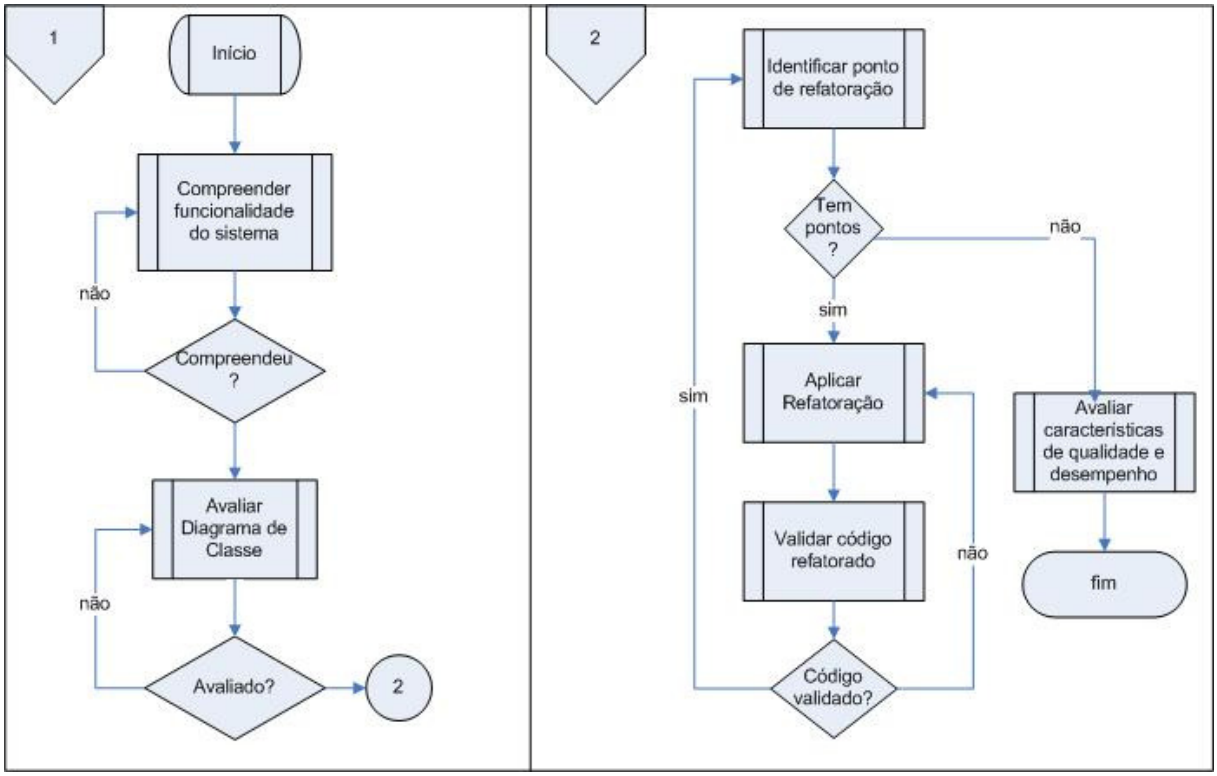

Figura 35 – Fluxograma da metodologia proposta

Fonte: Autoria própria

Essas etapas são descritas detalhadamente nas próximas subseções.

5.2.1 Compreender a funcionalidade do sistema

Inicialmente é feita uma análise de utilização do aplicativo *Manipulador ArcaboMK*, dos possíveis dados de entrada em cada tela e sua respectiva saída. Esta análise é feita manualmente por meio de uma tabela associativa que contém as respostas do sistema a determinadas ações feitas pelo usuário. Esta atividade está presente na metodologia apresentada por Rapeli (2006).

5.2.2 Avaliar diagrama de classe

Como explicado na metodologia de Rapeli (2006), nesta etapa efetua-se uma avaliação de consistência no diagrama de classes. Essa avaliação é relativa a real apresentação de informação em relação ao código fonte, visando sua adaptação para o padrão de projeto desejado.

Essa avaliação visa a adaptação ao padrão e estrutura de projeto utilizada pelo framework de aplicação *Struts* (HUSTED 2004).

5.2.3 Identificar ponto de refatoração

Como o foco desta refatoração se detém na estrutura do aplicativo, deve-se criar um projeto utilizando a estrutura do framework *Struts*, e então migrar um a um os componentes do projeto *Manipulador ArcaboMK*.

5.2.4 Aplicar refatoração

Efetua-se as mudanças necessárias no código de maneira que ele funcione utilizando a nova estrutura, sendo que estas alterações serão avaliadas em etapas posteriores.

5.2.5 Validar o código refatorado

Após a refatoração é analisada novamente as entradas e saídas das telas projeto e também alguns dos documentos como diagrama de classes e diagrama de pacotes do aplicativo.

A análise desta etapa é comparativa, utilizando os resultados e artefatos gerados antes e após a refatoração.

5.2.6 Avaliar características de qualidade e desempenho

A antiga versão do aplicativo *Manipulador ArcaboMK e sua versão refatorada* serão avaliados em características de qualidade através do *plugin* para eclipse *Metrics* (2012).

Optou-se pelo uso deste *plugin* pela fácil utilização e por sua compatibilidade com a IDE de desenvolvimento Eclipse, plataforma de desenvolvimento utilizada na refatoração do aplicativo *Manipuldor ArcaboMK*.

O *Metrics* analisa diversas características de qualidade em relação ao código e apresenta vários índices para análise. Os índices que serão avaliados estão listados a seguir:

- *Number of classes*
- *Number of children*
- *Number of Overridden Methods* (NORM)
- *Number of Methods* (NOM)
- *Number of Fields*
- *Lines of Code*
- *Weighted Methods per Class* (WMC)
- *Lack of Cohesion of Methods* (LCOM*)

O próximo capítulo descreve a aplicação da metodologia proposta no Manipulador *ArcaboMK*, bem como uma avaliação de sua versão anterior e nova.

6 APLICAÇÃO E RESULTADOS DO PROCESSO DE REFATORAÇÃO NO MANIPULADOR ARCABOMK

Este capítulo apresenta a aplicação e os resultados atingidos por meio da metodologia proposta no capítulo anterior. A seção 6.1 descreve a etapa "Compreender funcionalidade do sistema". A seção 6.2 relata a etapa "Avaliar diagrama de classe". A seção 6.3 narra a etapa "Identificar pontos de refatoração". A seção 6.4 apresenta a etapa "Aplicar a refatoração". A seção 6.5 descreve a etapa "Validar código refatorado". Por fim, a seção 6.6 apresenta a avaliação do software refatorado usando o *plugin Metrics* descrita na etapa "Avaliar características de qualidade e desempenho", permitindo uma análise entre as versões.

6.1 COMPREENDER A FUNCIONALIDADE DO SISTEMA

De acordo com o método apresentado por Rapeli (2006), devem ser listadas as entradas e saídas de software, como apresentado no Quadro 1.

Quadro 1 – Interações do usuário com o sistema Manipulador ArcaboMK antes da refatoração

N° de interação	**Interação do usuário no sistema (entrada)**	**Resposta do sistema na tela do usuário**
Operações de acesso do sistema		
1	Executar o sistema	A tela principal é apresentada contendo os dados cadastrados na tabela i*tem* , um menu que está presente em todas as páginas que contém *links* para as páginas de cadastro, a página principal e uma página que apresenta a estrutura dos menus em uma tabela. Ao lado de cada linha da tabela que apresenta os dados cadastrados existem dois botões, um que direciona para a página de edição e outro para exclusão do item.
2	Link "*Cadastrar Item"*	A tela de cadastro é apresentada contendo três campos de formulário: nome, endereço e pai. Existe também um botão chamado "*enviar*" que envia os dados para serem cadastrados.
3	Opção "*Clique aqui"* na dica de ajuda para cadastro de itens da tela de cadastro de itens	Uma nova página é apresentada, a qual contém um exemplo de geração e funcionamento dos menus.

Fonte: Autoria própria

Quadro 1 – Interações do usuário com o sistema Manipulador ArcaboMK antes da refatoração (cont.)

4	Botão "*enviar*" da tela de cadastro de item	É apresentada em uma nova tela a mensagem de confirmação "O registro foi inserido com sucesso".
5	Botão "*editar*" da tela principal no item "*Raiz"*	Um formulário é apresentado em uma nova tela com os campos preenchidos com os dados do item raiz. Os campos deste formulário são: o nome e o endereço.
6	Botão "*enviar"* da tela de edição do item "*Raiz"*	É apresentada em uma nova tela a mensagem "O registro foi editado com sucesso".
7	Botão "*editar"* da tela principal nos demais itens	Um formulário é apresentado em uma nova tela com os campos preenchidos com os dados do item escolhido na tela principal. Os campos deste formulário são: o nome e o endereço e pai.
8	Botão "*enviar*" da tela de edição dos demais itens	É apresentada em uma nova tela a mensagem "O registro foi editado com sucesso".
9	Botão "*deletar"* da tela principal no item "*Raiz"*	Uma caixa de confirmação é apresentada ao usuário com a mensagem "Tem certeza que deseja excluir" e as opções "Ok" e "Cancelar".
10	Opção "*Ok"* da caixa de confirmação de exclusão do item "*Raiz"*	É apresentada uma nova tela com a mensagem "ERRO: A raiz não pode ser deletada.".
11	Opção "*Cancelar"* da caixa de confirmação de exclusão	A exclusão é cancelada e o usuário passa a ser direcionado para a página principal.
12	Botão "*deletar"* da tela principal nos demais itens	Uma caixa de confirmação é apresentada ao usuário com a mensagem "Tem certeza que deseja excluir" e as opções "Ok" e "Cancelar".
13	Opção "*Ok"* da caixa de confirmação de exclusão dos demais itens	É apresentada uma nova tela com a mensagem: "O registro foi deletado com sucesso".
14	Link "*Estrutura"*	É apresentada uma nova tela que contém uma tabela que relaciona todos os itens cadastrados com seus respectivos filhos.

Fonte: Autoria própria

A partir dessas informações será possível determinar se a refatoração causou alguma mudança de comportamento no aplicativo *Manipulador ArcaboMK* que será descrita na seção 6.6.

6.2 AVALIAR DIAGRAMA DE CLASSE

O diagrama de classes do aplicativo *Manipulador ArcaboMK* antes da refatoração está ilustrado pela figura 36.

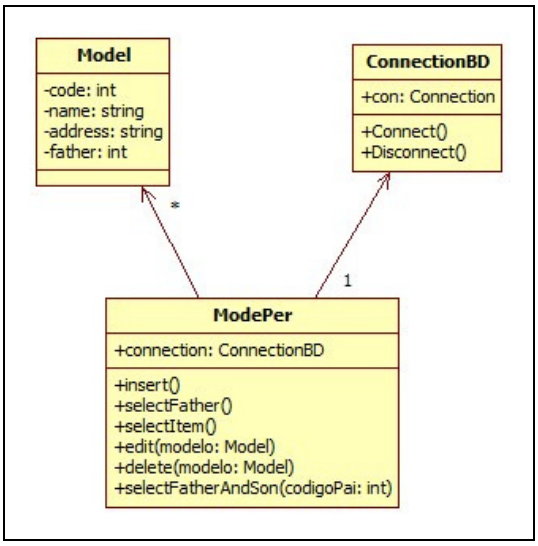

Figura 36 – Diagrama de classe da aplicação antes da refatoração

Fonte: Autoria própria

Por se tratar de um sistema simples, possui apenas três classes, uma classe para conexão com o banco de dados, uma para o modelo de dados do formulário e outra para as operações do banco de dados. No diagrama de classe da Figura 36 considerando a classe *Model* foram suprimidos os métodos *sets* e *gets* para evitar poluição no conteúdo.

Quanto a refatoração, existirão algumas mudanças neste diagrama, visto que o framework *Struts* utiliza classes que funcionam como *Servlets* para onde as requisições do software são apontadas pelo controlador principal.

6.3 IDENTIFICAR PONTO DE REFATORAÇÃO

O principal foco desta refatoração é a mudança de estrutura do aplicativo *Manipulador ArcaboMK* para uma estrutura do framework *Struts* de maneira que possa comportar novas funcionalidades de maneira mais clara e padronizada.

A estrutura de um projeto *Struts* pode ser visualizada na figura 37.

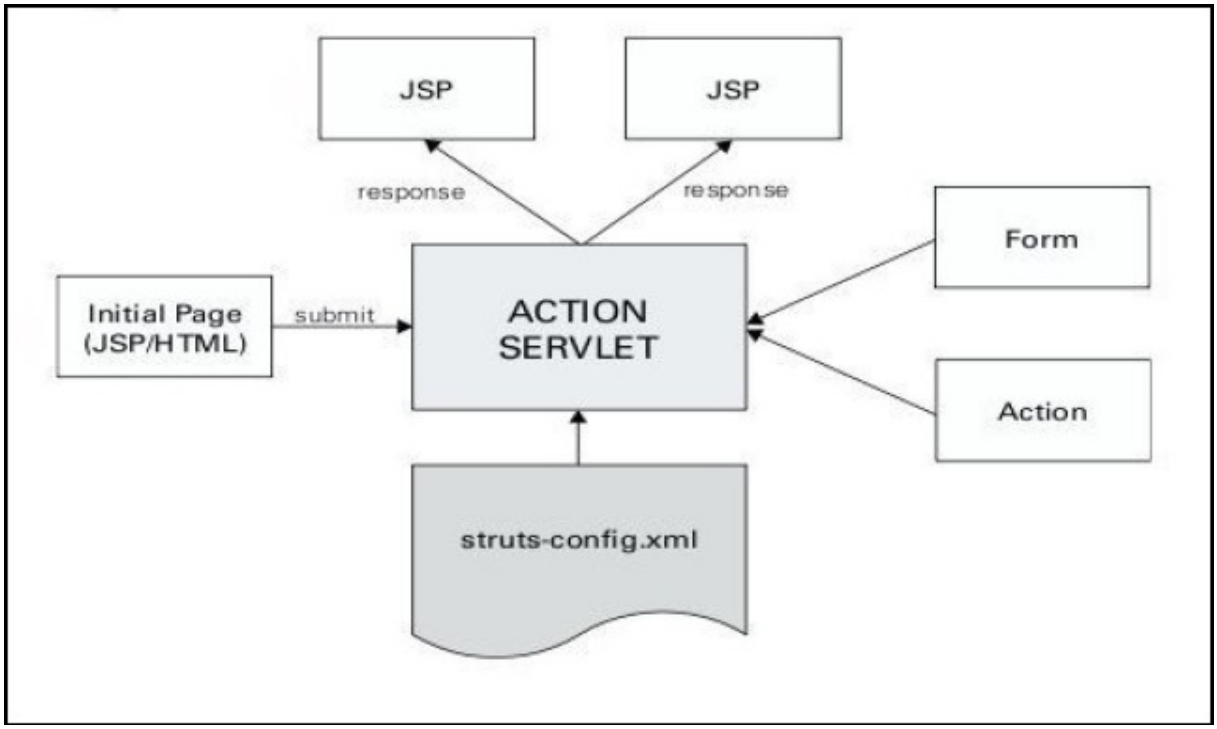

Figura 37 – Estrutura de um projeto Struts
Fonte: HUSTED et al. (2004, p. 15).

Como é possível observar nesta figura, as páginas JSP fazem requisições ao *ActionServlet* do *Struts* e este acessa o *struts-config.xml* onde estão declarados todas as classes *Action* e os *Forms* do projeto.

As classes do tipo *Form* são apenas uma reprodução dos formulários presentes no sistema, assim os componentes do formulário são associados diretamente a um tipo de objeto. As classes do tipo *Action* recuperam o objeto *form*, executam as regras de negócio e direcionam as requisições que estão mapeadas no *struts-config.xml*.

A partir destes conceitos se pode definir que a classe *Model* que possui o modelo presente na base de dados pode ser refatorada para uma classe do tipo *form*.

No aplicativo *Manipulador ArcaboMK*, as requisições eram *links* diretos entre as páginas JSP. Desta forma, essas requisições serão refatoradas para as classes do tipo *Action* mapeadas pelo *struts-config.xml*, sendo adotado por padrão que existirá uma classe do tipo *Action* para cada tela no sistema.

As regras de negócio e as estruturas para montar as páginas estão todas dentro das páginas JSP através da utilização de *scriplets*, logo essas regras de negócio serão transportadas para as classes *Action*.

As estruturas para a montagem das tabelas e população dos formulários serão refatoradas para *taglibs* do framework *Struts.* Isto permite que as páginas funcionem da maneira esperada.

Outra refatoração que foi efetuada se refere a parte das mensagens de confirmação das operações de inserção, edição e exclusão. Essas mensagens eram apresentadas em uma página diferente, pois as atualizações no banco de dados eram feitas em cada página JSP do pacote *control* correspondente a operação.

A ideia é montar uma única página que faça aparecer a mensagem correspondente a operação de acordo com um parâmetro passado as *Actions* destas operações e que essas utilizem o arquivo de mensagens nativo do *Struts*.

Por fim, é importante relembrar de que essas mudanças não alteraram em nada a disposição das páginas nem o funcionamento do aplicativo, mas possui como objetivo a padronização do código.

Para o ordenamento de tarefas na refatoração serão usadas iterações por tela e funcionalidade, de modo a aprender a aplicação do framework mais facilmente a cada etapa. A ordem das iterações é a seguinte:

- Tela Principal
- Cadastro de itens
- Edição de itens e do registro Raiz
- Exclusão de itens
- Tela de estrutura das páginas
- Tela de exemplo de geração e funcionamento de menus

Nesta seção foram descritas todas as refatorações possíveis para o manipulador, lembrando que a cada ponto de refatoração identificado realizam-se as etapas de *Aplicar refatoração* e *Validar código refatorado*.

6.4 APLICAR REFATORAÇÃO

Inicialmente se construiu um projeto web com a estrutura do *Struts*. Neste trabalho optou-se pela versão 1.3.10 porque o grupo de pesquisa a usa para criação dos aplicativos desenvolvidos.

Para se criar a estrutura básica é necessário copiar as bibliotecas do *Struts* para a pasta *lib* que se encontra dentro do diretório: *WEB_INF/ WebContent* do

projeto web dinâmico do Eclipse. Em seguida, configurou-se o arquivo *web.xml* para que ele utilize o arquivo de configurações padrões do *Struts*, o arquivo *struts-config.xml*.

Com isso, foi possível construir uma aplicação simples com *Struts*, criando as *Actions* e os *Forms* mapeando-os com os *forwards* no arquivo de configuração.

A seguir são apresentadas as descrições das iterações do processo de refatoração.

6.4.1 Tela Principal

Na página principal existe a tabela que contém todos os itens cadastrados, para montá-la foi necessário o uso do método da classe *ModelPer* chamado *selectItem()*. Este método seleciona todos os itens e os retorna em um vetor.

Na página antiga os objetos necessários para a execução e utilização deste método foram criados dentro da página JSP, como ilustrado pela figura 9.

A primeira refatoração foi criar uma classe *Action* para essa tela, e transportar a criação dos objetos e execução do método *selectItem()* para esta classe. O código da classe *PrincipalAction* está ilustrado na figura 38.

No framework *Struts*, as classes *Action* funcionam como as *servlets* e estendem a classe *org.apache.struts.action.Action*. Quando chamadas elas executam o código até encontrar um *forward* que está demonstrado pela linha *return mapping.findForward("principal")*.

```java
package Actions;
import java.util.ArrayList;
import java.util.List;


import Model.Model;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;


import Model.ModelPer;

public class PrincipalAction extends Action {
    @Override
    public ActionForward execute(ActionMapping mapping, ActionForm form,
    HttpServletRequest request, HttpServletResponse response)
    throws Exception {

        ModelPer modelo = new ModelPer();
        List<Model> vetorItens = new ArrayList<Model>();

        vetorItens = modelo.selectItem();

        request.getSession().setAttribute("vetorItens", vetorItens);

    return mapping.findForward("principal");
    }
}
```

Figura 38 – Código da classe PrincipalAction

Fonte: Autoria própria

Para que a classe *PrincipalAction* possa ser chamada nas páginas JSP é necessário que a coloque no arquivo *struts-config.xml*. Essa declaração é feita dentro da tag *<action-mappings>* e está ilustrada na figura 39.

```xml
<action-mappings>
    <action name="Principal" path="/Principal" type="Actions.PrincipalAction"
        scope="request">
        <forward name="principal" path="/View/principal.jsp"
            redirect="true" />
    </action>
```

Figura 39 – Declaração da classe PrincipalAction no struts-config.xml

Fonte: Autoria própria

Dentro das propriedades da tag <*action*> se declara o nome que será utilizado para chamar a classe (*path="/Principal"*) e associa-o a classe desejada (*type="Actions.PrincipalAction"*).

Quando se declara uma *Action* dentro do *struts-config.xml*, deve-se também declarar os *forwards* que serão as páginas que podem ser chamadas de dentro da *Action*. Isto é visualizado pelas propriedades da tag <*forward*>, onde se associa a página *principal.jsp* ao nome *"principal"*, o qual foi usado na linha de código *return mapping.findForward("principal")* da figura 38.

Ao iniciar o aplicativo *Manipulador ArcaboMK 2.0*, que será a versão refatorada, a primeira página chamada é a página *index.jsp* que tem o código ilustrado na figura 40.

```
<jsp:forward page="/Principal.do"></jsp:forward>
```

Figura 40 – Código da página índex.jsp
Fonte: Autoria própria

Este trecho de código faz uma requisição para a classe *PrincipalAction* para que os métodos necessários para montar a página principal sejam executados antes de sua chamada.

Para fazer requisições às *Actions* utiliza o nome declarado na propriedade *path* da tag <*action*> do *struts-config.xml* juntamente com o sufixo "*.do"*. Por padrão, o *Struts* em todas as *Actions* são chamadas assim pelas páginas JSP, por causa da configuração do arquivo padrão de projetos web (*web.xml)*, que está ilustrado na figura 41.

Todas as classes do tipo *Action* utilizam por padrão o sufixo de url *".do"*. Após a utilização do método *selectItem()* na classe *PrincipalAction*, é chamada a página *principal.jsp*.

No aplicativo antigo, as estruturas de repetição utilizadas para montar a tabela que contém todos os itens cadastrados que estavam sendo utilizadas por meio de *scriplets*, conforme mostrado na figura 10.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com
  <display-name>HelloWordWithStruts1</display-name>
  <servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
      <param-name>config</param-name>
      <param-value>
          /WEB-INF/struts-config.xml
       </param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

Figura 41 – Código do arquivo web.xml

Fonte: Autoria própria

Como o vetor que contém os dados para essa tabela é montado na classe *PrincipalAction*, é necessário que se salve um atributo de sessão para que possa ser utilizado na página JSP, o que é representado pelo trecho de código *request.getSession().setAttribute("vetorItens", vetorItens);* da figura 38.

No aplicativo refatorado, a estrutura de repetição para construção da tabela está ilustrada na figura 42. Para a criação da tabela foi utilizada uma *taglib* lógica do *Struts* representada pelo código *<logic:iterate>*. Esta estrutura executa o laço até que o vetor declarado na propriedade *name* tenha sido percorrido em sua totalidade.

Na propriedade *id* é declarado um nome para que se possa utilizar o objeto presente dentro de cada posição do vetor, assim pode-se ler qualquer propriedade do objeto, como ilustrado na figura 42 pelo trecho de código *<bean:write name="vetorItensId" property="code"/>*, por exemplo.

```
<logic:iterate name="vetorItens" id="vetorItensId">

    <tr>
        <td align="center"><bean:write name="vetorItensId"
            property="code" /></td>

        <td align="center"><bean:write name="vetorItensId"
            property="name" /></td>

        <td align="center"><bean:write name="vetorItensId"
            property="address" /></td>

        <td align="center"><bean:write name="vetorItensId"
            property="father" /></td>


        <td align="center"><html:form action="/Edit">
            <input type="hidden" name="codigo"
                value="<bean:write name="vetorItensId" property="code"/>" />
            <html:submit>Editar</html:submit>
        </html:form></td>
        <td align="center"><html:form action="/Delete">
            <input type="hidden" name="codigo"
                value="<bean:write name="vetorItensId" property="code"/>" />
            <html:submit>Deletar</html:submit>
        </html:form></td>
    </tr>
</logic:iterate>
```

Figura 42 – Código para montagem da tabela da página principal.jsp que contém os itens cadastrados

Fonte: Autoria própria

No menu principal do aplicativo, existem também *links* para diversas funcionalidades, sendo o primeiro deles para a página principal. A página responsável por estes *links* no aplicativo antigo era a página *menu_horizontal.jsp* e está representada pela figura 43.

```
<a name="menu_principal"></a>
<div id = "menu_horizontal" title="Menu Horizontal. Contém as ligações: Página inici

     <div id="links">
          <a href="./index.jsp?link=View/principal#titulo">Itens Cadastrados</a>
          <a href="./index.jsp?link=View/insertView#titulo">Cadastrar Item</a>
          <a href="./index.jsp?link=View/estrutura#titulo">Estrutura</a>
          <!--<li><a href="Conteudo.jsp?var=21">Mapa do Sítio</a></li>-->
     </div>

</div>
```

Figura 43 – Código do menu do aplicativo Manipulador ArcaboMK antes da refatoração

Fonte: Autoria própria

A página para os menus foi renomeada para *menu*.jsp e os *links* para as páginas JSP foram substituídos por chamadas para as classes *Action* de cada tela, como ilustra a figura 44.

```
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html"%>
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean"%>
<div id = "menu_horizontal" title="Menu Horizontal. Contém as ligações: Página inic

     <div id="links">
          <html:link action="/Principal.do">Itens Cadastrados</html:link>
         <html:link action="/Insert.do">Cadastrar Item</html:link>

          <html:link action="/Estrutura.do">Estrutura</html:link>
          <bean:define id="cod" value="1"></bean:define>
          <html:link action="/MenuExample.do" paramId="codigo" paramName="cod"
          property="${cod}">Exemplo de geração de menus</html:link>
     </div>

</div>
```

Figura 44 – Código do menu do aplicativo Manipulador ArcaboMK após a refatoração

Fonte: Autoria própria

Como os links são *Actions*, utiliza-se as *taglibs* de HTML do *Struts*. O *link* para a página principal é representado pela linha *<html:link action="/Principal.do">Itens Cadastrados</html:link>*.

6.4.2 Cadastro de Itens

O nome da *Action* criada para esta página é *InsertAction* e a única chamada para ela se encontra no menu apresentado na figura 44. O código da classe *InsertAction* está ilustrado pela figura 45. Para evitar o uso de duas *Actions* para essa funcionalidade, uma para o direcionamento da página e outra para o envio do formulário, foi adotado o uso de um parâmetro *hidden* no formulário HTML da página de cadastro. Esse parâmetro só é enviado quando o formulário é submetido.

Assim sendo, quando se chama a classe *InsertAction* é verificado se o valor do parâmetro é nulo. Caso afirmativo, é feito um *forward* para a página *insert.jsp*. Do contrário, não é feita a execução do método para inserir no banco de dados e o *forward* para a página *sucess.jsp*.

```
package Actions;
import java.util.ArrayList;
import java.util.List;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import Model.Model;
import Model.ModelPer;

public class InsertAction extends Action {
    @Override
    public ActionForward execute(ActionMapping mapping, ActionForm form,
    HttpServletRequest request, HttpServletResponse response)
    throws Exception {

          String submited = request.getParameter("submited");

       if (submited == null || !submited.equals("true")){
       ModelPer modelo = new ModelPer();
       List<Model> vetorCombo = new ArrayList<Model>();

       vetorCombo = modelo.selectFather();

       request.getSession().setAttribute("vetorCombo", vetorCombo);

       return mapping.findForward("insert");
       }
       else{
       int pai= ((Model) form).getFather();
       String nome = ((Model) form).getName();
       String endereco = ((Model) form).getAddress();

       Model item = new Model();
       ModelPer modelo = new ModelPer();

       item.setName(nome);
       item.setAddress(endereco);
       item.setFather(pai);
       modelo.insert(item);

       request.getSession().setAttribute("operacao", "insert");
       return mapping.findForward("sucess");
       }
  }
```

Figura 45 – Código da classe InsertAction

Fonte: Autoria própria

No antigo aplicativo, o método que fazia a busca no banco para preencher o botão de seleção do campo *Pai* era executado na página JSP. Na refatoração esta

parte de código foi transportada para a classe *InsertAction*, de maneira similar ao processo utilizado na seção anterior para a página principal.

A declaração da classe *InsertAction* no arquivo de configuração do *Struts* está ilustrada na figura 46.

```
<action name="ModelForm" path="/Insert" type="Actions.InsertAction"
    scope="request">
    <forward name="insert" path="/View/insert.jsp" redirect="true" />
    <forward name="sucess" path="/View/sucess.jsp" redirect="true" />
</action>
```

Figura 46 – Código do arquivo struts-config.xml da classe InsertAction
Fonte: Autoria própria

Como esta *Action* também é uma requisição de um formulário, a propriedade *name* receberá o nome de um *Form* que esteja declarado no arquivo de configuração do *Struts*. O mapeamento dos *Forms* no *struts-config.xml* está ilustrado pela figura 47.

```
<form-beans>

    <form-bean name="ModelForm" type="Model.Model" />
    <form-bean name="ModelForm2" type="Model.Model" />
    <form-bean name="ModelForm3" type="Model.Model" />
</form-beans>
```

Figura 47 – Código da declaração dos Forms no arquivo struts-config.xml
Fonte: Autoria própria

Na propriedade *type* dos *Forms*, pode-se verificar que a classe declarada é a mesma utilizada como modelo de dados no antigo aplicativo, a saber, a classe *Model*. A refatoração nessa classe se deve pelo fato de que ela agora estende a classe *org.apache.struts.action.ActionForm;* assim como todo modelo de formulário no *Struts*.

Após a utilização do método *selectFather()* para listagem do botão de seleção do formulário na classe *InsertAction* é feito um *forward* para a página *insert.jsp*. O código responsável pelo formulário da antiga página de inserção está ilustrado na figura 48.

```
<form id ="formulario" method=post action="index.jsp?link=Control/insert">
<div class="row">
    <label for="txt_nome" id="nome-ariaLabel">Nome</label>
    <input id="input" name="txtNome" type="text" aria-labelledby="nome-ariaLabel" />
</div>
<div class="row">
    <label for="txt_endereco" id="endereco-ariaLabel">Endereco</label>
    <input id="input" name="txtEndereco" type="text" aria-labelledby="endereco-ariaLabel"
</div>
<div class="row">
    <label for="sel_pai" id="pai-ariaLabel">Pai</label>
        <select id="sel_pai" name="txtPai" aria-labelledby="pai-ariaLabel">
            <%
            for(int i=0; i < modelo.size(); i++){
              Model m = new Model();
              m = (Model) modelo.elementAt(i);
               String p1,p2,opt=new String();
              if(m.getName().length()>=42){
                  p1=m.getName().substring(0, 21);
                  p2=m.getName().substring(m.getName().length()-21, m.getName().length());
                  opt=p1+" ..... "+p2;
                  }else{
                  opt=m.getName();
                  }
            %>
                <option value=<% out.println(m.getCode()); %> >
                <% out.println(opt);  }  %></option>
    </select>
</div>
<div class="row">
<input type="submit" value="Enviar" />
</div>
</form>
```

Figura 48 – Código da página de inserção no Aplicativo Manipulador ArcaboMK antes da refatoração

Fonte: Autoria própria

Nesta figura se pode visualizar que a propriedade *action* do formulário direciona para outra página JSP. Na página refatorada, a *action* do formulário é a classe *InsertAction* e o formulário é aplicado utilizando *taglibs* do *Struts*, como está ilustrado na figura 49. Utilizando os *taglibs* de HTML do *Struts*, cada campo do formulário é facilmente associado a um atributo do objeto declarado como *ActionForm* da *Action*. Para chamar a *Action* como ação do formulário não é necessário utilizar o sufixo *".do"*.

```
<html:form action="/Insert">
<div class="row">
    <label for="txt_nome" id="nome-ariaLabel">Nome</label><br/>
    <html:text property="name" size="54" />

</div>
<div class="row">
    <label for="txt_endereco" id="endereco-ariaLabel">Endereco</label>
    <html:text property="address" size="54" />

</div>
<div class="row">
    <label for="sel_pai" id="pai-ariaLabel">Pai</label>
        <html:select property="father">
            <logic:iterate name="vetorCombo" id="vetorComboId">
            <option value="<bean:write name="vetorComboId" property="code"/>">
                <bean:write name="vetorComboId" property="code"/> - <bean:write
                name="vetorComboId" property="name"/>
             </option>
             </logic:iterate>
    </html:select>
</div>
<div class="row">
<input type="hidden" name="submited" value="true"/>
<html:submit>Enviar</html:submit>

</div>
</html:form>
```

Figura 49 – Código da página de inserção no Aplicativo Manipulador ArcaboMK após a refatoração

Fonte: Autoria própria

Nota-se que no código responsável pela listagem dos itens no campo de seleção *Pai* possui apenas a listagem de um vetor, conforme mostrado na figura 42, enquanto na página de inserção, ilustrada na figura 48, havia uma lógica que diminuía a *String* apresentada no campo de seleção. A aplicação dessa lógica se fez necessária, pois existem nomes de itens cadastrados que fazem o campo de seleção “estourar” as dimensões da página, conforme mostrado na figura 50.

Por usar uma lógica complexa com a separação de *Strings* como apresentado na figura 48, ao invés do uso de *taglibs* na formulação deste algoritmo, optou-se por transferir o algoritmo que diminui as *Strings* dos nomes para o método *selectFather()* da classe *ModelPer*, visto que este método só é utilizado quando é necessário popular a caixa de seleção do campo *Pai* em um formulário.

Figura 50 – Código da página de inserção no Aplicativo Manipulador ArcaboMK após a refatoração

Fonte: Autoria própria

A figura 51 ilustra o código do método *selectFather()* após a refatoração.

```
public List<Model> selectFather() {
    String sql;
    connection = new ConnectionBD();
    sql = "select codigo,nome from item order by codigo";
    PreparedStatement pstm = null;
    ResultSet rs;
    String name = new String();
    List<Model> modelo = new ArrayList<Model>();
    try {
        pstm = connection.connect().prepareStatement(sql);
        rs = pstm.executeQuery();
        while (rs.next()) {
            // criar o objeto
            Model model = new Model();
            model.setCode(rs.getInt("codigo"));

            name = rs.getString("nome");

            String p1, p2, opt = new String();
            if (name.length() >= 36) {
                p1 = name.substring(0, 18);
                p2 = name.substring(name.length() - 18, name.length());
                opt = p1 + " ..... " + p2;
            } else {
                opt = name;
            }
            model.setName(opt);
            modelo.add(model);
        }
        connection.disconnect();
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return modelo;
}
```

Figura 51 – Código do método selectFather() da classe ModelPer

Fonte: Autoria própria

Outra pequena refatoração implantada foi a opção de listar o código do item ao lado do nome na caixa de seleção do campo *Pai*, pois um erro pode ocorrer ao cadastrar dois itens com o mesmo nome e então não saber qual selecionar. A figura 52 ilustra a visualização da página de inserção refatorada.

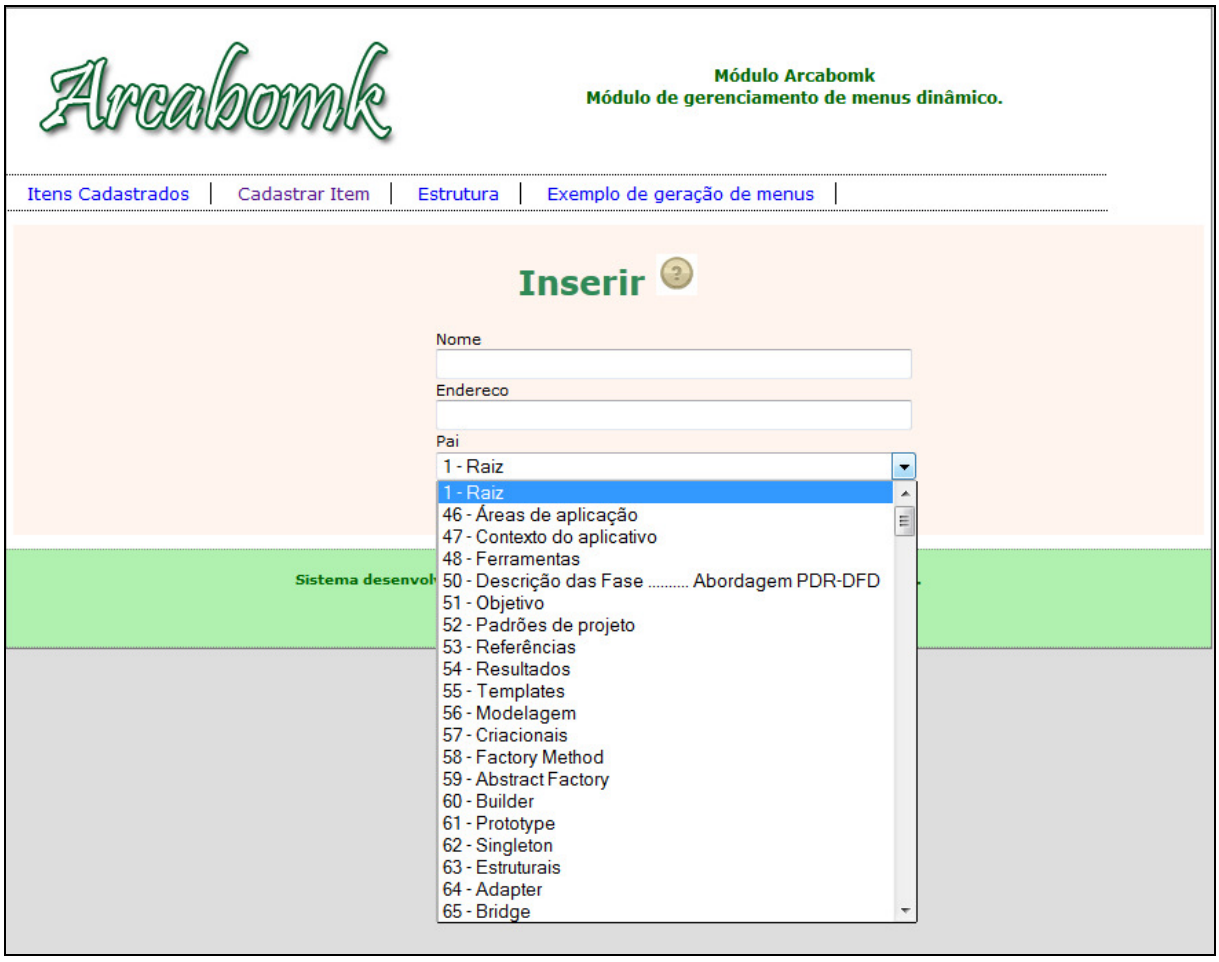

Figura 52 – Visualização da página de inserção refatorada

Fonte: Autoria própria

Ao clicar no botão enviar, a submissão fará nova requisição para a classe *InsertAction*. Na página de inserção o atributo *submited* é instanciado pelo trecho de código <input type=*"hidden"* name=*"submited"* value=*"true"*/>.

Voltando a classe *InsertAction*, ilustrado na figura 45, como o parâmetro *submited* possui o valor *true* será executado o trecho de código mostrado na figura 53.

```
        else{
        int pai= ((Model) form).getFather();
        String nome = ((Model) form).getName();
        String endereco = ((Model) form).getAddress();

        Model item = new Model();
        ModelPer modelo = new ModelPer();

        item.setName(nome);
        item.setAddress(endereco);
        item.setFather(pai);
        modelo.insert(item);

        request.getSession().setAttribute("operacao", "insert");
        return mapping.findForward("sucess");
        }
    }
```

Figura 53 – Trecho de código da classe InsertAction

Fonte: Autoria própria

Os valores do formulário são recuperados e então o objeto da classe *Model* é enviado para inserção no banco de dados por meio do método *insert(Model modelo)* da classe *ModelPer*.

Após a inserção é atribuído em um atributo de sessão a variável *operação* como o valor *"insert"* e é feito um *forward* para a página *sucess.jsp*.

O código da página da página *sucess.jsp* está ilustrado na figura 54.

```
<div align="center" id="corpo">
<%@include file = "corpo.jsp"%>
<div id="texto">
<logic:equal name="operacao" value="insert">
<h1><bean:message key="insert.sucess" /></h1>
</logic:equal>

<logic:equal name="operacao" value="edit">
<h1><bean:message key="edit.sucess" /></h1>
</logic:equal>

<logic:equal name="operacao" value="delete">
<h1><bean:write name="msg" /></h1>
</logic:equal>

</div>
```

Figura 54 – Trecho de código da página sucess.jsp

Fonte: Autoria própria

Desta forma, são utilizadas *taglibs* de lógica que verificam o valor do atributo de sessão atribuído após a submissão de um formulário e então é exibida a mensagem de confirmação correspondente a operação realizada.

Para escrever as mensagens foi utilizado o arquivo de mensagens do *Struts* chamado *MessageResources.properties*. Este arquivo deve ser criado dentro do diretório de classes *src* do projeto e para utilizá-lo é necessário adicionar o código ilustrado na figura 55 ao arquivo de configuração *struts-config.xml*.

```
<message-resources parameter="MessageResources" />
```

Figura 55 – Configuração do arquivo de mensagens no struts-config.xml
Fonte: Autoria própria

O código utilizado no arquivo de mensagens *MessageResources.properties* está ilustrado na figura 56.

```
insert.sucess = O registro foi inserido com sucesso

edit.sucess = O registro foi editado com sucesso
```

Figura 56 – Código do arquivo de mensagens MessageResources.properties
Fonte: Autoria própria

Ao utilizar este arquivo de mensagens, pode-se centralizar em um único local todas as mensagens do sistema, o que facilita durante a manutenção.

Para ler as mensagens foi utilizada a tag *<bean:message key="insert.sucess" />* conforme ilustrado na figura 54.

6.4.3 Edição de Itens e do registro Raiz

No antigo aplicativo *Manipulador ArcaboMK*, a edição era chamada por meio de uma função em *javascript* em um botão conforme mostrado nas figuras 10 e 16. O uso de *javascript* não é indicado em funções vitais de qualquer sistema, visto que se o navegador estiver com o *javascript* desligado o sistema não funcionará.

Ao clicar no botão editar em algum item que não seja a Raiz um parâmetro é enviado com o *link* conforme mostrado na figura 16, este parâmetro é recuperado na

página do formulário de edição para que o item seja recuperado e os campos do formulário sejam preenchidos corretamente, conforme já exibido na figura 17.

No aplicativo *Manipulador ArcaboMK 2.0*, a chamada da função de edição acontece conforme já mostrado na figura 42 e destacado na figura 57.

```
<td align="center"><html:form action="/Edit">
    <input type="hidden" name="codigo"
        value="<bean:write name="vetorItensId" property="code"/>" />

    <html:submit>Editar</html:submit>


</html:form></td>
```

Figura 57 – Código da chamada da função de edição no software Manipulador ArcaboMK 2.0
Fonte: Autoria própria

Para a chamada foi utilizado um formulário que possui como *Action* a classe *EditAction* e para enviar o parâmetro foi utilizado um campo do tipo *hidden* que receberá o valor do código do número selecionado.

A figura 58 ilustra a configuração do arquivo *struts-config.xml* da classe *EditAction*.

```
<action name="ModelForm2" path="/Edit" type="Actions.EditAction"
    scope="request">
    <forward name="edit" path="/View/edit.jsp" redirect="true" />
    <forward name="sucess" path="/View/sucess.jsp" redirect="true" />
</action>
```

Figura 58 – Configuração do arquivo struts-config.xml para a classe EditAction
Fonte: Autoria própria

Como mostrado na figura 46, a configuração da *Action* da classe *EditAction* é similar a utilizada na classe *InsertAction*.

Na página de edição de itens do aplicativo antigo foram feitas criações dos objetos da mesma maneira utilizada na página de inserção antiga, porém além disso ainda foram recuperados na página JSP os valores utilizados para o preenchimento dos campos do formulário de edição conforme mostrado na figura 17.

A classe *EditAction* utiliza uma lógica parecida com a usada na classe *InsertAction* e seu código está ilustrado na figura 59.

```
public class EditAction extends Action {
    @Override
    public ActionForward execute(ActionMapping mapping, ActionForm form,
            HttpServletRequest request, HttpServletResponse response)
            throws Exception {

        String submited = request.getParameter("submited2");

        if (submited == null || !submited.equals("true")) {
            int codigo = Integer.parseInt(request.getParameter("codigo"));
            ModelPer modelo = new ModelPer();

            List<Model> vetorCombo = new ArrayList<Model>();
            List<Model> vetor = new ArrayList<Model>();

            if (codigo != 1) {
                vetor = modelo.selectFather();

                for (int i = 0; i < vetor.size(); i++) {

                    Model item = new Model();
                    item = (Model) vetor.get(i);

                    if ((item.getCode() != codigo)
                            && (item.getFather() != codigo)) {
                        vetorCombo.add(item);

                    }
                }
            }
            Model itemForm = new Model();
            Model itemPai = new Model();
            itemForm = modelo.selectItemEdicao(codigo);
            itemPai = modelo.selectItemEdicao(itemForm.getFather());

            request.getSession().setAttribute("vetorCombo", vetorCombo);
            request.getSession().setAttribute("codigoForm", itemForm.getCode());
            request.getSession().setAttribute("nomeForm", itemForm.getName());
            request.getSession().setAttribute("enderecoForm",
                    itemForm.getAddress());
            request.getSession().setAttribute("paiForm", itemForm.getFather());
            request.getSession().setAttribute("paiNomeForm", itemPai.getName());
            return mapping.findForward("edit");
        } else {

            int codigo = ((Model) form).getCode();

            int pai = ((Model) form).getFather();
            String nome = ((Model) form).getName();
            String endereco = ((Model) form).getAddress();

            Model item = new Model();
            ModelPer modelo = new ModelPer();
            item.setCode(codigo);
            item.setName(nome);
            item.setAddress(endereco);
            item.setFather(pai);
            modelo.edit(item);

            request.getSession().setAttribute("operacao", "edit");
            return mapping.findForward("sucess");
        }

    }
```

Figura 59 – Código da classe EditAction

Fonte: Autoria própria

Quando chamada a primeira vez, a classe *EditAction* verifica se o item não é a Raiz. Caso não seja, o método *selectFather()* é executado para que o campo de seleção *Pai* do formulário seja preenchido. Isto é verificado, pois o índice Raiz é o primeiro item cadastrado e é a origem de todos os outros.

Importante ressaltar que são adicionados ao vetor do combo todos os itens, exceto o que foi selecionado para a edição. No aplicativo antigo, essas lógicas foram aplicadas em um algoritmo na página JSP conforme ilustrada pela figura 18.

Para recuperar os valores do item selecionado para a edição, foi criado um novo método na classe *ModelPer* chamado *selectItemEdicao(int código)* que está exibido na figura 60.

```
public Model selectItemEdicao(int codigo) {
    String sql;
    connection = new ConnectionBD();
    sql = "select * from item where codigo=" + codigo;
    PreparedStatement pstm = null;
    ResultSet rs;
    Model model = new Model();
    try {
        pstm = connection.connect().prepareStatement(sql);
        rs = pstm.executeQuery();
        while (rs.next()) {
            // criar o objeto

            model.setCode(rs.getInt("codigo"));
            model.setName(rs.getString("nome"));
            model.setAddress(rs.getString("endereco"));
            model.setFather(rs.getInt("pai"));

        }
        connection.disconnect();
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return model;
}
```

Figura 60 – Código do método selectItemEdicao daclasse ModelPer

Fonte: Autoria própria

Este método seleciona o item de acordo com o parâmetro *código* enviado. Além de recuperar os dados do item selecionado para edição, também é necessário recuperar o nome do item pai para que ele possa ser preenchido no campo *Pai* do formulário, o que é feito na linha de código *itemPai = modelo.selectItemEdicao(itemForm.getFather())*.

Após a recuperação de todos os dados, são criados os atributos de sessão usados para popular os campos do formulário de edição conforme mostrado na figura 59 e pode ser visualizado com maior precisão na figura 61.

```
request.getSession().setAttribute("vetorCombo", vetorCombo);
request.getSession().setAttribute("codigoForm", itemForm.getCode());
request.getSession().setAttribute("nomeForm", itemForm.getName());
request.getSession().setAttribute("enderecoForm",
        itemForm.getAddress());
request.getSession().setAttribute("paiForm", itemForm.getFather());
request.getSession().setAttribute("paiNomeForm", itemPai.getName());
```

Figura 61 – Código de criação e atribuição dos atributos de sessão

Fonte: Autoria própria

Após a atribuição dos atributos, é feito um *forward* para a página *edit.jsp*. O código do formulário da página *edit.jsp* está ilustrada na figura 62.

```
<html:form action="/Edit">

<html:hidden property="code"
    value="${codigoForm}" />
    <div class="row">
     <label for="txt_nome" id="nome-ariaLabel">Nome</label><br/>
<html:text property="name" size="54"  value="${nomeForm}" />
</div>

<div class="row"><label for="txt_endereco"
    id="endereco-ariaLabel">Endereco</label> <html:text property="address"
    size="54" value="${enderecoForm}" /></div>
<logic:notEqual name="codigoForm" value="1">
    <div class="row"><label for="sel_pai" id="pai-ariaLabel">Pai</label>

    <html:select property="father">

        <html:option value="${paiForm}">
            <bean:write name="paiForm" /> - <bean:write name="paiNomeForm" />
        </html:option>
        <logic:iterate name="vetorCombo" id="vetorComboId">
            <option value="<bean:write name="vetorComboId" property="code"/>">
            <bean:write name="vetorComboId" property="code" /> - <bean:write
                name="vetorComboId" property="name" /></option>
        </logic:iterate>

    </html:select></div>
</logic:notEqual>

<div class="row"><input type="hidden" name="submited2"
    value="true" /> <html:submit>Enviar</html:submit></div>
</html:form>
```

Figura 62 – Código do formulário da página edit.jsp

Fonte: Autoria própria

Os valores do item selecionado são recuperados e atribuídos as suas variáveis por meio do uso de *"${ nome do atributo de sessão }*. Também é possível notar que existe uma condição lógica para que o campo de seleção *Pai* seja mostrado na tela. Isto é feito caso o item Raiz seja selecionado para edição, então o campo *Pai* não será mostrado pois este é o índice de origem dos outros.

Este formulário também tem um campo do tipo *hidden* que recebe o valor do código do item selecionado para edição.

No aplicativo antigo, caso o item selecionado para edição fosse o item Raiz, era chamada uma página que foi criada com uma cópia do formulário de edição sem o campo *nome* e o campo de seleção *Pai*, como já mostrado na figura 16.

Após a submissão do formulário o processo é similar ao utilizado na classe *InsertAction*, onde são recuperados os atributos do *Form* e então é executado o método *edit(Model modelo)* da classe *ModelPer* que faz o *update* no banco de dados. Em seguida é feito um *forward* para a página *sucess.jsp* onde é mostrada a mensagem de confirmação.

6.4.4 Exclusão de itens

Antes da refatoração, a exclusão foi confirmada apenas por uma caixa de confirmação feita em *javascript*, conforme apresentado na figura 23. Caso a escolha seja positiva na caixa de confirmação, no aplicativo anterior era feita uma requisição para a página *delete.jsp* como mostrado na figura 22. Então, na página eram criados os objetos necessários para a exclusão conforme o mostrado na figura 24.

Após a refatoração a chamada para a exclusão de itens é feita conforme já ilustrado na figura 42 e realçado na figura 63.

```
<td align="center"><html:form action="/Delete">
    <input type="hidden" name="codigo"
        value="<bean:write name="vetorItensId" property="code"/>" />

    <html:submit>Deletar</html:submit>


</html:form></td>
```

Figura 63 – Chamada da função de exclusão do aplicativo Manipulador ArcaboMK 2.0

Fonte: Autoria própria

A propriedade *action* do formulário da figura anterior aponta para a classe *DeleteAction*. A configuração do arquivo *struts-config.xml* para a classe *DeleteAction* está ilustrada na figura 64.

```
<action name="ModelForm3" path="/Delete" type="Actions.DeleteAction"
    cancellable="true" scope="request">
    <forward name="delete" path="/View/delete.jsp" redirect="true" />
    <forward name="sucess" path="/View/sucess.jsp" redirect="true" />
    <forward name="principal" path="/View/principal.jsp"
        redirect="true" />
</action>
```

Figura 64 – Declaração da classe DeleteAction no arquivo de configuração do Struts

Fonte: Autoria própria

O código da classe *DeleteAction* está ilustrado na figura 65.

```
public class DeleteAction extends Action {
    @Override
    public ActionForward execute(ActionMapping mapping, ActionForm form,
            HttpServletRequest request, HttpServletResponse response)
            throws Exception {

        String submited = request.getParameter("submited3");

        if (submited == null || !submited.equals("true")) {
            int codigo = Integer.parseInt(request.getParameter("codigo"));
            ModelPer modelo = new ModelPer();
            Model itemForm = new Model();
            itemForm = modelo.selectItemEdicao(codigo);

            request.getSession().setAttribute("itemForm", itemForm);
            return mapping.findForward("delete");
        } else {
            if (isCancelled(request)) {
                return mapping.findForward("principal");
            } else {
                int codigo = ((Model) form).getCode();
                Model item = new Model();
                ModelPer modelo = new ModelPer();
                item.setCode(codigo);
                String msg = modelo.delete(item);
                request.getSession().setAttribute("msg", msg);
                request.getSession().setAttribute("operacao", "delete");
                return mapping.findForward("sucess");

            }

        }
    }
}
```

Figura 65 – Código da classe DeleteAction

Fonte: Autoria própria

Assim como o processo utilizado na edição de itens, o código do item escolhido para a exclusão é recuperado a primeira vez quando a classe *DeleteAction* é chamada.

Em seguida é utilizado o método *selectItemEdicao(int codigo)* para recuperar os dados do item selecionado para a exclusão e então ele é atribuído a uma variável de sessão chamada *itemForm*. É feita então um *forward* para a página *delete.jsp*.

Diferentemente da versão antiga do aplicativo, a decisão da exclusão foi empregada em um formulário em uma página para que isso não dependesse mais do uso de *javascript*. A tela de exclusão do novo aplicativo está ilustrada na figura 66.

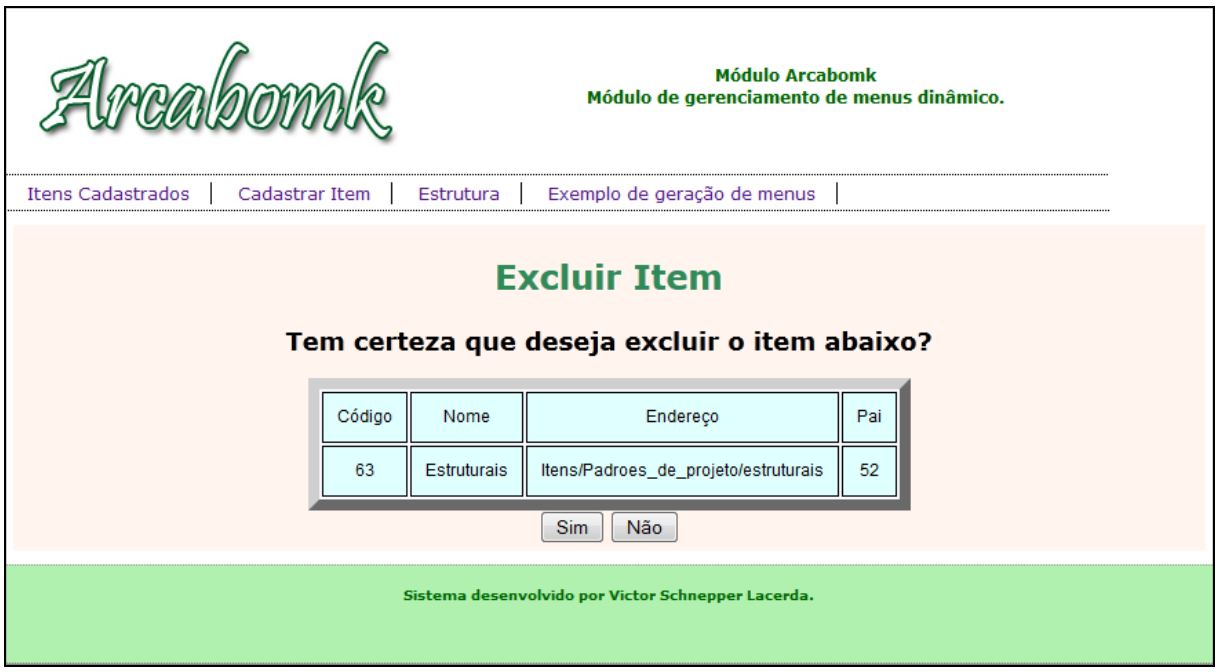

Figura 66 – Visualização da página de exclusão

Fonte: Autoria própria

O código do formulário da página *delete.jsp* está ilustrado na figura 67. Nota-se que o atributo de sessão *itemForm* é recuperado e atribuído a um *taglib* HTML do tipo *hidden*. O atributo *itemForm* também é utilizado para que o item que será excluído seja visualizado.

No formulário da página de confirmação de exclusão têm-se dois botões, as opções *"Sim"* e *"Não"*, sendo que a primeira utiliza a *tag <html:submit>* e a outra a *tag <html:cancel>*. Os dois botões passam a requisição para o *Action* do formulário que neste caso é a classe *DeleteAction*.

```
<html:form action="/Delete">
    <div class="row">
    <bean:define name="itemForm" id="codigoForm" property="code"/>
    <html:hidden property="code" value="${codigoForm}" />
    <h2 align="center">Tem certeza que deseja excluir o item abaixo?</h2>

    <table id="tabela" align="center">
    <tr>
        <td id="td" align="center">Código</td>

        <td align="center">Nome</td>

        <td align="center">Endereço</td>

        <td align="center">Pai</td>
    </tr>
    <tr>
        <td id="td" align="center"><bean:write name="itemForm" property="code" /></td>

        <td align="center"><bean:write name="itemForm" property="name" /></td>

        <td align="center"><bean:write name="itemForm" property="address" /></td>

        <td align="center"><bean:write name="itemForm" property="father" /></td>
    </tr>
    </table>
    <div class="row">
     <input type="hidden" name="submited3"  value="true" />
     <html:submit>Sim</html:submit>
     <html:cancel>Não</html:cancel>
     </div>
     </div>
</html:form></div>
```

Figura 67 – Parte do código da página delete.jsp

Fonte: Autoria própria

Para a utilização da tag *<html:cancel>* é necessário que seja adicionada a propriedade *cancellable=”true”* no arquivo de configuração do *Struts* conforme exibido na figura 64.

A classe *DeleteAction* após a submissão do formulário, por meio da linha de código ***if*** *(isCancelled(request))*, verifica se o usuário escolheu a opção “*Não”* do formulário de exclusão. Se a condição for verdadeira é feito um *forward* para a página *principal.jsp*. Caso contrário, é recuperado o código do item selecionado e são criados os objetos necessários para a manipulação do banco de dados para a execução do método *delete(Model modelo)* da classe *ModelPer* como mostrado na figura 66.

A página *sucess.jsp* é chamada para a apresentação da mensagem de confirmação. Como já mostrado na figura 28, a tabela do banco de dados possui

uma *trigger* que é disparada a cada exclusão, que verifica se o item excluído é o índice Raiz e retorna uma mensagem de erro.

Por este motivo, o arquivo de mensagens não foi utilizado para a exclusão do arquivo. A mensagem de confirmação é atribuída a um atributo de sessão e então recuperada na página *sucess.jsp* como mostrado na figura 54.

6.4.5 Tela de estrutura das páginas

A página de estrutura apresenta uma tabela que relaciona cada item com seus filhos, como ilustra a figura 68.

Figura 68 – Parte da visualização da página de estrutura das páginas cadastradas
Fonte: Autoria própria

No aplicativo antes da refatoração os objetos foram instanciados e a lógica também foi aplicada na página *estrutura.jsp*, como ilustra a figura 69.

```
<%
ModelPer conexao = new ModelPer();
Vector modelo = new Vector();
modelo = conexao.selectItem();
%>

<html>
    <head>

       <title>Manipulador Arcabomk</title>
    </head>

    <body >
       <h1 align="center">Estrutura dos itens</h1>
       <table id="tabela" align="center">
       <tr>
            <td id="td" align="center">
    Nome do Item
    </td>

     <td align="center">
   Filhos
    </td>
       </tr>
        <%
            for(int i=0; i < modelo.size(); i++){
              Model m = new Model();

              m = (Model) modelo.elementAt(i);

            %>
    <tr>
        <td align="center">
            <% out.println(m.getName()); %>
        </td>

        <td>
             <%
            for(int j=0; j < modelo.size(); j++){
                Model m2 = new Model();
                m2 = (Model) modelo.elementAt(j);

                if(m2.getFather()==m.getCode()){
                    out.println(m2.getCode());
            %> - <% out.println(m2.getName());
                        %> <br> <%
              }
              }
             %>
        </td>
    </tr>
    <% } %>
       </table>

    </body>
</html>
```

Figura 69 – Código da página estrutura.jsp no Aplicativo Manipulador ArcaboMK antes da refatoração

Fonte: Autoria própria

Assim como nas outras páginas refatoradas, foi criada uma classe *Action* para esta página denominada de *EstruturaAction*. A criação dos objetos e execução do método *selectItem()* foram então transportados para a classe como ilustrado na figura 70.

```
public class EstruturaAction extends Action {
    @Override
    public ActionForward execute(ActionMapping mapping, ActionForm form,
    HttpServletRequest request, HttpServletResponse response)
    throws Exception {

        ModelPer modelo = new ModelPer();
        List<Model> vetorItens = new ArrayList<Model>();
        vetorItens = modelo.selectItem();

        request.getSession().setAttribute("vetorItens", vetorItens);

    return mapping.findForward("estrutura");

    }
}
```

Figura 70 – Código da classe EstruturaAction

Fonte: Autoria própria

A configuração do arquivo *struts-config.xml* para a classe *EstruturaAction* está ilustrada na figura 71.

```
<action name="Estrutura" path="/Estrutura" type="Actions.EstruturaAction"
    scope="request">
    <forward name="estrutura" path="/View/estrutura.jsp"
        redirect="true" />
</action>
```

Figura 71 – Código de configuração do struts-config.xml para a classe EstruturaAction

Fonte: Autoria própria

Após a execução do método *selectItem()*, é feito um *forward* para a página *estrutura.jsp*. A lógica apresentada na figura 69 foi reformulada com o uso de *taglibs* e seu código está ilustrado na figura 72.

```
<div id="texto">
<h1 align="center">Estrutura dos itens</h1>
<table id="tabela" align="center">
    <tr>
        <td id="td" align="center">Nome do Item</td>

        <td align="center">Filhos</td>
    </tr>
<logic:iterate name="vetorItens" id="vetorItensId">
<bean:define id="codigo" name="vetorItensId" property="code"></bean:define>

    <tr>
        <td align="center">
        <bean:write name="vetorItensId"
                property="name" />
        </td>

        <td>
        <logic:iterate name="vetorItens" id="vetorItensId2">
        <bean:define id="pai" name="vetorItensId2" property="father"></bean:define>
        <logic:equal name="pai" value="${codigo}">

        <bean:write name="vetorItensId2"    property="code" /> - <bean:write
         name="vetorItensId2"   property="name" />
         <br/>

        </logic:equal>


        </logic:iterate>

        </td>
    </tr>
    </logic:iterate>
</table>
</div>
```

Figura 72 – Código da página estrutura.jsp

Fonte: Autoria própria

A lógica aplicada é a mesma utilizada no aplicativo *Manipulador ArcaboMK* antigo, porém foi escrita com o uso de *taglibs*. A chamada para a classe *EstruturaAction* se encontra no menu principal do aplicativo mostrado na figura 44.

6.4.6 Tela de exemplo de geração e funcionamento de menus

Essa funcionalidade é responsável por apresentar um exemplo prático de utilização dos menus conforme mostrado na figura 31.

A página funciona por meio da passagem de parâmetros como mostrado na figura 33 e sua única chamada no sítio antigo foi feita na página de cadastro por meio do *link* de ajuda "C*lique Aqui"* como ilustrado na figura 73.

Figura 73 – Chamada da página de exemplos de menu

Fonte: Autoria própria

Com a refatoração incorporou-se também uma chamada no menu principal através do *link "Exemplo de geração de menus"*, como ilustra a figura 74.

Figura 74 – Chamada da página de exemplos de menu incorporda ao menu principal do aplicativo Manipulador ArcaboMK 2.0

Fonte: Autoria própria

Como nas outras páginas, foi criada uma classe *Action* chamada *MenuExampleAction* para a criação dos objetos e utilização do método *selectFatherAndSon(int codigo)* para a geração dos menus. O código da classe *MenuExampleAciton* está ilustrado na figura 75.

```
public class MenuExampleAction extends Action {
    @Override
    public ActionForward execute(ActionMapping mapping, ActionForm form,
    HttpServletRequest request, HttpServletResponse response)
    throws Exception {
        int codigo= Integer.parseInt(request.getParameter("codigo"));

        ModelPer modelo = new ModelPer();
        List<Model> vetorItens = new ArrayList<Model>();
        vetorItens = modelo.selectFatherAndSon(codigo);
        Model pai  = new Model();
        pai= vetorItens.get(0);
        vetorItens.remove(0);

        request.getSession().setAttribute("paiCodigo", pai.getFather());
        request.getSession().setAttribute("paiNome", pai.getName());
        request.getSession().setAttribute("vetorItens", vetorItens);

    return mapping.findForward("menuExample");

    }
}
```

Figura 75 – Código da classe MenuExampleAction

Fonte: Autoria própria

A configuração do arquivo *struts-config.xml* para a classe *MenuExampleAction* está ilustrada na figura 76.

```
<action name="MenuExample" path="/MenuExample" type="Actions.MenuExampleAction"
    scope="request">
    <forward name="menuExample" path="/View/menuExample.jsp"
        redirect="true" />
</action>
```

Figura 76 – Código da classe MenuExampleAction

Fonte: Autoria própria

A lógica de passagem de parâmetros utilizada na página do aplicativo antigo foi implantada nesta classe, sendo que ela recupera esse parâmetro e então executa o método *selectFatherAndSon(int codigo)* novamente para a geração de um novo menu. Por isso, na primeira chamada desta classe, como mostrado na figura 44, é passado como parâmetro o valor 1, para que o menu seja gerado a partir da Raiz.

Outra mudança ocorre quando o item pai é separado do vetor de retorno do método *selectFatherAndSon(int codigo)* para que possa ser disposto corretamente os valores de parâmetros na página *menuExample.jsp*.

Após a execução da classe *MenuExampleAction* e a atribuição as variáveis de sessão, é feito um *forward* para a página *menuExample.jsp* e seu código está ilustrado na figura 77.

```
<h1>Exemplo de Estrutura de menu</h1>

<ul>
    <li><bean:define id="cod" value="${paiCodigo}"></bean:define>

    <html:link action="/MenuExample.do" paramId="codigo" paramName="cod"
        property="${cod}">
        <bean:write name="paiNome"/>
    </html:link></li>
    <ul>

        <logic:iterate name="vetorItens" id="vetorItensId">
        <li>

        <bean:define id="codFilho" name="vetorItensId" property="code"></bean:define>
        <html:link action="/MenuExample.do" paramId="codigo" paramName="codFilho"
        property="${codFilho}">

        <bean:write name="vetorItensId" property="name" />

        </html:link></li>
            </logic:iterate>
    </ul>
</ul>
```

Figura 77 – Código da página menuExample.jsp

Fonte: Autoria própria

Os *links* fazem chamada para a classe *MenuExampleAction* com passagem de parâmetros para a geração dos menus.

O primeiro item do menu passa como parâmetro o código do *pai* para que o menu anterior seja gerado, enquanto os *filhos* são dispostos utilizando a lógica de listagem padrão utilizada nas outras páginas refatoradas. Os itens *filhos* no menu passam o seu código como parâmetro para que um novo menu seja gerado caso o item seja clicado.

6.5 VALIDAR O CÓDIGO REFATORADO

Utilizando o método apresentado por Rapeli (2006), listam-se as entradas e saídas do software refatorado para comparação com o antigo, como apresentado no Quadro 2.

Quadro 2 – Interações do usuário com o sistema Manipulador ArcaboMK após a refatoração

N° de interação	**Interação do usuário no sistema (entrada)**	**Resposta do sistema na tela do usuário**
Operações de acesso do sistema		
1	Executar o sistema	A tela principal é apresentada contendo os dados cadastrados na tabela i*tem.* Um menu está presente em todas as páginas que contém *links* para as páginas de: cadastro, principal e estrutura dos menus em uma tabela. Ao lado de cada linha da tabela que apresenta os dados cadastrados existem dois botões, um que direciona para a página de edição e outro para exclusão do item.
2	Link "*Cadastrar Item"*	A tela de cadastro é apresentada contendo três campos de formulário: nome, endereço e pai. Existe também um botão chamado "*enviar*" que envia os dados para serem cadastrados.
3	Opção "*Clique aqui"* na dica de ajuda para cadastro de itens da tela de cadastro de itens	Uma nova página é apresentada onde é exibido um exemplo de geração e funcionamento dos menus.
4	Botão "*enviar*" da tela de cadastro de item	É apresentada em uma nova tela a mensagem de confirmação "O registro foi inserido com sucesso".
5	Botão "*editar*" da tela principal no item "*Raiz"*	Um formulário é exibido em uma nova tela com os campos preenchidos com os dados do item raiz. Os campos deste formulário são: o nome e o endereço.
6	Botão "*enviar"* da tela de edição do item "*Raiz"*	É apresentada em uma nova tela a mensagem "O registro foi editado com sucesso".
7	Botão "*editar"* da tela principal nos demais itens	Um formulário é mostrado em uma nova tela com os campos preenchidos com os dados do item escolhido na tela principal. Os campos deste formulário são: nome, endereço e pai.
8	Botão "*enviar*" da tela de edição dos demais itens	É apresentada em uma nova tela a mensagem "O registro foi editado com sucesso".

Fonte: Autoria própria

Quadro 2 – Interações do usuário com o sistema Manipulador ArcaboMK após a refatoração (cont.)

9	Botão “deletar” da tela principal no item “Raiz”	Uma nova tela é exibida contendo o item que será deletado em uma linha de tabela e um formulário contendo as opções “Sim” e “Não”.
10	Opção “*Sim”* da tela de confirmação de exclusão do item “*Raiz”*	É apresentada uma nova tela com a mensagem “ERRO: A raiz não pode ser deletada.”
11	Opção “*Não”* da tela de confirmação de exclusão	A exclusão é cancelada e o usuário é direcionado para a página principal.
12	Botão “*deletar”* da tela principal nos demais itens	Uma nova tela é apresentada contendo o item que será excluído em uma linha de tabela e um formulário contendo as opções “Sim” e “Não”.
13	Opção “*Ok”* da caixa de confimaçãode exclusão dos demais itens	É mostrada uma nova tela com a mensagem: “O registro foi deletado com sucesso”.
14	*Link* “*Estrutura”*	Uma nova página é exibida onde é apresentado um exemplo de geração e funcionamento dos menus.
15	*Link* Exemplo de geração de menus	Uma nova página é mostrada onde é colocado um exemplo de geração e funcionamento dos menus.

Fonte: Autoria própria

Avaliando e comparando os Quadros 1 e 2, pode-se concluir que os comportamentos do aplicativo foram mantidos com sua refatoração, pois as funcionalidades, entradas e saídas permaneceram as mesmas da aplicação anterior.

Analisando do ponto de vista visual, a página apresentou apenas pequenas mudanças tais como: a visualização dos códigos dos itens nos campos de seleção, o menu principal que teve a adesão e um novo *link* e a exclusão de itens na qual foi incorporada uma nova página para a sua confirmação.

O diagrama de classes do projeto precisou ser atualizado, pois novas classes foram adicionadas ao projeto devido ao uso das *Actions*. O diagrama de classes para o *Manipulador ArcaboMK* 2.0 está ilustrado na figura 78.

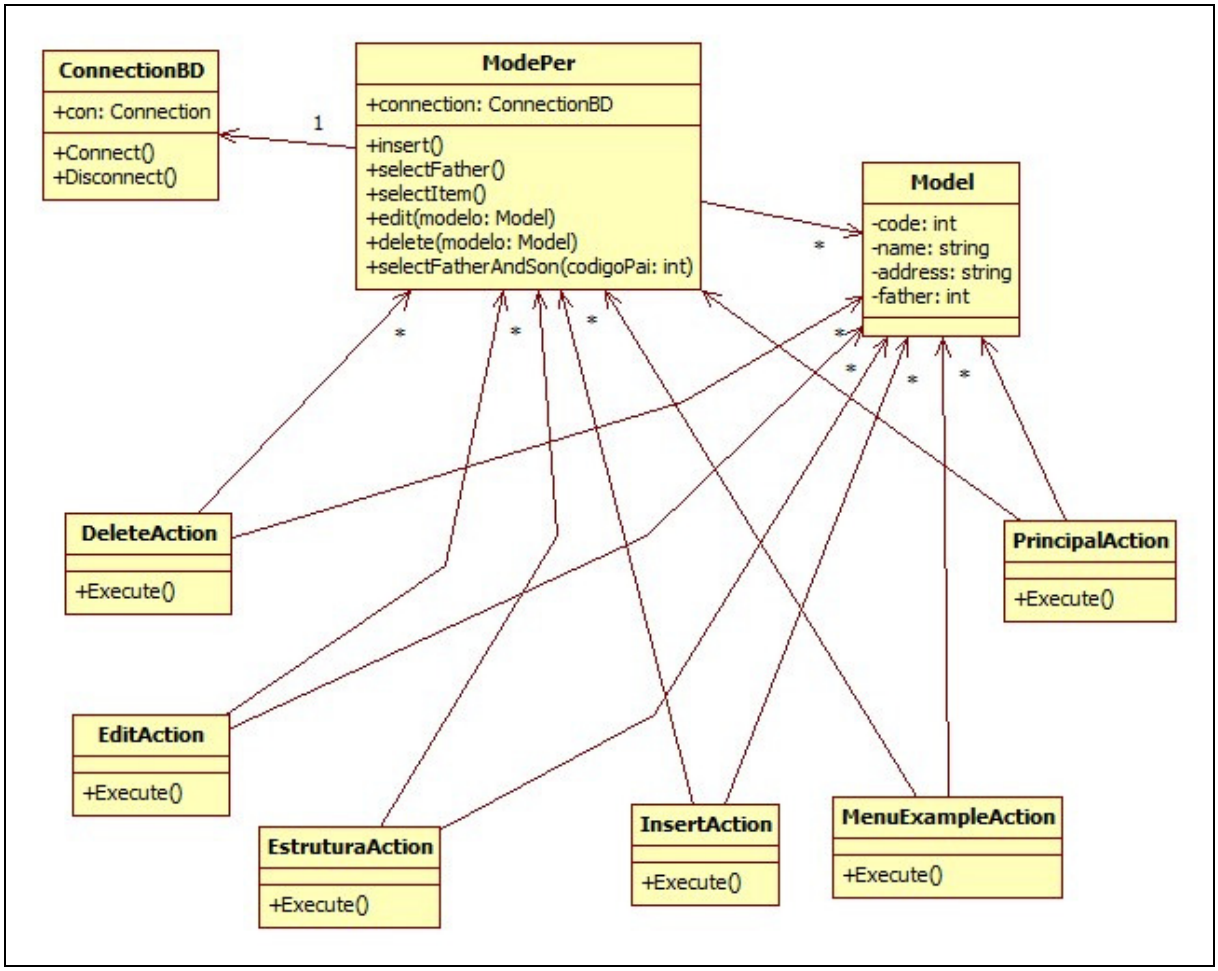

Figura 78 – Diagrama de classes do aplicativo Manipulador ArcaboMK 2.0

Fonte: Autoria própria

A figura 79 apresenta o diagrama de pacotes com seus respectivos componentes do aplicativo refatorado.

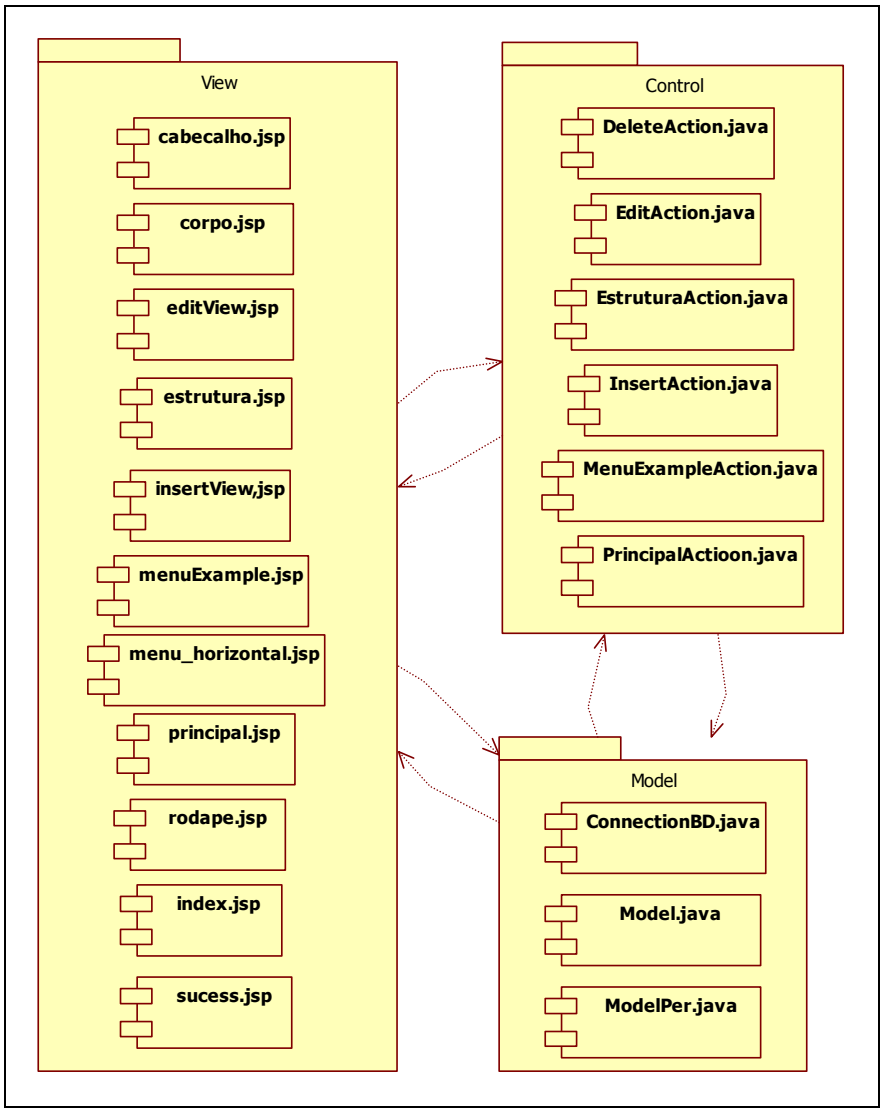

Figura 79 – Diagrama de pacotes do aplicativo Manipulador ArcaboMK 2.0
Fonte: Autoria própria

6.6 AVALIAR CARACTERÍSTICAS DE QUALIDADE E DESEMPENHO

Para avaliação das características de qualidade de software foi utilizado o *plugin Metrics* (Metrics 2012) para a plataforma Eclipse. A figura 80 apresenta os resultados das métricas calculadas pelo *plugin* no aplicativo antes da refatoração.

Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maximum	Method
Number of Overridden Methods (avg/max per type)	0	0	0	0	/ManipuladorArcabomk_definitivo/src/Model/ModelPer.java	
Number of Attributes (avg/max per type)	10	3.333	1.7	5	/ManipuladorArcabomk_definitivo/src/Model/ConnectionBD.java	
Number of Children (avg/max per type)	0	0	0	0	/ManipuladorArcabomk_definitivo/src/Model/ModelPer.java	
Number of Classes (avg/max per packageFragment)	3	3	0	3	/ManipuladorArcabomk_definitivo/src/Model	
Method Lines of Code (avg/max per method)	127	6.35	8.07	28	/ManipuladorArcabomk_definitivo/src/Model/ModelPer.java	selectPaieFilho
Number of Methods (avg/max per type)	20	6.667	2.625	9	/ManipuladorArcabomk_definitivo/src/Model/Model.java	
Nested Block Depth (avg/max per method)		1.55	0.865	4	/ManipuladorArcabomk_definitivo/src/Model/ModelPer.java	selectPaieFilho
Depth of Inheritance Tree (avg/max per type)		1	0	1	/ManipuladorArcabomk_definitivo/src/Model/ModelPer.java	
Number of Packages	1					
Afferent Coupling (avg/max per packageFragment)		0	0	0	/ManipuladorArcabomk_definitivo/src/Model	
Number of Interfaces (avg/max per packageFragment)	0	0	0	0	/ManipuladorArcabomk_definitivo/src/Model	
McCabe Cyclomatic Complexity (avg/max per method)		1.55	0.865	4	/ManipuladorArcabomk_definitivo/src/Model/ModelPer.java	selectPaieFilho
Total Lines of Code	193					
Instability (avg/max per packageFragment)		1	0	1	/ManipuladorArcabomk_definitivo/src/Model	
Number of Parameters (avg/max per method)		0.45	0.497	1	/ManipuladorArcabomk_definitivo/src/Model/ModelPer.java	ExecuteUpdateBD
Lack of Cohesion of Methods (avg/max per type)		0.552	0.391	0.857	/ManipuladorArcabomk_definitivo/src/Model/Model.java	
Efferent Coupling (avg/max per packageFragment)		0	0	0	/ManipuladorArcabomk_definitivo/src/Model	
Number of Static Methods (avg/max per type)	0	0	0	0	/ManipuladorArcabomk_definitivo/src/Model/ModelPer.java	
Normalized Distance (avg/max per packageFragment)		0	0	0	/ManipuladorArcabomk_definitivo/src/Model	
Abstractness (avg/max per packageFragment)		0	0	0	/ManipuladorArcabomk_definitivo/src/Model	
Specialization Index (avg/max per type)		0	0	0	/ManipuladorArcabomk_definitivo/src/Model/ModelPer.java	
Weighted methods per Class (avg/max per type)	31	10.333	4.989	17	/ManipuladorArcabomk_definitivo/src/Model/ModelPer.java	
Number of Static Attributes (avg/max per type)	0	0	0	0	/ManipuladorArcabomk_definitivo/src/Model/ModelPer.java	

Figura 80 – Métricas obtidas utilizando o plugin Metrics para versão antiga do manipulador

Fonte: Autoria própria

O resultado da aplicação do *plugin* para o manipulador após o processo de refatoração está ilustrado na figura 81.

Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maximum	Method
Number of Overridden Methods (avg/max per type)	6	0.667	0.471	1	/ManipuladorArcabomk_2.0/src/Actions/DeleteAction.java	
Number of Attributes (avg/max per type)	10	1.111	1.853	5	/ManipuladorArcabomk_2.0/src/Model/ConnectionBD.java	
Number of Children (avg/max per type)	0	0	0	0	/ManipuladorArcabomk_2.0/src/Actions/DeleteAction.java	
Number of Classes (avg/max per packageFragment)	9	4.5	1.5	6	/ManipuladorArcabomk_2.0/src/Actions	
Method Lines of Code (avg/max per method)	275	10.185	11.327	44	/ManipuladorArcabomk_2.0/src/Actions/EditAction.java	execute
Number of Methods (avg/max per type)	27	3	3.266	9	/ManipuladorArcabomk_2.0/src/Model/ModelPer.java	
Nested Block Depth (avg/max per method)		1.815	1.123	5	/ManipuladorArcabomk_2.0/src/Actions/EditAction.java	execute
Depth of Inheritance Tree (avg/max per type)		1.889	0.314	2	/ManipuladorArcabomk_2.0/src/Actions/DeleteAction.java	
Number of Packages	2					
Afferent Coupling (avg/max per packageFragment)		3	3	6	/ManipuladorArcabomk_2.0/src/Model	
Number of Interfaces (avg/max per packageFragment)	0	0	0	0	/ManipuladorArcabomk_2.0/src/Actions	
McCabe Cyclomatic Complexity (avg/max per method)		1.963	1.427	7	/ManipuladorArcabomk_2.0/src/Actions/EditAction.java	execute
Total Lines of Code	454					
Instability (avg/max per packageFragment)		0.625	0.375	1	/ManipuladorArcabomk_2.0/src/Actions	
Number of Parameters (avg/max per method)		1.259	1.53	4	/ManipuladorArcabomk_2.0/src/Actions/DeleteAction.java	execute
Lack of Cohesion of Methods (avg/max per type)		0.184	0.345	0.857	/ManipuladorArcabomk_2.0/src/Model/Model.java	
Efferent Coupling (avg/max per packageFragment)		4	2	6	/ManipuladorArcabomk_2.0/src/Actions	
Number of Static Methods (avg/max per type)	0	0	0	0	/ManipuladorArcabomk_2.0/src/Actions/DeleteAction.java	
Normalized Distance (avg/max per packageFragment)		0.375	0.375	0.75	/ManipuladorArcabomk_2.0/src/Model	
Abstractness (avg/max per packageFragment)		0	0	0	/ManipuladorArcabomk_2.0/src/Actions	
Specialization Index (avg/max per type)		1.333	0.943	2	/ManipuladorArcabomk_2.0/src/Actions/DeleteAction.java	
Weighted methods per Class (avg/max per type)	53	5.889	6.279	22	/ManipuladorArcabomk_2.0/src/Model/ModelPer.java	
Number of Static Attributes (avg/max per type)	2	0.222	0.416	1	/ManipuladorArcabomk_2.0/src/Model/ModelPer.java	

Figura 81 – Métricas do aplicativo Manipulador ArcaboMK 2.0 após refatoração

Fonte: Autoria própria

Comparando as figuras 80 e 81, observou-se uma melhoria de código proporcionada pela refatoração. A comparação entre os índices está apresentado na tabela 1.

Tabela 1 – Comparação dos índices das métricas

Nome da métrica	Definição	Manipulador ArcaboMK	Manipulador ArcaboMK 2.0
Métricas			
Number of classes	Número de classes	3	9
Number of children	Número de subclasses utilizadas	0	0
Number of Overridden Methods (NORM)	Número de métodos que sobrescrevem uma classe extendida	0	6
Number of Methods (NOM)	Número de métodos	20	27
Lines of Code	Quantidade de linhas de código	193	454
McCabe Cyclomatic Complexity	Conta o número de fluxos através de um pedaço de código. Cada vez que um ramo ocorre (se, por, enquanto que, se, de captura caso, :? Operador ternário, bem como os && e | | operadoresde lógica condicional em expressões) esta métrica é incrementado em um.	Média=1,55 Desvio padrão=0,865 Máximo=4	Média=1,963 Desvio padrão=1,427 Máximo=7
Weighted Methods per Class (WMC)	Soma da McCabe Cyclomatic Complexity para todos os métodos em uma classe	Total=31 Média=10,333 Desvio padrão=4,989 Máximo=17	Total=53 Média=5,889 Desvio padrão=6,279 Máximo=22
Lack of Cohesion of Methods (LCOM*)	Uma medida para a coesão de uma classe. Um valor baixo indica uma classe coesa e um valor próximo de 1 indica uma falta de coesão e sugere a classe pode melhor ser dividida em um número de (sub) classes.	Média=0,552 Desvio padrão=0,391 Máximo=0,857	Média=0,184 Desvio padrão=0,345 Máximo=0,857

Fonte: Autoria própria

Por meio dos coeficientes apresentados na tabela se verifica que os índices do aplicativo refatorado são maiores que os do primeiro aplicativo, além do coeficiente de coesão da classe que apresentou índices melhores.

O número de classes e de métodos sobrescritos cresceu, pois foram adicionadas uma classe *Action* para cada funcionalidade do aplicativo, como já mostrado na figura anterior. Cada classe extende o *org.apache.struts.action.Action*.

No índice *McCabe Cyclomatic Complexity*, que também houve um aumento, pois foram adicionadas 6 novas classes contendo a regra de negócio que antes era apresentada nas páginas JSP, logo não sendo contabilizadas pelo plugin.

A diminuição do índice *Lack of Cohesion of Methods*, que faz a medição da coesão dos métodos de uma classe, indica que após a refatoração os métodos estão mais bem distribuídos entre as classes do programa. O valor máximo permaneceu o mesmo, pois ainda deve ser possível a divisão dos métodos de uma das classes antigas do programa que não foram muito alteradas na refatoração.

Na análise de desempenho e complexidade pode parecer que não houve ganho com a refatoração, porém se avaliar a estrutura que o aplicativo possui após o processo de refatoração concluí-se que este aumento dos índices é pequeno e aceitável tendo em vista que a coesão dos métodos e classes diminuiu e que o padrão MVC (HOLMES, 2006) foi aplicado com sucesso.

7 CONCLUSÃO

A refatoração de software compreende muitas etapas, desde a parte técnica quanto a metodologia empregada. Este trabalho adaptou algumas metodologias e propôs outra para realização do processo de refatoração do Manipulador *ArcaboMK*.

A refatoração do aplicativo *Manipulador ArcaboMK* teve como objetivo a mudança de estrutura do projeto afim de que ele empregasse o modelo MVC de forma eficaz e a adição de funcionalidades fosse facilitada.

O processo foi iniciado com a análise do aplicativo em que foram identificadas as entradas e saídas da sua primeira versão baseando-se no método de Rapeli (2006). Nesta etapa foi possível compreender um dos pontos essenciais em toda refatoração, que é a garantir que o comportamento do software não mude.

Na etapa de análise do diagrama de classes foi possível contabilizar o número de classes que iriam ser utilizadas e criadas, confrontando o modelo de classes do aplicativo com o modelo de funcionamento do framework *Struts* para então realizar as mudanças necessárias.

Na identificação dos pontos a estrutura do projeto utilizando *Struts* foi criada e foram migrados os componentes antigos. Através desta etapa pode-se começar a adaptar as classes e páginas para funcionar conforme o framework especifica.

Alcançando com sucesso cada uma das iterações que foram separadas nessa etapa, foi possível construir uma linha de conhecimento de base para aplicação do *Struts* nas outras funcionalidades do aplicativo.

Por meio do processo de refatoração houve uma melhora na estrutura do aplicativo, como também a reorganização do código que estava confuso e mal distribuído.

Analisando o quesito de desempenho de software, o aplicativo obteve resultados inferiores com a refatoração, exceto pelo índice que mede a coesão das classes, mas isto é esperado conforme o estudo exposto na seção 2, devido a utilização de padrões. A aplicação de padrões no código pode tornar o desempenho de processamento de código mais lento que uma solução simples de código não padronizado.

Observando por este enfoque, com a refatoração do aplicativo *Manipulador Arcabomk* diminui-se os índices de desempenho, porém aumenta-se o índice de

coesão das classes e métodos aplicando uma estrutura de software mais adequada a adição de funcionalidades no aplicativo.

7.1 TRABALHOS FUTUROS

Os seguintes trabalhos futuros podem ser desenvolvidos a partir desta pesquisa tais como:

- Validar os formulários utilizando o framework *Struts* por meio dos validadores da WCAG 2.0.
- Aplicar uma funcionalidade que permita a criação de páginas de conteúdo no aplicativo.
- Criar uma funcionalidade de edição do conteúdo das páginas.
- Utilizar validação de critérios de acessibilidade na criação de páginas.
- Desenvolver uma funcionalidade que permita ao usuário escolher a ordem em que os itens aparecerão no menu.
- Criar de um Sistema de *Login* para o aplicativo, de modo que este possa ficar ativo no servidor e acessível de qualquer lugar.

REFERÊNCIAS

ARCABOMK. Página do Framemk – Um framewor para Otimização de Preço de Venda. Disponível em <http://200.134.81.19:8080/arcabomk> acesso em 11 maio 2011.

CMS. **Content Management System**. Disponível em <http://pt.kioskea.net/contents/www/cms.php3> acesso em 04 de novembro de 2011.

EMDEN, E. MOONEN, L. Java quality assurance by detecting code smells. **Proc. Working Conf. Reverse Engineering – IEEE Computer Society**, 2002.

FOWLER, Martin. **Aperfeiçoando o projeto de código existente**. Porto Alegre: Bookman, 2004.

FOWLER, Martin. **Padrões de Arquiteturas de Aplicações Corporativas**. Porto Alegre: Bookman, 2006.

GAMMA, Eric et al. **Design patterns – Elements of Reusable Object-Oriented Software**. New York: Addison-Wesley, 1995.

GPES. Grupo de Pesquisa de Engenharia de Software - GPES. Disponível em <http://www.pg.utfpr.edu.br/coinf/gpes/> acesso em 11 maio 2011.

HOLMES, James. **Struts: The complete Reference**. McGraw-Hill, 2006.

HUSTED, Ted et al. **Struts em Ação: Como construir aplicações web com o principal framework Java**. Rio de Janeiro: Ciência Moderna, 2004.

JOOMLA. **Joomla CMS**. Disponível em <http://www.joomla.org/> acesso em 11 maio 2011.

JÚNIOR, Francisco. B. **JSP – A tecnologia Java para a internet**. Editora Érica, 2003.

KATAOKA, Y. Automated support for program refactoring using invariants. **Proc. Int'l Conf. Software Maintenance – IEEE Computer Society**, 2001.

KERIEVSKY, Joshua. **Refatoração para Padrões**. Porto Alegre. Bookman, 2008.

LACERDA, V.S.;RIBAS, J.H. **Estrutura do sítio Framemk**. Disponível em:<http://200.134.81.19:8080/gpes/arquivos/material/engenharia/interface/relatorios/2010/I.Lacerda.Ribas.2010.F.pdf> acesso em: 16 de agosto de 2010.

LACERDA, Victor S. Acessibilidade – Um Estudo de Caso no Sítio do Arcabomk. **SICITE,** Ponta Grossa (PR), ago. 2011.

LEE, Berners. **Web Accessibility Initiative**. Disponível em <http://www.w3.org/WAI/> acesso em 11 maio 2011.

Manuais Postgree. Disponível em <http://www.postgresql.org/docs/8.2/static/ddl-constraints.html> acesso em 04 de novembro de 2011.

MENS, Tom.TOURWÉ, Tom. A Survey of Software Refactoring. **IEEE Transactions on Software Engineering**, v. 30, n. 2, 2004

Metrics. **Metrics 1.3.6 Plugin para Eclipse**. Disponível em <http://metrics.sourceforge.net/> acesso em 18 abril 2012.

Netbeans 6.8. **NetBeans IDE 6.8**. Disponível em <http://netbeans.org/community/releases/68/> acesso em 11 maio 2011.

RAPELI, Leide R. **Refatoração de sistemas Java utilizando padrões de projeto: um estudo de caso**. 2006. 127 f. Dissertação (Mestrado, Universidade Federal de São Carlos. São Paulo, 2006.

W3C. **World Wide Web Consortium**. Disponível em <http://www.w3.org/> acesso em 11 maio 2011.

WCAG 1.0. **Web Content Accessibility Guidelines 1.0**. 1999. Disponível em <http://www.w3.org/TR/WCAG10/> acesso em 11 maio 2011.

WCAG 2.0. **Web Content Accessibility Guidelines 2.0**. 2008. Disponível em <http://www.w3.org/TR/WCAG/> acesso em 11 maio 2011.

WORDPRESS. **Gerenciador de conteúdo**. Disponível em <http://br.wordpress.org/> acesso em 11 maio 2011.