Instituto de Pesquisas Tecnológicas do Estado de São Paulo

Luiz Henrique Rorato Decaro

Foundation: facilitando a implementação de operações do tipo CRUD em aplicações JavaEE.

São Paulo

2008

Luiz Henrique Rorato Decaro

Foundation: facilitando a implementação de operações do tipo CRUD em aplicações JavaEE

Luiz Henrique Rorato Decaro

Foundation: facilitando a implementação de operações do tipo CRUD em aplicações JavaEE

Dissertação apresentada ao Instituto de Pesquisas Tecnológicas de São Paulo - IPT para obtenção do título de mestre em Engenharia de Computação.

Área de Concentração: Engenharia de Software.

Orientador: Prof. Dr. Alan Mitchell Durham

São Paulo

Novembro/2008

Luiz Henrique Rorato Decaro

Foundation: facilitando a implementação de operações do tipo CRUD em aplicações JavaEE

Dissertação de Mestrado apresentada ao Instituto de Pesquisas Tecnológicas do Estado de São Paulo - IPT, como parte dos requisitos para a obtenção do título de Mestre em Engenharia de Computação.

Data da aprovação: _____/_____/_____

_____
Prof. Dr. Alan Mitchell Durham (orientador)
USP - Universidade de São Paulo

Membros da Banca Examinadora:

Prof. Dr. Alan Mitchell Durham (Orientador)
USP - Universidade de São Paulo

Prof. Dr. Fabio Kon (Membro)
USP - Universidade de São Paulo

Prof. Dr. Marcelo Novaes de Rezende (Membro)
IPT - Instituto de Pesquisas Tecnológicas do Estado de São Paulo

Dedico esta dissertação a meus pais,
que sempre me ensinaram sobre a importância do
estudo e, em especial, à minha noiva Patricia Rodrigues,
que sempre me deu todo o apoio necessário.

Agradecimentos

Meus sinceros agradecimentos para:

. Prof. Dr. Alan Mitchell Durham por todo aprendizado e pela ótima experiência de trabalho e pesquisa.

. Minha noiva Patricia Rodrigues pelas revisões.

. Minha mãe Maura Antônia Rorato Decaro e meu irmão Luiz Guilherme Rorato Decaro pelo apoio durante todo este trajeto.

. Adilson, da secretaria do Instituto de Pesquisas Tecnológicas de São Paulo - IPT, por toda presteza nos assuntos relacionados ao curso de Mestrado em Engenharia de Software.

Resumo

Aplicações corporativas são conhecidas pela sua necessidade de coletar, transformar e disponibilizar dados para geração de informação. A linguagem Java é amplamente utilizada para desenvolvimento de aplicações corporativas por prover uma melhor portabilidade entre sistemas operacionais e por prover bom suporte a sistemas baseados na *web*, a aplicações distribuídas e protocolos de rede com a ajuda de diversas APIs padronizadas. O desenvolvimento de sistemas *web* em Java pode ser dividido em duas partes: definição e implementação de domínio e definição e implementação arquitetural. Em aplicações que contém formulários *web*, a modelagem de domínio determina a criação de classes que definem os conjuntos de dados gerenciados pelos formulários. Neste mesmo contexto, a definição arquitetural deverá prover abstrações para deta- lhes expostos pelas especificações JavaEE e Java Database Connectivity para promover a utilização de classes do domínio no transporte interno de dados. Além disso, espera-se que a definição arquitetural proveja soluções a problemas específicos relacionados à implementação de formulários HTML, tais como o gerenciamento da persistência de dados agregados e representados por relacionamentos do tipo Um-Para-Um ou Um-Para-Muitos não limitando a quantidade de agregações gerenciadas. Existem diversos sistemas de middleware focados no desenvolvimento de aplicações JavaEE e que provêem, dentre outras funcionalidades, a abstração de detalhes expostos pelas especificações JavaEE e JavaDatabase Connectivity. Porém, ainda existe a necessidade de um middleware específico, focado na implementação de formulários HTML e que proponha um modelo de dados para o atendimento de requisições HTTP e execução automática de operações de persistência.O objetivo deste trabalho é a implementação de um *middleware* que solucione problemas específicos relacionados à implementação de formulários HTML em aplicações JavaEE. Existem alguns *middlewares* que ja provêem parte das funcionalidades, porém este *middleware* deve promover a reutilização de classes de domínio criadas por aplicações e deve gerenciar a persistência de dados apresentados em tela. Este middleware é contruído com o Struts, um *middleware* que é muito utilizado no mercado. A utilização do Struts não apenas evita o re-projeto de soluções já existentes no mercado, como também facilita a curva de aprendizado desta nova solução. O *mi- ddleware* também pode ser utilizado com qualquer provedor de persistência que obedeça a especificação JPA. A implementação atual utiliza o Hibernate.

Palavras-chaves: J2EE, JavaEE, Java, *middleware*, arquitetura de *software*, formulários HTML, JPA, Struts, Hibernate, *Design Patterns*, MVC, JUnit, testes de performance, testes de estresse.

Abstract

Foundation: promoting the implementation of CRUD operations in JavaEE applications.

Enterprise applications are known for collecting, transforming and providing data to generate information. The Java language is widely used for the development of enterprise applications because it offers better portability across operational systems and because it provides good support for web-based systems, for distributed applications and for network protocols with the help of many standardized APIs. The development of web systems in Java can be divided in two parts: domain definition/implementation and architectural definition/implementation. In applications that provide any web forms, the domain modeling determines the implementation of classes that define the managed data suite. In the same context, the architectural definition must provide abstractions of details exposed by the JavaEE and Java Database Connectivity specifications to promote the use of domain classes in the internal data transport. In addition, it is expected from the architectural definition to provide solutions to specific problems related to the implementation of HTML forms, such as managing the persistence of aggregated data represented by One-To-One or One-To-Many relationships not limiting the amount of managed aggregations. There are many middleware systems focused on the development of JavaEE applications and that provide, among other functionalities, the abstraction of details exposed by the JavaEE and by the Java Database Connectivity specifications. However, there is still a need for a specific middleware, focused on the implementation of HTML forms, that proposes a data model to the attendance of HTTP requests and automatic execution of persistence operations. The goal of this work is the implementation of a middleware capable of solving specific problems related to the implementation of HTML forms in JavaEE applications. There are some systems that provide part of those capabilities, however this middleware must promote the reuse of domain classes created by applications and must manage the persistence of the data presented in screen. The middleware is build on top of Struts, a middleware that is widely used in the market. Using Struts not only avoid re-designing solutions already adopted in the market, but also eases the learning curve on our solution. The middleware can also be used with any persistence providers that complies with the JPA specification. The current implementation uses Hibernate.

Keywords: J2EE, JavaEE, Java, middleware, software architecture, stress tests, performance tests, HTML forms, JPA, Struts, Hibernate, Design Patterns, MVC, JUnit.

Lista de Figuras

Lista de Tabelas

Lista de abreviaturas

API	Application Programming Interface
CRUD	Create Retrieve Update Delete
DAO	Data Access Object
E-R	Entidade Relacionamento
IoC	Inversion of Control
JavaEE	Java Enterprise Edition
JDBC	Java Database Connectivity
JDO	Java Data Objects
JPA	Java Persistence API
JSF	Java Server Faces
JSP	Java Server Page
HTML	Hyper Text Markup Language
HTTP	Hyper Text Transfer Protocol
MVC	Model View Controller
O-R	Objeto-Relacional
ORM	Object Relacional Mapping
RoR	Ruby On Rails
SQL	Structured Query Language
UML	Unified Modeling Language
URL	Unified Resource Locator
XML	Extensible Markup Language

Sumário

1 Introdução

1.1 Motivação

A especificação JavaEE é um conjunto de APIs para desenvolvimento, implantação e execução de aplicações em Java. A especificação promove o desenvolvimento de aplicações em camadas, sendo duas camadas comuns a todas as aplicações *web* que gerenciam a persistência de dados: a camada de interface com servidores de aplicações (camada *web*) e a camada de acesso a dados de repositórios.

O desenvolvimento de aplicações pode ser dividido em duas partes: definição/implementação de domínio e definição/implementação arquitetural. Uma vez definido o domínio de uma aplicação, espera-se que a definição arquitetural facilite a criação de formulários HTML que gerenciam conjuntos de dados definidos por atributos de classes que compõem parte do domínio.

No entanto, a definição arquitetural de aplicações deve considerar que a interface com servidores de aplicações ou camada *web* não promove esta abordadem. Isto porque, de acordo com a especificação JavaEE, a interface que define o acesso a dados recebidos de requisições HTTP determina o acesso aos dados de forma individual e sempre no formato texto.

A especificação JavaEE poderia permitir a aplicações o acesso a dados recebidos de requisições HTTP por meio de um objeto definido pelo domínio de uma aplicação e cujos atributos definam o conjunto de dados recebido (incluindo os tipos de dados). Assim, no atendimento de uma requisição HTTP pelo servidor de aplicações, cada aplicação poderia definir um objeto de domínio que seria automaticamente instanciado e preenchido com os dados recebidos antes de ser disponibilizado para acesso por aplicações. De acordo com Johnson:

> Acreditar que objetos de domínio devem ser fisicamente separados da camada *web* é a mais cara e prejudicial falácia entre os desenvolvedores JavaEE(JOHNSON; HOELLER, 2004, p. 38).

Assim, esperamos que o transporte dos dados por todas as camadas arquiteturais seja realizado por objetos de domínio. A fim de proporcionar a construção de aplicações com arquiteturas consistentes, o acesso a repositórios de dados também poderia ser realizado com a utilização de objetos do domínio de aplicações. No entanto, assim como a especificação JavaEE, a especificação *Java DataBase Connectivity* (JDBC) não privilegia este tipo de abordagem; sua interface determina o acesso individual a cada dado definido por um atributo de um objeto de domínio. A complexidade no desenvolvimento de aplicações não deveria ser técnica, mas se concentrar na modelagem do próprio domínio relacionado à atividade ou negócio do usuário (EVANS, 2003).

Além da abstração de detalhes tecnológicos, a criação de formulários HTML em aplicações Java envolve a criação de soluções para problemas mais específicos. Podemos dar como exemplo uma aplicação que gerencia clientes em uma vídeo locadora. Essa aplicação define a entidade *categoria de clientes* (ex. mensalista ou avulso) e a entidade *cliente* composta por *endereços, e-mails* e *telefones*. Para o contexto deste trabalho, definimos a implementação de um formulário como o conjunto de atividades necessárias à criação de uma página HTML e de métodos para atendimento a requisições HTTP que solicitam a execução, em repositórios, de quatro operações relacionadas com os dados submetidos: *incluir, obter, alterar e excluir* (em inglês CRUD - *create, retrieve, update, delete*).

Suponhamos que um requisito da aplicação em questão seja a criação de um formulário que gerencia categorias de clientes e clientes. Neste caso, teríamos um formulário do tipo "Mestre-Detalhe-SubDetalhe" que gerencia os dados definidos por *categoria, cliente, endereço, e-mail e telefone*, sendo que *categoria* possui um relacionamento do tipo Um-Para-Muitos com *cliente* que, por sua vez, possui relacionamento do tipo Um-Para-Muitos com *endereço*, *e-mail* e *telefone* (Figura 1).

A implementação de formulários como os da Figura 1 pode ser executada de diversas maneiras, dependendo da arquitetura de cada aplicação. No entanto, sua implementação envolve a criação de operações de persistência que manipulem os dados de registros relacionados e definidos por relacionamentos do tipo Um-Para-Muitos (relacionamento *categoria-clientes*) e esta é uma tarefa trabalhosa; especialmente se estes relacionamentos também definirem outros relacionamentos do tipo Um-Para-Muitos (*categoria-*

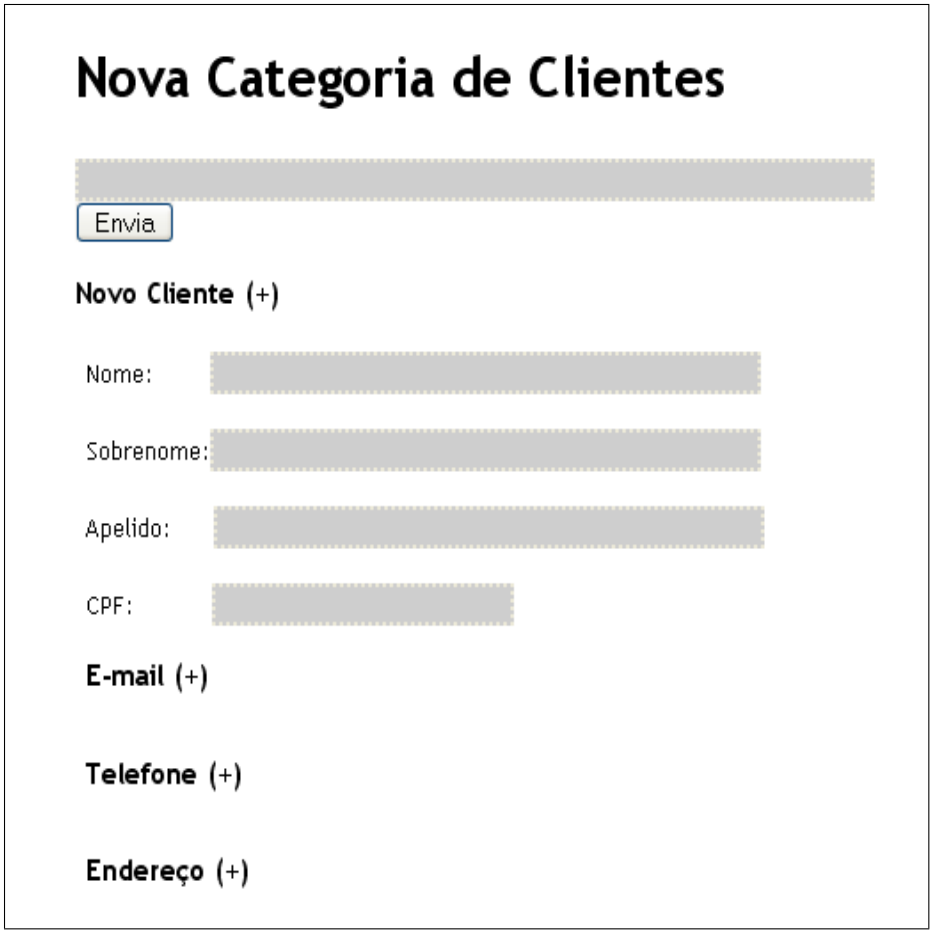

Figura 1: A aplicação que gerencia clientes em uma vídeo locadora possui um formulário para o cadastro de categoria, clientes, e-mails, telefones e endereços. A criação de uma categoria de clientes requer a criação de ao menos um cliente que pode ter dados pessoais de e-mails, endereços e telefones (botão +). O botão Envia realiza a submissão do formulário.

clientes-endereços). A implementação destes formulários envolve, além das atividades de abstração de detalhes tecnológicos citadas anteriormente, a implementação de alguma lógica que gerencie as operações de persistência a serem executadas nos dados agregados (usuários, e-mails, telefones e endereços), independente da operação de persistência executada no conjunto de dados agregador (categoria)[1].

Desta maneira, percebe-se a utilidade de um mecanismo que gerencie as operações de persistência em dados agregados, possibilitando a implementação de formulários que gerenciam automaticamente a persistência de dados representados em relacionamentos do

1 Isso significa que a alteração de uma categoria pode implicar na criação, alteração ou exclusão de clientes e a alteração em um cliente pode implicar na criação, alteração ou exclusão de endereços, telefones e e-mails

tipo Um-Para-Um ou Um-Para-Muitos.

Utilizando apenas o *kit* disponibilizado pela linguagem Java, o desenvolvimento da aplicação que contém o formulário HTML do exemplo acima (Figura 1.1) envolve a execução das seguintes atividades:

1. ***Implementar as classes de domínio***: Devem ser implementadas as classes de domínio que definem os conjuntos de dados apresentados no formulário HTML.

2. ***Implementar operações de persistência com a utilização de objetos de domínio***: A linguagem Java possui duas especificações para acesso a repositórios de dados. A especificação JDBC[2] e a especificação Java Persistence API (JPA)[3]. Independente da tecnologia utilizada, deve ser implementado um gerenciamento de conexões utilizadas por aplicações a fim de minimizar a ocorrência de erros em tempo de execução que podem acarretar em um consumo excessivo de recursos. A criação de uma camada de abstração dos detalhes tecnológicos relacionados com o acesso a dados de um repositório é fortemente aconselhada além de ser uma boa prática de programação para desenvolvimento de aplicações JavaEE (ALUR et al., 2003, p. 285). No caso específico do acesso por meio da especificação JDBC devem ser implementados métodos que determinam as operações de persistência executadas em relacionamentos do tipo Um-Para-Um e Um-Para-Muitos durante uma operação de alteração de dados.

3. ***Processar requisições HTTP***: Os dados recebidos por meio de requisições HTTP, sempre no tipo texto, devem ser convertidos (caso necessário) e associados a objetos de domínio. O processamento de requisições HTTP consiste em obter dados de uma tabela de associação entre nomes de elementos HTML e valores, converter o tipo dos dados e associá-los a atributos do objeto de domínio ou de um objeto de domínio agregado. Além disso, cabe a cada aplicação instanciar o objeto de domínio (e suas possíveis agregações) durante o processamento de requisições.

4. ***Implementar página JSP***: Deve ser criada uma página dinâmica JSP que extrai dados de objetos de domínio e gera páginas HTML.

2 Especifica a comunicação com repositórios de dados por meio de *drivers* que implementam as interfaces de comunicação.
3 Define a comunicação com repositórios de dados através de mapeamento objeto-relacional.

5. ***Configurar a aplicação web***: Devem ser definidas e configuradas, no arquivo de configuração de aplicações JavaEE, as URLs que a aplicação atenderá para o recebimento de requisições que solicitam a execução das operações de persistência.

Abstrair detalhes do acesso à camada *web* ou do acesso a repositórios de dados de cada aplicação ou formulário desenvolvidos é muito trabalhoso. As atividades 2 e 3 são repetitivas e poderiam ser automatizadas com a utilização de um *middleware*. O tempo de desenvolvimento de soluções específicas a cada aplicação, aliado à quantidade de testes necessários justificam a implementação de um *middleware* que simplifique a construção de formulários em aplicações JavaEE.

Além disso, a implementação de formulários HTML do tipo Mestre-Detalhe (ou Mestre-Detalhe-Subdetalhe) requer a utilização de algum mecanismo que permita identificar quais agregações devem ser alteradas, incluídas ou excluídas do repositório de dados quando um formulário de alteração de dados for submetido. Estas operações podem ser identificadas pela diferença entre o conjunto de dados utilizado para o preenchimento inicial do formulário e o conjunto de dados recebido com sua submissão.

Uma técnica comum é a criação de controles, escondidos em página, para cada formulário que, na sua submissão, informam a aplicação das agregações inseridas, alteradas ou apagadas no cliente. Esta abordagem possui uma desvantagem por ser específica para o contexto dos dados de cada formulário HTML.

Existe a necessidade de um mecanismo de controle de estado para os dados apresentados em formulários de alteração de dados. Este mecanismo deve comparar os dados utilizados para o preenchimento inicial de um formulário de alteração de dados com os dados recebidos pela sua submissão. Assim, torna-se possível generalizar a identificação e automatizar a execução de operações de persistência para agregações de dados em qualquer formulário HTML de alteração de dados.

Atualmente, existem diversas ferramentas de *middleware* que proporcionam a aplicações a comunicação com ao menos uma destas duas camadas em um nível mais elevado. É o caso dos arcabouços Struts, Webwork, Spring e Stripes que, porém, possuem funcionalidades limitadas como veremos no restante deste trabalho.

1.2 Objetivo

O objetivo deste trabalho é desenvolver um *middleware* capaz de gerenciar a persistência de dados definidos por objetos de domínio e apresentados em formulários HTML. Em particular, o *middleware* deve:

- Controlar internamente o ciclo de vida dos objetos de domínio que definem os dados gerenciados por formulários HTML. Internamente, objetos de domínio deverão ser instanciados, preenchidos e utilizados no transporte dos dados gerenciados durante as operações de persistência.

- Implementar quatro operações a serem executadas automaticamente em mecanismos de persistência: incluir, obter, alterar e excluir. As operações devem ser executadas no atendimento a requisições HTTP e devem utilizar objetos definidos por classes do domínio de aplicações.

Propomos atingir o objetivo com a criação de uma solução que atenda a quatro requisitos principais:

1. ***Modelar o atendimento de requisições que determinam a execução de operações de persistência***: Prover um modelo de dados para o atendimento de requisições que solicitam a execução das operações *incluir, obter, alterar e excluir* as quais são executadas em mecanismos de persistência. Caso necessário, operações de persistência executadas em um objeto de domínio podem ser propagadas a objetos agregados.

2. ***Processar requisições HTTP:*** As requisições HTTP podem transportar, no formato texto, dados submetidos por formulários HTML. Estes dados devem ser convertidos automaticamente para o tipo de dados dos atributos definidos em uma classe do domínio configurada como a classe que representa o conjunto de dados recebidos. Um atributo pode estar definido como uma coleção de dados, isto é, uma classe ou interface do pacote java.util. Neste caso, o *middleware* deve prover um mecanismo que instancie o objeto da classe ou interface definida e popule automaticamente a coleção de dados com objetos relacionados ao objeto agregador.

Popular objetos de domínio com dados recebidos de clientes é uma operação que pode ser automatizada proporcionando a diminuição no tempo de desenvolvimento de aplicações. Popular automaticamente agregações definidas como coleções de dados permite a reutilização de código legado.

3. ***Disponibilizar pontos de interceptação***: A sequência de operações que compreende o atendimento a uma requisição HTTP e a execução de uma operação de persistência deve prover pontos de interceptação para uma possível especialização do processamento de requisições ou execução de operações do domínio antes da execução de uma operação de persistência. A existência de pontos específicos para a execução de operações comuns visa propor uma diminuição da erosão arquitetural (PERRY; WOLF, 1992) e aumento da manutenibilidade em aplicações.

4. ***Possibilitar a construção de formulários com leiaute específico***: O leiaute de um formulário HTML deve ser definido de acordo com requisitos de aplicações e não do *middleware*. Isto inclui a liberdade de escolha sobre a quantidade de elementos apresentados, disposição de componentes HTML e folhas de estilos. Este requisito determina que o leiaute de formulários deve ser um requisito da aplicação e não do *middleware* utilizado para seu desenvolvimento.

Para cada classe de domínio gerenciada, o *middleware* deve atender às quatro requisições HTTP do tipo CRUD, que determinam, cada uma, a execução de uma operação de persistência. Como mostramos na Figura 2, a classe *Categoria* representa um registro em uma tabela e suas agregações representam outros registros em tabelas relacionadas. O *middleware* deve ser capaz de executar as operações pertinentes nas tabelas corretas de acordo com os dados recebidos. Além disso, o *middleware* deve ser capaz de propagar as consultas (conforme definido no requisito 1) em objetos agregados ao objeto de domínio gerenciado, no caso da figura, *Cliente* e *Telefone*.

1.3 Resultados Esperados e Contribuições

Espera-se construir um *middleware* que proporcione as seguintes facilidades no desenvolvimento de aplicações *web*:

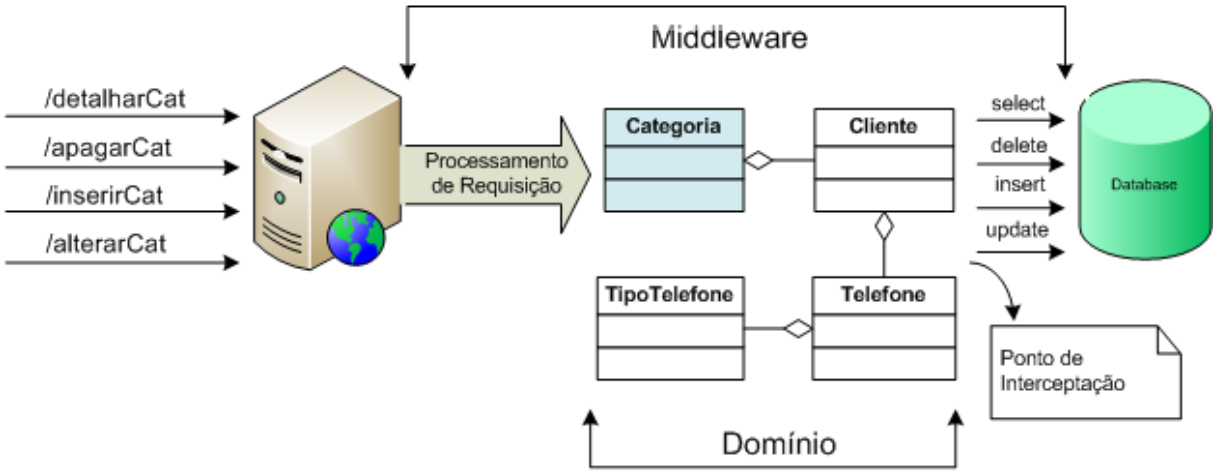

Figura 2: Dados enviados pelas requisições determinam a instanciação dos objetos de domínio envolvidos. Estes objetos são preenchidos com os dados recebidos e utilizados para a criação e execução automáticas de instruções SQL. O *middleware* deve prover pontos de interceptação para a execução de operações de domínio, caso necessárias, entre o processamento de requisições e a persistência.

- Modelar o gerenciamento de dados presentes em formulários HTML por meio das operações que definem seu ciclo de vida em repositórios: inserir, detalhar, alterar e excluir.
- Proporcionar a diminuição na quantidade de código escrito para implementação de aplicações *web* que gerenciam a persistência de dados.
- Definir uma arquitetura padrão que tenha baixa erosão e facilite a manutenção de aplicações.

Espera-se também que objetos de uma classe de domínio possam ser gerenciados pelo *middleware* com a implementação de:

1. Uma classe de domínio que represente um conjunto de dados em uma tabela de um repositório do tipo relacional.
2. Uma página dinâmica JSP com um leiaute qualquer.
3. Configurações que permitam:

   A associação entre requisições HTTP e classes de domínio.

A associação entre classes de domínio e tabelas em um repositório de dados relacional.

Por fim, esperamos verificar que aplicações desenvolvidas com o *middleware* são escaláveis, confiáveis e com boa performance e bom desempenho de acordo com a "Regra dos 8 Segundos"[4] (SUBRAYA, 2006, p. 5).

1.4 Metodologia de Desenvolvimento

O processo de desenvolvimento será baseado na metodologia ICONIX (COLLINS-COPE; ROSENBERG; STEPHENS, 2005). Dentro desta metodologia, algumas atividades não serão realizadas por não se adequarem ao desenvolvimento do *middleware*[5]. Assim, o processo de desenvolvimento será definido de acordo com as seguintes atividades:

1. Elaboração dos requisitos para a construção do *middleware*, já definidos no objetivo deste trabalho.
2. Criação dos diagramas de classes que representam a arquitetura proposta.
3. Criação de diagramas de seqüência que ilustram as interações entre os objetos que compõem a solução.
4. Implementação.
5. Testes que compreendem a criação e execução de casos de teste unitários (caixa-branca) e a execução de testes de estresse a fim de medirmos o desempenho da solução. Neste caso, o teste deverá ser executado em uma aplicação desenvolvida com a solução final.

A documentação resultante do processo de modelagem arquitetural, definições estruturais e interações será elaborada com a utilização da linguagem UML (GROUP, 06.02.2007).

4 Determina que aplicações devem responder a requisições antes do tempo máximo de espera de um usuário que é de 8 segundos.
5 As atividades de análise de Robustez e a definição de Requisitos Comportamentais ou Casos de Uso não serão realizadas

1.5 Organização do Trabalho

O restante deste trabalho está dividido em cinco capítulos e dois apêndices: estado da arte (Capítulo 2), apresentação da solução proposta (Capítulo 3), avaliação de desempenho (Capítulo 4), conclusão (Capítulo 5), manual da solução proposta (Apêndice A) e diagramas UML (Apêndice B).

2 Estado da Arte

2.1 Introdução

Procuramos analisar projetos Java que provessem, além de uma arquitetura para o desenvolvimento de aplicações *web*, a implementação dos quatro requisitos definidos no objetivo deste trabalho. Nossa pesquisa detectou a existência dos seguintes projetos: Struts, Spring, Tapestry, Webwork, Stripes, Slingshot, Expresso, Wicket, Jaffa, Rife, Millstone, Canyamo, Folium, Bishop, TeaServlet, jStatemachine, Scope, Chiba, Genie, JFormular, Dinamica, Baritus, OXF, Maverick, Jucas, Barracuda, wingS, jZonic, Warfare, Macaw, JBanana, Melati, Xoplon, WebOnSwing, JPublish, Verge, OpenEmcee, Smile, Jeenius, Dovetail, Japple, Nacho, Click, Cocoon, SOFIA, JATO, Niggle, Shocks, Bento, Turbine, JWarp, Cameleon, Helma, Cassandra, GWT, Ruby on Rails e JSF.

Porém, utilizamos alguns critérios para reduzir o escopo deste capítulo:

- **Quantidade de referências ao projeto em artigos e sítios especializados em programação de aplicações JavaEE**: Os projetos menos expressivos e que, por este motivo, não serão comentados são: Millstone, Canyamo, Folium, Bishop, TeaServlet, jStatemachine, Scope, Chiba, Genie, JFormular, Dinamica, Baritus, OXF, Jucas, Barracuda, jZonic, Warfare, Macaw, Melati, Xoplon, WebOnSwing, JPublish, Verge, OpenEmcee, Smile, Jeenius, Dovetail, Japple, Nacho, Click, JATO, Niggle, Shocks, Bento, Cameleon, Helma, Cassandra.

- **Documentação oferecida**: Alguns projetos não possuem documentação, outros não descrevem sua arquitetura em detalhes dificultando a pesquisa. Estes projetos são: Jaffa, Espresso, Wicket, Rife, Maverick, wingS, JBanana, SOFIA, Turbine, JWarp, Wicket e GWT.

- **Uso da linguagem Java**: Rails é um arcabouço para desenvolvimento de aplicações *web* baseado na linguagem Ruby. Um interpretador para a linguagem Ruby foi desenvolvido em Java, trata-se do projeto JRuby que gera bytecodes escritos na lin-

guagem Ruby para serem executados em máquinas virtuais Java. Existe uma forma de utilizar o Rails juntamente com o JRuby, porém a versão disponível do JRuby (0.9.2) ainda contém bugs e sua utilização em produção não é recomendada.

- **Uso de persistência**: Java Server Faces é uma especificação para criação de componentes visuais em aplicações *web* na plataforma Java. Existem diversas implementações disponíveis, dentre elas, Tomahawk, Myfaces, IceFaces, Richfaces e ADF Faces. Este levantamento do estado atual da arte propõe mostrar que ainda não existe implementação para a solução definida no objetivo deste trabalho. Nesta, um de seus requisitos é a execução automática de operações de persistência e a especificação JSF (ou as implementações citadas) não aborda persistência de dados.

Dos projetos restantes, Ford (FORD, 2004) e Johnson (JOHNSON; HOELLER, 2004) destacam quatro que consideram como o estado da arte em desenvolvimento de aplicações *web* para a plataforma Java: Struts, Spring, Tapestry e Webwork. Serão estes quatro os trabalhos estudados em maior detalhe neste capítulo. É importante destacar que estes projetos ainda estão em desenvolvimento e constante evolução.

Os projetos foram avaliados considerando a implementação completa, parcial ou não implementação de cada um dos critérios apresentados em detalhes no objetivo deste trabalho e enumerados a seguir:

1. *Modelar o atendimento de requisições que determinam a execução de operações de persistência*
2. *Processar requisições HTTP*;
3. *Disponibilizar pontos de interceptação*;
4. *Permitir a construção de formulários com leiaute livre.*

2.2 Struts

O Struts é uma implementação de código aberto do modelo *Model-View-Controller* (GAMMA et al., 1995) para desenvolvimento de aplicações *web* na plataforma Java. A

implementação do modelo MVC é representada em sua arquitetura pelos seguintes componentes:

- Modelo: objetos da classe *ActionForm* que representam o conjunto de dados apresentados em um formulário HTML;
- Visão: páginas JSP;
- Controle: classes que implementam o atendimento a requisições HTTP.

As requisições atendidas por aplicações desenvolvidas com o Struts determinam a execução de ações. As ações solicitadas devem estar implementadas em métodos de subclasses da classe *Action* (classes de ações). Cada requisição que solicita a execução de uma ação pode transportar dados de um formulário submetido. Neste caso, os dados recebidos juntamente com uma requisição são transferidos automaticamente para um objeto da classe *ActionForm*. Este objeto representa, do lado do servidor, os dados do formulário submetido. O *ActionForm* preenchido, é disponibilizado para a implementação da ação solicitada. Em seguida, a requisição é encaminhada para uma página JSP que deverá gerar a página HTML de resposta com os dados do *ActionForm* (figura 3).

O Struts possui um arquivo de configuração do tipo XML, processado na inicialização de cada aplicação. Este arquivo contém definições de ações, de *ActionForms* e de possíveis fontes de dados utilizadas pela aplicação. As ações definidas no arquivo possuem os seguintes parâmetros:

- A URL atendida;
- A classe que implementa a ação;
- As possíveis páginas de resposta;
- O nome da configuração de um *ActionForm* que será preenchido com os dados recebidos pelas requisições;

A seguir a avaliação das funcionalidades do Struts de acordo com os requisitos definidos no objetivo:

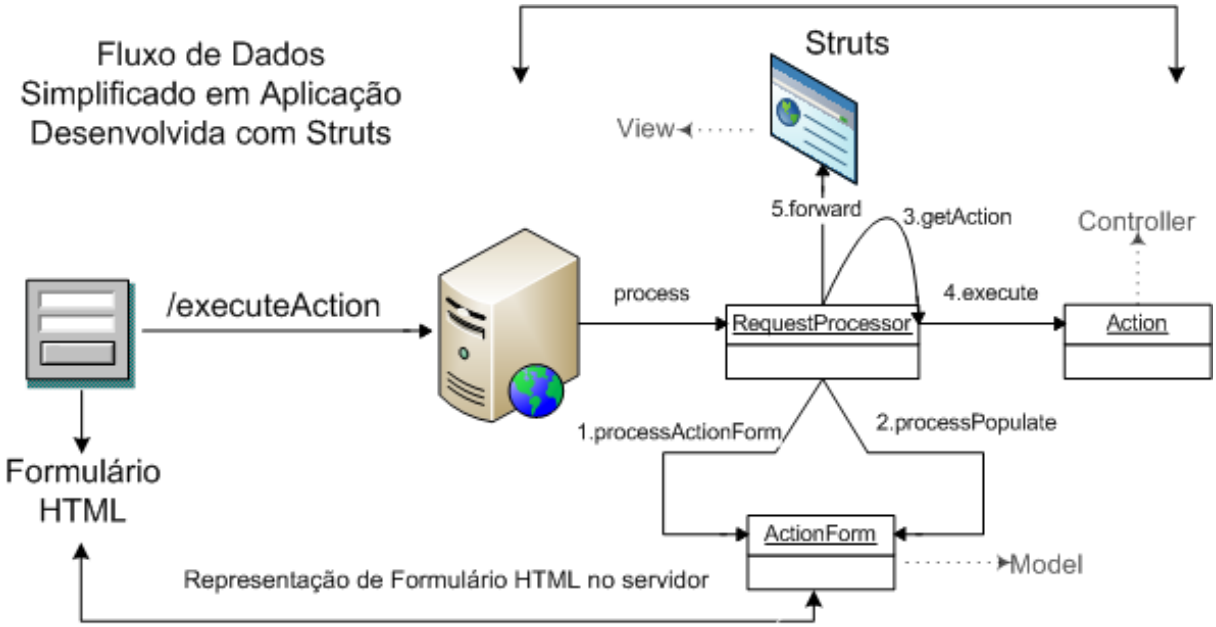

Figura 3: Diagrama de colaboração para a execução de uma determinada ação em uma aplicação desenvolvida com o Struts. Um formulário é submetido com uma URL que solicita a execução de uma ação no servidor (/executeAction). O Struts processa a requisição, preenche um ActionForm com os dados recebidos do formulário e executa a ação implementada na classe Action. Ao final, o fluxo é redirecionado para uma página de resposta.

1. *Modelar o atendimento de requisições que determinam a execução de operações de persistência*: o Struts limita-se a prover apenas um modelo de dados para a implementação de ações executadas no atendimento de requisições.

2. *Processar requisições HTTP*: O Struts processa requisições HTTP com algumas considerações. Transforma os dados recebidos no formato texto em tipos de dados definidos na declaração do "objeto de formulário" (da classe*ActionForm*). Preenche o *ActionForm* associado à ação com os dados do formulário submetido [1]. No entanto, para o caso de agregações de dados, somente associa automaticamente valores em agregações do tipo Um-Para-Muitos caso os atributos que definam as associações sejam definidos como vetores ou coleções de dados que implementam a interface *java.util.List* e contenham dados no formato texto. Isto limita a utilização de código legado que pode definir a associação como um objeto da interface *java.util.Set*.

3. *Disponibilizar pontos de interceptação*: O Struts não implementa a persistência

1 Para tanto, os atributos do *ActionForm* devem ser definidos com os mesmos nomes dos elementos HTML presentes no formulário submetido.

de dados. Portanto, não provê pontos de interceptação entre o atendimento de requisições e a execução de operações de persistência.

4. *Permitir a construção de formulários com leiaute específico*: O Struts permite a construção de páginas JSP com leiaute livre.

O Struts é incompleto em relação ao gerenciamento de objetos de domínio. Limita-se apenas a padronizar e controlar o atendimento a requisições HTTP. Não implementa comunicação com mecanismos de persistência e não especializa tipos de ações de acordo com as requisições recebidas. No entanto, possui uma modelagem que oferece uma redução do hiato existente entre o cliente e o servidor determinando a utilização de "objetos de formulários" para representar o estado de um formulário submetido após o processamento de requisições HTTP.

2.3 Webwork

O Webwork é um *middleware* de código aberto para o desenvolvimento de aplicações *web* na plataforma Java. Sua arquitetura implementa o padrão de projeto *Command* (GAMMA et al., 1995) ao definir um modelo para o atendimento de requisições.

Ao comparar sua arquitetura com a de arcabouços que implementam o modelo MVC, por exemplo Struts e Spring MVC, é possível identificar o objeto que implementa um comando realizando dois papéis, o da camada de controle e da camada de modelo. A camada de visão continuaria sendo representada por páginas JSP.

No Webwork, cada requisição atendida determina a criação de uma nova instância de um comando. Um comando é uma subclasse de *Action* e deve conter um método com sua implementação. É possível também definir atributos em comandos que serão preenchidos automaticamente (com os dados do formulário submetido[2]) antes da sua execução. Após a execução de um comando, os seus atributos poderão ser acessados pela página JSP que gerará a página HTML de resposta (Figura 4).

Os comandos do Webwork devem ser configurados em um arquivo XML processado na inicialização de aplicações. Neste arquivo, para cada comando, devem estar definidas:

2 Os nomes dos atributos devem ser os mesmos dos elementos contidos no formulário HTML submetido.

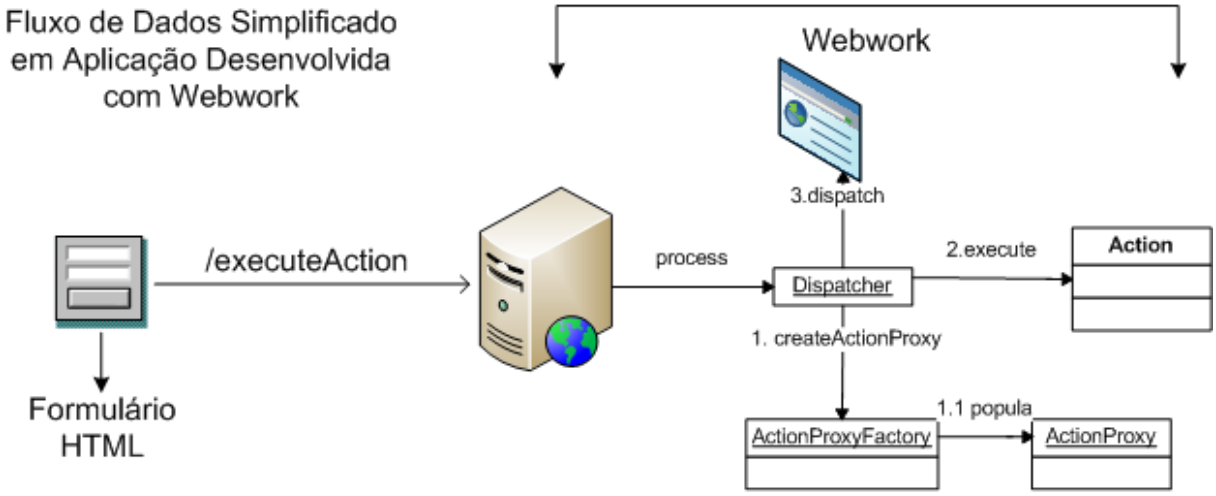

Figura 4: A figura ilustra o atendimento de requisições com o Webwork. A cada requisição atendida uma nova instância de *Action* é criada. Caso este objeto contenha atributos, estes serão preenchidos com os dados do formulário submetido. Após a execução do comando, o fluxo de dados é direcionado para a página JSP de resposta.

- A URL das requisições atendidas pelo comando;
- A classe que implementa o comando e que será instanciada a cada requisição atendida;
- As possíveis páginas de resposta que poderão ser retornadas após a execução do comando.

A seguir, a avaliação das funcionalidades do *middleware* de acordo com os requisitos definidos no objetivo:

1. *Modelar o atendimento de requisições que determinam a execução de operações de persistência*: O Webwork não provê a implementação de comandos que executem automaticamente operações de persistência utilizando objetos de domínio. No entanto, o Webwork fornece integrações com mecanismos de persistência Objeto-Relacionais, como por exemplo, o Hibernate.
2. *Processar requisições HTTP*: O processamento de requisições do Webwork é completo. Classes que implementam comandos podem definir objetos de domínio como

atributos. Assim, o Webwork preenche os objetos de domínio com os dados do formulário submetido antes da execução do comando [3]. O processamento de requisições ainda preenche atributos que representam relacionamentos do tipo Um-Para-Muitos. As classes que definem estes relacionamentos podem ser vetores, coleções de dados ou interfaces do pacote *java.util* da linguagem Java[4].

3. *Disponibilizar pontos de interceptação*: O Webwork provê uma gama de interceptadores para seus comandos. Dentre eles, interceptadores para conversão de tipos de dados, validação, execução de tarefas antes e após a execução de um comando. Porém, o Webwork não implementa comandos que executam operações de persistência. Dessa forma, não provê pontos de interceptação entre o atendimento de requisições e a execução de operações de persistência.

4. *Permitir a construção de formulários com leiaute específico*: O Webwork permite a construção de páginas JSP com leiaute livre.

O Webwork possui mais funcionalidades que o projeto Struts. Oferece meios para integração com mecanismos de persistência, porém não implementa comandos que executam automaticamente operações de persistência envolvendo os dados submetidos.

2.4 Spring

O Spring é um *middleware* de código aberto construído com diversos propósitos. O projeto é bastante amplo e foi dividido em módulos, dentre estes:

- Spring *Model-View-Controller* MVC: Implementa o modelo MVC para desenvolvimento de aplicações *web*.

- Spring *Inversion of Control* (IoC): Gerencia objetos de domínio por meio da instanciação e inserção de dependências.

3 Para tanto, é necessário que os nomes dos elementos contidos no formulário HTML submetido sejam iguais ao nomes dos atributos da classe de comando.
4 Para atributos que definem coleções de dados do pacote java.util, o Webwork permite definir em um arquivo de configuração para o comando as classes dos objetos que devem ser instanciados, preenchidos e associados a cada coleção de dados.

- Spring *Object-Relational Mapping* (ORM): Abstrai detalhes tecnológicos de diferentes mecanismos de persistência objeto-relacionais.

Juntos, estes módulos fornecem um alicerce para o desenvolvimento de aplicações *web* que executam operações de persistência em repositórios de dados. Apenas os módulos Spring MVC e Spring IoC são relevantes a este trabalho, pois estão relacionados com os requisitos definidos no objetivo.

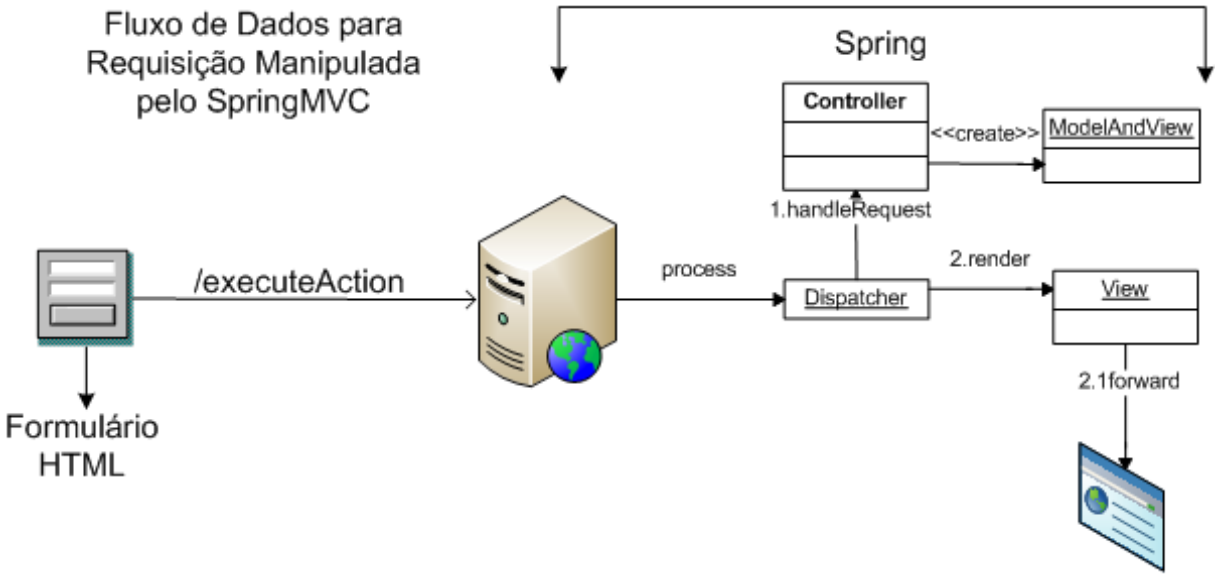

Figura 5: Diagrama de Colaboração para o atendimento de requisições do Spring MVC. Um objeto da classe *Dispatcher* direciona as requisições para objetos da classe *Controller*. Estes preenchem objetos da classe *ModelAndView* com os dados do formulário submetido e executam as ações solicitadas. Por fim, o fluxo de dados é direcionado pelo *Dispatcher* para as páginas JSP de resposta.

2.4.1 Spring MVC

Ao implementar o modelo MVC o Spring define três estruturas básicas:

- *Controller*: Representa a camada de Controle. No Spring MVC, as requisições atendidas por aplicações determinam a execução de ações, implementadas em métodos de classes que estendem a classe *Controller*.

- *Model*: Representa a camada de modelo e é responsável pela transferência de dados entre as camadas de controle e visão. Implementada por objetos que, em tempo de execução, devem ser preenchidos com os dados disponíveis na página de resposta.

- *View*: Representa a camada de Visão e deve extrair dados de um objeto da camada de modelo. Esta camada é implementada por páginas JSP.

O Spring MVC define um objeto da classe *Dispatcher* responsável por receber as requisições HTTP atendidas pela aplicação (Figura 5). O *Dispatcher* recebe as requisições atendidas e as delega para o objeto da classe *Controller* que deve executar a ação solicitada (Figura 5). O *Controller* cria uma instância da classe *ModelAndView* que é preenchida com os dados submetidos pelo formulário HTML e executa a ação. Por fim, o *Dispatcher* delega a um objeto da classe *View* a apresentação da página HTML de resposta.

2.4.2 Spring IoC Container

O Spring IoC gerencia objetos do domínio de aplicações. Ele deve ser configurado em um arquivo XML e, para cada objeto gerenciado, devem estar definidos:

- A classe do objeto de domínio gerenciado.
- Modo de obtenção do objeto. Por exemplo, instanciação ou obtenção por meio de um servidor de nomes.
- Dependências dos objetos de domínio que também serão gerenciadas e inseridas por meio de métodos ou construtores.
- Modo de obtenção de dependências de um objeto de domínio. Por exemplo, instanciação ou obtenção através de um servidor de nomes.
- Escopo de um objeto obtido. Por exemplo, *um único por contêiner* ou instanciado sempre que solicitado.
- Modo de inserção de dependências para um objeto. Por exemplo, construtor ou método.

A utilização do Spring IoC em aplicações caracteriza a inversão de controle dos objetos de domínio que passam a ser gerenciados pelo container. Em aplicações *web*, o Spring IoC pode ser utilizado para definição arquitetural determinando o modo de acesso às camadas inferiores à camada de Controle (responsável pelo atendimento de requisições).

O Spring MVC é integrado ao Spring IoC e, como resultado, classes que implementam ações possuem acesso à fábrica de objetos do Spring IoC.

O Spring será avaliado, de acordo com os requisitos definidos no objetivo, para identificarmos se implementa as funcionalidades propostas:

1. *Modelar o atendimento de requisições que determinam a execução de operações de persistência*: O Spring MVC não implementa ações que gerenciam a persistência de objetos de domínio. O projeto limita-se a prover um modelo de dados para a implementação destas ações, além de uma integração com mecanismos de persistência Objeto-Relacionais, tais como Hibernate e *Java Data Objects* (JDO).

2. *Processar requisições HTTP*: O Spring MVC processa requisições HTTP e permite a criação de classes que implementam ações e definem um objeto de domínio para ser preenchido com os dados do formulário submetido. Neste caso, por convenção, os atributos do objeto de domínio devem ter os mesmos nomes dos parâmetros recebidos pela requisição.

3. *Disponibilizar pontos de interceptação*: O Spring provê interceptadores para as operações executadas durante o atendimento a requisições HTTP. Aplicações podem implementar interceptadores para validação, segurança e execução de tarefas antes e após a execução de uma ação. O Spring, no entanto, não implementa ações que executam operações de persistência. Ainda assim, não provê pontos de interceptação entre o atendimento de requisições e a execução de operações de persistência.

4. *Permitir a construção de formulários com leiaute específico*: O Spring permite a construção de páginas JSP com leiaute livre.

O Spring é atualmente o *middleware* que provê o maior número de funcionalidades para facilitar o desenvolvimento de aplicações na plataforma Java. Propõe diminuir a complexidade na construção das funcionalidades mais comuns em aplicações JavaEE: acesso a dados, processamento de requisições HTTP, gerenciamento de objetos de domínio, segurança, dentre outras. Porém, o Spring MVC não implementa ações genéricas que utilizam objetos do domínio de aplicações para a execução automática das quatro operações de persistência: *incluir, obter, alterar e excluir*.

2.5 Tapestry

Tapestry é um *middleware* de código-aberto para a implementação de aplicações JavaEE. Diferente dos projetos orientados ao atendimento de requisições HTTP, o Tapestry aplica, no ambiente *web*, padrões conhecidos de programação para interfaces de usuário em aplicações do tipo *desktop*. Portanto, ao invés do atendimento a requisições HTTP propõe o atendimento à eventos disparados por componentes em páginas HTML. Um evento é uma requisição HTTP disparada por um componente de página. Estes componentes podem ser: um formulário, um calendário, uma tabela, etc. Sua arquitetura determina a separação total entre código Java e páginas HTML, dividindo a implementação de cada página HTML em 3 artefatos:

- *Template* ou Modelo HTML: Arquivo que contém apenas marcações HTML. As partes dinâmicas (por exemplo, linhas em uma tabela) são marcadas com atributos do tipo jwcid=“” dentro de marcações da linguagem HTML. Em tempo de execução, as marcações “jwcid” são substituídas pelo código HTML dinamicamente gerado.

- Especificação de Página: Trata-se de um arquivo de configuração do tipo XML que define os componentes que geram código HTML dinâmico para um determinado *template* HTML. Cada componente produz um trecho de código HTML que, em tempo de execução, é incluído na página HTML gerada. Os componentes podem especificar tabelas, formulários, calendários e é possível criar componentes personalizados para cada aplicação. Os componentes também podem ser aninhados.

- Classe de Implementação: uma classe que define os objetos da camada de modelo e os *listeners* para a página. Um *listener* implementa um tratador para um evento disparado por uma página HTML. Um evento pode ser, por exemplo, a submissão de um formulário ou um redirecionamento de página.

A arquitetura do Tapestry não faz referência a nenhum modelo tradicional porém, se pudéssemos compará-la ao modelo MVC, teríamos uma página HTML que representaria a Visão, uma Classe de Implementação que representaria as camadas de Controle e Modelo e uma especificações de página e aplicação onde são mantidas as configurações.

No Tapestry, a cada requisição recebida de um novo cliente HTTP, um objeto da classe *ApplicationEngine* é criado (Figura 6). Este objeto lê a especificação da aplicação (arquivo XML onde estão declaradas as páginas disponíveis), acessa a especificação da página solicitada e identifica qual sua classe de implementação e o seu *template* HTML. Em seguida, instancia as classes de implementação da página e de cada possível componente e executa os métodos adequados para obter o conteúdo que deve estar presente na página de resposta. Por fim, o Tapestry insere o conteúdo dinâmico no *template* HTML e apresenta ao cliente HTTP.

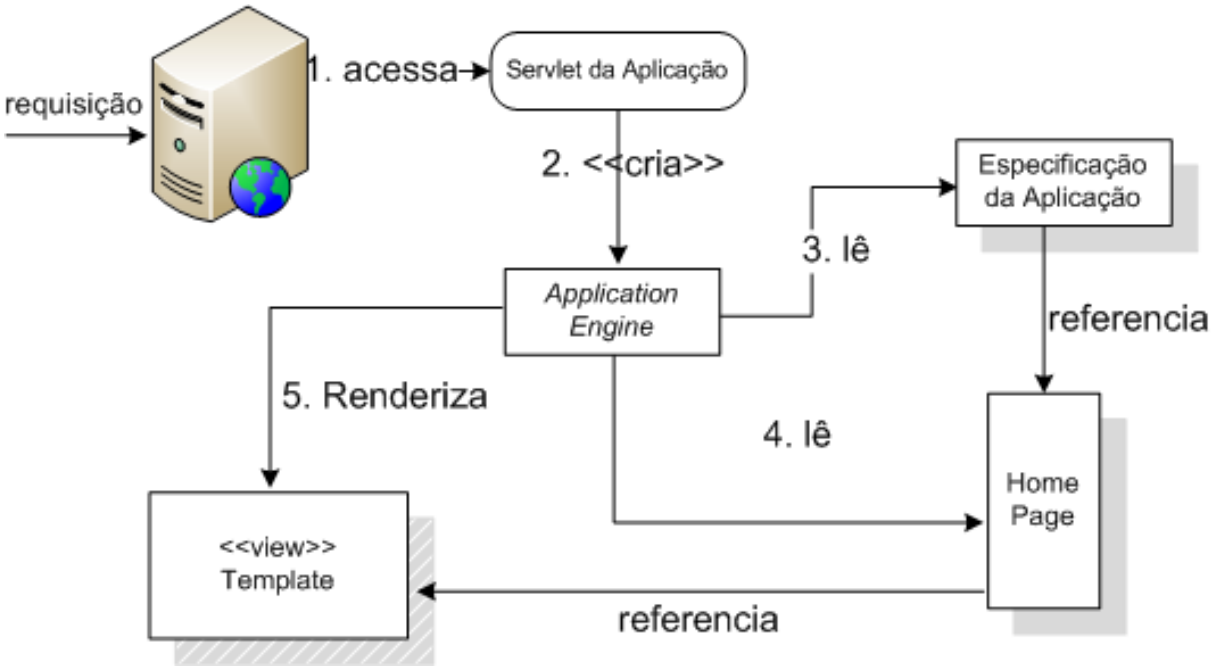

Figura 6: O servlet de aplicação cria um *application engine* que lê as especificações para a aplicação e apresenta os resultados

Listeners e eventos disparados por páginas HTML devem ser implementados em classes que contém implementações para eventos disparados por páginas HTML. Estas, podem conter atributos que representam os dados manipulados pelas páginas HTML e que são automaticamente preenchidos com dados de formulários submetidos (processamento de requisições). As classes de implementação são utilizadas para a geração de páginas HTML e mantidas em memória para o atendimento de qualquer evento que possa ser disparado pela página que está no cliente. Isto significa que, para cada página HTML apresentada em um cliente, uma instância de sua Classe de Implementação é mantida no servidor. Estas instâncias são removidas assim que os clientes solicitarem outras páginas,

tiverem sua sessão expirada ou saírem da aplicação.

Manter em memória as instâncias de Classes de Implementação para cada página HTML apresentada no cliente pode comprometer a escalabilidade das aplicações. Em situações onde a escalabilidade é um requisito primordial, utilizar o Tapestry significa mais investimentos em infra-estrutura se comparado à utilização do Struts, Spring MVC e Webwork.

A seguir, a avaliação do Tapestry de acordo com os requisitos apresentados no objetivo para identificação das funcionalidades apresentadas:

1. *Modelar o atendimento de requisições que determinam a execução de operações de persistência*: O Tapestry não implementa eventos que determinam a execução de operações de persistência, limitando-se a prover um mecanismo para a criação de eventos disparados por componentes de páginas HTML, além de uma integração com o mecanismo de persistência Objeto-Relacional Hibernate.

2. *Processar requisições HTTP*: O Tapestry processa requisições HTTP. Converte os dados recebidos como texto em dados definidos por atributos nas classes que implementam tratadores de eventos de páginas. Um objeto de domínio declarado como atributo em uma destas classes é preenchido com dados de um formulário submetido para então ser disponibilizado para acesso por métodos que implementam tratadores de eventos[5].

3. *Disponibilizar pontos de interceptação*: O Tapestry é totalmente orientado a eventos disparados por componentes de páginas HTML. Define a implementação de *listeners* executados no atendimento a eventos de componentes ou páginas.

4. *Permitir a construção de formulários com leiaute específico*: O Tapestry permite a construção de formulários com leiautes específicos e a criação de páginas em editores visuais HTML.

5 O processamento de requisições do Tapestry considera que elementos de páginas HTML submetidas possuem o mesmo nome dos atributos definidos em Classes de Implementação de páginas para que a associação automática de valores seja realizada com sucesso.

2.6 Outros Projetos

Além dos quatro projetos analisados acima, podemos ainda destacar outros 3 projetos com documentação disponível: Stripes, Slingshot, e Rife. Por serem projetos de menor impacto, será feita apenas uma descrição resumida de cada um.

2.6.1 Stripes

O Stripes é um middleware para o desenvolvimento de aplicações JavaEE. Sua arquitetura implementa o modelo MVC. Durante o processamento de requisições HTTP, instancia e preenche objetos de domínio com dados recebidos por requisições HTTP. No entanto, este processamento não preenche objetos de domínio em relacionamentos do tipo Um-Para-Muitos definidos por interfaces do pacote java.util. O Stripes disponibiliza pontos de interceptação para o atendimento à requisições HTTP e permite a criação de páginas HTML com leiaute livre, porém não modela o atendimento de requisições que determinam a execução de operações de persistência.

2.6.2 Slingshot

O Slingshot é um *middleware* para o desenvolvimento de aplicações JavaEE. Sua arquitetura também implementa o modelo MVC. O Slingshot implementa o atendimento de requisições e a execução automática de operações de persistência envolvendo os dados contidos em objetos de domínio. No entanto, esta funcionalidade requer um acoplamento entre objetos de domínio e classes do *middleware*. Esta abordagem é ruim, pois não promove a utilização de código legado na construção da funcionalidade. Além disso, o Slingshot não implementa de forma genérica ações que gerenciam a persistência de objetos de domínio preenchidos com dados de requisições HTTP. O seu processamento de requisições é o mesmo do Struts, não preenche objetos de domínio contidos em coleções de dados do pacote java.util que sejam diferentes de *java.util.List*. As páginas HTML podem ser elaboradas com quaisquer leiautes.

2.6.3 Rife

O Rife é um arcabouço que implementa o modelo MVC e é utilizado para o desenvolvimento de aplicações JavaEE. Possui um módulo chamado Rife/CRUD que processa automaticamente requisições HTTP, cria objetos de domínio com os dados recebidos e executa automaticamente operações de persistência. Esta funcionalidade requer a criação de classes de meta-dados para mapeamento Objeto-Relacional. Isto aumenta em muito o tempo de aprendizado e desenvolvimento com o *middleware* uma vez que a linguagem Java já provê uma especificação de mapeamento Objeto-Relacional chamada Java Persistence API. O leiaute de páginas que contém formulários é determinado pelo *middleware* e, portanto, restrito ao uso de folhas de estilos.

2.7 Conclusão

A pesquisa realizada pôde levantar dois modelos arquiteturais utilizados nos projetos avaliados: arquiteturas orientadas à execução de ações ou comandos como consequência do atendimento de requisições HTTP e arquiteturas orientadas à execução de eventos disparados em componentes presentes em páginas HTML. Nenhum dos projetos avaliados implementa todos os requisitos definidos no objetivo deste trabalho. O resultado deste estudo pode ser conferido na Tabela 1.

	Modelo de Dados	Processamento HTTP	Interceptação	Leiaute Livre
Struts	Não	Parcialmente	Não	Sim
Spring	Não	Sim	Não	Sim
Webwork	Não	Sim	Não	Sim
Tapestry	Não	Sim	Não	Sim
Stripes	Não	Sim	Não	Sim
Slingshot	Não	Parcialmente	Não	Sim
Rife/CRUD	Sim	Parcialmente	Sim	Não

Tabela 1: A tabela apresenta a classificação dos principais projetos estudados levando em consideração os requisitos definidos no objetivo: modelar atendimento de requisições para execução de persistência (requisito 1), processar requisições HTTP (requisito 2), presença de pontos de interceptação (requisito 3), formulários com leiaute livre (requisito 4).

A maioria dos projetos estudados apresenta integração com mecanismos de persistência. Porém, nenhum projeto provê, simultaneamente, a integração entre a camada

web (interface com servidores de aplicações) e a camada de persistência, além de definir uma arquitetura que implemente, de forma genérica, estas operações promovendo o uso de objetos de domínio.

Independente da arquitetura de cada projeto, a maioria poderia ser estendida para a implementação de ações, comandos ou eventos genéricos que recebam dados de formulários HTML submetidos, populem objetos de domínio relacionados e executem operações de persistência automaticamente. Verificamos a existência de projetos que implementam alguns dos requisitos desejados na solução final. Portanto, reutilizamos um destes projetos na definição arquitetural da solução proposta. Embora o projeto Spring seja o mais completo (mais funcionalidades) dentre os estudados, a solução proposta, apresentada no capítulo 3, reutiliza em sua arquitetura o projeto Struts.

O Struts é um projeto pioneiro e utilizado em muitas das aplicações existentes. Sua arquitetura é facilmente extensível e muitos profissionais conhecem ou já o utilizaram em algum projeto. Assim, o Struts foi escolhido para integrar o Foundation por possibilitar que parte das aplicações já existentes possam ter novas funcionalidades desenvolvidas e que estejam de acordo com suas arquiteturas já existentes. Além disso, acreditamos que o conhecimento prévio do Struts diminui consideravelmente seu tempo de aprendizado e facilita sua adoção.

3 Foundation

O Foundation é um *middleware* desenvolvido para a construção de aplicações JavaEE. Seu principal objetivo é definir um modelo de dados para o atendimento de requisições HTTP relacionadas à execução de operações de persistência, além de abstrair detalhes tecnológicos relacionados à comunicação via protocolo HTTP e à comunicação com repositórios de dados relacionais.

A arquitetura definida determina a reutilização de objetos do domínio de aplicações para o gerenciamento da persistência de dados apresentados em formulários. Utilizamos o modelo MVC para separar a criação de páginas da implementação do domínio de aplicações e do processamento de requisições. Para tanto, incorporamos o projeto Struts, já conhecido na comunidade de desenvolvimento de *software* pela sua qualidade e utilizado em larga escala no desenvolvimento de aplicações JavaEE.

O acesso a repositórios de dados é implementado de acordo com a especificação JPA e, por padrão, o Hibernate é utilizado como provedor da implementação. O Foundation permite a utilização de qualquer outra tecnologia de persistência, desde que siga a especificação JPA. A escolha do Hibernate como mecanismo de persistência padrão se deve à sua utilização em larga escala para desenvolvimento de aplicações Java, além de ser uma solução de *software* livre.

3.1 Arquitetura

Seguindo a orientação de Fowler para a criação de arquiteturas em camadas (FOWLER et al., 2002), estruturamos o Foundation em quatro camadas: Acesso a Dados, Domínio, Controle e Visão. Dentro do modelo MVC, as camadas de Acesso a Dados e Domínio compõem o Modelo.

- Acesso a Dados. Esta camada possui as seguintes responsabilidades:
  - Definir a interface para o acesso a dados de repositórios.

  - Implementar o acesso a repositórios relacionais obedecendo a especificação JPA.
  - Abstrair, das camadas superiores, detalhes relacionados à utilização de tecnologias de persistência como gerenciamento de conexões e transações.

- Domínio: Acima da camada de Acesso a Dados. Define e implementa as operações do domínio de acordo com o contexto do Foundation: *create, retrieve, update* e *delete*. Esta camada desacopla o acesso a dados do controle de elementos visuais e provê um mecanismo para interceptação do fluxo de dados e execução de operações específicas ao domínio de cada entidade gerenciada.

- Controle: Acima da camada de Domínio, implementa o controle de componentes visuais e atende requisições HTTP; o que envolve as seguintes responsabilidades:
  - Prover meios para especialização de ações executadas automaticamente.
  - Definir e automatizar o atendimento a requisições HTTP para a execução de operações de persistência envolvendo os dados recebidos.
  - Abstrair detalhes tecnológicos da comunicação entre aplicações e o protocolo HTTP.

- Visão: Acima da camada de Controle, é definida pela implementação de páginas JSP para geração do conteúdo dinâmico de páginas HTML.

No modelo arquitetural proposto, para cada requisição HTTP atendida, dois objetos são responsáveis pelo transporte de dados entre as camadas (Figura 7):

1. *ActionForm*: É a interface entre as camadas de Controle e Visão definida no modelo de dados proposto pelo Struts. Transporta dados nos dois sentidos. Assim, no recebimento de uma requisição, um objeto *ActionForm* é instanciado e preenchido com os dados do formulário submetido. Após a execução de alguma tarefa pela aplicação, o mesmo objeto deve ser preenchido com os dados que deverão estar presentes na página HTML enviada como resposta. Na solução proposta, os objetos *ActionForm* são preenchidos automaticamente com os dados recebidos de requisições HTTP.

2. Objetos de Domínio: São as entidades gerenciadas pelo Foundation e definidas por cada aplicação. Por exemplo, em um determinado domínio, objetos do tipo *Usuario, Aluno, NotaFiscal* cujos atributos definem os conjuntos de dados gerenciados. Estes objetos são reutilizados pelo Foundation para o transporte de dados entre as camadas de Controle, Domínio e Acesso a Dados. Após o recebimento de uma requisição HTTP, a camada de Controle preenche um objeto do tipo ActionForm com os dados recebidos e, sem seguida, um objeto de domínio que será encaminhado às camadas que compõem o Modelo. Após a execução de alguma tarefa pela aplicação, este mesmo objeto será utilizado para preencher o *ActionForm* que deverá alimentar a página HTML enviada como resposta.

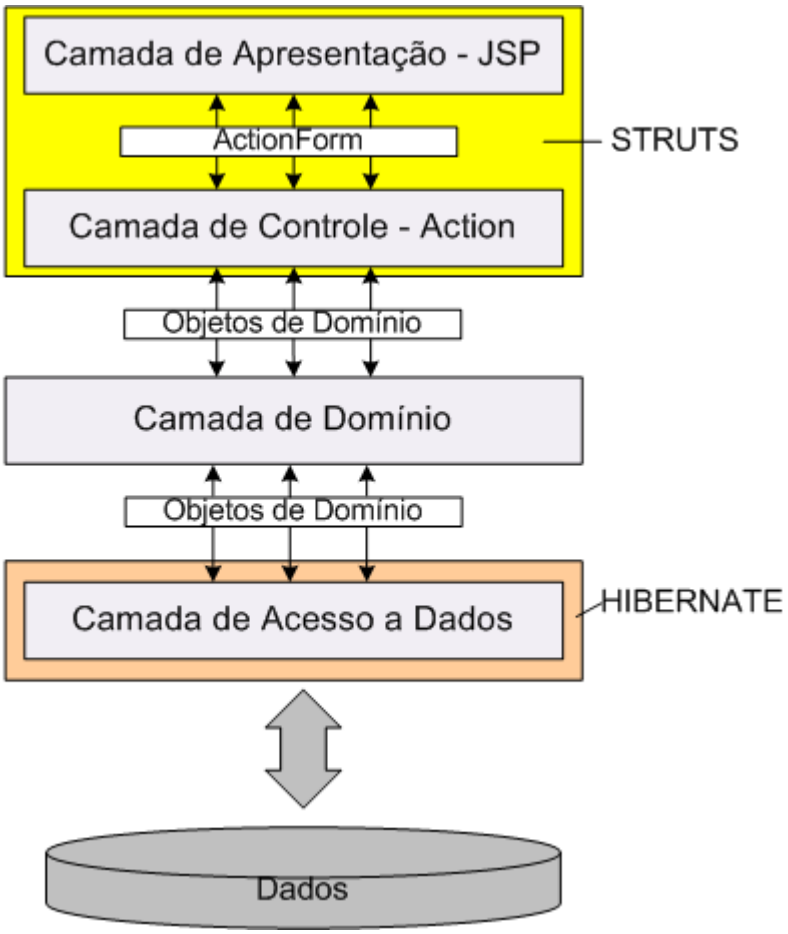

Figura 7: Arquitetura do Foundation

3.2 As camadas do Foundation

As camadas de Controle e Acesso a Dados instanciam e preenchem objetos de domínio e suas dependências definidas por relacionamentos do tipo Um-Para-Um ou Um-Para-Muitos. A camada de Controle instancia e preenche objetos de domínio contendo dados recebidos de clientes HTTP e a camada de Acesso a Dados instancia e preenche os objetos com dados recebidos de repositórios. Estas camadas determinam assim a inversão de controle (JOHNSON; HOELLER, 2004) sobre os objetos de domínio e, por consequência, promovem o baixo acoplamento entre o domínio de aplicações e as camadas.

3.2.1 Controle

A camada de Controle é responsável pela transferência de dados entre as camadas de Domínio e Apresentação e pela abstração de detalhes tecnológicos referentes à comunicação entre aplicações e clientes do protocolo HTTP. No modelo de dados da camada, toda requisição HTTP atendida determina a execução de uma ação relacionada aos dados submetidos. Estas ações podem ser uma das operações *Create, Retrieve, Update e Delete* definidas na interface *CRUD*. A classe *CRUDImpl* fornece uma implementação padrão para estas ações e pode ser estendida para a implementação de outras ações de acordo com requisitos de cada aplicação.

Uma ação é disparada no atendimento a requisições HTTP especificadas por uma URL[1]. Cada ação é implementada dentro de um método. Os métodos ou ações dentro de uma mesma classe devem manipular dados definidos por uma mesma classe do domínio. Por exemplo, uma classe que contenha ações relacionadas a uma entidade *Usuario* pode implementar métodos com assinaturas do tipo *incluir, detalhar, processarInadimplentes, inativar*, todas executando operações que envolvem os dados definidos pela classe *Usuario*, do domínio da aplicação.

A configuração de uma ação é definida em um arquivo XML (*struts-config.xml*) que define para cada ação:

- A URL de acesso a ação por clientes do protocolo HTTP.

1 Por exemplo, uma URL terminada em /logout.do determina a execução da ação *logout* implementada em uma classe de ações.

- A configuração do objeto da classe *ActionForm*, que será preenchido com os dados recebidos por cada requisição HTTP.
- A classe e o método que contém a implementação da ação.
- A classe do domínio que define o conjunto de dados manipulado pela ação.
- Um interceptador. O interceptador é uma classe Java utilizada para especialização dos algorítmos que determinam as ações definidas pela interface *CRUD*. Esta configuração é opcional.
- A classe da camada de Domínio que será utilizada durante a execução da ação. Esta configuração é opcional e deve ser utilizada caso seja necessário especializar a camada de Domínio para a ação. A especialização pode tornar-se necessária em casos onde a aplicação deve executar regras de negócio antes do acesso ao repositório de dados, podendo ou não alterar o fluxo padrão de dados.
- A classe da camada de Acesso a Dados utilizada para a execução da ação. Esta configuração é opcional e utilizada caso seja necessário especializar a camada de Acesso a Dados para a ação.

A Figura 8 apresenta o diagrama UML das principais entidades que compõem a camada de Controle. A camada de Controle do Struts é estendida com os objetivos de melhorar o processamento de requisições HTTP, implementar um controle de exceções próprias do Foundation e automatizar o atendimento a requisições que solicitam a execução das quatro operações de persistência definidas no objetivo deste trabalho.

A camada de Controle é definida por nove artefatos principais:

- *FoundationRequestProcessor*: sobrescreve o processamento de requisições HTTP implementado pelo Struts. Acrescenta a funcionalidade de preencher automaticamente com os dados recebidos, atributos de objetos da classe *ActionForm* definidos pelas interfaces e classes do pacote *java.util*[2]. Esta classe é de uso interno ao Foundation e não deve ser referenciada por classes de aplicações.

2 A interface java.util.Collection define coleções de dados na plataforma Java. As coleções mantém referências a objetos do tipo java.lang.Object. Porém, o Foundation permite definir, no arquivo XML de configuração de ações, qual a classe do objeto que deve ser adicionado a cada coleção

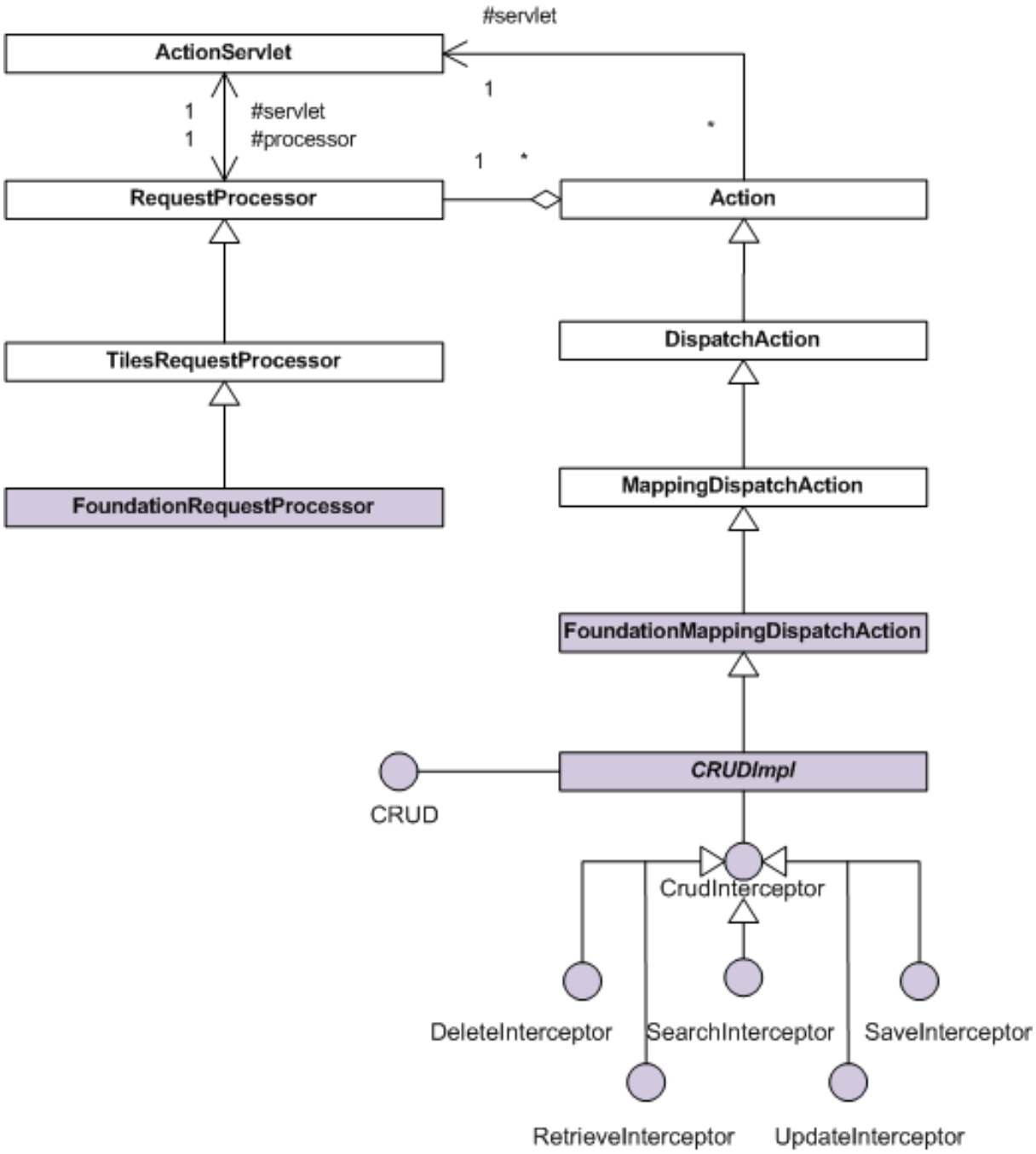

Figura 8: Diagrama de classes simplificado da camada de controle. As classes representadas em tom claro compõem o Struts. As classes em tom escuro pertencem ao Foundation.

- *FoundationMappingDispatchAction*: esta classe implementa um mecanismo de exceções próprio do Foundation além de gerenciar detalhes relacionados à especificação JPA.

- *CRUD*: Esta interface define as ações que executam operações de persistência com os dados recebidos.A interface *CRUD* atualmente não possui utilidade prática, porém

no futuro pode servir para diferenciar internamente outras classes de ações[3].

- *CRUDImpl*: Classe que implementa as ações definidas em *CRUD*. Provê pontos de interceptação para especialização das ações.
- *CrudInterceptor*: Interface que define interceptadores para as ações definidas em *CRUD*.
- *SaveInterceptor*: A interface define dois pontos de interceptação para especialização da ação *save*, implementada em *CRUDImpl*.
- *RetrieveInterceptor*: Interface que define dois pontos de interceptação para a especialização da ação *retrieve*, implementada em *CRUDImpl*.
- *UpdateInterceptor*: Interface que define dois pontos de interceptação para a especialização da ação *update*, implementada em *CRUDImpl*.
- *DeleteInterceptor*: Interface que define dois pontos de interceptação para a a especialização da ação *delete*, implementada em *CRUDImpl*.

Ao atender requisições, a camada de Controle deve carregar a configuração da ação (definida no arquivo XML) solicitada, preencher objetos da classe *ActionForm* com os parâmetros recebidos e encaminhá-los como parâmetro para a as ações solicitadas. Entre o recebimento de uma requisição e a execução de uma ação implementada na camada de Controle, uma série de operações são sempre executadas:

1. A URL de cada requisição que chega ao servidor de aplicações é definida no seguinte padrão: `http://<servidor>:<porta>/<nomeDaAplicação>/<ação>`. O método *encontraDefinicaoDaAcao* utiliza a parte final da URL (<ação>) para encontrar a definição da ação a ser executada. O objeto devolvido por este método, da classe *ActionMapping*, contém todos os elementos presentes na definição da ação e, dentre eles, a definição do *ActionForm* que deverá ser preenchido com os dados recebidos, o método que contém a ação, sua classe e as páginas de resposta que poderão ser devolvidas[4].

3 Por exemplo, diferenciar internamente uma classe de ações do tipo CRUD de uma classe de ações para gerenciamento de relatórios.
4 Esses dados fazem parte da configuração de cada ação definida no arquivo struts-config.xml.

2. O *ActionForm* associado à definição da ação solicitada poderá ou não ser criado a cada requisição. Por padrão, para cada requisição atendida, uma nova instância é criada[5]. O método *processaActionForm* faz esta verificação e retorna a instância de um objeto *ActionForm*.

3. Após a conversão dos tipos de dados (todos recebidos como texto) o *ActionForm* é preenchido[6].

4. Uma classe que implementa uma ou mais ações deve estender a classe *Action*. A primeira requisição que solicita a execução de uma de suas ações (métodos) determina sua instanciação e a inclusão de sua referência em um *cache* para acessos futuros[7].

5. A classe *RequestProcessor* delega a execução da ação solicitada para um método específico de uma determinada classe, ambos definidos no arquivo XML de configuração de ações.

O diagrama de execução destas operações pode ser visto na Figura 9.

3.2.1.1 CRUD

A interface *CRUD* define ações que executam as operações de persistência *save*, *retrieve*, *update* e *delete*. Estas ações, implementadas na classe *CRUDImpl*, são genéricas, uma vez que manipulam conjuntos de dados definidos por quaisquer classes do domínio de aplicações. Em uma aplicação, o objeto de domínio gerenciado por cada ação é definido no arquivo XML de configuração de ações.

As ações "genéricas" tornam-se específicas ao serem utilizadas em uma aplicação qualquer. Parte da implementação genérica de uma ação pode ser especializada por meio da implementação de interceptadores. Para tanto, é necessário definir na configuração de

5 Na configuração de cada ação é possível definir diferentes escopos para o *ActionForm* utilizado. Exemplos de escopos são: requisição, sessão, aplicação.
6 O *ActionForm* pode conter agregações do tipo Um-Para-Um ou Um-Para-Muitos. A conversão dos tipos de dados recebidos como texto para tipos definidos pela linguagem Java, a instanciação de objetos agregados e atribuição dos valores recebidos para seus atributos internos é chamada de processamento de requisição que é executado dentro do método que preenche o *ActionForm*.
7 Esta é uma abordagem que visa aumento da escalabilidade de aplicações em situações de grande volume de requisições. Portanto, as classes que implementam ações não são *thread-safe*.

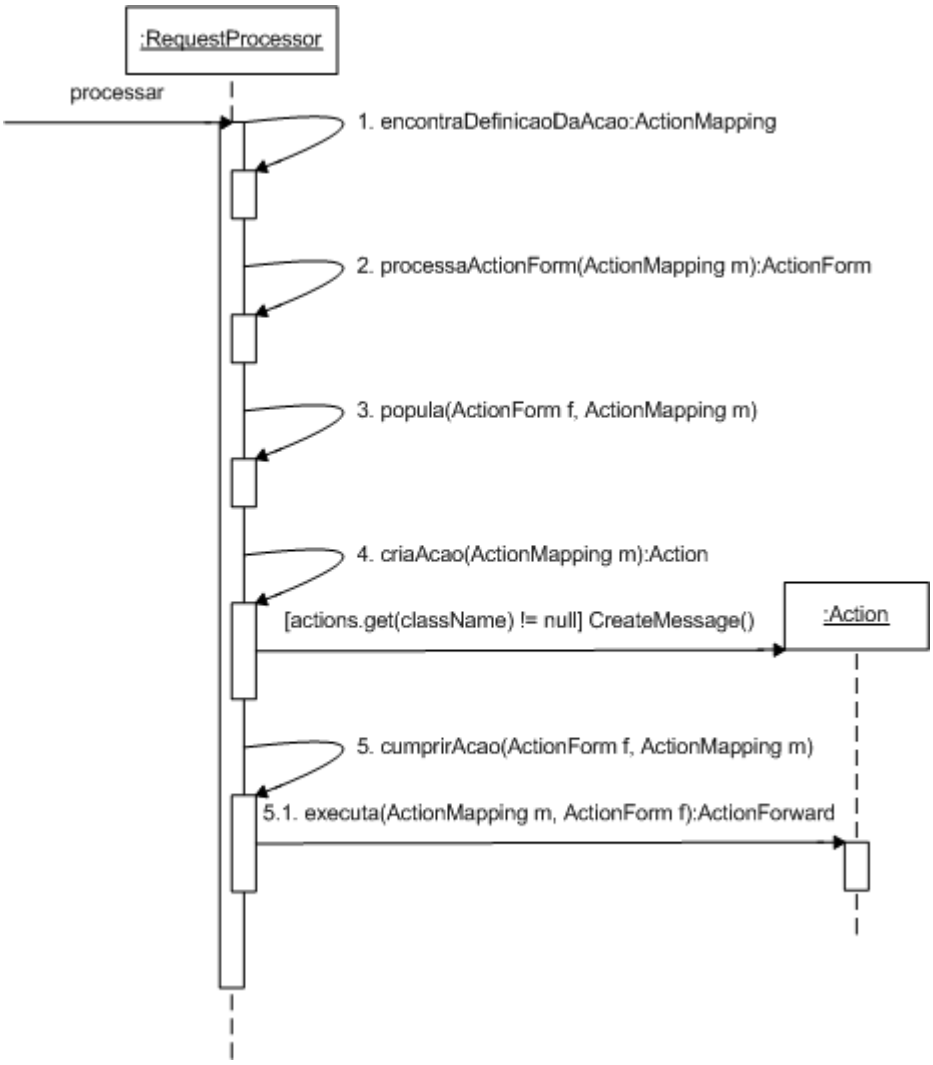

Figura 9: Diagrama de sequência simplificado da camada de controle. O diagrama de sequência não apresenta detalhes tecnológicos e contém nomes de métodos traduzidos do inglês para o português.

cada ação a classe que implementa alguma das interfaces do pacote *br.foundation.mvc.crud .interceptor*.

A definição de uma ação especifica em uma aplicação consiste apenas na edição do arquivo de configuração, na codificacao de um JSP de resposta e de uma classe de domínio persistente[8] que define o conjunto de dados gerenciado.

Interceptadores foram criados para cada ação implementada em *CRUDImpl*. Assim, para a ação de inclusão (*save*) existe o interceptador *SaveInterceptor* e analogamente para as outras ações. Os interceptadores são interfaces cujos nomes são os nomes das ações

8 Contém anotações da especificação JPA que definem a persistência de uma classe Java.

acrescidos do sufixo *Interceptor* (Figura 8).

Um interceptador contém dois métodos: um método com o mesmo nome da ação interceptada e outro método com sufixo *InitializeForm* que deve ser utilizado para inicialização de campos na página de resposta como, por exemplo, um elemento do tipo combo que deverá conter registros de países. Neste caso, o método deve conter uma instrução que executa a busca dos registros de países e associa o resultado ao *ActionForm* (o *ActionForm* referencia os dados apresentados em página). Classes que implementam interceptadores não são *thread-safe* e, portanto, não devem conter atributos de instância.

As operações de persistência estão implementadas na camada de Acesso a Dados, abaixo da camada de Domínio. Visando aumentar a usabilidade, todos os interceptadores para as ações definidas em *CRUD* possuem métodos com nomes iguais aos nomes das ações. Estes recebem como um de seus parâmetros um objeto da camada de Domínio. Quaisquer regras de negócio que devam ser executadas antes das operações de persistência devem ser inseridas em classes da camada de Domínio.

Ao atender uma requisição HTTP que solicita a execução de operações de persistência do tipo CRUD (Figura 10), é responsabilidade da camada de Controle:

- Processar requisições, preencher objetos da classe *ActionForm* e disponibilizá-los como parâmetro ao método que implementa a ação.
- Transferir dados entre o objeto da classe *ActionForm* (preenchido com os dados recebidos) e a entidade gerenciada (e suas possíveis agregações) definida na configuração da ação. Isso é feito no caso das ações *save, update e delete*.
- Transferir dados entre o objeto da entidade gerenciada (e suas possíveis agregações) para o objeto da classe *ActionForm* que irá preencher a página de resposta.

A classe *CRUDImpl* implementa também um controle de estado para formulários. Este controle é responsável por permitir que a ação *update* consiga apagar automaticamente registros agregados que não são retornados pelo cliente quando um formulário de alteração de dados é submetido. Este controle de estado para formulários é implementado de forma a tornar válida a seguinte interação:

- Usuário executa a ação *retrieve* e obtém um formulário de alteração de dados.

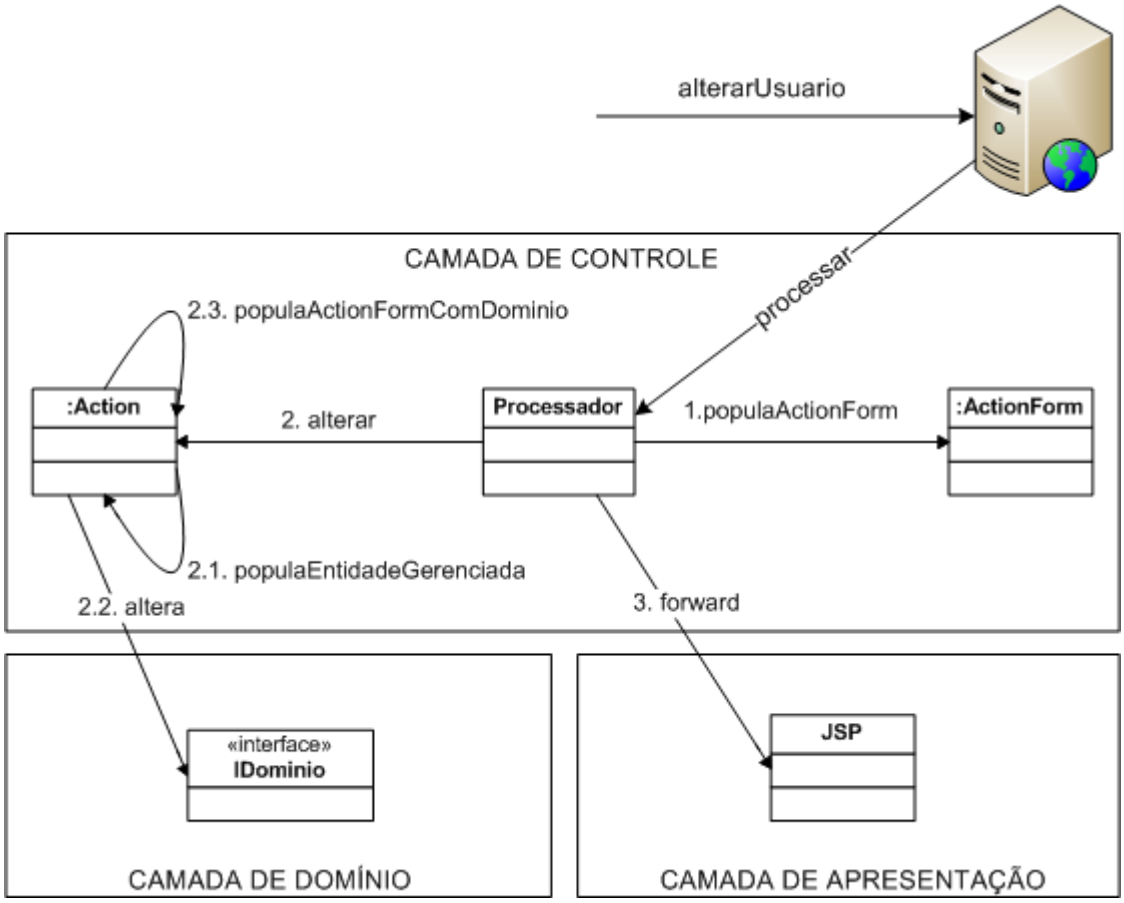

Figura 10: Diagrama de colaboração para uma ação de alteração de dados solicitada ao *Foundation*. A camada de Controle preenche a entidade gerenciada com os dados recebidos pela requisição e armazenados no *ActionForm*. A figura não apresenta os parâmetros dos métodos. O apêndice apresenta um diagrama completo.

- O Foundation mantém em memória (sessão do usuário no servidor de aplicações) os objetos utilizados para geração do formulário e atribui um identificador único para o conjunto de dados armazenado em memória e para a página retornada ao cliente.
- O cliente altera os dados e submete o formulário.
- O Foundation processa a requisição, popula um objeto da classe *ActionForm* e depois popula as entidades gerenciadas configuradas para a ação de alteração de dados. Em seguida, o Foundation identifica os dados utilizados na geração do formulário submetido (que estavam armazenados em sessão) e os utiliza para fazer comparações com o conjunto de dados recebido do cliente. Verificações do tipo "existia antes, não existe agora" são responsáveis pela geração automática de operações de exclusão de dados para dados agregados a um conjuto de dados gerenciado.

Este controle de estado para formulários permite identificar conjuntos de dados que devem ser excluídos do repositório de dados, permitindo a criação de formulários com diversas agregações de dados definidas por relacionamentos do tipo Um-Para-Muitos e representadas visualmente por níveis (formulários do tipo Mestre-Detalhe-SubDetalhe). Esta abordagem facilita a implementação dos formulários, pois identifica as operações de exclusão de dados sem a necessidade de inclusão de controles especiais em páginas JSP.

O Apêndice A descreve como deve ser realizada a configuração de ações do tipo CRUD implementadas na classe *CRUDImpl*.

3.2.2 Visão

A camada de Visão é responsável pela geração de conteúdo dinâmico e pela definição de leiautes de páginas HTML. As páginas HTML são geradas automaticamente pelo servidor de aplicações JavaEE ao processar páginas dinâmicas JSP. O transporte de dados entre as camadas de Controle e Visão é realizado por objetos da classe *ActionForm*. Esta classe faz parte da camada de Modelo na implementação do modelo MVC proposta pelo Struts e incorporada ao Foundation.

Objetos da classe *ActionForm* devem ser configurados no arquivo *struts-config.xml*. Nesta configuração devem ser especificados os dados manipulados pelo *ActionForm* e apresentados em uma página HTML. Estes dados devem ser definidos com um *nome* e um *tipo*. O *nome* deve ser o mesmo nome do atributo da classe de domínio gerenciada. O *tipo* é a classe que define o dado. Caso o tipo seja uma coleção de dados, a configuração do *ActionForm* deverá conter um outro elemento que definirá a classe dos objetos contidos nos atributos que são coleções de dados.

O conjunto de dados de um objeto da classe *ActionForm* é o mesmo conjunto de dados presente no formulário HTML e não necessariamente o conjunto definido pelo objeto de domínio gerenciado por uma ação. Dados auxiliares podem ser apresentados em página, assim como um conjunto de dados menor que o definido pela classe de domínio. Por este motivo, para qualquer ação implementada, a camada de controle transfere automaticamente dados contidos em atributos de objetos de domínio para um objeto da classe *ActionForm* antes do fluxo de dados chegar às páginas JSP.

Esse é um algorítmo específico que promove a reutilização de objetos de domínio respeitando a definição do modelo MVC proposto pelo Struts. Os conjuntos de dados auxiliares, em um formulário HTML, podem ser preenchidos no *ActionForm*, durante a execução de uma determinada ação, por meio da implementação de interceptadores fornecidos para cada ação genérica.

O Foundation implementa ações genéricas, porém não define nenhum critério para a criação de formulários HTML porque o preenchimento de campos na página de resposta deve ser implementado para cada ação específica com a criação de uma página JSP. Esta abordagem permite que o leiaute de formulários atenda a requisitos das aplicações e não do *middleware* utilizado.

O Apêndice A descreve como deve ser feita a configuração de ações e *ActionForms* em aplicações.

3.2.3 Domínio

A camada de Domínio corresponde ao Modelo na implementação da arquitetura MVC implementada pelo Foundation. Sua responsabilidade é prover meios para interceptação de fluxo e execução de regras de negócio anteriores à execução de operações de persistência. Esta camada também funciona como uma fachada entre a camada de Controle e a camada de Acesso a Dados

A interface *IDomain* define as operações *save, retrieve, update e delete* que são implementadas pela classe *DomainImpl*. Aplicações podem especializar a camada de Domínio para uma determinada ação que gerencia uma entidade especifica. Neste caso, aplicações devem estender a classe *DomainImpl* e sobrescrever algum de seus métodos (ou criar uma classe que implemente *IDomain*) para definir a execução de regras de negócio antes do fluxo de dados chegar ao repositório de dados.

A classe *CRUDImpl*, por padrão, sempre direciona o fluxo de dados para a classe *DomainImpl*. Uma determinada ação de uma aplicação qualquer pode ser configurada no arquivo de especificação de modo a determinar qual classe específica da camada de Domínio será utilizada durante o atendimento de requisições. Esta configuração também pode ser feita pela criação de uma classe de ação que estenda a classe *CRUDImpl* e

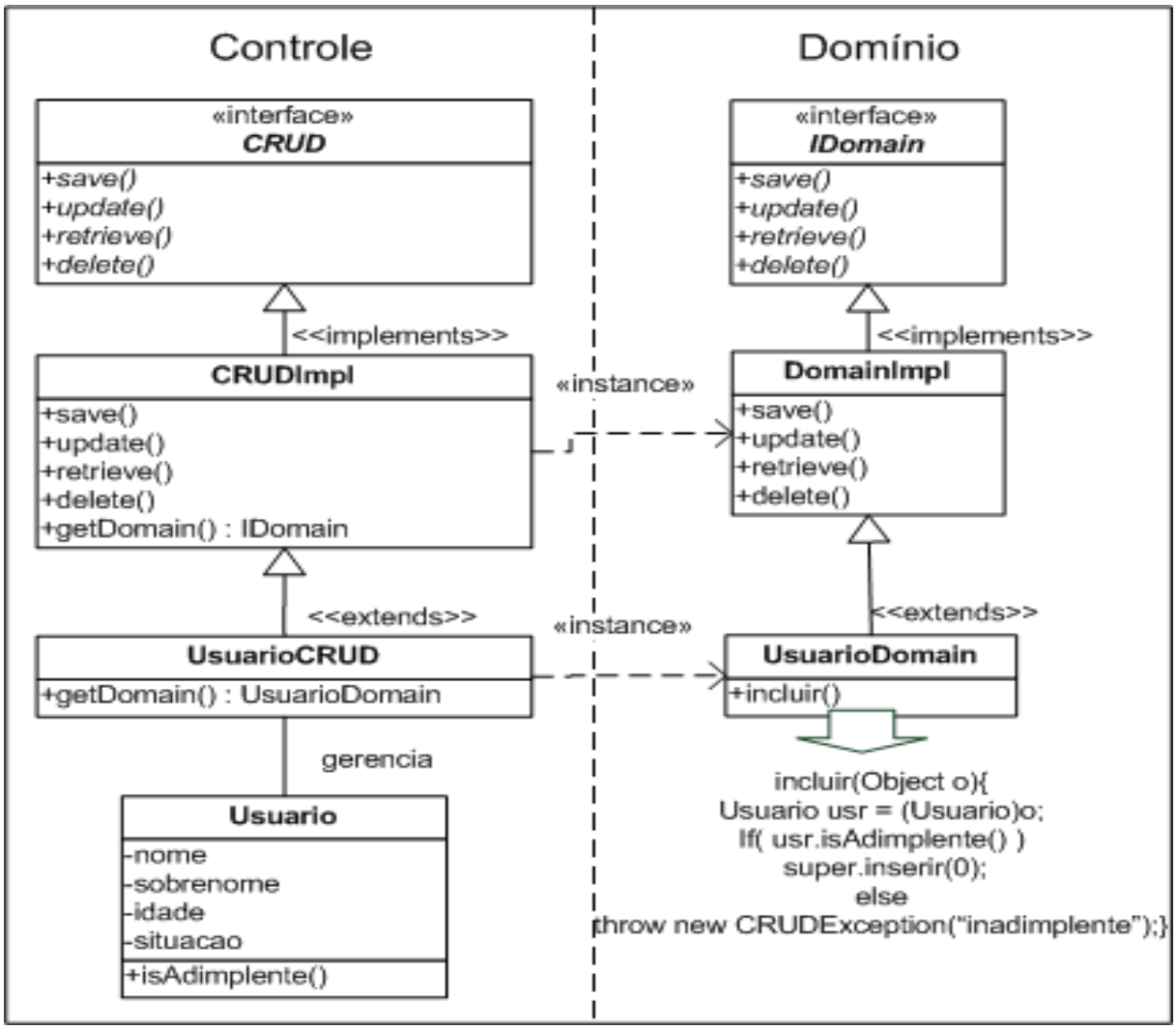

Figura 11: A camada de Domínio fornece pontos de interceptação anteriores à execução de operações de persistência envolvendo os dados de uma classe gerenciada. No exemplo acima, novos usuários são inseridos no repositório somente em caso de adimplência.

sobrescreva o método *getDomain* (Figura 11). Vale lembrar que a configuração para a camada de Domínio feita de forma declarativa possui precedência sobre a configuração feita em subclasse de CRUDImpl, caso as duas configurações existam para uma mesma ação.

Ao contrário da camada de Controle, as instâncias da camada de Domínio (e Acesso a Dados) são *thread-safe* (Isso significa que elas podem manter estado por meio de atributos internos, pois o ciclo de vida destes objetos termina com a execução de cada requisição HTTP).

A camada de Domínio delega a execução das operações de persistência para a camada

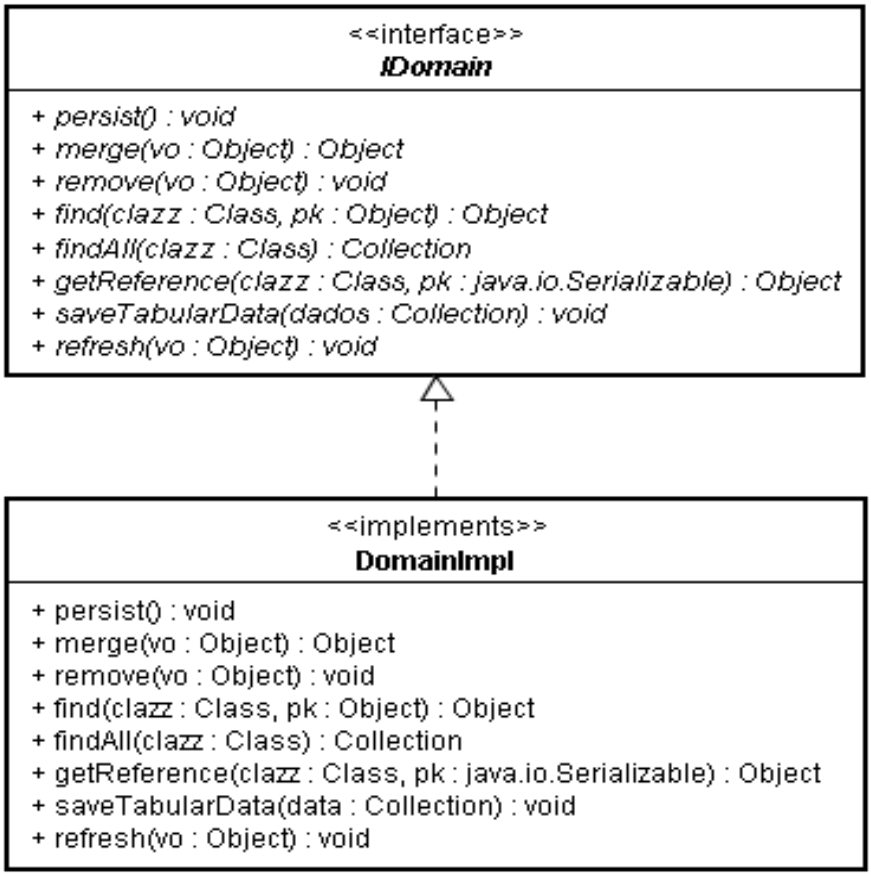

Figura 12: Diagrama de classes da camada de Domínio que contém dois artefatos: a interface *IDomain* e a classe *DomainImpl*. Os nomes dos métodos estão em conformidade com a especificação JPA.

de Acesso a Dados. Por padrão, uma instância da classe *DaoImpl*[9] é disponibilizada como atributo da classe *DomainImpl* sempre que o Foundation, internamente, cria uma nova instância de *DomainImpl*. No arquivo de configuração é possível personalizar qual classe implementa o acesso a dados para cada ação executada. Para tanto, basta inserir o nome da classe que implementa a interface Dao ou estende a classe DaoImpl como uma propriedade de nome `dao` na configuração da ação desejada. O Apêndice A mostra como configurar ações em arquivo XML e também como especializar as camadas do Foundation para uma determinada ação.

9 Dao é a abreviação do padrão de projeto *Data Access Object*

3.2.4 Acesso a Dados

A camada de Acesso a Dados implementa a comunicação entre aplicações e repositórios de dados. Sua arquitetura é definida por um objeto que implementa o padrão de projeto *singleton*, uma interface e uma classe de implementação (Figura 13). A principal finalidade da camada é determinar a abstração de detalhes tecnológicos na comunicação entre aplicações e repositórios de dados, além de prover uma padronização no acesso a dados por meio da especificação JPA. Como resultado, a camada abstrai o acesso a conexões e transações de dados e executa operações de persistência do tipo CRUD sem a prévia definição de *queries* para entidades persistentes [10].

A principal estrutura que representa a camada de Acesso a Dados é a interface Dao (*Data Access Object*). Esta interface define o contrato dos objetos de acesso a dados utilizados pelo Foundation. Estes são responsáveis pela abstração de detalhes relacionados ao acesso a dados, como por exemplo, o gerenciamento de conexões e transações. A interface Dao é definida de acordo com o padrão de projeto *DAO* (ALUR et al., 2003).

A arquitetura da camada de Acesso a Dados permite a construção de aplicações que utilizam um ou mais provedores [11] de persistência diferentes. Por padrão, a distribuição do Foundation provê o acesso a dados com a utilização do provedor Hibernate. Qualquer provedor que implemente a especificação JPA pode ser utilizado em substituição ou em conjunto com o Hibernate, possibilitando o acesso a mais de um repositório durante uma mesma transação. O acesso a dados por meio de tecnologias de persistência que não sejam do tipo Objeto-Relacional inviabiliza a utilização da classe DaoImpl. Nestes casos, é necessária uma nova implementação da interface Dao, não fornecida pelo Foundation.

A classe *JpaEntityManagerFactory* (Figura 13) abstrai detalhes relacionados à especificação JPA e gerencia o acesso a fábricas de objetos da classe *EntityManager* (estes determinam o ponto de acesso a um repositório). Existem alguns padrões para gerenciamento de um *EntityManager* (especificação JPA) dentro de um projeto; e o padrão implementado pelo Foundation chama-se *session-per-request*. Neste padrão, o ciclo de vida de um *EntityManager* é atrelado ao escopo de uma requisição, que é o mesmo de

10 Uma entidade persistente é uma classe que contém anotações da especificação JPA para mapeamento Objeto-Relacional.
11 Um provedor é um projeto que implementa a especificação JPA. Por exemplo: Hibernate, TopLink.

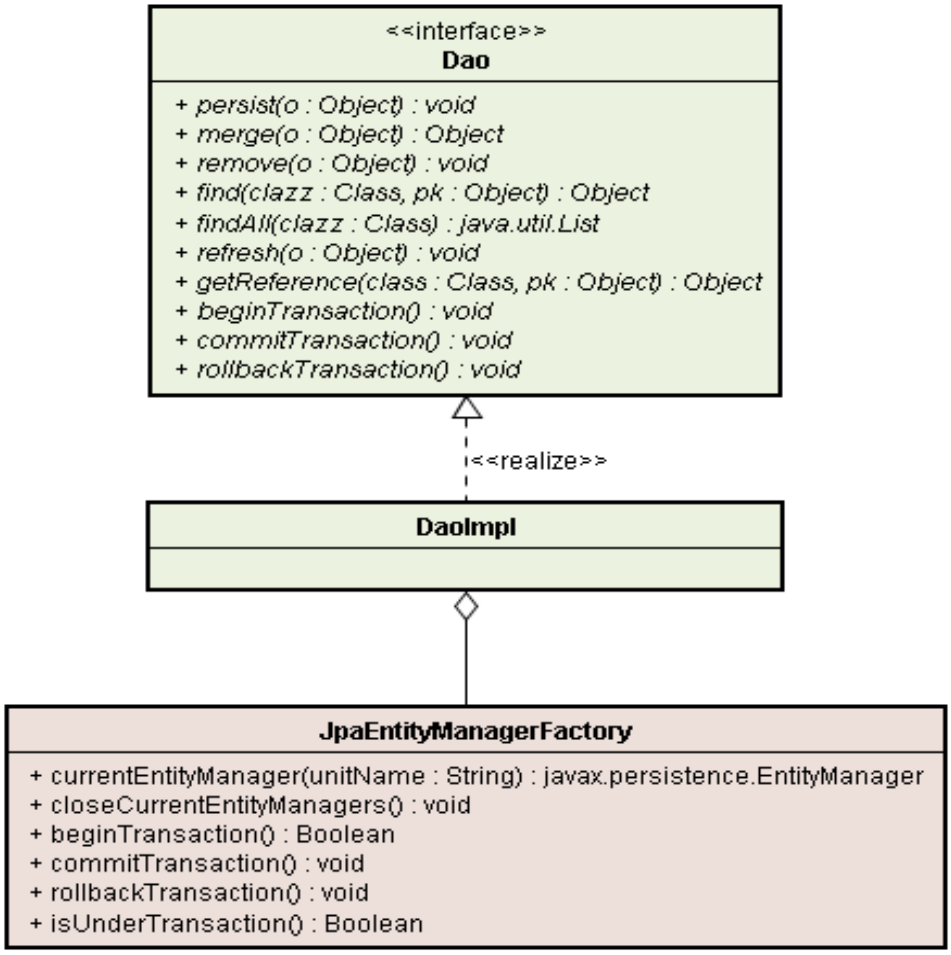

Figura 13: Arquitetura da camada de persistência. Na figura, a classe *DaoImpl* não apresenta os métodos implementados da interface *Dao*.

uma *thread* em cada aplicação.

A classe *JpaEntityManagerFactory* garante ainda a criação de apenas um objeto da classe *EntityManager* para acesso a um determinado repositório dentro de uma mesma *thread*[12].

Todos os objetos persistentes utilizados na execução de operações em repositórios de dados permanecem associados ao *EntityManager* até que este seja finalizado. Objetos criados pelo *EntityManager*, após buscas, podem ter seus relacionamentos carregados por *lazy-loading* desde que o *EntityManager* não esteja fechado ou finalizado. Portanto, o Foundation recomenda a finalização de *EntityManagers* ao final de cada requisição.

12 Em uma mesma *thread* é possível executar operações em duas bases de dados e, no caso, utilizando dois *EntityManagers* durante uma mesma transação de acordo com a especificação *Java Transaction API*. Neste caso, mais de uma fábrica de *EntityManagers* deve ser configurada.

Esta abordagem é conhecida como *open-session-in-view* que viabiliza o *lazy-loading* de relacionamentos em páginas JSPs. Portanto, o Foundation provê a implementação de um filtro para requisições HTTP (classe *FilterJpa*) que deve ser utilizado em todas as aplicações. Este filtro serve apenas para executar o método *closeCurrentEntityManagers* ao final de cada requisição atendida.

3.3 Implementação de aplicações com o Foundation

A implementação de um formulário HTML que submete dados para a execução de quatro operações em repositórios (incluir, detalhar, alterar e excluir) consiste na execução de quatro atividades. Os formulários implementados gerenciam conjuntos de dados definidos em classes do domínio de aplicações. As ações de persistência implementadas pela classe *CRUDImpl* podem executar operações de persistência em cascata para dados definidos em classes agregadas a uma classe de domínio gerenciada. As agregações podem ser definidas por relacionamentos do tipo *Um-Para-Um ou Um-Para-Muitos*.

Um formulário HTML simples gerencia os dados definidos por uma classe de domínio. Um formulário complexo gerencia os dados definidos por uma classe de domínio e suas agregações, definidas em relacionamentos do tipo Um-Para-Muitos. Um formulário complexo pode representar agregações do tipo Um-Para-Muitos que podem também conter dados agregados em relacionamentos do tipo Um-Para-Muitos. Estes são formulários complexos denominados Mestre-Detalhes-SubDetalhes e possuem ao menos três níveis.

O Foundation permite a construção de formulários complexos de qualquer nível com a execução da mesma quantidade de atividades necessárias para a implementação de um formulário simples. Portanto, com a execução de quatro atividades é possível implementar um fomulário do tipo Mestre-Detalhes-SubDetalhes que gerencia três classes de domínio agregadas em relacionamentos do tipo Um-Para-Muitos ao invés de serem executadas doze atividades para a implementação de três diferentes formulários que gerenciam os mesmos dados; caso comum na implementação da maioria das aplicações JavaEE.

3.3.0.1 Ações do tipo CRUD

São necessárias quatro atividades para a implementação de um formulário HTML (Figura 14).

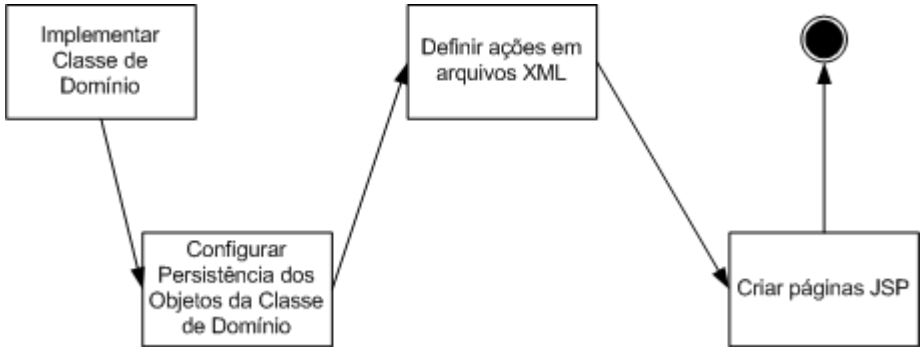

Figura 14: Fluxo de atividades para implementação de ações do tipo CRUD com o Foundation

A seguir é apresentada a descrição de cada atividade de acordo com a ordem em que devem ser executadas:

1. Implementar classe de domínio: Implementação da classe e, caso necessário, classes agregadas, que define o conjunto dos dados presentes no formulário e que abstrai parte do domínio da aplicação.

2. Configurar a Persistência dos objetos da classe de domínio: Utilizar anotações[13] da linguagem Java5 e a especificação de persistência Java Persistence API (JPA) para declarar na classe de domínio propriedades relacionadas à sua persistência[14].

3. Definir ações em arquivo XML: Configurar a implementação do modelo MVC para o atendimento às URLs (a serem definidas) que solicitarão à aplicação a execução de quatro operações de persistência nos dados submetidos.

4. Criar páginas JSP. Deve ser criada uma ou mais páginas JSP de acordo com a quantidade de formulários utilizados para as ações. Uma única página JSP pode gerar uma página HTML contendo formulários para inclusão, detalhamento por chave-primária, alteração e exclusão de dados.

13 Metadados interpretados pela máquina virtual Java que, no caso, permitem a configuração do mecanismo de persistência.
14 Por exemplo, associar a classe a uma tabela do repositório e seus atributos a colunas.

O Anexo A mostra por meio de um exemplo prático como devem ser executadas as quatro atividades apresentadas.

3.3.0.2 Ações específicas

Ações diversas podem ser criadas com a execução das cinco atividades descritas a seguir:

1. Implementar classe de domínio: Implementação da classe que define o conjunto dos dados gerenciados.
2. Criar classe de ações: Criar uma classe de ações que estenda a classe *Foundation-MappingDispatchAction*.
3. Implementar a ação. A classe de ações criada na atividade anterior deve conter um método com a implementação da ação.
4. Definir ações em arquivo XML: Configurar a implementação do modelo MVC para o atendimento à URL que solicita a execução da ação.
5. Criar página JSP: Deve ser criada uma página JSP que, em tempo de execução, gerará a página HTML enviada como resposta à requisição HTTP.

O Anexo A detalha, por meio de um exemplo, cada uma das quatro atividades necessárias para a implementação de ações específicas em aplicações.

4 Avaliação de Desempenho

4.1 Introdução

Codificamos uma aplicação para avaliar o desempenho das principais funcionalidades implementadas pelo Foundation. Os resultados obtidos foram analisados para medir o desempenho em carga máxima de acordo com a regra dos 8 segundos[1] (SUBRAYA, 2006, p. 5).

De acordo com Subraya (SUBRAYA, 2006, p. 14) os testes de performance são divididos em:

- Testes de Carga: A carga é definida pelo número de usuários concorrentes utilizando a aplicação. O teste é utilizado para determinar se a aplicação é capaz de executar várias atividades pré-estabelecidas e realizadas concorrentemente por diferentes usuários.

- Testes de Resistência: Este teste determina a confiabilidade de uma aplicação. Ele é executado em uma aplicação sob carga normal ou sob estresse (muitos usuários concorrentes). O foco deste teste é a sua duração. Ele deve ser executado por horas ou até mesmo dias. A aplicação deve ser capaz de atender requisições durante oscilações no número de usuários por minutos ou horas e mostrar que o gerenciamento de recursos (por exemplo, conexões com repositórios) é realizado corretamente. Este teste permite diagnosticar o consumo excessivo de memória dentro da aplicação podendo fazer com que a mesma negue-se a atender requisições.

- Testes de Estresse: Este teste identifica a carga máxima suportada pelo sistema até que o mesmo falhe ou degrade drasticamente. Neste tipo de teste o tempo de resposta não é importante uma vez que o sistema será testado com uma carga maior que a esperada.

1 A regra define este tempo como sendo o máximo esperado pela grande maioria dos usuários por uma página solicitada em um sistema *web*.

- Testes de Pico: Estes testes são conduzidos para testar um sistema rapidamente em pequenos períodos de tempo. A carga do sistema deve ser alterada em intervalos constantes para testar se o sistema não bloqueia ou aborta ao atender picos de usuários. Ao contrário do teste de estresse, as cargas utilizadas devem ser menores que a carga máxima do sistema.

O objetivo desta avaliação de desempenho é descobrir a carga máxima de uma aplicação de testes, não importando o tempo gasto durante a sua execução. Já o teste de Resistência avalia a ocorrência de vazamentos de memória que podem ocorrer com a execução prolongada dos testes. Assim, descartamos a execução dos Testes de Resistência.

O Teste de Pico serve para diagnosticar erros funcionais que podem bloquear ou abortar a aplicação durante a oscilação na carga de usuários do sistema. Portanto este teste é relevante apenas para contextos especificos relativos à aplicação final.

Sabemos que os tempos de resposta para o atendimento de requisições em aplicações com carga máxima são maiores que em cargas menores. Gostaríamos de medir os tempos médios de resposta (para as funcionalidades) da aplicação de testes em carga máxima para verificar se atendem à regra dos 8 segundos. Portanto, dentre os testes de Carga e Estresse, o teste de Estresse é o mais adequado por apresentar para testar a aplicação de testes.

4.2 Ferramenta de Testes

Todos os testes foram realizados com a utilização de uma ferramenta que simula usuários executando ações de forma concorrente na aplicação. Realizamos um levantamento bibliográfico que revelou a existência de diversas ferramentas com estas características, dentre elas, LoadRunner, JMeter, Grinder e OpenSTA. A ferramenta escolhida foi o JMeter por apresentar as seguintes características:

- Documentação clara e descritiva.
- Ferramenta de uso livre. Não é necessária a compra de licença para a utilização da ferramenta.

- Bom suporte. Disponibilizada pela fundação Apache, possui diversas listas de discussão sobre questões relacionadas às suas boas práticas de utilização além de suporte da comunidade de desenvolvedores de *software*.

- Independente de plataforma. Desenvolvida em Java, pode ser utilizada em diversos sistemas operacionais.

O JMeter considera cada requisição enviada como um usuário que solicita a execução de uma determinada tarefa. Um teste consiste no envio de um conjunto fixo de requisições durante um determinado intervalo de tempo. Após o término, o JMeter apresenta os tempos médios de resposta e a quantidade de requisições atendidas por minuto. Para obter a carga máxima, deve-se diminuir o intervalo de tempo para o envio de um mesmo conjunto de requisições e comparar as quantidades máximas de requisições atendidas por minuto (também conhecido como *throughput*).

4.3 Aplicação de Teste

A aplicação de teste apresenta funcionalidades de gerenciamento da persistência de dados definidos pelo domínio de uma agenda de contatos. Ela gerencia categorias e seus relacionamentos com contatos, endereços, emails, telefones e tipos de telefones. Ela é composta por um único formulário do tipo Mestre-Detalhe-SubDetalhe para o cadastro de categorias e seus relacionamentos simulando uma agenda (Figura 15).

O Foundation gerencia a persistência de entidades relacionadas a uma entidade gerenciada. Portanto, no único formulário existente, é possível executar uma ação de alteração de categoria de contatos (por exemplo, sua descrição) e que envolve também a criação de um novo contato, a alteração de um contato existente e a exclusão de um contato antigo. Registros relacionados a contatos, como endereço, telefone e e-mail também podem ser gerenciados durante a mesma ação de alteração de categoria. Assim, o formulário criado representa e gerencia textualmente até quatro níveis de relacionamentos para a associação entre categoria, contato, telefone e tipo de telefone[2].

2 Não existe limite para a quantidade de relacionamentos que podem ser gerenciados em um mesmo formulário. É importante deixar claro que o formulário não possui recursos javascript para configuração na aplicação das operações de persistência a serem executadas em registros relacionados. No caso de uma ação de alteração em uma entidade qualquer, o Foundation compara os dados recebidos com os dados

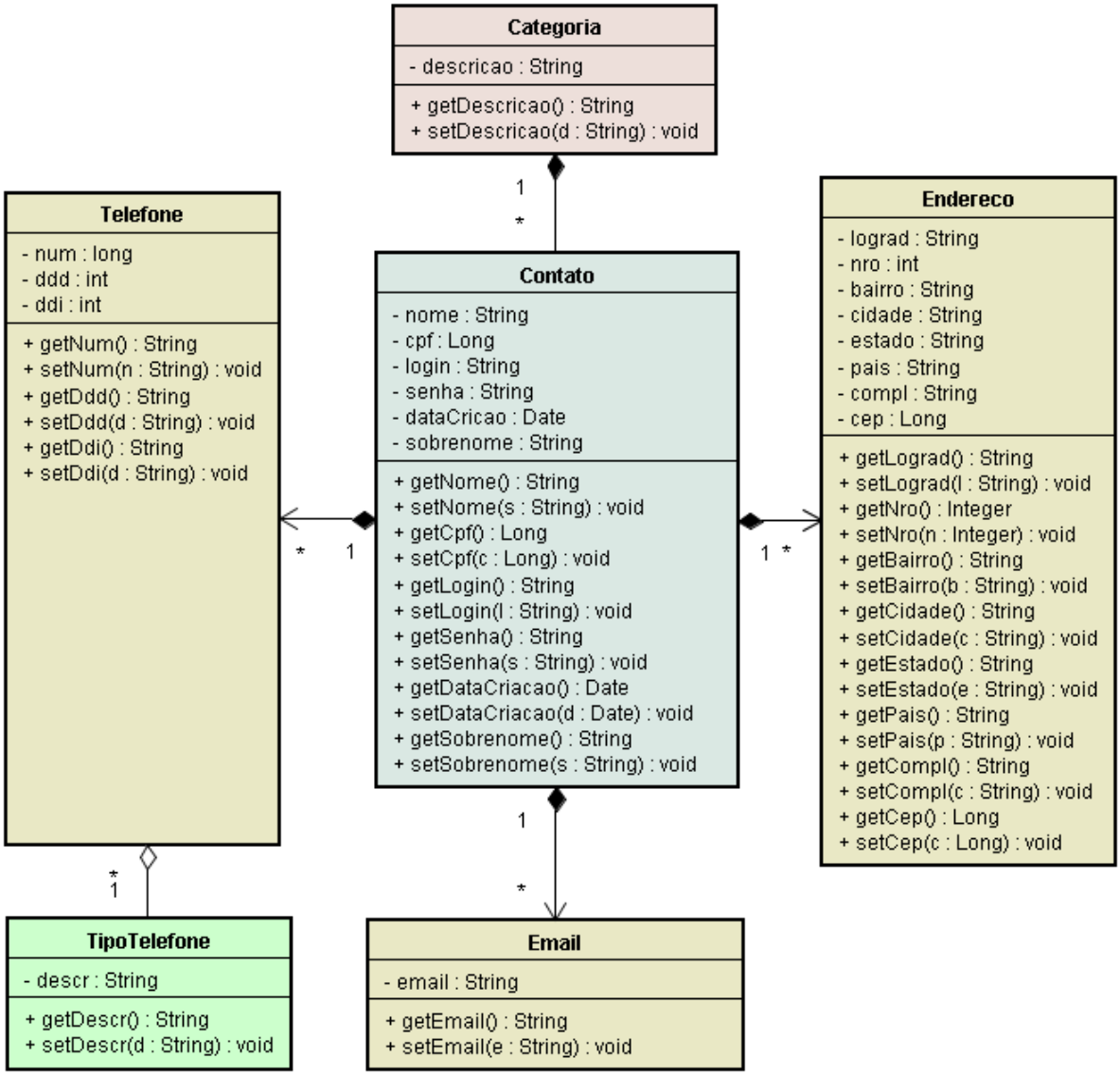

Figura 15: Modelo de classes da aplicação utilizada nos testes de carga. As diferentes cores indicam os níveis de agregação de cada classe com relação à entidade Categoria.

4.4 Cenários de Teste

Os testes de estresse foram executados em quatro cenários. Em cada cenário, os usuários simulados executam apenas uma das quatro operações envolvendo *Categoria*: inclusão, busca, alteração ou exclusão de categorias de contatos e seus relacionamentos.

Para cada cenário foi executada uma bateria de testes na aplicação sendo que, em cada teste, para um mesmo conjunto de usuários, o tempo de envio de todas as requisições é

utilizados para a geração do formulário submetido e determina qual operação de persistência deve ser executada em cada registro agregado.

diminuido até que obtenha-se a quantidade máxima de requisições atendidas por minuto (maior vazão ou *"throughtput"*).

Em cada cenário ou bateria de testes apenas uma operação de persistência é executada na entidade *Categoria*. Porém, esta operação pode não ser a mesma executada nos registros relacionados à *Categoria*. Portanto, definimos as seguintes operações de persistência executadas em cada cenário:

1. Inserção: Cada usuário simulado (requisição HTTP enviada ao servidor de aplicação) determina a inserção de um registro de uma categoria de contatos e dois contatos. Cada contato contém um endereço, um e-mail e um telefone. Portanto, cada "usuário" executa a inserção de uma categoria de contatos, dois contatos, dois e-mails, dois telefones e dois endereços. No total são nove operações de inclusão de dados envolvendo cinco tabelas.

2. Busca: Cada "usuário" simulado é responsável por executar a busca de um contato que relaciona-se à dois e-mails, dois telefones e quatro endereços. No total são nove operações de busca de dados por chave-primária envolvendo quatro tabelas.

3. Exclusão: Cada "usuário" simulado é responsável por excluir uma categoria de contatos da aplicação. Esta exclusão compreende a exclusão dos registros dependentes desta categoria que são: dois contatos, sendo que cada um contém um endereço, um telefone e um e-mail. Portanto, cada "usuário" executa a exclusão de uma categoria de contatos, dois contatos, dois e-mails, dois telefones e dois endereços. No total são nove operações de exclusão de dados envolvendo cinco tabelas.

4. Alteração: Diferente das outras baterias, a bateria de alteração de dados determina a execução de diferentes operações de persistência para os registros relacionados ao registro de *categoria*. Assim, cada "usuário" simulado determina:

   A alteração de um registro de categoria de contatos. A alteração de um contato antigo e de seu endereço, seu telefone e seu e-mail. A exclusão de um contato antigo e que possui um endereço, um telefone e um e-mail. A inclusão de um novo contato que possui um endereço, um telefone e um e-mail.

   Um total de 5 operações de alteração de registros envolvendo cinco tabelas, quatro operações de exclusão de registros envolvendo quatro tabelas e quatro operações

de inclusão de registros envolvendo quatro tabelas.

4.5 Execução e Resultados

Os testes foram executados em uma máquina contendo um disco rígido de 5400 RPM, um processador pentium core2duo de 1.6 GHz e 1,5GB de memória. O repositório de dados utilizado foi o MySql versão 5. Esta máquina foi utilizada para executar de forma concorrente a ferramenta JMeter, o servidor de aplicações JBoss-4.0.3SP1 e o repositório de dados.

Uma bateria de testes foi executada em cada cenário, sendo que, em cada teste, foram configurados diferentes valores para o tempo total de envio de todas as requisições (*ramp-up*). Ao final de cada teste obtivemos a quantidade de requisições atendidas por minuto. A Tabela 2 apresenta a configuração e o resultado dos testes que geraram valores de carga máxima para cada bateria.

Cada bateria determinou a execução de um grande número de testes para alteração de parâmetros, correções na aplicação ou ajustes de configuração no repositório de dados.

Cenário	*Ramp-up*	Total Usuários	Usuários Atendidos/min.
Inserção	76s	4000	3159
Busca	64s	5000	3964
Alteração	60s	2000	2954
Exclusão	100s	10092	5880

Tabela 2: Configurações da ferramenta JMeter para a execução do teste que gerou o resultado de carga máxima em cada bateria. A coluna ramp-up é o tempo gasto para o envio de todas as requisições enviadas durante o teste. A coluna Usuários Atendidos/min. é o resultado do teste e informa a quantidade de requisições atendidas durante um minuto.

As baterias de testes foram executadas para cargas crescentes até que a carga máxima pudesse ser encontrada. Em cargas maiores que a carga máxima o desempenho da aplicação degrada drasticamente. Esta medida é obtida comparando-se os tempos médios de resposta para o atendimento das requisições durante um teste (Tabela 3).

Os dados da Tabela 3 foram utilizados para a geração de gráficos. Os gráficos das Figuras 16 e 17 demonstram claramente que, após a carga máxima da aplicação para cada cenário, seu desempenho degrada drasticamente.

Resultados dos Testes				
Baterias	Carga (req./s)	Média (ms)	Desvio P. (ms)	Vazão (usr/min).
Inserção	15	24	8	905
	30	24	8	1803
	45	24	8	2695
	52,94 (máx)	29	8	3159
	60	400	288	2778
	75	1016	584	2455
Busca	15	22	10	905
	30	21	7	1805
	45	20	7	2699
	60	29	9	3577
	66,66 (máx)	36	13	3964
	75	417	225	34447
	90	580	362	3484
Alteração	15	31	41	904
	30	29	8	1800
	45	37	9	2709
	50 (máx)	39	16	2974
	60	1055	584	2146
	75	1311	65	2227
Exclusão	15	44	8	902
	30	16	7	1808
	45	15	7	2707
	60	14	7	3600
	75	25	25	4447
	90	18	7	5366
	100 (máx)	26	17	5880
	115,71	474	254	4425
	130,64	574	294	4416

Tabela 3: A tabela apresenta a performance da aplicação durante a execução de determinados testes para as quatro baterias. A coluna Carga representa a quantidade de requisições enviadas pelo JMeter durante um segundo. A coluna Média indica o tempo médio gasto pela aplicação para o atendimento de cada requisição.

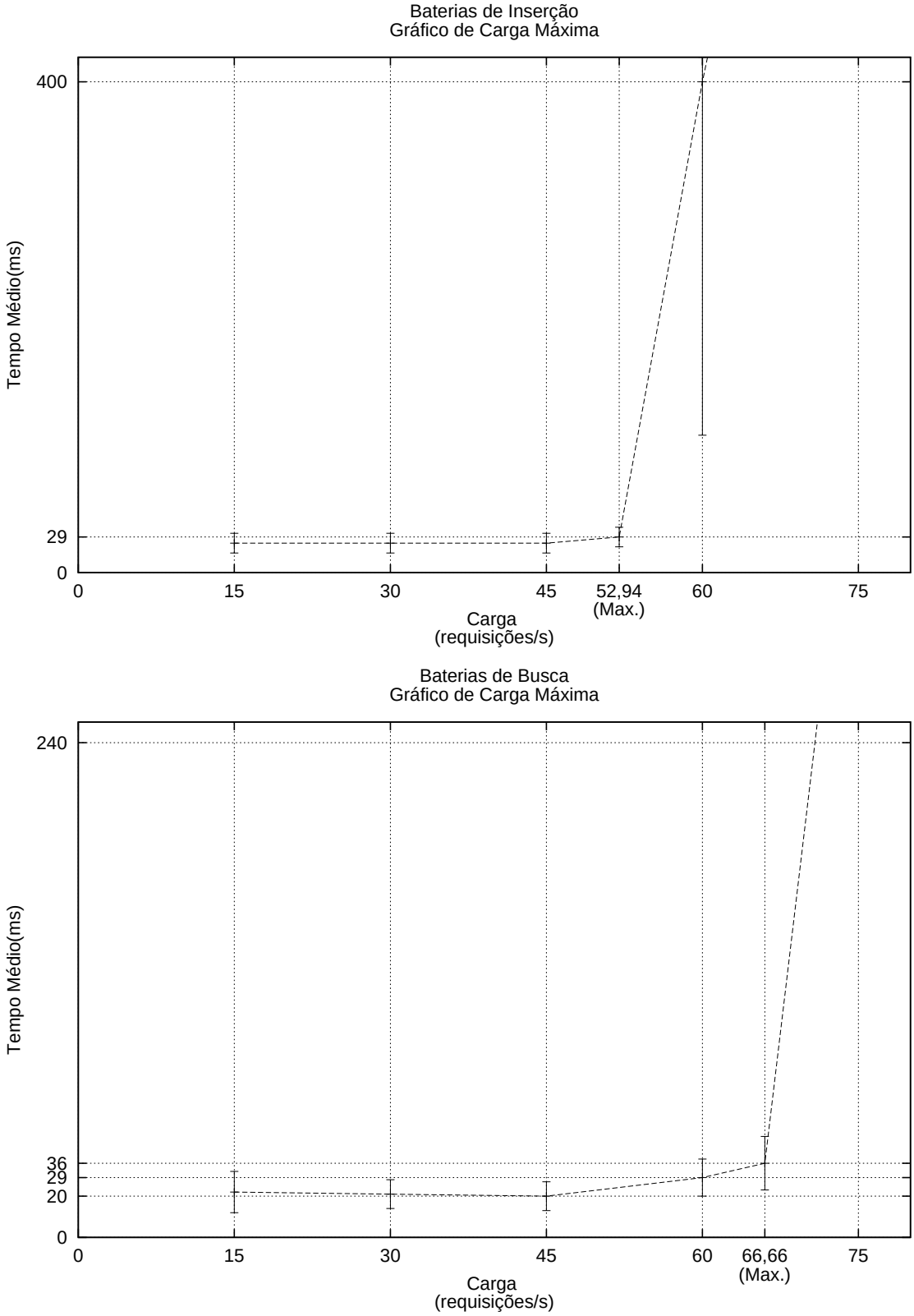

Figura 16: Gráfico de variação de desempenho da aplicação durante as baterias de Inserção e Busca.

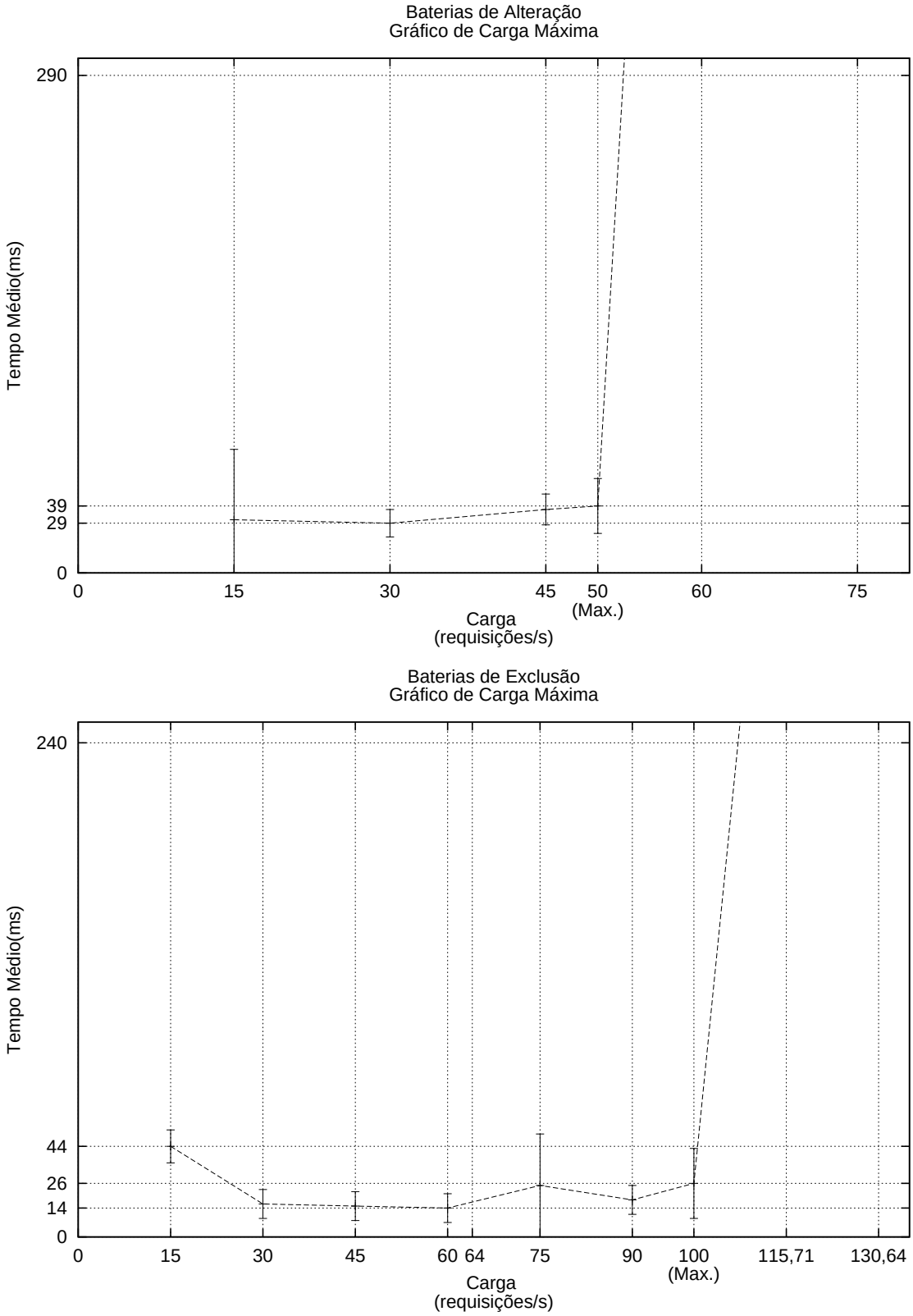

Figura 17: Gráfico de variação de desempenho da aplicação durante as baterias de Alteração e Exclusão.

A vazão ou quantidade de usuários atendidos por minuto ("throughtput") em cada teste realizado em carga máxima pode ser projetado para um período de 24 horas. Assim, obtém-se a quantidade de requisições atendidas por dia (Tabela 4).

Bateria	Vazão Req./Min.	Total Requisições/Dia
Inserção	3159	4.548.960
Busca	3964	5.708.160
Alteração	2974	4.282.560
Exclusão	5880	8.467.200

Tabela 4: Total de Requisições atendidas por minuto e por dia (24 horas) para cada cenário de testes. Os dados da coluna Total Requisições/Dia foram obtidos por meio de uma projeção da Carga Máxima para um período de 24 horas.

4.6 Conclusão

Neste capítulo descrevemos os experimentos realizados no Foundation para medir sua performance em uma aplicação desenvolvida especificamente para este fim. Verificamos que, em carga máxima, todas suas funcionalidades estão de acordo com a regra dos 8 segundos. É válido lembrar que, durante os testes, tanto a ferramenta de testes quanto o servidor de aplicações foram executados na mesma máquina, o que significa que, dos valores apresentados, é necessário acrescentar o "tempo de rede", ou tempo em que os dados demoram para trafegar pela rede.

Os dados da Tabela 4 mostram que aplicações desenvolvidas com o Foundation possuem boa performance podendo atender a milhões de requisições durante o período de 24 horas.

5 Conclusão

O código-fonte do Foundation está disponível sob licença GPL no seguinte endereço: http://assembla.com/spaces/foundation.

O Foundation propõe um modelo para a implementação de formulários em aplicações JavaEE baseado no modelo MVC proposto pelo Struts e na especificação JPA para acesso à repositórios de dados. Gerencia objetos de domínio por meio da inversão de controle, ao instanciar objetos de domínio e instanciar e injetar suas dependências, além de prover uma arquitetura em camadas extensíveis. A versão atual foi construída utilizando-se o arcabouço Struts, que é pioneiro dentre os arcabouços estudados, porém sua utilização faz parte da estratégia de adoção do Foundation, pois o Struts ainda é muito utilizado atualmente.

Conseguimos atingir o objetivo proposto com a criação de um *middleware* que atende a quatro requisitos: modelar o atendimento de requisições HTTP que determinam a execução de operações de persistência, processar requisições HTTP preenchendo inclusive coleções de dados da linguagem Java, disponibilizar pontos de interceptação de fluxo para especialização de operações automáticas e possibilitar a criação de formulários HTML com leiaute livre.

A implementação dos quatro requisitos anteriores determinou também a criação de outras funcionalidades, como o mecanismo de controle de estado para formulários. Este mecanismo permite à ação *update* executar automaticamente operações de exclusão de dados em agregações do tipo Um-Para-Muitos após comparação entre os conjuntos de dados utilizados na geração do formulário e conjuntos de dados enviados após alteração no cliente.

Dos resultados esperados, o Foundation proporciona a diminuição de código necessário para a implementação de aplicações *web* ao automatizar operações antes executadas manualmente para cada conjunto de dados definido por classes do domínio: processamento de requisições para preenchimento de coleções de dados da linguagem Java, gerenciamento de

conexões e transações com repositórios de dados e implementação de consultas genéricas para as operações de persistência: *create, retrieve, update* e *delete*. Ainda, de acordo com os resultados da avaliação de desempenho, pode ser utilizado na criação de sistemas que executam diariamente milhões de operações de persistência.

Além disso, a arquitetura inclui pontos de interceptação (entre o atendimento de requisições e execução automática de operações de persistência) para a implementação das operações mais comuns relacionadas à criação de formulários HTML: controle de componentes visuais, execução de operações específicas do domínio, especialização do acesso à repositórios. Desta forma, sua arquitetura visa minimizar os efeitos de erosão causados principalmente pela falta de lugares específicos para a implementação de operações comuns.

Mais de cinco projetos já foram desenvolvidos em uma mesma empresa com a utilização do Foundation. Dentre eles, um sistema de controle de acessos à sistemas da empresa e um sistema de gerenciamento de fornecedores da empresa. Sua utilização nestes projetos demonstrou ser possível a criação de aplicações com formulários contendo diversos níveis (agregações do tipo Um-Para-Muitos) e grande quantidade de campos (mais de 50) em um tempo de desenvolvimento comparável ao tempo gasto para a criação de formulários menores.

5.1 Trabalhos Futuros

O Foundation é um *framework* caixa-cinza (BüCHI; WECK, 1999) e sua utilização é restrita a desenvolvedores Java. Podemos utilizar a metodologia proposta por Durham e Johnson (DURHAM; JOHNSON, 1996) para desenvolver uma linguagem visual de especificação que possibilitaria a criação de sistemas por analistas de negócio.

Devemos criar uma nova versão do Foundation baseada na implementação do padrão MVC proposta pelo Struts 2. Além disso, esta nova versão do Foundation deve oferecer novas implementações das camadas atuais, a saber:

- Visão: *templates* ou modelos de páginas contendo leiautes padrão para os formulários de busca, inserção/alteração e exclusão de registros.

- Controle: classes de ações que implementem outras funcionalidades, como por exemplo, a geração de relatórios em formato pdf ou gráficos contendo registros obtidos por meio de pesquisas.

- Domínio: implementação do padrão *Session Facade* (ALUR et al., 2003) para distribuição de aplicações em servidores Java. Isto permitiria a aplicações a distribuição de carga em diferentes servidores. No caso, sugerimos a utilização de componentes *Enterprise Java Beans* (BURKE; MONSON-HAEFEL, 2006) que implementem a interface *IDomain*.

- Acesso à Dados: implementação da interface *Dao* para acesso à repositórios por meio de outras tecnologias de persistência, como por exemplo, *Java Database Connectivity*.

Referências

ALUR, D. et al. *Core J2EE Patterns: Best Practices and Design Strategies*. 2. ed. Londres: Prentice Hall PTR, 2003. 528 p.

APACHE SOFTWARE FOUNDATION. *Logging Services*. EUA, 2006. Disponível em: <http://logging.apache.org/log4j/docs>. Acesso em: 4 jan, 2007.

APACHE SOFTWARE FOUNDATION. *Struts Framework Documentation*. EUA, 2006. Disponível em: <http://struts.apache.org/1.2.9/index.html>. Acesso em: 4 jan, 2007.

BAUER, C.; KING, G. *Java Persistence with Hibernate*. Greenwich, CT, USA: Manning Publications Co., 2006. ISBN 1932394885.

BüCHI, M.; WECK, W. *The Greybox Approach: When Blackbox Specification Hide Too Much*. Turku, Finland, Aug 1999.

BORLAND INC. *Using InternetBeans Express*. EUA, 2005. Disponível em: <http://info.borland.com/techpubs/jbuilder/jbuilder2005/webapps/internetbeans.html>. Acesso em: 4 jan, 2007.

BURKE, B.; MONSON-HAEFEL, R. *Enterprise JavaBeans 3.0 (5th Edition)*. EUA: O'Reilly Media, Inc., 2006. ISBN 059600978X.

CODEHAUS. *Java Powered Ruby Implementation*. EUA: [s.n.], 2007. Disponível em: <http://jruby.codehaus.org/>. Acesso em: 18 nov, 2008.

COLLINS-COPE, M.; ROSENBERG, D.; STEPHENS, M. *Agile Development with ICONIX Process: People, Process, and Pragmatism*. Berkely, CA, USA: Apress, 2005. ISBN 1590594649.

CRYSTAL REPORTS. *Crystal Reports Specifications*. EUA, 2006. Disponível em: <http://www.businessobjects.com/products/reporting/crystalreports/default.asp>. Acesso em: 25 set, 2006.

DURHAM, A. M.; JOHNSON, R. E. A framework for run-time systems and its visual programming language. In: *OOPSLA*. [s.n.], 1996. p. 406–420. Disponível em: <http://dblp.uni-trier.de/db/conf/oopsla/oopsla96.html#DurhamJ96>.

EVANS, E. *Domain Driven Design*. EUA: Addison Wesley, 2003. 560 p.

FORD, N. *Art of Java Web Development*. EUA: Manning, 2004. 593 p.

FOWLER, M. et al. *Patterns of Enterprise Application Architecture*. EUA: Addison Wesley, 2002. 560 p.

GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1. ed. EUA: Addison Wesley, 1995.

GOSLING BILL JOY, G. S. J.; BRACHA, G. *The Java Language Specification, Third Edition*. Prentice Hall PTR, 2005. Disponível em: <http://java.sun.com/docs/books/jls/>. Acesso em: 18 nov, 2008.

GROUP, O. *Unified Modeling Language (UML), version 2.1.1*. 06.02.2007. Disponível em: <http://www.omg.org/technology/documents/formal/uml.htm>. Acesso em: 18 nov, 2008.

HOHMANN, L. *Beyond Software Architecture: Creating and Sustaining Winning Solutions*. Boston, MA, EUA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN 0201775948.

JASPERSOFT. *Jasper Reports Key Features*. EUA, 2007. Disponível em: <http://www.jaspersoft.com/pr_jasperreports.html>. Acesso em: 4 jan, 2007.

JBOSS INC. *Hibernate Reference Documentation*. EUA, 2006. Disponível em: <http://www.hibernate.org/hib_docs/v3/reference/en/pdf/hibernate_reference.pdf>. Acesso em: 18 out, 2008.

JOHNSON, R.; HOELLER, J. *Expert One-on-One: J2EE Development Without EJB*. Indianapolis, Indiana, EUA: Wiley Publishing Inc., 2004. 552 p.

JOHNSON, R. et al. *Professional Java Development with the Spring Framework*. Birmingham, UK: Wrox Press Ltd., 2005. ISBN 0764574833.

JOHNSON, R. E.; FOOTE, B. Designing reusable classes. *Journal of Object-Oriented Programming*, v. 1, n. 2, p. 22–35, 1988.

MICROSYSTEMS, S. *Java Server Faces Technology*. EUA: [s.n.], 2008. Disponível em: <http://java.sun.com/javaee/javaserverfaces/>. Acesso em: 18 nov, 2008.

ORACLE INC. *Oracle Toplink Documentation*. EUA, 2006. Disponível em: <http://www.oracle.com/technology/products/ias/toplink/index.html>. Acesso em: 4 jan, 2007.

PERRY, D. E.; WOLF, A. L. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, v. 17, n. 4, p. 40–52, 1992.

SUBRAYA, B. M. *Integrated Approach To Web Performance Testing*. EUA: IRM Press, 2006. 368 p.

SUN MICROSYSTEMS. *Enterprise JavaBeans 3.0 Documentation*. EUA, 2008. Disponível em: <http://java.sun.com/products/ejb/docs.html>. Acesso em: 18 out, 2008.

VANBRABANT, R. *Google Guice: Agile Lightweight Dependency Injection Framework (Firstpress)*. [S.l.]: APress, 2008. ISBN 1590599977, 9781590599976.

ZUKOWSKI, J. *Java Collections*. Berkely, CA, EUA: Apress, 2001. ISBN 1-893115-92-5.

APÊNDICE A – Criação de ações com o Foundation

A.1 Código-Fonte

O código-fonte do Foundation está disponível no endereço http://www.assembla.com/spaces/foundation.

A.2 Ações Específicas

Ações específicas em aplicações desenvolvidas com o Foundation são as ações que não determinam a execução de operações de persistência do tipo CRUD. Para a implementação de uma ação específica são necessárias quatro atividades:

1. Implementação da classe de ações, classe que contém métodos que implementam ações.

2. Implementação da ação; implementar o método da classe de ações criada na primeira atividade.

3. Configuração da ação em arquivo XML: definição da ação e do *ActionForm* que, em tempo de execução, conterá os dados recebidos por requisições HTTP.

4. Criação da página JSP de resposta.

Das atividades descritas, apenas a criação da página JSP de resposta não será demonstrada. No caso, será apresentada a criação de uma classe de ações, *UsuarioAction*, que deverá conter métodos relacionados ao comportamento de usuários. Cada método de *UsuarioAction* define uma ação executada no atendimento a requisições HTTP. Os métodos da classe *UsuarioAction* podem ser: *inserir, pesquisar, processarInadimplentes, excluir*.

O código abaixo ilustra a implementação de uma classe de ações, *UsuarioAction*, que atende requisições para o processamento de usuários inadimplentes. A classe de ações *UsuarioAction* possui apenas uma ação implementada: *processaInadimplentes*.

```
package br.appTeste;
public class UsuarioAction extends FoundationMappingDispatchAction {

  public ActionForward processaInadimplentes(
                                    ActionMapping mapping,
                                    ActionForm form,
                                    HttpServletRequest request,
                                    HttpServletResponse response)
                                                throws Exception{
        //processar inadimplentes.
        return mapping.findForward("jspDeResposta");
  }
}
```

Toda classe de ações deve estender a classe *FoundationMappingDispatchAction*. Esta é a classe base para a criação de ações. Todos os métodos de classes de ações devem retornar um objeto da classe *ActionForm* e conter os mesmos parâmetros de entrada:

- *ActionMapping*. Classe que, em tempo de execução, mantém os valores para os parâmetros que configuram a ação.
- *ActionForm* é a classe que mantém os dados do formulário HTML enviado pelo cliente.
- *HttpServletRequest* e *HttpServletResponse* são objetos da especificação JavaEE criados pelo servidor de aplicações e representam, respectivamente, a requisição enviada pelo cliente e a resposta que será retornada. Normalmente utilizados para acesso a detalhes do protocolo de comunicação ou à sessão de um usuário no servidor de aplicações.

- *ActionForward* contém as informações sobre a página de resposta retornada após a execução da ação.

Para que o método *processaInadimplentes* da classe *UsuarioAction* seja executado no atendimento a requisições HTTP é necessária a configuração da ação. Esta configuração define a classe e o método executados no atendimento a requisições com URL no padrão definido pela aplicação. Esta configuração é realizada no arquivo *struts-config.xml* por meio do elemento *action*:

```
<action     name="InadimplentesForm"
    className="br.foundation.struts.FoundationActionMapping"
            path="/processarUsuariosInadimplentes"
            type="br.appTeste.UsuarioAction"
            parameter="processarInadimplentes">
  <forward name="jspDeResposta" path="/jsps/usuario/inadimplentes.jsp"/>
</action>
<form-bean name="InadimplementesForm"
           type="org.apache.struts.validator.DynaValidatorActionForm">
</form-bean>
```

O mapeamento anterior determina a configuração de uma ação e a definição de um *ActionForm*. Em detalhes:

- O atributo *name* associa a ação que está sendo definida com um alias que representa a definição do *ActionForm*. A denominação *InadimplentesForm* é uma referência para a definição do *ActionForm*, declarada no mesmo documento.

- O atributo *className* determina a classe (do tipo ActionMapping) que mantém as configurações da ação durante o fluxo de dados dentro do Foundation.

- O atributo *path* define que o servidor de aplicações irá atender requisições pela URL no padrão *http://¡meuServer¿:¡porta¿/¡minhaApp¿/processarOsInadimplentes.do* [1]

1 O Foundation atende requisições que terminam com o padrão *.do. Este padrão é configurável no arquivo web.xml da aplicação, embora não seja necessária sua alteração

- O atributo *type* define a classe contendo o método que implementa a ação solicitada. Seu nome é *br.appTeste.UsuarioAction*.

- O atributo *parameter* define o método (*processarInadimplentes*) da classe definida em *type* e que contém a implementação da ação solicitada.

- O elemento *forward* associa o *alias* "jspDeResposta" com a página localizada em "/jsps/usuario/inadimplentes.jsp". Isto evita a recompilação de código nas classes de ação sempre que o caminho de acesso para a página jsp precisar ser alterado.

- O *ActionForm InadimplentesForm* não contém elementos, pois o processamento da ação não utiliza dados recebidos por requisições HTTP.

Uma classe de ações que define ações do tipo CRUD para um conjunto de dados definido por classes do domínio pode conter implementações de ações específicas definidas dentro de uma subclasse de *CRUDImpl*. A classe *CRUDImpl* estende a classe *FoundationMappingDispatchAction*, classe base para a criação de ações com o Foundation e implementa ações que executam operações de persistência em um repositório.

Para maiores informações sobre a criação de ações específicas com o Foundation, recomenda-se a leitura da documentação do *middleware* struts (APACHE SOFTWARE FOUNDATION, 2006b).

A.3 Criação de ações do tipo CRUD

Dadas duas classes de um domínio, *Produto* e *CategoriaProduto*, mostraremos as cinco atividades necessárias para a criação de uma ação de inclusão de um novo produto em um repositório de dados. A inclusão de um produto é composta por um conjunto de duas ações: uma ação que atende requisições que solicitam o formulário de inclusão de produto e outra ação que atende a requisições que solicitam a inclusão de produtos em um repositório de dados.

Neste caso específico, serão necessárias cinco atividades para implementar as ações, porém o mínimo são quatro. No caso em questão, a ação que solicita um formulário para inclusão de um novo produto precisará popular um "combo-box" de categorias de

produtos; o que implica na criação de uma classe que implemente um interceptador para a ação de busca de formulário para inclusão de registros. As atividades necessárias são:

1. Implementação das classes de domínio: Produto e CategoriaProduto.
2. Configuração de Persistência das classes Produto e CategoriaProduto.
3. Criação de classe que implemente um interceptador para a ação de busca de formulário para inclusão de registros e que defina um método que carrega uma lista de categorias sempre que um formulário para inclusão de novo produto for solicitado.
4. Definição das ações e dos *ActionForms* em arquivos XML.
5. Criação da página JSP contendo o formulário de inclusão de produtos. Não é o intuito demonstrar a criação de páginas JSP, portanto esta atividade não será realizada.

A implementação das classes *Produto* e *CategoriaProduto* contém anotações Java[2] que especificam suas configurações de persistência.

```
package br.exemplo;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
@Entity
@Table("Produto")
public class Produto{

 private Long   id;
 private String nome;
 private CategoriaProduto categoria;
```

2 Recurso disponível na versão Java 1.5

```
 @Id
 @GeneratedValue(strategy=GenerationType.IDENTITY)
 public Long getId(){
  return id;
 }

 @Column(name="NOME")
 public String getNome(){
  return nome;
 }

 @OneToOne
 @JoinColumn(name="fk_categoria")
 public CategoriaProduto getCategoria(){
  return categoria;
 }

 public void setId(Long id){
  this.id = id;
 }

 public void setNome(String nome){
this.nome = nome;
 }

 public void setCategoria(CategoriaProduto categoria){
  this.categoria = categoria;
 }
}

package br.exemplo;
```

```
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
@Entity
@Table("Categoria")
public class CategoriaProduto{

private Long id;
private String nome;
@Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
public Long getId(){
 return id;
}
@Column(name="NOME")
public String getNome(){
 return nome;
}

public void setId(Long id){
 this.id = id;
}

public void setNome(String nome){
   this.nome = nome;
  }
}
```

A classe *Produto* contém dois atributos (id e nome) e um relacionamento do tipo

Um-Para-Um com a classe CategoriaProduto especificado pelo atributo categoria. Já a classe *CategoriaProduto* contém apenas dois atributos: id e nome.

As duas ações (busca de formulário e inserção de dados) manipulam dados relacionados com o mesmo formulário HTML. Portanto, suas configurações irão referenciar a mesma definição de *ActionForm*:

```
<form-bean name="ProdutoForm"
           type="org.apache.struts.validator.DynaValidatorActionForm">
 <form-property name="id"              type="java.lang.Long" />
 <form-property name="nome"            type="java.lang.String" />
 <form-property name="categoria"       type="br.exemplo.CategoriaProduto"/>
 <form-property name="listaCategorias" type="java.util.Collection"/>
</form-bean>
```

A declaração do *ActionForm* define elementos com os mesmos nomes dos atributos declarados na classe *Produto*: id, nome e categoria. O *ActionForm ProdutoForm* possui mais campos que a classe de domínio *Produto*. Isto porque o cadastro de produto possuirá um componente do tipo *drop-down* contendo uma lista de categorias para o usuário selecionar. O algorítimo de transferência de dados entre *ActionForm* e objetos de domínio desconsidera os campos contidos no *ActionForm* e não contidos no objeto de domínio no momento da transferência de dados vindos do cliente[3]. O *ActionForm ProdutoForm* foi declarado como sendo do tipo *org.apache.struts.validator.DynaValidatorActionForm*. Todos os *ActionForm* de ações construídas com o Foundation devem ser da mesma classe[4].

Uma ação de inclusão de um produto é precedida por uma ação de solicitação do formulário de produto para inclusão de novo registro. A classe *CRUDImpl* implementa esta ação de busca de formulário e a camada de Controle define um interceptador para esta ação. O interceptador define um método utilizado na inicialização de campos do formulário.

```
<action     name="ProdutoForm"
    className="br.foundation.struts.FoundationActionMapping"
```

3 O inverso é recíproco ao transferir dados do objeto de domínio para o *ActionForm*
4 Maiores detalhes na documentação do Struts

```
            path="/iniciarFormularioProduto"
            type="br.foundation.mvc.crud.CRUDImpl"
            parameter="retrieveSaveForm">
  <set-property property="managedEntity" value="br.exemplo.Produto"/>
  <set-property property="interceptor" value="br.exemplo.ProdutoInterceptor"/>
  <set-property property="domainLayer" value="br.exemplo.ProdutoDominio"/>
  <forward name="SUCCESS"       path="/jsps/usuario/cadastroProduto.jsp"/>
  <forward name="SERVICE_ERROR" path="/jsps/home/home.jsp"/>
</action>
```

Conforme a declaração da ação acima, requisições atendidas pelo servidor de aplicações e com URL cujo final é /iniciarFormularioProduto.do são atendidas pelo método *retrieveSaveForm* da classe *CRUDImpl* que, em caso de sucesso, retorna o formulário definido em cadastroProduto.jsp e preenchido com dados do *ActionForm ProdutoForm*. A classe ProdutoInterceptor deve implementar um interceptador para a ação *retrieveSaveForm* e implementar a inicialização de campos presentes no formulário de resposta. Caso o formulário de inclusão de produtos não tivesse campo algum que necessitasse inicialização, seria possível criarmos as duas ações utilizando a classe CRUDImpl sem a necessidade de especificarmos um interceptador para as ações.

O mapeamento anterior para a ação genérica *retrieveSaveForm* possui um elemento que define o conjunto de dados definidos pelo domínio e que será manipulado pelo Foundation durante a execução da ação. No caso, o elemento *set-property* com a propriedade *managedEntity* determina a classe do domínio para a ação.

O formulário retornado conterá uma lista de categoria de produtos. A ação *retrieveSaveForm* é um método da classe *CRUDImpl* cujo interceptador é *RetrieveSaveFormInterceptor*[5]. A classe *ProdutoInterceptor*, deve implementar o interceptador e, consequentemente, o método *initializeSaveForm* que deve popular o *ActionForm* com a listagem de categorias que deverá estar presente no formulário de inclusão de produtos.

Repare que no código a seguir o Foundation insere como dependência um objeto específico da camada de domínio implementado pela própria aplicação. A configuração

5 Todos os interceptadores de ações são interfaces que possuem como prefixo o nome da ação acompanhado do sufixo Interceptor

da ação anterior contém o elemento *set-property* com a propriedade *domainLayer* que serve para informar o Foundation sobre esta especialização.

```
package br.exemplo.ProdutoAction;

import br.exemplo.ProdutoDominio;
import br.foundation.domain.*;
public ProdutoAction implements RetrieveSaveFormInterceptor{

  protected void initializeSaveForm(
                                        IDomain dominio,
                                        ActionForm form,
                                        ActionMapping mapping,
                                        HttpServletRequest request,
                                        HttpServletResponse response)
                                                throws CRUDException {

              DynaActionForm f  = (DynaActionForm)form;
              ProdutoDominio d = (ProdutoDominio)dominio;
              Collection categorias = d.getCategorias();
              f.set("listaCategorias", categorias);
  }
}
```

A ação de inclusão de produto não precisa de implementação. Ela está implementada de forma genérica na classe *CRUDImpl*. Assim, é necessária apenas a configuração da ação de inclusão de produtos:

```
<action     name="ProdutoForm"
    className="br.foundation.struts.FoundationActionMapping"
            path="/incluirProduto"
            type="br.foundation.mvc.crud.CRUDImpl"
            parameter="save">
```

```
  <set-property property="managedEntity" value="br.exemplo.Produto"/>
  <forward name="SUCCESS"      path="/listarProdutos.do"/>
  <forward name="SERVICE_ERROR" path="/jsps/usuario/cadastroProduto.jsp"/>
</action>
```

A declaração da ação de inclusão de produto determina a associação entre o *ActionForm ProdutoForm* com o método *save* da classe *CRUDImpl* e que atende requisições com URL cuja terminação é */incluirProduto.do*. As ações do tipo CRUD sempre retornarão **dois** tipos de resposta: sucesso e erro representadas, respectivamente, por `SUCCESS` e `SERVICE_ERROR` na configuração da ação. No caso, uma inclusão bem sucedida deverá retornar uma lista contendo todos os produtos. Esta ação é especificada pela URL */listarProdutos.do* cuja ação não está definida. Em caso de erro na inclusão de produto a aplicação deverá retornar a página de cadastro com os dados submetidos.

A página JSP cadastroProduto.jsp deverá gerar uma página HTML que conterá um formulário equivalente ao formulário abaixo:

```
<form action="/incluirProduto.do" method="post">
 <input type="text" name="nome"/>
 <select name="categoria.id">
    <option value="1">CATEGORIA 1</option>
    <option value="2" selected>CATEGORIA 2</option>
 </select>
</form>
```

Ao atender requisições que solicitam a inclusão de produtos, o Foundation transfere os dados do formulário submetido para o *ActionForm ProdutoForm*. Por tratar-se de uma classe de ações que estende a classe CRUD, os dados do *ActionForm* serão transferidos novamente para a entidade Produto. Suas agregações também serão criadas, neste caso, um objeto da classe *CategoriaProduto*. O objeto *CategoriaProduto* é representado no formulário HTML pelo elemento de nome *categoria.id*. Em detalhes, este elemento HTML possui um nome composto, o que caracteriza a agregação. A primeira parte, *categoria*, representa o nome do atributo em *Produto* que especifica a agregação. A segunda, *id*, representa o nome do atributo em *CategoriaProduto* que deve conter o valor do elemento

HTML *categoria.id*. Sempre que um atributo de uma agregação for representado nos formulários HTML deve-se utilizar um ponto (.). Por fim, o processamento do objeto *Produto* é delegado para as camadas inferiores e sua inclusão é efetuada em um repositório de dados.

Para ilustrar melhor a representação de objetos Java em formulários HTML será criado um cenário onde um produto possa pertencer a mais de uma categoria. Neste caso, a classe *Produto* será modificada da seguinte forma [6]:

```
package br.exemplo;
import java.util.List;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
@Table("Produto")
public class Produto{

 private Long   id;
 private String nome;
 private List<CategoriaProduto> categorias;

 @Id
 @GeneratedValue(strategy=GenerationType.IDENTITY)
 public Long getId(){
  return id;
 }

 @Column(name="NOME")
 public String getNome(){
```

6 A codificação é realizada utilizando-se notação da linguagem Java 1.5

```
  return nome;
 }

 @OneToMany
 public List<CategoriaProduto> getCategoria(){
  return categorias;
 }

 public void setId(Long id){
  this.id = id;
 }

 public void setNome(String nome){
  this.nome = nome;
 }

 public void setCategorias(List<CategoriaProduto> categorias){
  this.categorias = categorias;
 }
}
```

Neste novo cenário, a definição do *ActionForm* sofre uma alteração de acordo com a alteração realizada na classe Produto:

```
<form-bean name="ProdutoForm"
           type="org.apache.struts.validator.DynaValidatorActionForm">
 <form-property name="id"              type="java.lang.Long"/>
 <form-property name="nome"            type="java.lang.String"/>
 <form-property name="categorias" type="br.exemplo.CategoriaProduto[]"/>
 <form-property name="listaCategorias" type="java.util.Collection"/>
</form-bean>
```

Assim como a classe *Produto*, a nova definição do *ActionForm ProdutoForm* possui

uma propriedade chamada *categorias*. No entanto, a classe declarada para a propriedade categorias não condiz com a classe declarada em *Produto* para o atributo categorias. Ao definir um *ActionForm* o Foundation permite a declaração de coleções de dados como uma das interfaces java.util.Collection, java.util.List ou java.util.Set ou como uma classe do pacote java.util. Portanto, a propriedade categorias poderia estar declarada no *ActionForm* como sendo a interface java.util.List.

Caso a declaração da propriedade categorias em *ProdutoForm* seguisse a definição do atributo na classe *Produto*, o tipo declarado seria a interface java.util.List. Neste caso, um outro elemento XML deveria ser criado para informar ao Foundation a classe dos objetos que devem ser criados e inseridos dentro da lista.

O exemplo a seguir ilustra a situação:

```
<form-bean name="ProdutoForm"
           type="org.apache.struts.validator.DynaValidatorActionForm">
  <form-property name="id"                type="java.lang.Long" />
  <form-property name="nome"              type="java.lang.String" />
  <form-property name="categorias"        type="java.util.List" />
  <form-property name="listaCategorias"   type="java.util.Collection"/>
  <form-property name="tiposColecoes"     type="java.lang.String"
 initial="categorias=br.exemplo.CategoriaProduto"/>
</form-bean>
```

Esta última definição de *ProdutoForm* representa exatamente o domínio gerenciado. O elemento tiposColecoes, declarado no *ActionForm*, é utilizado como configuração. Ele é utilizado pelo Foundation para identificar, em tempo de execução, a classe dos objetos que deverão ser inseridos nas coleções que representam relacionamentos do tipo Um-Para-Muitos e que são definidas na classe Produto por interfaces ou classes Java do pacote java.util. Para inserir mais de um par chave-valor no campo tiposColecoes é necessário separá-los por ponto-e-vírgula (;). A elementos HTML que representam a agregação Um-Para-Muitos devem ser nomeados conforme o padrão seguir o padrão: listaCategorias[0].id, listaCategorias[1].id e assim por diante.

A criação das ações detalhar, excluir, alterar para os dados da classe *Produto* consiste

apenas na execução de uma atividade: definição das ações em arquivo XML. Estas ações podem ainda reaproveitar a definição do *ActionForm ProdutoForm*.

A criação de formulários do tipo Mestre-Detalhe-SubDetalhe consiste na execução das mesmas cinco atividades. O Foundation controla a complexidade no gerenciamento da persistência de dados agregados. Isto permite a aplicações a construção de menos páginas HTML contendo formulários que manipulam dados relacionados e o oferecimento de formulários mais completos aos seus usuários.

A.4 Especializando ações do tipo CRUD com o Foundation

Utilizando o exemplo anterior (Produto e CategoriaProduto) desejamos especializar o fluxo de dados para uma ação em especial e incluir uma classe na camada de domínio e outra na camada de acesso à dados para especializar, por exemplo, uma ação de inclusão de produto.

Dado o seguinte mapeamento:

```
<action     name="ProdutoForm"
          className="br.foundation.struts.FoundationActionMapping"
            path="/incluirProduto"
            type="br.foundation.mvc.crud.CRUDImpl"
            parameter="save">
  <set-property property="managedEntity" value="br.exemplo.Produto"/>
  <forward name="SUCCESS"       path="/listarProdutos.do"/>
  <forward name="SERVICE_ERROR" path="/jsps/usuario/cadastroProduto.jsp"/>
</action>
```

Temos que, na aplicação em questão, a inclusão de produtos não pode ocorrer sem que os mesmos existam em estoque. Para tanto, a classe ProdutoDominioImpl estende a classe DomainImpl e sobrescreve o método *persist* de acordo com o trecho abaixo:

```
package br.exemplo.dominio;
```

```
import br.foundation.domain.*;
public class ProdutoDominioImpl extends DomainImpl implements ProdutoDominio{

  public void persist(Object novo) throws CRUDException{
    Produto p = (Produto) novo;
    if( p.isInStock() ){
      super.persist(novo);
    }else{
      throw new ProdutoException("Produto não existe em estoque");
    }
  }
}
```

Temos também que o mecanismo de persistência só permite a inclusão de novos registros durante o período compreendido entre 22 e 23 horas. Portanto, a ação requer uma especialização na camada de acesso à dados a fim de verificar se o horário no mecanismo de persistência está compreendido entre 22 e 23 horas. Assim, a classe ProdutoDaoImpl é responsável pela inclusão de produtos. De acordo com o trecho abaixo:

```
package br.exemplo.dao;

import br.foundation.dao.*;
public class ProdutoDaoImpl extends DaoImpl implements Dao, ProdutoDao{

  public void persist(Object novo) throws DAOException{

    if( isNowOkToInsert() ){
      super.insert();
    }else{
      throw new ProdutoDAOException("Horário não apropriado
                                    para inserção de produto.");
    }
  }
```

```
  public Boolean isNowOkToInsert(){
     //buscar horario no repositorio e
     //confirmar se está compreendido entre 22 e 23 horas.
  }
}
```

A ação definida *incluirProduto* precisa especializar o acesso às camadas de Domínio e Acesso a Dados. Para tanto são necessários dois elementos no mapeamento da ação:

```
<action     name="ProdutoForm"
            className="br.foundation.struts.FoundationActionMapping"
            path="/incluirProduto"
            type="br.foundation.mvc.crud.CRUDImpl"
            parameter="save">
  <set-property property="managedEntity" value="br.exemplo.Produto"/>
  <set-property property="domainLayer" value="br.exemplo.ProdutoDominioImpl"/>
  <set-property property="dao"           value="br.exemplo.ProdutoDaoImpl"/>
  <forward name="SUCCESS"       path="/listarProdutos.do"/>
  <forward name="SERVICE_ERROR" path="/jsps/usuario/cadastroProduto.jsp"/>
</action>
```

Os elementos *set-property* que definem as propriedades *domainLayer* e *dao* permitem que a ação possa determinar durante a sua execução o objeto utilizado para as camadas de Domínio e Acesso à Dados respectivamente.

Como o *framework* Guice é utilizado internamente para a criação de objetos e injeção de dependências é possível definirmos as propriedades dao e dominio com valores que representam interfaces, no caso, ProdutoDominio e ProdutoDao. Neste caso, as interfaces devem ser definidas com a utilização de uma anotação presente na biblioteca do Guice, responsável por definir a classe de implementação da interface. Assim teríamos as seguintes modificações na definição da ação textitincluirProduto:

```
<action     name="ProdutoForm"
```

```
            className="br.foundation.struts.FoundationActionMapping"
            path="/incluirProduto"
            type="br.foundation.mvc.crud.CRUDImpl"
            parameter="save">
  <set-property property="managedEntity" value="br.exemplo.Produto"/>
  <set-property property="domainLayer"  value="br.exemplo.ProdutoDominio"/>
  <set-property property="dao"           value="br.exemplo.ProdutoDao"/>
  <forward name="SUCCESS"       path="/listarProdutos.do"/>
  <forward name="SERVICE_ERROR" path="/jsps/usuario/cadastroProduto.jsp"/>
</action>
```

Devemos também acrescentar uma anotação na interface *ProdutoDominio* para definir a classe de implementação:

```
package br.exemplo.dominio;
import com.google.inject.ImplementedBy;

@ImplementedBy(ProdutoDominioImpl.class)
public class ProdutoDominio extends IDominio{}
```

A interface ProdutoDao também deve conter uma anotação que determina a classe de implementação:

```
package br.exemplo.dao;
import com.google.inject.ImplementedBy;

@ImplementedBy(ProdutoDaoImpl.class)
public class ProdutoDao extends Dao{}
```

Um terceiro modo de definir classes de implementação para interfaces definidas por aplicações e que devam ser resolvidas pelo Guice é a criação de uma classe que configura o Guice. Esta classe deverá ser implementada de acordo com a implementação da classe *GuiceModule* do pacote *br.foundation.guice.modules* presente na distribuição do Founda-

tion. Esta classe deverá ser configurada como um parâmetro de contexto da aplicação em questão[7] sob o nome de *module* conforme trecho abaixo:

```
<context-param>
<param-name>module</param-name>
<param-value>meuPacote.MinhaClasseQueImplementaGuiceModule</param-value>
<description>Módulo que deve ser carregado na inicialização</description>
</context-param>
```

O Foundation permite a utilização de quatro elementos específicos na definição de ações com a utilização do Struts. Os elementos definidos abaixo foram retirados de configurações de ações já apresentadas anteriormente:

```
<set-property property="managedEntity" value="br.exemplo.Produto"/>
<set-property property="interceptor"   value="br.exemplo.Produto"/>
<set-property property="domainLayer"  value="br.exemplo.ProdutoDominio"/>
<set-property property="dao"           value="br.exemplo.ProdutoDao"/>
```

A.5 Estendendo a classe *CRUDImpl*

De acordo com os elementos acima podemos extrair as seguintes informações:

1. *managedEntity*: Identifica a entidade que modela parte do domínio da aplicação e que representa o conjunto de dados manipulado pela ação.

2. *interceptor*: Identifica uma classe que implementa um interceptador para a especialização da ação em questão. O Foundation provê um interceptador para cada ação definida em *br.foundation.mvc.crud.CRUD*. Os interceptadores disponíveis encontram-se no pacote *br.foundation.mvc.crud.interceptor*.

3. *domainLayer*: Define uma interface ou classe de implementação para a camada de Domínio e que será utilizada durante esta ação. Caso seja definida uma interface, o *framework* Guice precisa estar configurado corretamente para que possa ser descoberta sua classe de implementação.

7 De acordo com a especificação JavaEE, no arquivo web.xml

4. *dao*: Define uma interface ou classe de implementação para a camada de Acesso a Dados e que será utilizada durante esta ação. Caso seja definida uma interface, o *framework* Guice precisa estar configurado corretamente para que possa ser descoberta sua classe de implementação.

Embora artefatos arquiteturais do Foundation possam ser definidos para cada ação de forma declarativa, é possível também sua definição por meio de programação; podendo esta inclusive incluir lógicas complexas para a escolha de um artefato arquitetural utilizado para uma ação dentre diversos possíveis. Para tanto, é possível a criação de uma classe de ações que gerencie uma entidade, que estenda a classe *br.foundation.mvc.crud.CRUDImpl* e também implemente interceptadores.

A classe *CRUDImpl* possui atributos e métodos de conveniência que podem ser úteis na implementação de classes de ações:

- *public IDomain getDomain(HttpServletRequest req, HttpServletResponse resp, ActionMapping map).* Este método permite a inclusão de lógicas utilizando parâmetros recebidos pela requisição vinda de um cliente para decidir qual o objeto da camada de domínio deverá ser utilizado durante a ação.[8]

- *getManagedEntity(HttpServletRequest req, HttpServletResponse resp, ActioMapping map, ActionForm f).* Este método permite a inclusão de lógicas utilizando parâmetros recebidos pela requisição vinda de um cliente para decidir qual o objeto a ser instanciado para ser preenchido com os dados recebidos do cliente. Este mecanismo permite criar formulários que manipulam dados definidos por classes e subclasses.[9]

- *protected com.google.inject.Injector injector.* Este atributo deve ser utilizado internamente para a inserção de dependências em objetos que forem criados pelo método *getDomain*.

8 Não esquecer que o objeto da camada de domínio deve utilizar o atributo *injector* para inserir suas dependências antes de ser retornado pelo método.
9 Cadastro de um usuário. Caso um radio button escolha um usuário do tipo Funcionario, outros dados deverão ser enviados ao cliente e outra classe deverá ser instanciada.

APÊNDICE B – Diagramas

Este apêndice apresenta os diagramas de classe e sequência do Foundation, construídos durante seu desenvolvimento e seguindo a metodologia de pesquisa.

B.1 Classes

B.1.1 Camada de Controle

A figura 18 apresenta o modelo de dados definido para o atendimento de requisições. No diagrama, a interface *CRUD* define as operações de persistência disponíveis para implementação. A classe *CRUDImpl* provê uma implementação padrão para todas as ações definidas em *CRUD* e define pontos de interceptação para especialização do processamento de requisições em interceptadores. O Foundation provê um interceptador para cada ação definida em *CRUD*. Caso necessário, aplicações devem interagir com a camada de Controle por meio da implementação de interceptadores.

B.1.2 Camada de Domínio

A camada de Domínio desacopla a camada de Controle, a interface de comunicação com servidores de aplicação, do acesso ao repositório de dados. Assim, provê um ponto de interceptação entre o processamento de requisições e a execução de operações de persistência para a execução de operações de domínio.

A figura 19 apresenta a classe *DomainImpl* que deve ser estendida por aplicações quando necessária a interceptação do fluxo de dados durante a execução de ações. Aplicações podem prover outras implementações para a interface *IDomain* para a execução, por exemplo, de operações remotas em aplicações distribuídas.

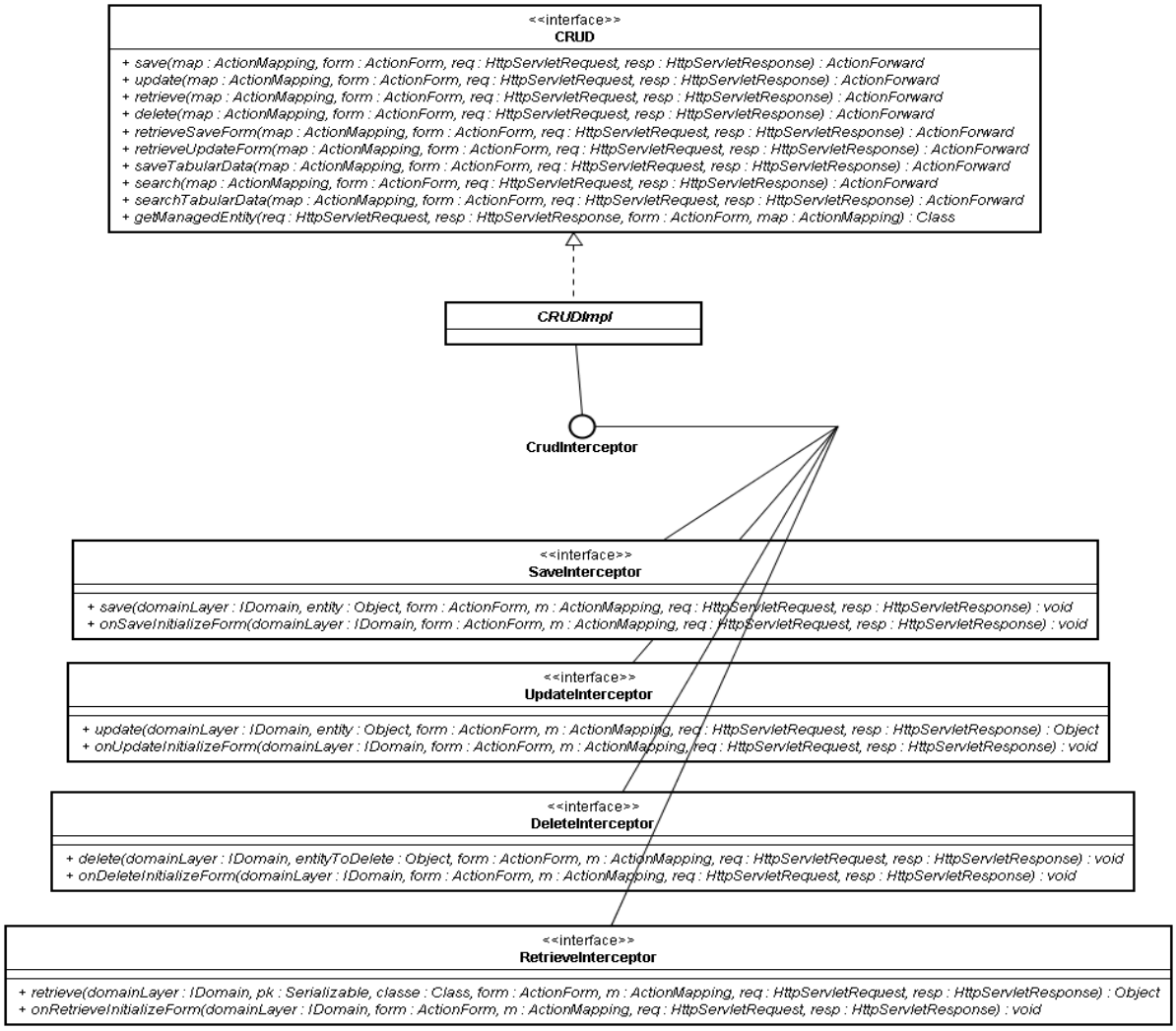

Figura 18: A Camada de Controle modela o atendimento de requisições para a execução de operações de persistência e provê interceptadores para especialização do atendimento de requisições, controle de componentes visuais e inicialização de componentes visuais que deverão ser apresentados na página de resposta.

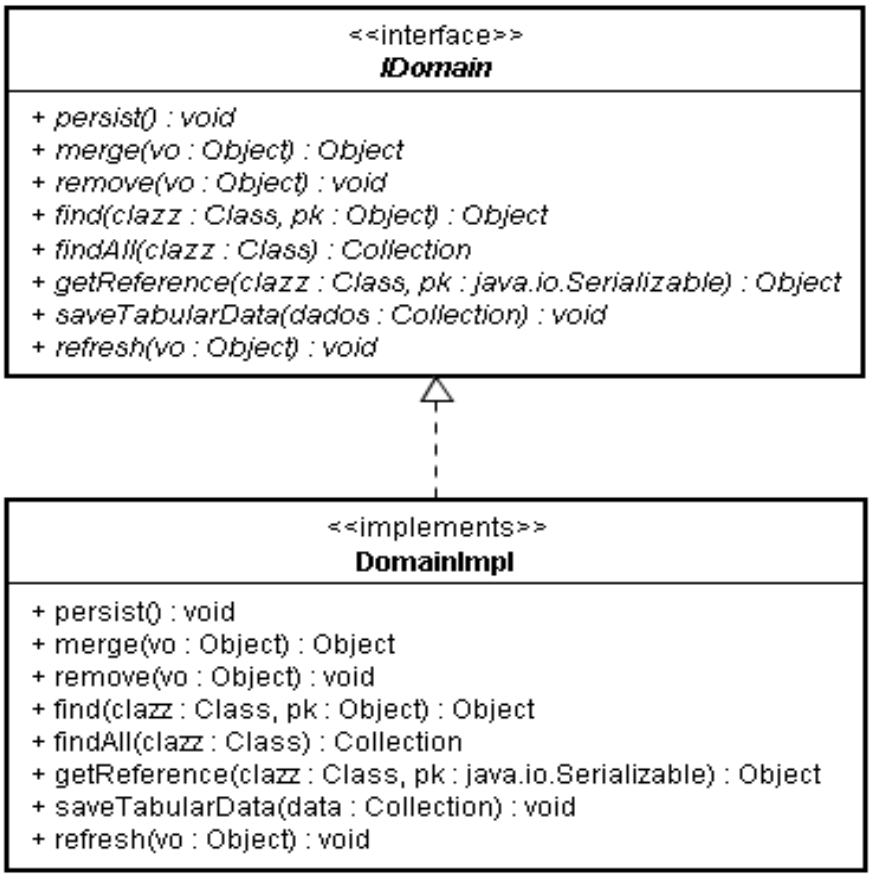

Figura 19: Diagrama de classes da camada de Domínio que contém dois artefatos: a interface *IDomain* e a classe *DomainImpl*. A camada de Domínio situa-se entre as camadas de Controle e Acesso à Dados.

B.1.3 Camada de Acesso à Dados

A camada abstrai detalhes da especificação JPA e implementa o acesso genérico a repositórios de dados fazendo uso de objetos de domínio. Na figura 20 a classe *DaoImpl* pode ser estendida por aplicações para especialização ou criação de novas operações de persistência.

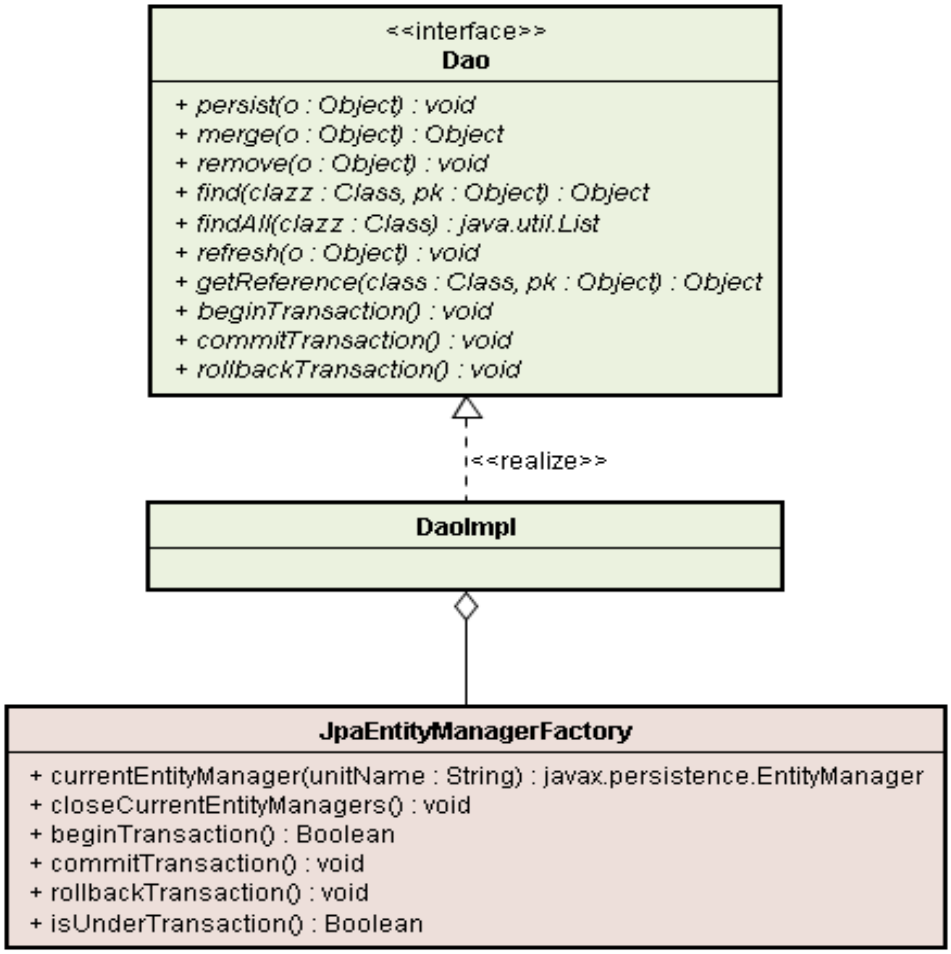

Figura 20: Arquitetura da camada de persistência. Na figura, a classe *DaoImpl* não apresenta os métodos implementados da interface *Dao*.

B.2 Sequência

B.2.1 Atendimento de Requisições

O Foundation executa uma série de operações comuns durante o atendimento de requisições. São operações internas e seu conhecimento deve servir para objeto de estudo,

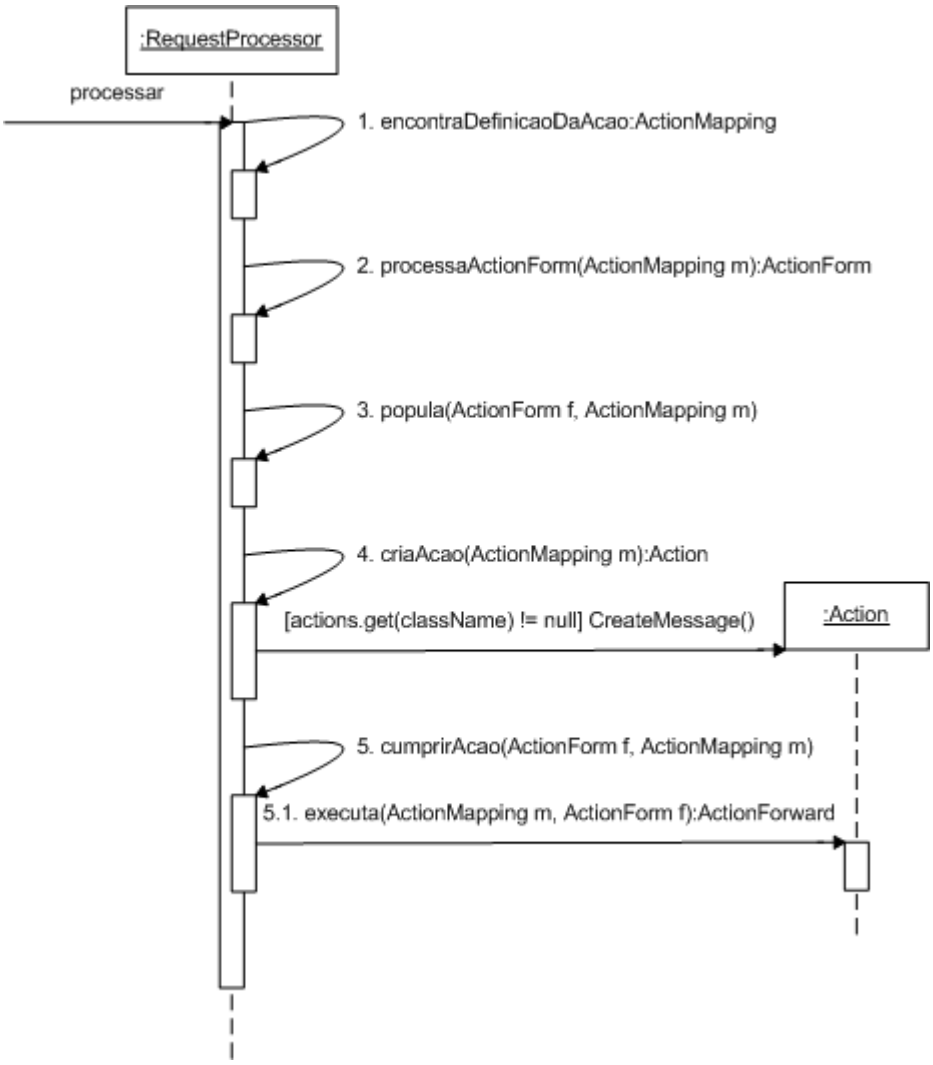

Figura 21: Diagrama de seqüência simplificado da camada de controle

não sendo necessário para o desenvolvimento de aplicações. Durante o atendimento de requisições, estas operações são executadas antes das ações definidas na interface *CRUD* (figura 18).

1. A URL de cada requisição que chega ao servidor de aplicações é definida no seguinte padrão: `http://<servidor>:<porta>/<nomeDaAplicacao>/<acao>`. O método *encontraDefinicaoDaAcao* utiliza a parte final da URL (/¡acao¿) para encontrar a definição da ação a ser executada. O objeto retornado da classe *ActionMapping* contém todos os elementos presentes na definição da ação e, dentre eles, a definição do *ActionForm* que deverá ser preenchido com os dados recebidos, o método que

contém a ação, sua classe e as páginas de resposta que poderão ser retornadas[1].

2.O *ActionForm* associado à definição da ação solicitada poderá ou não ser criado a cada requisição. Por padrão, a cada requisição atendida uma nova instância é criada[2]. O método processaActionForm faz esta verificação e retorna a instância de um objeto *ActionForm*.

3.Após a conversão dos tipos de dados (todos recebidos como texto) o *ActionForm* é preenchido.[3].

4.A classe que contém implementação de uma ou mais ações é do tipo *Action*. Apenas a primeira requisição que solicita a execução de algum de seus métodos determina a instanciação de um objeto da classe e inclusão de sua referência em um *cache* para acessos futuros. [4]

5.A classe *RequestProcessor* delega a execução da ação solicitada para um determinado método de uma determinada classe, ambos definidos na configuração da ação.

B.3 Diagramas de Sequência das ações de persistência

As ações *save, retrieve, update* e *delete* executam algumas operações em comum. São as operações:

- *getDomain():IDomain*, retorna um novo objeto da camada de Domínio para ser utilizado durante a ação.

- *getManagedEntity():Object*, retorna um objeto da classe que define o conjunto de dados manipulado durante a ação. A classe deve ser informada na configuração da ação.

1 Estes dados fazem parte da configuração de cada ação definida no arquivo struts-config.xml
2 Na configuração de cada ação é possível definir diferentes escopos para o *ActionForm* utilizado. Exemplos de escopos são: requisição, sessão, aplicação
3 O *ActionForm* pode conter agregações do tipo Um-Para-Um ou Um-Para-Muitos. A conversão dos tipos de dados recebidos como texto para tipos definidos pela linguagem Java, a instanciação de objetos agregados e atribuição dos valores recebidos para seus atributos internos é chamada de processamento de requisição que é executado dentro do método que popula o *ActionForm*.
4 Esta é uma abordagem que visa a escalabilidade de aplicações em situações de grande volume de requisições. Portanto, as classes que implementam ações não são *thread-safe*.

- *getInterceptor():CrudInterceptor*, retorna uma instância de um interceptador configurado para a ação. Somente um objeto de cada classe de interceptadores é criada por aplicação. Esta é uma abordagem que visa performance, seguindo a arquitetura do Struts. Portanto, assim como as classes da camada de Controle, os interceptadores não são *thread-safe*.

- *populateEntity():void*, preenche a entidade gerenciada com os dados do formulário. Caso o formulário possua mais atributos que a entidade gerenciada, estes serão desconsiderados.

B.3.0.1 Operação Incluir

A figura 22 apresenta a sequência de operações executadas pela ação de inclusão de dados *save*.

B.3.0.2 Operação Detalhar

A figura 23 apresenta a sequência de operações executadas pela ação de inclusão de dados *retrieve*.

B.3.0.3 Operação Alterar

A figura 24 apresenta a sequência de operações executadas pela ação de inclusão de dados *update*.

B.3.0.4 Operação Excluir

A figura 25 apresenta a sequência de operações executadas pela ação de inclusão de dados *delete*.

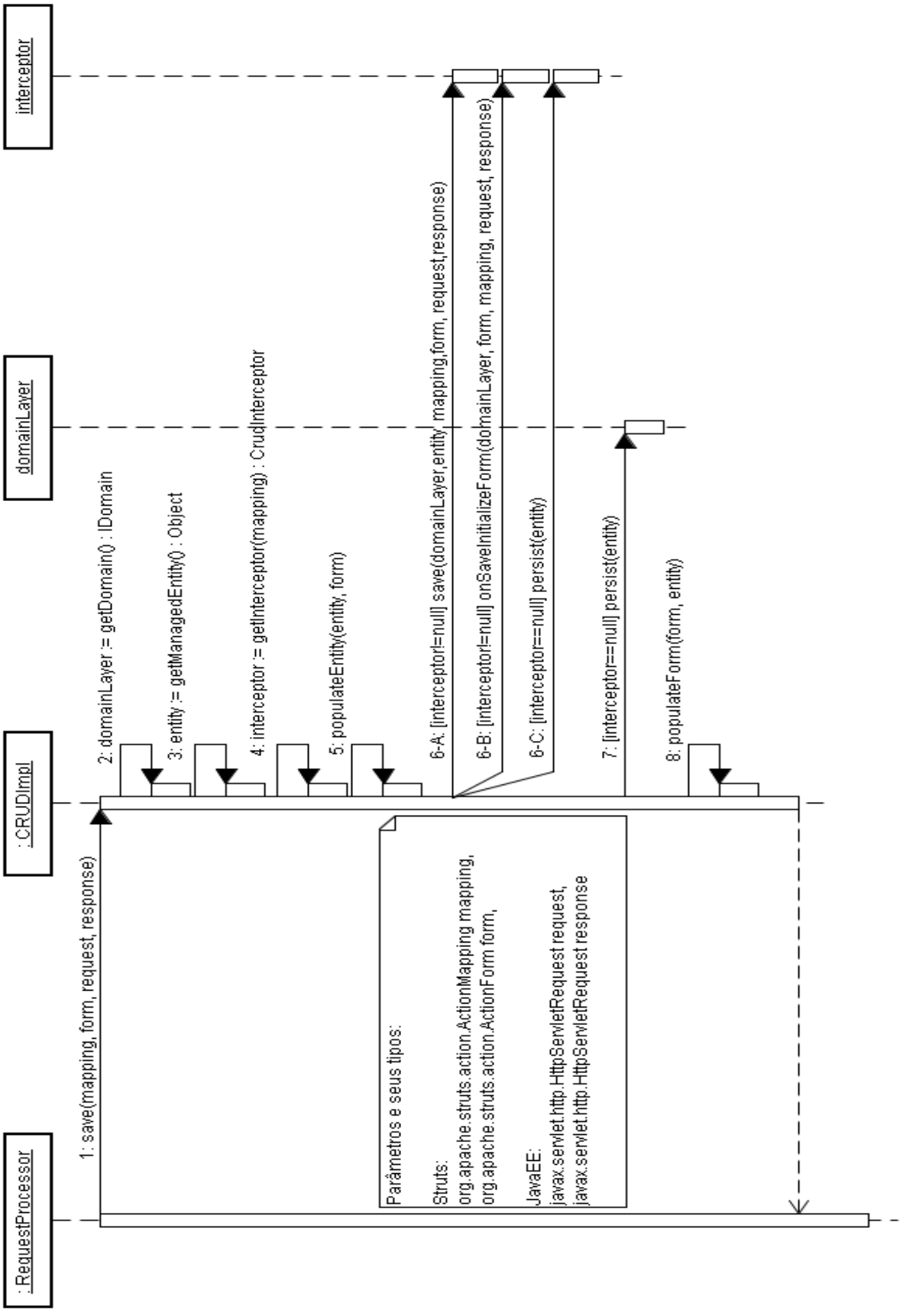

Figura 22: Diagrama de seqüência das operações executadas pela ação *save*.

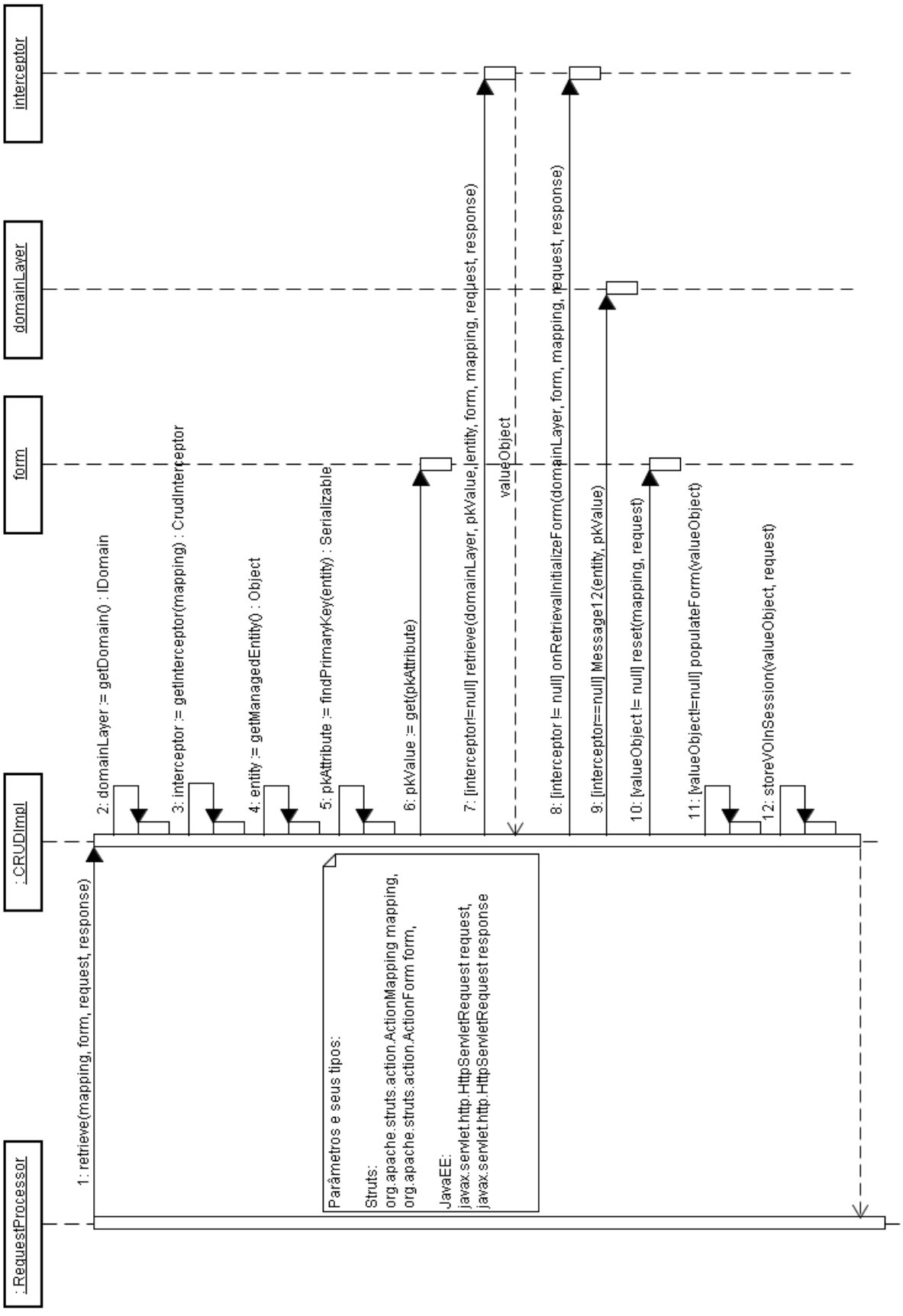

Figura 23: Diagrama de seqüência das operações executadas pela ação *retrieve*.

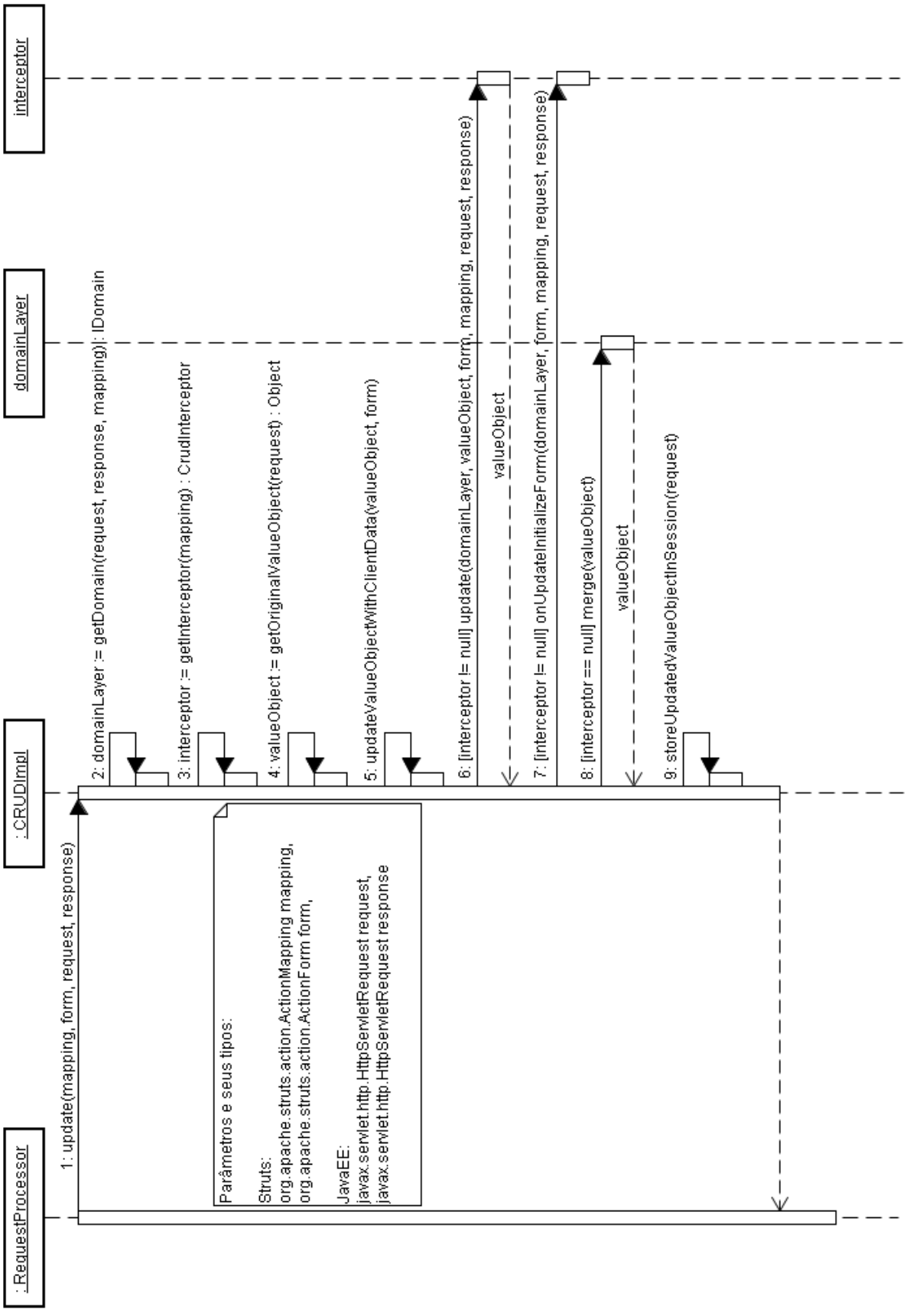

Figura 24: Diagrama de seqüência das operações executadas pela ação *update*.

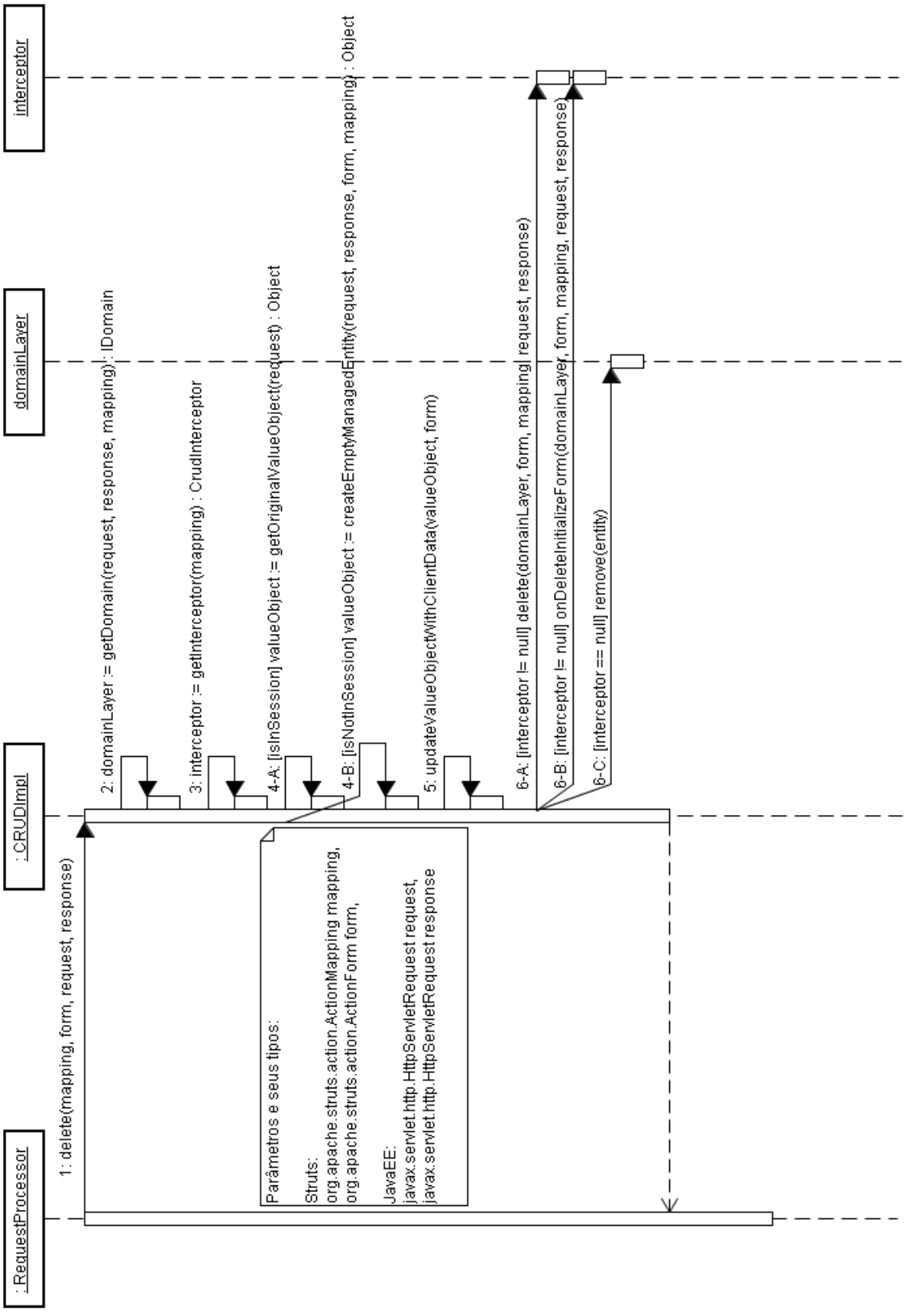

Figura 25: Diagrama de seqüência das operações executadas pela ação *delete*.