THIAGO DA ROSA DE BUSTAMANTE

CRUX: UM ARCABOUÇO DE SOFTWARE PARA DESENVOLVIMENTO DE APLICAÇÕES WEB

Belo Horizonte,
Agosto de 2008

THIAGO DA ROSA DE BUSTAMANTE
Orientador: Rodolfo Sérgio Ferreira de Resende

CRUX: UM ARCABOUÇO DE SOFTWARE PARA DESENVOLVIMENTO DE APLICAÇÕES WEB

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Belo Horizonte,
Agosto de 2008

UNIVERSIDADE FEDERAL DE MINAS GERAIS

FOLHA DE APROVAÇÃO

Crux: Um Arcabouço de Software para
Desenvolvimento de Aplicações Web

THIAGO DA ROSA DE BUSTAMANTE

Dissertação de mestrado defendida e aprovada pela banca examinadora constituída por:

Ph. D. Rodolfo Sérgio Ferreira de Resende – Orientador
Universidade Federal de Minas Gerais

Ph. D. Clarindo Isaías P. da Silva e Pádua
Universidade Federal de Minas Gerais

Ph. D. Jussara Marques de Almeida
Universidade Federal de Minas Gerais

Dedico este trabalho ao meu filho Vinícius, à minha esposa Thalita e aos meus pais, Cristina e Jorge.

Agradecimentos

Gostaria de agradecer a todas as pessoas que me ajudaram a realizar este trabalho.

A todos os professores e funcionários do Departamento de Ciências da Computação da UFMG, por tudo o que aprendi aqui.

Ao orientador Rodolfo Sérgio Ferreira de Resende, por toda a ajuda dada durante esse período.

Aos meus amigos da Sysmap Solutions, que me deram imenso apoio. Um agradecimento especial ao Anderson dos Santos e ao Cláudio Holanda, que me ajudaram de forma decisiva.

Aos meus pais, Cristina e Jorge, que sempre me apóiam em todos os momentos.

À minha esposa Thalita e ao meu filho Vinícius, pela alegria e companheirismo.

Resumo

Para produzir *softwares* de qualidade, de uma forma eficiente, é necessário um reuso sistemático de modelos, desenhos e implementações que tenham sido previamente testados.

Dentre as principais formas de reuso, destacam-se os arcabouços de *software* (ou *frameworks*). Estes, combinando reuso de desenho com reuso de código, representam o estado da arte em termos de reutilização em *softwares* orientados a objetos.

Este trabalho apresenta o Crux, um arcabouço criado para auxiliar o desenvolvimento de aplicações no domínio da Web. Estas aplicações são sistemas distribuídos complexos que estão crescendo em número e em importância, tornando-se, para muitas empresas, um recurso chave para realizar seus negócios.

O Crux visa auxiliar a criação de interfaces ricas, proporcionando uma maior qualidade de interação para os usuários e um modelo de desenvolvimento mais simples para os desenvolvedores.

Abstract

To produce quality software, in a cost effective manner, is necessary a systematic reuse of software models, designs and implementations previously tested.

Among all reuse methods, we can emphasize the application frameworks. Combining design and code reuse, they represent the state of the art in terms of reuse in object oriented softwares.

We present Crux, an application framework created to help the development of applications on the Web domain. These applications are complex distributed systems that are arising fast in number and importance. For many companies they are a key factor for success in business.

The Crux framework aims to help rich interfaces creation, improving quality of interaction for users and simplifying the development for programmers.

Sumário

CAPÍTULO 1 - INTRODUÇÃO **14**

1.1 OBJETIVO DO TRABALHO 15
1.2 CONTRIBUIÇÕES DO TRABALHO 16
1.3 ORGANIZAÇÃO DA DISSERTAÇÃO 16

CAPÍTULO 2 - REVISÃO BIBLIOGRÁFICA **18**

2.1 ARCABOUÇOS E REUSO SISTEMÁTICO 18
2.1.1. Arcabouços 19
2.1.1.1. Definição 19
2.1.1.2. Classificação 20
2.1.1.3. Conseqüências do Uso de Arcabouços 22
2.1.2. Padrões 26
2.1.3. Bibliotecas de classes 27
2.1.4. Componentes 27
2.2 WWW 28
2.2.1. Internet e Web 28
2.2.2. Aplicações Web 29
2.2.3. Conseqüências da Utilização da Web para Aplicações 30
2.2.3.1. Benefícios da Utilização da Web para Aplicações 30
2.2.3.2. Desafios na Utilização da Web por Aplicações 31
2.2.4. Tecnologias e Conceitos na Web 32
2.2.4.1 URL 32
2.2.4.2 HTML 33
2.2.4.3 HTTP 33
2.2.4.4 Navegador 33
2.2.4.5 DOM 34
2.2.4.6 CSS 34
2.2.4.7 SCRIPT 35
2.2.4.8 DHTML 35
2.2.4.9 AJAX 36
2.2.5. Linguagens de Programação Para Web 37
2.3 ARCABOUÇOS PARA WEB 38
2.3.1. MVC e Web 38
2.3.2. Características Desejáveis 39
2.3.3. Principais Arcabouços para Web 40
2.3.3.1. Struts 41
2.3.3.2. JSF 49
2.3.3.3. GWT 58
2.3.3.4. Análise das Soluções 68

CAPÍTULO 3 - O ARCABOUÇO CRUX **71**

3.1 CARACTERÍSTICAS GERAIS 71
3.1.1. Objetivo 71
3.1.2. Plataforma 72
3.1.3. Escopo 72
3.2 ARQUITETURA 72
3.2.1. Visão Geral 73
3.2.1.1 Contexto 73
3.2.1.2 Desenvolvimento 75
3.2.2. Detalhamento da Arquitetura 79
3.2.2.1 Árvore de Componentes 81
3.2.2.2 Motor Servidor 85
3.2.2.3 Motor Cliente 97
3.3 ANÁLISE DO ARCABOUÇO 106
3.3.1 O Crux e os Desafios Relacionados aos Arcabouços 106
3.3.1.1 Desenvolvimento 106

3.3.1.2 Utilização ... 106
3.3.1.3 Composição ... 107
3.3.1.4 Manutenção ... 108
3.3.2 O Crux e os Desafios Relacionados à Web ... 108
3.3.3 Considerações Gerais ... 109
3.3.4 O Crux e Outras Soluções ... 110
3.3.4.1 Padrões de Projeto no Crux ... 110
3.3.4.2 Comparação do Crux com as Outras Soluções ... 111
3.3.5 O Desempenho do Crux ... 112

CAPÍTULO 4 - DETALHAMENTO DA SOLUÇÃO ... 116

4.1 MOTOR SERVIDOR ... 116
4.1.1 Início da Aplicação ... 116
4.1.2 Árvore de Componentes ... 118
4.1.3 Tratamento de Eventos ... 119
4.1.3.1 Evento Server-RPC ... 120
4.1.3.2 Evento Server-Auto ... 121
4.1.4 Internacionalização ... 121
4.2 MOTOR CLIENTE ... 123
4.2.1 Início da Aplicação ... 123
4.2.2 Árvore de Componentes ... 124
4.2.3 Tratamento de Eventos ... 125

CAPÍTULO 5 - CONCLUSÃO ... 130

5.1 LIÇÕES APRENDIDAS ... 130
5.2 TRABALHOS FUTUROS ... 130

APÊNDICE A – PADRÕES DE PROJETO. ... 132

APÊNDICE B – DETALHES DE CÓDIGOS. ... 134

APÊNDICE B.1 – EVENTOS E COMPONENTES 134
APÊNDICE B.2 – TRATAMENTO DE EVENTOS NO SERVIDOR. ... 134
APÊNDICE B.3 – ASSOCIAÇÃO DE UMA CLASSE DE ESCRITA A UM COMPONENTE. ... 136
APÊNDICE B.4 – EXEMPLO DE RESPOSTA ENVIADA AO MOTOR CLIENTE. ... 136
APÊNDICE B.5 – ASSOCIAÇÃO DE UMA CLASSE DE ANÁLISE A UM COMPONENTE. ... 137
APÊNDICE B.6 – CONFIGURAÇÃO DO MECANISMO DE GERÊNCIA DE ESTADO DAS ÁRVORES DE COMPONENTES. ... 137
APÊNDICE B.7 – ASSOCIAÇÃO DE GERADORES NO CRUX. ... 137
APÊNDICE B.8 – CONFIGURAÇÃO DA CLASSE DE FÁBRICA DOS PROCESSADORES DE EVENTO. ... 138
APÊNDICE B.9 – BUSCA DE CLASSES POR ANOTAÇÃO. ... 139
APÊNDICE B.10 – MÉTODO DE ANÁLISE PADRÃO PARA COMPONENTES NO CRUX. ... 140
APÊNDICE B.11 – GERENCIADORES DE ESTADO DA ÁRVORE DE COMPONENTES. ... 141
APÊNDICE B.12 – CLASSES DO CICLO DE VIDA DO SERVLET PARA TRATAMENTO DE EVENTOS SERVER-RPC 142
APÊNDICE B.13 – CLASSES DO CICLO DE VIDA DO SERVLET PARA TRATAMENTO DE EVENTOS SERVER-AUTO ... 145
APÊNDICE B.14 – ADAPTER PARA DESENHO DE COMPONENTES DO GWT NO CRUX. ... 147
APÊNDICE B.15 – MÉTODOS PARA SERIAÇÃO DOS PARÂMETROS DA TELA. ... 148

REFERÊNCIAS 149

Índice de Figuras

Figura 2-1 – Os Dois Modelos de Aplicações Web (Garrett 2005). 37
Figura 2-2 – MVC Clássico (Husted et al., 2003).. 38
Figura 2-3 – Camadas em Aplicações Web (Husted et al., 2003).. 39
Figura 2-4 – MVC2 no *Struts* (Dudney et al., 2004)... 42
Figura 2-5 – MVC2 no JSF (Dudney et al., 2004)... 50
Figura 2-6 – Ciclo de Vida de uma requisição no JSF (Dudney et al., 2004)......................... 51
Figura 2-7 – Implementação do *Composite* no JSF (Dudney et al., 2004). 58
Figura 2-8 – Principais Componentes do GWT (Hanson-Tacy, 2007). 60
Figura 3-1 – Visão Geral da Arquitetura do Crux.. 74
Figura 3-2 – Desenvolvimento com o Crux. 77
Figura 3-3 – Principais Componentes do Arcabouço Crux... 80
Figura 3-4 – Estrutura de uma Tela no Crux.. 81
Figura 3-5 – Carregando uma Aplicação com o Crux. 87
Figura 3-6 – Tratamento de Eventos no Motor Servidor. 88
Figura 3-7 – Classes Envolvidas no Registro de Componentes no Cliente. 98
Figura 3-8 – Classes Envolvidas no Registro de Tratadores de Evento no Cliente. 99
Figura 3-9 – Classes Envolvidas no Tratamento de Evento no Cliente. 101
Figura 3-10 – Classes Envolvidas na Análise da Resposta de um Evento Server-Auto........ 104
Figura 3-11 – Classes Envolvidas na Criação da Árvore de Componentes. 105
Figura 4-1 – Classes Envolvidas com as Fases de Eventos Processados no Servidor. 120

Índice de Listagens com Código-Fonte

Listagem 2-1 – Configuração do Controlador do *Struts* (Husted et al., 2003)........................ 43
Listagem 2-2 – Exemplo do Arquivo struts-config.xml. 44
Listagem 2-3 – Exemplo de *Action* do *Struts*.. 44
Listagem 2-4 – Exemplo de Arquivo de Mensagens no *Struts*. 45
Listagem 2-5 – Exemplo de *ActionForm* do *Struts*.. 45
Listagem 2-6 – Exemplo do Arquivo validation.xml... 46
Listagem 2-7 – Exemplo de Página com marcações do *Struts*. 47
Listagem 2-8 – Configuração do Controlador do JSF (Dudney et al., 2004). 52
Listagem 2-9 – Exemplo de um *Managed Bean* no JSF (Dudney et al., 2004)....................... 53
Listagem 2-10 – Exemplo de Regras de Navegação no JSF (Dudney et al., 2004)................ 54
Listagem 2-11 – Declarando um *Managed Bean* no JSF (Dudney et al., 2004)...................... 54
Listagem 2-12 – Exemplo de Página com Componentes do JSF (Dudney et al., 2004).......... 55
Listagem 2-13 – Exemplo de *Validador* no JSF (Dudney et al., 2004). 56
Listagem 2-14 – Exemplo Recuperação de Mensagem no JSF. 56
Listagem 2-15 – Exemplo de Configuração de um Módulo GWT (Hanson-Tacy, 2007)....... 59
Listagem 2-16 – Exemplo de JSNI do GWT. 61
Listagem 2-17 – Exemplo de Tratamento de Eventos no GWT (Hanson-Tacy, 2007). 63
Listagem 2-18 – Exemplo de Arquivo de Mensagens no GWT (Hanson-Tacy, 2007). 63
Listagem 2-19 – Exemplo de Interface de Mensagens no GWT (Hanson-Tacy, 2007). 64
Listagem 2-20 – Exemplo de Uso de Arquivo de Mensagens no GWT 64
Listagem 2-21 – Exemplo de Interface Remota (Hanson-Tacy, 2007).................................. 65
Listagem 2-22 – Exemplo de Classe Remota de Serviço (Hanson-Tacy, 2007). 65
Listagem 2-23 – Exemplo de Chamada RPC no GWT (Hanson-Tacy, 2007). 65
Listagem 2-24 – Exemplo de Controle do Histórico no GWT (Hanson-Tacy, 2007). 66
Listagem 2-25 – Exemplo de Criação de Ficha de Histórico no GWT.................................. 67
Listagem 3-1 – Exemplo de Tela no Crux. 76
Listagem 3-2 – Exemplo de Tratador de Eventos no Crux... 78
Listagem 3-3 – Exemplo de Tratador de Evento Cliente no Crux.. 79
Listagem 3-4 – Exemplo de Registro de Componentes no Crux.. 83
Listagem 3-5 – Exemplo de Arquivo Crux.properties. 86
Listagem 3-6 – Exemplo de Transferência de Dados Entre Componente e Servidor............. 90
Listagem 3-7 – Exemplo de Tratador de Evento Server-RPC no Crux. 90
Listagem 3-8 – Exemplo de Método Tratador de Resposta de um Evento Server-RPC. 92
Listagem 3-9 – Exemplo de Chamada de Evento Server-Auto no Crux. 93
Listagem 3-10 – Exemplo de Tratador de Evento Server-Auto no Crux............................... 93
Listagem 3-11 – Exemplo de Uso de Mensagens Internacionalizadas. 96
Listagem 3-12 – Exemplo de Interface Usada para Internacionalização............................... 96
Listagem 3-13 – Exemplo de Arquivo de Propriedades Usado para Internacionalização....... 97
Listagem 3-14 – Classe Gerada para Registro dos Componentes... 99
Listagem 3-15 – Classe Gerada para Registro dos Tratadores de Evento 100
Listagem 4-1 – Método Chamado para Criação de um Componente na *ScreenFactory*....... 118
Listagem 4-2 – Interface *ComponentParser*. 118
Listagem 4-3 – Interface *ScreenStateManager*. 119
Listagem 4-4 – Interface *ComponenteRenderer*. 121
Listagem 4-5 – Método *initProxy*, da Classe *MessagesFactory*. 122
Listagem 4-6 – Método de Fábrica da Classe *MessagesFactory*. 123

Listagem 4-7 – Interface *CruxConfig* 123
Listagem 4-8 – Método *generatePreserveStateMethod*, da Classe *CruxConfigGenerator*. .. 124
Listagem 4-9 – Método *newComponent*, da Classe *ScreenFactory* 124
Listagem 4-10 – Método *callEvent*, da Classe *EventFactory*. 125
Listagem 4-11 – Método *createEventProcessor*, da Classe *EventProcessorFactoryImpl*. ... 126
Listagem 4-12 – Método *createServerAutoEventProcessor*, da Classe *EventProcessorFactoryImpl* 127
Listagem 4-13 – Métodos *getPostData* e *updateScreen*, da Classe *ScreenSerialization* 128
Listagem 4-14 – Método *serialize*, da Classe *ComponentBase*. 129
Listagem 4-15 – Método confirmSerialization, da Classe *ScreenSerialization*. 129
Listagem B-1 – Exemplo de Eventos no Crux. 134
Listagem B-2 – *BaseServlet*. 135
Listagem B-3 – EventServerAutoServlet 135
Listagem B-4 – Exemplo de Tratador de Evento com Ciclo de Vida Personalizado. 136
Listagem B-5 – Associação de uma Classe de Escrita a um Componente 136
Listagem B-6 – Exemplo de Alteração na Árvore de Componentes Enviada pelo Crux. 136
Listagem B-7 – Associação de uma Classe de Análise a um Componente. 137
Listagem B-8 – Mecanismo de Gerência de Estado das Árvores de Componentes 137
Listagem B-9 – Associação dos Geradores. 137
Listagem B-10 – Configuração da Classe de Fábrica dos Processadores de Evento. 138
Listagem B-11 – Classe *ClassScanner*. 139
Listagem B-12 – Método de Análise da Classe *ComponentParserImpl*. 140
Listagem B-13 – Classe *ScreenStateManagerServerImpl*. 141
Listagem B-14 – Classe *ScreenStateManagerClientImpl*. 141
Listagem B-15 – Classe *AbstractScreenStateManager*. 142
Listagem B-16 – Classe *RPCDispatchPhase*. 142
Listagem B-17 - Classe *RPCRenderResponsePhase*. 143
Listagem B-18 – Classe *AbstractRenderResponsePhase* 144
Listagem B-19 – Classe *AutoDispatchPhase*. 145
Listagem B-20 – Classe *AutoRenderResponsePhase*. 146
Listagem B-21 – Método *render*, da Classe *ComponentBase*. 147
Listagem B-22 – Métodos para Seriação dos Parâmetros da Tela, da classe *ScreenSerialization* 148

Índice de Tabelas

Tabela 2-1 – Padrões de Projeto no *Struts* (Husted et al., 2003)............ 48
Tabela 2-2 – Padrões de Projeto no JSF............ 57
Tabela 2-3 – Modos de Saída do Compilador GWT............ 61
Tabela 2-4 – Padrões de Projeto no GWT............ 67
Tabela 3-1 – Padrões de Projeto no Crux............ 111
Tabela A-1 – Padrões de Projeto (Gamma et al, 2002)............ 132
Tabela A-2 – Padrões de Projeto (Deepak et al, 2001). 133

CAPÍTULO 1 - Introdução

A construção de *softwares* de qualidade, de uma forma efetiva em termos de custo, necessita de um reuso sistemático de modelos de software, desenhos e implementações que foram previamente desenvolvidos e testados (Schmidt-Buschman, 2003).

A expressão “reuso sistemático” não se refere aqui a formas de reuso oportunistas, onde desenvolvedores simplesmente copiam e colam trechos de programas existentes para criarem novos. Ao contrário, essa expressão remete a um esforço intencional em criar e utilizar artefatos multiuso de software no processo de desenvolvimento de uma organização (Schmidt-Buschman, 2003).

Em um processo bem estruturado de reuso sistemático, reduz-se em cada novo projeto o tempo para desenho e implementação, adicionando somente novos códigos específicos para um aplicativo particular. Reestruturam-se arquiteturas existentes apenas quando estas se tornam inadequadas para cobrir as evoluções de negócio em um domínio específico (Schmidt-Buschman, 2003).

Estudos mostram que as falhas de reengenharia podem ocorrer em uma taxa de até 70% (Fingar, 1996) (Fayad, 2000). Reuso sistemático é essencial para aumentar a produtividade do desenvolvimento e a qualidade do software, evitando ter de redescobrir, reinventar e revalidar artefatos comuns. O reuso, no entanto, não deve se restringir a componentes, classes e códigos fonte. A reutilização de desenhos de soluções para problemas recorrentes também é fundamental para aumentar a qualidade e a produtividade (Schmidt-Buschman, 2003).

Dentre as principais formas de reuso sistemático, combinando reuso de desenho com reuso de código, destacam-se os arcabouços para aplicações (*application frameworks* ou ainda *software frameworks*), que representam o estado da arte em termos de estruturação e reutilização em *softwares* orientados a objetos (Brugali-Sycara, 2000) e rapidamente estão se tornando necessidade estratégica para organizações em todo o mundo (Fayad, 2000).

Funcionando como um esqueleto para a criação de uma aplicação, um bom arcabouço pode reduzir o esforço para desenvolvê-la em uma ordem de grandeza (Johnson, 1997).

Esta dissertação destaca pontos importantes a serem considerados na elaboração de um arcabouço para o desenvolvimento de aplicações no domínio da Web. Com base nas diversas peculiaridades deste tipo de sistema e no estudo dos arcabouços enquanto técnica

de reuso, foi construído um arcabouço para facilitar a criação de aplicações com interfaces ricas, que proporcionem uma melhor qualidade de interação com o usuário.

A escolha desse domínio como alvo do trabalho realizado foi motivada pelo fato de o desenvolvimento de aplicações Web ser particularmente complexo e dispendioso de tempo (Zhang-Buy, 2003), necessitando, ainda mais que em outros domínios, de técnicas de reuso sistemático, para que possam ter qualidade e custo compatíveis com as exigências atuais do mercado (Schwabe et al., 2001).

Além da complexidade, outro fator que motiva a criação de um arcabouço para aplicações Web é o enorme aumento na demanda por aplicações deste tipo (Nielsen, 1999), (Ginige-Murugesan, 2001). Atualmente, muitas empresas vêem a Web como um recurso chave para realizar seus negócios (Baresi-Morasca, 2007).

1.1 Objetivo do Trabalho

O objetivo deste trabalho foi criar uma solução para o problema de desenvolvimento de aplicações no domínio da Web. Para isto, decidimos adotar a abordagem dos arcabouços de software.

Desta forma, desenvolvemos o arcabouço denominado Crux. Este visa melhorar a qualidade das interfaces gráficas e a comunicação dos dados entre cliente e servidor em aplicações neste domínio, proporcionando uma melhor interação com o usuário.

Vale ressaltar que não foi nosso intuito, neste trabalho, produzir um produto acabado, mas vivenciar o seu processo de construção. Desta forma, as implementações apresentadas buscam apenas ilustrar as idéias elaboradas e funcionar como prova de conceito para as mesmas.

Também não tivemos a pretensão de construir uma solução definitiva para o problema de se desenvolver aplicações Web, uma vez que os navegadores – *middlewares*[1] utilizados para interface com usuário destas aplicações – assim como diversas outras tecnologias envolvidas em seu processo de criação, encontram-se em um estágio de desenvolvimento muito acelerado (Lowe et al., 2001). Analisando-se as listas de funcionalidades previstas para serem adicionadas nas próximas versões de navegadores, como o firefox (Firefox, 2008), pode-se perceber como os navegadores estão evoluindo para uma plataforma de

[1] Softwares reutilizáveis que visam cobrir o buraco existente entre os requisitos funcionais das aplicações e os sistemas operacionais, protocolos de rede, bancos de dados e outras interfaces de programação de mais baixo nível (Schmidt-Buschman, 2003).

desenvolvimento, adicionando inúmeros recursos projetados para suportar aplicações e não apenas navegação de páginas desconectadas. Esta evolução também é comentada por Jazayeri (Jazayeri, 2007).

Desta forma, neste trabalho, considerou-se o atual estado dos *middlewares* e das demais tecnologias disponíveis. No entanto, acreditamos que alternativas melhores devem surgir em curto prazo para amparar este desenvolvimento.

1.2 Contribuições do Trabalho

Este trabalho trás algumas contribuições importantes:

Em primeiro lugar, apresenta um estudo dos arcabouços, destacando vantagens e desafios relacionados a esta técnica de reuso.

Em segundo lugar, mostra uma comparação de alguns dos mais utilizados arcabouços atuais para desenvolvimento na Web, mostrando os principais padrões de projeto empregados, assim como as soluções adotadas para os problemas mais comuns.

Uma terceira contribuição é o próprio arcabouço produzido, que, apesar de ainda não ser um produto totalmente acabado, pode ser utilizado como base para construção de um produto final, para utilização em aplicações reais.

1.3 Organização da Dissertação

Esta dissertação está organizada da seguinte forma:

O capítulo 2 faz uma revisão da literatura. A seção 2.1 trata reuso em desenvolvimento de *softwares*, enfatizando arcabouços e o seu relacionamento com outras formas de reuso. A seção 2.2 apresenta as aplicações Web, destacando características deste tipo de sistemas, assim como os principais desafios no seu processo de desenvolvimento. A seção 2.3 mostra outros trabalhos na área, destacando alguns arcabouços mais conhecidos e aceitos pelo mercado atualmente.

O capítulo 3 apresenta o Crux, nosso arcabouço criado para auxiliar o desenvolvimento de aplicações no domínio da Web. A seção 3.1 apresenta as características gerais desta solução, a seção 3.2 mostra a arquitetura do arcabouço e a seção 3.3 mostra uma análise da solução apresentada.

O capítulo 4 apresenta um detalhamento da solução apresentada, mostrando trechos de código gerados. A seção 4.1 detalha aspectos relacionados à parte servidora da aplicação, enquanto a seção 4.2 detalha aspectos relacionados à parte cliente.

O capítulo 5 conclui a dissertação, destacando o nosso aprendizado e nossa visão de trabalhos futuros nesta área.

CAPÍTULO 2 - Revisão Bibliográfica

Este capítulo apresenta uma revisão bibliográfica sobre arcabouços e aplicações Web. Inicialmente, são abordados os arcabouços, comparando-os a outras formas de reuso e mostrando como estes se relacionam com cada uma dessas formas apresentadas.

Em seguida, são abordados as principais características e desafios das aplicações Web. A construção dessas aplicações é um processo complexo que requer conhecimento do domínio da aplicação, assim como um cuidadoso desenho de sua estrutura navegacional e de suas interfaces com usuário. (Schwabe et al., 2001).

Na seqüência são apresentados arcabouços para Web, destacando alguns que foram selecionados dentre os diversos existentes atualmente.

2.1 Arcabouços *e Reuso Sistemático*

Como visto no capítulo 1, o reuso sistemático é fundamental para garantir a construção de *softwares* de qualidade, de uma forma efetiva em termos de custo (Schmidt-Buschman, 2003). Ao longo deste trabalho, o termo "reuso" é utilizado sempre se referindo a "reuso sistemático".

Existem diversas formas de reuso que podem ser utilizadas na criação de *softwares*. Os arcabouços para aplicações são o estado da arte em termos de estruturação e reutilização em *softwares* orientados a objetos (Brugali-Sycara, 2000) e relacionam-se e com diversas técnicas de reuso que são exploradas ao longo deste capítulo.

A seção 2.1.1 procura conceituar arcabouço, assim como apresentar as suas diferentes classificações e as conseqüências decorrentes de se escolher tal abordagem em um processo de uma organização.

A seção 2.1.2 apresenta os "Padrões de Projeto" ou *Design Patterns*, que expressam experiências de desenho reutilizáveis e estão intrinsecamente relacionados aos arcabouços. Pode-se considerar um arcabouço como uma implementação de diversos padrões, relacionando-os de forma a montar um esqueleto para aplicações em um determinado domínio (Schmidt-Buschman, 2003). Nesta seção, também são apresentadas as "Linguagens de Padrões" e é abordada a forma como estas podem ser utilizadas para auxiliar na construção de arcabouços.

A seção 2.1.3 mostra o uso de bibliotecas de classes e como esta forma de reuso está relacionada aos arcabouços.

A seção 2.1.4 trata de componentes e do relacionamento destes com os arcabouços. Segundo (Johnson, 1997), a visão original de reuso de software sempre esteve baseada em componentes. De maneira geral, componentes podem ser usados para construir arcabouços e arcabouços podem auxiliar no desenvolvimento de componentes (Fayad, 2000).

2.1.1. Arcabouços

Arcabouços são técnicas de reuso orientadas a objetos (Johnson, 1997). Estes, apesar de já serem usados com sucesso há algum tempo, ainda não são bem entendidos fora da comunidade de desenvolvedores aderentes ao paradigma da Orientação a Objetos e, frequentemente, são mal utilizados (Johnson, 1997).

A própria definição de arcabouço varia bastante. Nas seções seguintes, discutem-se as principais definições para o termo, assim como suas classificações, benefícios e desafios.

2.1.1.1. Definição

O conceito de arcabouço tem sido definido de diversas maneiras pela literatura da área. As principais definições encontradas por nós são descritas a seguir:

Para (Johnson-Foote, 1998) e (Fayad, 2000), *"arcabouços são aplicações semi-completas reutilizáveis que podem ser especializadas para produzir novas aplicações"*.

Para (Brugali-Sycara, 2000), *"um arcabouço é um conjunto integrado de componentes reutilizáveis e extensíveis para um domínio específico de aplicações. Uma aplicação semi-definida que deve ser personalizada a fim de construir aplicações concretas"*.

Para (Johnson, 1997), *"arcabouço é um desenho reutilizável de todas as partes de um sistema que é representado por um conjunto de classes abstratas e pelo modo como as instâncias interagem"*.

Para (Schwabe et al., 2001), *"um arcabouço é um desenho reutilizável construído através de um conjunto de classes abstratas e concretas e um modelo de colaboração entre objetos em um domínio específico"*.

As definições citadas não são conflitantes, mas, ao contrário, complementam-se. Enquanto algumas descrevem a estrutura de um arcabouço, outras explicam o seu propósito.

A definição adotada nesta dissertação é a de (Buschmann, 1996) e (Pree, 1995), que considera que *"um arcabouço é definido como um software parcialmente completo projetado para ser instanciado. O arcabouço define uma arquitetura para uma família de subsistemas e oferece os construtores básicos para criá-los. Também são explicitados os lugares ou pontos de extensão (hot-spots) nos quais adaptações do código para um funcionamento específico de certos módulos devem ser feitas"*.

2.1.1.2. Classificação

Pode-se classificar um arcabouço de acordo com diferentes critérios. Nesta seção, são mostradas duas formas de classificação: por escopo e pela forma de extensão utilizada.

Classificação por Escopo

De acordo com o escopo, segundo (Fayad, 2000) e (Fayad-Schmidth, 1997), pode-se dividir os arcabouços em três categorias: (1) infra-estrutura de sistemas, (2) integração de *middleware* e (3) arcabouços de aplicações corporativas.

Arcabouços de infra-estrutura de sistemas simplificam o desenvolvimento de sistemas de infra-estrutura portáveis e eficientes, como sistemas operacionais, arcabouços de comunicação, de interfaces gráficas e ferramentas de processamento de linguagens (Fayad, 2000).

Arcabouços de integração de *middleware* são usados para integrar aplicações e componentes distribuídos. Estes arcabouços escondem o baixo nível da comunicação entre componentes distribuídos, possibilitando que os desenvolvedores trabalhem em um ambiente distribuído de forma semelhante a que trabalham em um ambiente não distribuído. São exemplos de arcabouços de integração de *middleware:* Java RMI, arcabouços ORB (*Object Request Broker*) e bancos de dados transacionais (Fayad, 2000).

Arcabouços de aplicações corporativas são voltados para um domínio de aplicação específico, como por exemplo, os domínios da aviação, telecomunicações e financeiro. São mais caros de serem desenvolvidos ou comprados quando comparados aos dois tipos anteriores, pois são necessários especialistas do domínio de aplicação para construí-los. Entretanto, eles podem prover um retorno substancial já que encapsulam o conhecimento sobre o domínio em questão (Fayad, 2000).

Classificação Quanto a Forma de Extensão

De acordo com a forma de extensão, segundo (Fayad-Schmidth, 1997) e (Brugali-Sycara, 2000), pode-se dividir os arcabouços em duas categorias: caixa branca (*white box*) e caixa preta (*black box*).

Segundo (Fayad-Schmidth, 1997), pode-se definir estas duas categorias de arcabouços da seguinte forma:

"Arcabouços de caixa branca apóiam-se em características de linguagens orientadas a objetos, como herança e amarrações dinâmicas, para conseguir flexibilidade. As funcionalidades existentes são reutilizadas e estendidas através de (1) herança de classes básicas do arcabouço e (2) sobrescrevendo métodos-gancho[2] (*hook methods*) pré-definidos ou utilizando padrões como Métodos Template[3] (*Template Methods*) (Gamma et al., 1995)".

"Arcabouços de caixa preta oferecem flexibilidade através da definição de interfaces para componentes, que podem ser ligados ao arcabouço através de composição de objetos. Funcionalidades existentes são reutilizadas através: (1) da definição de componentes que seguem uma determinada interface e (2) da integração destes componentes ao arcabouço, utilizando padrões como *Strategy* e *Bridge* (Gamma et al., 1995)".

Alguns autores, como (Szyperski, 1997), também falam de arcabouços de componentes, que aqui são tratados por arcabouços de caixa preta.

Frequentemente, ambas as abordagens estão presentes em um mesmo arcabouço (Brugali-Sycara, 2000). Alguns autores classificam como caixa cinza os arcabouços que utilizam uma abordagem mista, procurando obter as vantagens existentes nas duas formas anteriores (Markiewicz-Lucena, 2001).

---

[2] Métodos com uma implementação padrão que pode ser redefinida na subclasse (Pree, 1995).
[3] Métodos abstratos que são implementados na subclasse (Gamma et al., 1995).

Os arcabouços de caixa branca exigem dos desenvolvedores de aplicações um conhecimento profundo da estrutura interna do arcabouço. Eles são amplamente difundidos, apesar de mais difíceis de utilizar. Os arcabouços de caixa preta, por sua vez, geralmente são mais fáceis de usar do que os de caixa branca, mas bem mais difíceis de serem criados, uma vez que exigem que seus criadores definam interfaces e ganchos que antecipem a maior faixa possível de potenciais casos de uso das aplicações (Johnson, 1995) (Fayad-Schmidth, 1997).

2.1.1.3. Conseqüências do Uso de Arcabouços

A adoção de arcabouços agrega uma gama de benefícios e desafios ao processo de desenvolvimento de uma organização. A seguir, são abordadas as principais conseqüências detectadas em nossa pesquisa.

Benefícios Obtidos com a Utilização de Arcabouços

Segundo (Fayad-Schmidth, 1997), os principais benefícios obtidos pela utilização de arcabouços são:

- **Modularidade:** Arcabouços encapsulam detalhes de implementação voláteis atrás de interfaces estáveis e bem definidas. A modularização proporcionada pelos arcabouços ajuda a aumentar a qualidade do software, pois os locais onde são feitas mudanças de projeto e de implementação ficam determinados, diminuindo o esforço para entender e manter a aplicação (Fayad-Schmidth, 1997).
- **Reuso:** Um arcabouço proporciona diversos tipos de reuso, ao: (1) definir componentes genéricos que podem ser utilizados para criar novas aplicações. (2) evitar recriar e revalidar desenhos de soluções para problemas recorrentes, reaproveitando o esforço e conhecimento de projetistas experientes. O reuso proporcionado pelos arcabouços pode aumentar substancialmente a produtividade dos programadores, assim como elevar a qualidade, desempenho, confiabilidade e interoperabilidade dos *softwares* (Fayad-Schmidth, 1997). Este reuso de desenho também ajuda a reduzir o esforço com manutenção, visto que ele trás uma uniformidade para as aplicações. Um programador pode dar manutenção em vários sistemas sem precisar aprender um novo desenho (Johnson, 1997).

- **Facilidade de extensão:** Arcabouços oferecem pontos de extensão explícitos que possibilitam aos desenvolvedores estenderem suas funcionalidades para gerar uma aplicação. Os pontos de extensão desconectam as interfaces estáveis do arcabouço e o comportamento de um determinado domínio das variações requeridas por uma determinada aplicação em um dado contexto (Fayad-Schmidth, 1997).
- **Inversão de Controle:** A arquitetura de um arcabouço é caracterizada por uma "inversão de controle" (*Inversion of Control* – IoC). Esta arquitetura permite que passos do processamento das aplicações sejam personalizados em objetos tratadores de eventos, que são chamados através de um mecanismo de disparo do arcabouço. Quando eventos ocorrem, o arcabouço reage chamando métodos pré-registrados em objetos tratadores, que realizam o processamento do evento, de acordo com as regras específicas da aplicação. Inversão de controle, também conhecida como princípio de Hollywood[4], permite ao arcabouço (ao invés da aplicação) determinar que método chamar em resposta a acontecimentos externos (como mensagens vindas de janelas, ou por alguma porta de comunicação) (Fayad-Schmidth, 1997).

Desafios Relacionados aos Arcabouços

Com base na experiência adquirida pela implementação de diversos arcabouços (Bosch et al., 2000) classificaram os principais desafios relacionados aos arcabouços em quatro categorias: (1) Desenvolvimento, (2) Utilização, (3) Composição e (4) Manutenção. Dentro de cada uma dessas categorias, são expostos diversos problemas. Os principais são tratados a seguir:

Desenvolvimento:

O desenvolvimento de um arcabouço de qualidade, em geral, é mais difícil que o desenvolvimento de aplicações, mesmo as mais complexas (Bosch et al., 2000). Entre os principais problemas relacionados ao desenvolvimento, segundo Bosch, pode-se destacar:

- **Modelos de negócio:** A criação de um arcabouço pode ser viável do ponto de vista tecnológico, mas não ser do ponto de vista de negócio. O retorno do investimento necessário para construir um arcabouço pode vir da sua posterior venda para outras companhias. Não existem muitos modelos de negócio para desenvolvimento de um arcabouço.

[4] Princípio de Hollywood : "*Don't call us, we will call you*" (Sparks et al., 1996)

- **Verificação de comportamentos abstratos:** Garantir a correção de um arcabouço é uma atividade complexa. Como eles definem comportamentos abstratos, não é possível testá-los completamente sem que sejam primeiramente instanciados.
- **Gerenciamento de versões:** Para que seja disponibilizada uma nova versão do arcabouço, é necessário que ele atenda a diversos requisitos. Entre estes, ele deve estar documentado. No entanto, não existe um método de documentação aceito que cubra todos os aspectos de um arcabouço e é difícil determinar quando o mesmo está suficientemente documentado para o seu público alvo.

Utilização:

Ao utilizar arcabouços de caixa branca – e mesmo os de caixa preta – é necessário entender seus conceitos e determinações de arquitetura, para que se possa desenvolver aplicações que estejam de acordo com o proposto pelo arcabouço (Bosch et al., 2000). Como as abordagens de documentação de arcabouços sofrem de diversos problemas, normalmente é difícil obter este conhecimento. Segundo Bosch, Isso causa os seguintes problemas:

- **Aplicabilidade do arcabouço:** Determinar se um arcabouço é adequado para o desenvolvimento de uma aplicação particular requer um esforço considerável em análise. No caso de apenas uma parte dos requisitos da aplicação ser contemplada, redesenhar o arcabouço, ou buscar uma solução de contorno pode ser inviável.
- **Estimativas de esforço de desenvolvimento:** É difícil prever o esforço necessário quando se utiliza um arcabouço, a menos que se tenha um profundo conhecimento do mesmo. Aplicações complexas podem ser escritas muito rapidamente. No entanto, um requisito não contemplado pelo arcabouço pode aumentar o esforço para o desenvolvimento, podendo superar o esforço de construir a aplicação do zero.
- **Depuração de aplicações:** Depuradores tradicionais têm problemas quando analisam aplicações que utilizam bibliotecas prontas. Arcabouços, especialmente os de caixa preta, sofrem deste problema. Além disso, como eles são geralmente baseados no princípio de Hollywood, os problemas são ainda maiores. Pode ser difícil seguir uma linha de execução que se afunda em códigos de um arcabouço.

Composição:

Cada vez mais, um arcabouço necessita de ser composto com outros arcabouços ou com componentes legados que são reutilizados. No entanto, normalmente, arcabouços são projetados considerando que terão o controle total da aplicação (Bosch et al., 2000). Os principais problemas associados à composição são:

- **Diferenças de arquitetura:** Quando os princípios adotados nas arquiteturas dos arcabouços envolvidos são diferentes, a composição fica complicada, e pode até tornar-se impossível. Explicitar e documentar todas as decisões de arquitetura tomadas no desenvolvimento do arcabouço é o primeiro passo para evitar este problema.
- **Sobreposição de entidades:** Ocorre quando arcabouços diferentes mapeiam uma mesma entidade do mundo real de formas diferentes, fazendo com que a aplicação tenha de integrar as duas representações para a entidade.
- **Composição de controle:** Este problema pode ocorrer quando dois arcabouços esperam controlar uma entidade na aplicação.
- **Composição com componentes legados:** Ocorre quando é necessário integrar classes ou componentes legados ao arcabouço, o que é algo complexo. Uma abordagem existente é utilizar o padrão *Adapter* (Gamma et al., 1995).
- **"Buraco" (*Gap*) entre Arcabouços:** Dois arcabouços compostos podem ainda falhar em cobrir todos os requisitos de uma aplicação.

Manutenção:

O desenvolvimento de um arcabouço tem que ser pensado como um investimento de longo prazo e, como tal, deve ser visto como um produto que necessita de manutenção (Bosch et al., 2000). Desta forma, segundo Bosch, os seguintes problemas podem ocorrer:

- **Escolha da estratégia de manutenção:** Quando um arcabouço precisa ser alterado é necessário decidir entre redesenhar o arcabouço ou aplicar uma solução de contorno (*work around*) para a aplicação específica. No caso de redesenhar um arcabouço, o desenvolvimento da aplicação pode ser atrasado, esperando por uma nova versão ficar disponível. Além disso, a organização acaba sendo obrigada a manter duas versões do arcabouço, pois podem existir aplicações antigas que estão baseadas na versão anterior. No caso de adotar uma solução de contorno na aplicação, o problema ainda poderá ocorrer novamente. Esta estratégia não deve ser adotada se o tempo de vida esperado para a aplicação é longo. No entanto,

pode ser aceitável se não é esperado que aplicações similares sejam desenvolvidas no futuro.

- **Mudanças no domínio:** Frequentemente, o domínio relacionado a um arcabouço é fracamente definido e evolui ao longo do tempo. Consequentemente, este domínio inicial precisa ser atualizado e isso afeta o arcabouço existente. A probabilidade de mudanças no domínio é um fator de risco importante a ser considerado na determinação do esforço de desenvolvimento do arcabouço. Existem basicamente três formas de tratar este problema. A primeira é definir o domínio do arcabouço como sendo muito mais abrangente do que no início seja útil, de modo que possa capturar futuras modificações. Outra abordagem é redesenhar o arcabouço a cada nova mudança de domínio. Uma terceira, é reutilizar as idéias do arcabouço original para desenvolver um segundo arcabouço para o novo domínio. Todas as três formas possuem desvantagens.

2.1.2. Padrões

Um Padrão de Projeto (*Design Pattern*), conceito inicialmente proposto pelo arquiteto Christopher Alexander (Alexander, 1977), descreve uma solução concreta para um problema de arquitetura que pode aparecer em um contexto específico (Brugali-Sycara, 2000). Dizendo de outra forma, padrões codificam experiências de desenho reutilizáveis que fornecem soluções para problemas comuns de *softwares*, encontrados em contextos e domínios particulares (Schmidt-Buschman, 2003).

Os padrões capturam e reusam a estrutura, estática e dinâmica, e a forma de colaboração entre os principais participantes no projeto de *softwares*. Eles são particularmente úteis para documentar micro-arquiteturas recorrentes. Pelo uso de padrões, desenvolvedores podem escapar de armadilhas que seriam evitadas apenas através de um longo e custoso aprendizado. Além disso, eles elevam o nível do discurso no desenho de um projeto e nas atividades de programação, o que ajuda a aumentar a qualidade da aplicação e a produtividade da equipe (Schmidt-Buschman, 2003).

Os padrões possuem um relacionamento estreito com os arcabouços. Segundo Schmidt e Buschman, pode-se afirmar que os arcabouços fornecem uma base arquitetural – guiadas por padrões – para uma família de aplicações e um conjunto integrado de componentes que implementam realizações concretas dessa arquitetura (Schmidt-Buschman, 2003).

Pode-se considerar um arcabouço como uma implementação de diversos padrões, relacionando-os de forma a montar um esqueleto para aplicações em um determinado domínio (Schmidt-Buschman, 2003).

Além disso, os arcabouços auxiliam na descoberta de novos padrões. Todos os padrões contidos no livro *Design Patterns* (Gamma et al., 1995), um dos principais catálogos de padrões atuais, foram descobertos através da análise de uma coleção de arcabouços (Johnson, 1997).

2.1.3. Bibliotecas de classes

Os arcabouços podem ser vistos como extensões das bibliotecas de classes (Fayad, 2000), com algumas diferenças importantes.

Os arcabouços definem aplicações semi-completas, que agrupam funcionalidades e estruturas de um domínio (ou aspecto de infra-estrutura/integração) específico. Já as bibliotecas de classes, por sua vez, são conjuntos de classes relacionadas com um propósito geral. Elas fornecem uma menor granularidade de reuso, se comparadas aos arcabouços e, normalmente, são independentes de domínio (Fayad, 2000).

Os arcabouços, além disso, influenciam e até determinam a arquitetura das aplicações. Eles são ativos e podem controlar o fluxo do sistema, através da inversão de controle. As bibliotecas de classes, ao contrário, são sempre passivas (Fayad, 2000).

2.1.4. Componentes

A visão original de reuso de software estava baseada em componentes (Johnson, 1997). No cenário ideal, segundo Johnson, o desenvolvedor não precisa saber como um componente é implementado e deve ser fácil para ele aprender a utilizá-lo.

Desta forma, o sistema criado por componentes como os descritos acima, será eficiente, fácil de manter e confiável (Johnson, 1997).

Johnson utiliza o sistema de energia elétrica como exemplo para descrever como seria um sistema onde ocorre um reuso ideal. Uma pessoa pode comprar uma televisão de um lugar e uma torradeira de outro. Ambas irão funcionar tanto em casa quanto no escritório de qualquer pessoa, mesmo que esta não conheça absolutamente nada sobre a Lei de Ohm

(Johnson, 1997). Infelizmente, *softwares* não podem ser compostos como o sistema de energia elétrica.

Cada componente assume algumas coisas a respeito de seu ambiente. Quando componentes diferentes assumem coisas muito diferentes, pode ficar difícil utilizá-los em conjunto. Um arcabouço cria uma forma padrão para os componentes tratarem erros, comunicarem dados e invocarem operações uns dos outros, de forma a estabelecer um contexto para o desenvolvimento de componentes. Desta forma, um arcabouço torna mais fácil desenvolver componentes (Johnson, 1997). Do mesmo modo, os componentes podem ser utilizados como estratégias conectáveis em um arcabouço de caixa preta (Fayad-Schmidth, 1997), auxiliando na sua criação e extensão.

2.2 WWW

Esta seção faz uma revisão sobre a WWW – (*World Wide Web* ou, simplesmente, Web) – e a sua utilização como plataforma para o desenvolvimento de aplicações.

O desenvolvimento para Web abrange diversos aspectos. Este trabalho foca em alguns dos seus aspectos técnicos, enfatizando pontos na arquitetura das aplicações, assim como algumas tecnologias e conceitos envolvidos. A construção de aplicações Web requer, além disso, métodos que abranjam todo o ciclo de desenvolvimento, conforme discutido por Conte e outros (Conte et al., 2005).

Esta seção está estruturada da seguinte forma:

A seção 2.2.1 fala sobre a diferença entre Web e Internet. A seção 2.2.2 apresenta as aplicações Web, mostrando como a Web pode ser utilizada para construção de sistemas. A seção 2.2.3 aborda os benefícios e desafios em se utilizar a Web como base para aplicações. A seção 2.2.4 apresenta algumas tecnologias e conceitos envolvidos com o desenvolvimento. A seção 2.2.5 fala sobre linguagens de programação utilizadas na Web e apresenta a linguagem escolhida para este trabalho.

2.2.1. Internet e Web

Inicialmente, é preciso separar claramente Internet e Web.

A Internet, segundo Nielsen, permite que qualquer computador no mundo troque informações com qualquer outro computador. Como resultado, um programa cliente em uma máquina pode acessar um programa servidor em outra (Nielsen, 98).

Já a Web é um sistema de hipertextos que executa sobre a Internet como um de seus serviços. Como resultado, um usuário pode navegar por documentos que estão em qualquer parte do mundo. Além disso, estes podem conter ligações para outros documentos localizados em computadores diferentes (Nielsen, 98).

Constata-se, desta forma, que, estruturalmente, a Web está baseada em uma computação cliente-servidor, onde os servidores armazenam documentos e os clientes os acessam (Jazayeri, 2007). A Web utiliza, segundo (Jazayeri, 2007), para completar essa computação cliente-servidor:

- Um método para nomear e referenciar documentos (URL);
- Uma linguagem para escrever documentos que contenham dados e ligações para outros documentos (HTML);
- Um protocolo para que as máquinas clientes e servidoras possam se comunicar (HTTP).

No entanto, a Web, apesar de ter sido concebida como uma plataforma para hipertexto, tornou-se, também, uma plataforma para aplicações. Do ponto de vista da Engenharia de Software, pode-se considerar que elas constituem um domínio de aplicações (Gellersen et al., 1997).

2.2.2. Aplicações Web

Construídas sobre o conceito original da Web, estas aplicações são formadas por documentos (hipertextos) que fazem a interface com o usuário (Jazayeri, 2007). No entanto, as ligações contidas nos documentos HTML referenciam, além de outros documentos, programas que são executados no servidor e retornam documentos criados dinamicamente para o cliente. Os documentos gerados podem conter, além de informações para serem exibidas, códigos em scripts que são executados no navegador do cliente (Jazayeri, 2007).

Desta forma, aplicações Web modernas, não são meros repositórios de informações em hipertexto. Elas são sistemas distribuídos complexos que utilizam a Web como forma

de interação com o usuário e a Internet como infra-estrutura de comunicação (Baresi-Morasca, 2007).

A experiência prática em desenvolvimento para Web mostra diferenças significativas entre aplicações Web e as aplicações tradicionais (Baresi-Morasca, 2007), (Ginige-Murugesan, 2001).

Alguns especialistas vêem o processo de criação das aplicações Web como uma nova disciplina (Ginige-Murugesan, 2001), enquanto outros (Gellersen et al., 1997) o descrevem como um novo domínio – ou ramo independente – da Engenharia de Software. Todos, no entanto, concordam em dizer que as aplicações Web não são apenas mais um exemplo de sistemas distribuídos convencionais, mas que seu desenvolvimento possui algumas características distintas (Baresi-Morasca, 2007) (Lowe et al., 2001). O dinamismo presente nas aplicações Web modernas requer que técnicas de modelagem e implementação apropriadas sejam concebidas (Baresi-Morasca, 2007).

Vários métodos voltados para o desenvolvimento de aplicações Web têm sido propostos atualmente (Conte et al., 2005), como OOHDM (Schwabe, 1996), WebML (Ceri, 2000), W2000 (Baresi, 2000), WAE (Conallen, 2002), UWE (Koch-Kraus 2002), OOWS (Fons, 2003) e ADM (Diaz, 2004), assim como diversos arcabouços para Web têm sido desenvolvidos, como o GWT (2008), o Struts (2008) e o JSF (2004).

2.2.3. Conseqüências da Utilização da Web para Aplicações

A utilização da Web como plataforma de desenvolvimento para aplicações agrega uma gama de vantagens e desvantagens. A seguir, são abordadas as principais conseqüências detectadas em nossa pesquisa.

2.2.3.1. Benefícios da Utilização da Web para Aplicações

A utilização da Web como interface de aplicações traz grandes vantagens. Dentre elas, pode-se destacar:

1. As aplicações não precisam ser copiadas e instaladas no computador de cada cliente, desde que o mesmo possua um navegador instalado em sua máquina. (Nielsen, 1998);
2. As aplicações não precisam ser reimplementadas para cada plataforma que se torne suficientemente popular (Nielsen, 1998);

3. A administração e atualização das aplicações são feitas em um único ponto e atingem todos os clientes (Nielsen, 1998);
4. Aumento do poder de alcance da aplicação. Segundo Nielsen, o usuário, ao navegar entre sítios, enxerga a Web como "um todo" e não como vários sistemas separados (Nielsen, 1997). Ele passa por várias páginas diferentes, mas tem a sensação de estar em apenas um programa. Ele utiliza uma máquina de busca e acha o que quer no meio deste "todo". Por isso, ao navegar na Web, o usuário não gasta muito tempo e não quer ler manuais de como utilizar uma página e nem ter de fazer muito esforço para acessar a mesma (Nielsen, 1998). Desta forma, uma aplicação de loja virtual, por exemplo, que necessite ser baixada e instalada é muito menos cômoda que uma que já tenha a aparência do restante da Web e que, portanto, esteja "contida no todo".
5. Impõem restrições de configurações de hardware mais modestas às máquinas dos usuários já que a maior parte do processamento ocorre no servidor (Johnson, 2002).

2.2.3.2. Desafios na Utilização da Web por Aplicações

Em contrapartida, este modelo traz consigo uma gama de grandes problemas, decorrentes do fato de a Web ter sido projetada para um intercâmbio de informações (documentos) e não para ser uma plataforma para aplicações (Ginige-Murugesan, 2001) (Garret, 2005). Dentre os principais desafios desta abordagem, pode-se destacar:

1. Uma interface de um navegador Web que seja ótima para navegar entre hipertextos nem sempre é ótima para utilizar qualquer aplicação (Nielsen, 98). Para exemplificar, podemos citar as funcionalidades "Voltar" e "Favoritos", presentes nos navegadores. Elas são muito úteis e apropriadas para navegar em hipertexto, mas causam grandes problemas para a usabilidade de aplicações (Nielsen, 99).
2. As páginas Web (em HTML) não são apropriadas para criação de interfaces gráficas, como as que temos em aplicações tradicionais. Várias soluções têm surgido para melhorar a qualidade das interfaces, mas ainda não se consegue a mesma qualidade de interação (Nielsen, 2002).
3. Pode existir uma grande diversidade de dispositivos utilizados como cliente para acessar uma determinada aplicação (Nielsen, 97). Cada um destes dispositivos

pode possuir um tamanho de tela diferente. Ex: desktops, notebooks, PDAs e celulares.

4. Existem diversos navegadores utilizados e, infelizmente, nem sempre há compatibilidade entre a forma como estes apresentam uma mesma página HTML (Chang et al., 2004).
5. Em uma aplicação cliente-servidor tradicional, o cliente pode abrir uma conexão com o servidor e utilizá-la para todas as comunicações necessárias. Uma aplicação Web, faz requisições e recebe respostas, através do protocolo HTTP, para cada comunicação que precisa fazer ao servidor (w3c, 1999b). Desta forma, não se atualiza um componente da interface automaticamente a partir de uma mudança na parte servidora da aplicação. A interface sempre deve requisitar informações para receber respostas (Chang et al., 2004).
6. Requisições HTTP carregam apenas parâmetros do tipo String, que, freqüentemente, precisam ser convertidos para outros tipos de dados (w3c, 1999b).
7. HTML oferece um conjunto limitado e não expansível de componentes de interface (Husted et al., 2003).
8. Interfaces HTML são executadas no navegador, do qual se tem controle limitado. Por conseqüência, são difíceis de testar e depurar (Xu-Xu, 2004).
9. Interfaces HTML tornam a validação da entrada do usuário mais importante, pois a aplicação tem controle limitado sobre o navegador onde o usuário insere os dados (Husted et al., 2003).

Além disso, elas apresentam os mesmos desafios de qualquer aplicação distribuída e multi-usuário, como, por exemplo, necessidade de cuidados extras com concorrência e segurança (Baresi-Morasca, 2007).

2.2.4. Tecnologias e Conceitos na Web

Esta seção apresenta alguns conceitos e tecnologias utilizados na construção de aplicações Web.

2.2.4.1 URL

URL (*Uniform Resource Locator*) são nomes compostos que identificam um computador (endereço IP), um documento no sistema de arquivos deste computador e o protocolo de

comunicação a ser utilizado para acessar o objeto alvo (Jazayeri, 2007). É utilizado para nomear e referenciar recursos na Web.

2.2.4.2 HTML

HTML (*HyperText Markup Language*) é a linguagem utilizada para publicação na Web (w3c, 1999a). Documentos HTML contém: (1) as informações a serem mostradas, (2) instruções de formatação, que instruem o navegador a como exibir as informações e (3) ligações para outros documentos ou recursos (Jazayeri, 2007).

2.2.4.3 HTTP

HTTP (*HyperText Transfer Protocol*) é o protocolo utilizado para comunicação na Web. É um protocolo simples, baseado em requisições / respostas, que não guarda estado (IETF RFC 2616) (w3c, 1999b). Requisições e respostas HTTP são texto puro que podem ser facilmente lidas por desenvolvedores (Husted et al., 2003).

Um servidor HTTP sempre irá aceitar qualquer requisição de qualquer cliente e sempre irá fornecer algum tipo de resposta, mesmo quando a resposta seja apenas para dizer “não”. Sem a sobrecarga de negociar e manter conexões, protocolos sem estado podem tratar um grande volume de requisições (Husted et al., 2003).

2.2.4.4 Navegador

É um programa utilizado para interpretar documentos HTML e exibir o seu conteúdo. Estes são utilizados como clientes universais para aplicações Web e tendem a se tornar uma plataforma sofisticada para aplicações (Jazayeri, 2007).

Pode-se considerar o navegador como um *middleware* de infra-estrutura utilizado pelas aplicações Web para exibição das interfaces. Os navegadores modernos, além da interpretação do HTML, começaram a oferecer recursos para suportar aplicações, como cookies[5], ou suporte a scripts. Alguns navegadores, como Safári, Firefox e Ópera,

[5] Mecanismo para armazenar informações sobre uma página na máquina do cliente, de modo que possa ser utilizado posteriormente pela aplicação (Jazayeri, 2007).

fornecem, também, recursos gráficos avançados invocados por marcações especiais no HTML (Jazayeri, 2007).

2.2.4.5 DOM

O DOM (*Document Object Model*) é uma API (*Application Programming Interface*) para trabalhar com documentos HTML válidos e XML bem formados. Ele define a estrutura lógica dos documentos e a forma como eles são acessados e manipulados (w3c, 2004).

Os navegadores permitem o acesso ao objeto DOM, que representa a estrutura da página HTML, para scripts, a fim de que estes possam acessar e modificar o conteúdo do documento. O DOM é uma especificação do W3C que tem como objetivo, fornecer uma interface de programação padrão para construção de páginas e aplicações. Através dele, programadores podem construir documentos, navegar sua estrutura, adicionar, remover ou modificar elementos e conteúdo (w3c, 2004).

No DOM, os documentos têm uma estrutura lógica que é muito parecida com uma árvore, onde cada elemento é um nodo. Na realidade, são mais parecidos com uma "floresta", visto que podem possuir mais de uma árvore (w3c, 2004).

A especificação do DOM é composta de um núcleo e de vários módulos. É possível para uma implementação, estar de acordo com o núcleo da especificação ou com um ou mais de seus módulos (w3c, 2004).

Um grande problema hoje para programação de aplicações Web reside no fato de os principais navegadores utilizados suportarem, cada um, partes diferentes da especificação. Outro problema, ainda mais grave, é que alguns navegadores introduzem elementos particulares, que não estão previstos na especificação (Husted et al., 2003).

2.2.4.6 CSS

O CSS (*Cascading Style Sheets*) é um mecanismo para se adicionar formatação aos elementos em um documento HTML (ex: cor, fonte, espaçamento etc.) (w3c, 1998).

Enquanto o HTML é utilizado para estruturar o conteúdo, o CSS é usado para formatar este conteúdo estruturado. Desta forma, com o CSS é possível modificar a aparência de uma página sem precisar, necessariamente, modificar a sua estrutura, além de possibilitar um controle da apresentação de vários documentos em uma única folha de estilo (w3c, 1998).

2.2.4.7 SCRIPT

Um script é um programa que pode acompanhar um documento HTML ou estar embutido diretamente nele. O programa executa na máquina do cliente quando o documento é carregado ou em qualquer outro momento, tal como quando uma ligação é ativada (w3c, 1999a).

Os scripts fornecem formas de estender o HTML, proporcionando uma maior interatividade. Por exemplo, (w3c, 1999a):

- Podem modificar o conteúdo do documento dinamicamente;
- Podem processar informações que são fornecidas pelos usuários nos campos de entrada para, por exemplo, fazer algum tipo de validação;
- Podem ser disparados por eventos que afetam o documento, tais como mudança de foco em um elemento ou movimentos de mouse.

O suporte a scripts do HTML é independente da linguagem utilizada. As linguagens mais comuns atualmente são o *Javascript* e o *Vbscript* (w3c, 1999a).

2.2.4.8 DHTML

O DHTML (*Dynamic HTML*) é um termo utilizado para descrever a combinação entre HTML, folhas de estilo (CSS) e scripts, permitindo aos documentos serem animados e manipulados (w3c, 2005).

A especificação do DOM diz como interagir com os documentos HTML. Através de scripts, podemos acessar o DOM e modificar os elementos de um documento, assim como o CSS desses elementos (w3c, 2005).

Utilizando DHTML, é possível montar elementos de uma tela inteiramente no lado cliente de uma aplicação Web. Dizendo de outra forma, o próprio navegador do usuário pode adicionar ao DOM o HTML necessário para construir determinado trecho da interface. Desta forma, a aplicação não precisa montar a interface inteira no servidor e trafegar o HTML completo pela rede.

No entanto, existem muitos problemas de compatibilidade entre os diferentes navegadores, o que representa uma dificuldade para o desenvolvimento.

2.2.4.9 AJAX

Segundo Garrett, AJAX (*Asynchronous Javascript* + XML) é um conjunto de tecnologias utilizadas para explorar melhor a interação com os usuários de aplicações Web (Garrett, 2005).

O AJAX utiliza as seguintes tecnologias:

1. Apresentação baseada em padrões, utilizando HTML e CSS;
2. Interação e apresentação dinâmica utilizando o DOM (*Document Object Model*);
3. Formato padrão para troca e manipulação de dados – XML;
4. Comunicação assíncrona com o servidor utilizando XMLHttpRequest;
5. Linguagem *Javascript* que interliga todas estas tecnologias.

A Figura 2.1 compara o modelo clássico de aplicações com o modelo proposto pelo AJAX. Conforme pode ser visto em detalhes em (Garrett 2005), existe uma grande diferença entre os dois.

No modelo clássico, o usuário realiza ações que disparam requisições para o servidor. O servidor processa cada requisição e produz uma nova página HTML, que é devolvida para o usuário. A partir do momento em que fez a requisição, o usuário passa a esperar até que a resposta seja enviada e processada por seu navegador (Garrett 2005).

No modelo do AJAX, um "motor AJAX" é adicionado à página e ele realiza as requisições para o servidor. Estas requisições são feitas de forma assíncrona, o que faz com que o usuário não fique esperando para poder continuar interagindo. O servidor responde e o motor AJAX atualiza na página apenas o que foi modificado em função de sua requisição (via *Javascript*) (Garrett 2005).

Estas requisições são feitas através do *XMLHttpRequest*. Este é um objeto, inicialmente implementado pela Microsoft no Internet Explorer, que permite a realização de requisições HTTP via *Javascript*, sem que se precise recarregar toda a página HTML. Posteriormente, ele foi adicionado aos principais navegadores como objeto nativo, apesar de não constar nos padrões para navegadores da W3C (WEISS, 2006).

Desta forma, pode-se dizer que o modelo do AJAX realiza requisições "menores", que processam apenas o que foi pedido (uma vez que não é necessário montar a página HTML inteira) e enviam pela rede apenas o que precisa ser atualizado (dados).

Este modelo, que Garrett chama de modelo centrado em dados, apresenta uma série de vantagens e de desvantagens em relação ao modelo tradicional, centrado na página (Garrett 2005).

Como vantagem, pode-se citar a melhor qualidade de interação proporcionada ao usuário, que terá menores tempos de resposta e poderá continuar executando tarefas enquanto uma requisição é realizada, já que as requisições são assíncronas (Garrett 2005).

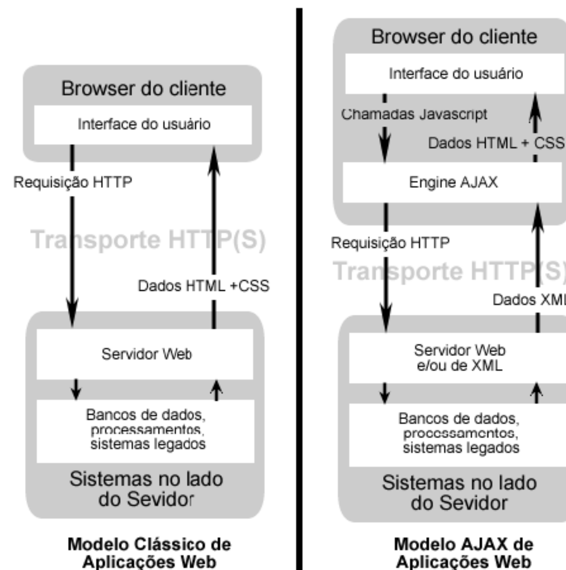

Figura 2-1 – Os Dois Modelos de Aplicações Web (Garrett 2005).

Como desafio desse modelo, pode-se citar os problemas que são ocasionados pelos botões de controle de navegação contidos no navegador (“voltar”, “favoritos”). Eles foram pensados para navegar entre páginas e não para funcionar em um modelo centrado em dados (Nielsen, 1998).

2.2.5. Linguagens de Programação Para Web

Diversas linguagens e plataformas estão disponíveis para construção de aplicações Web atualmente. Pode-se citar, a título de exemplo, Java, .Net e PHP. Neste trabalho, é utilizada a linguagem Java como base para criação do arcabouço proposto.

Esta escolha baseia-se, principalmente, na maior experiência do autor em trabalhar com esta linguagem e com suas APIs no desenvolvimento de aplicações Web. Além disso, a plataforma Java para desenvolvimento corporativo (JEE) define um padrão *de facto* para o desenvolvimento de aplicações multicamadas interativas para Web (Zhang-Buy, 2003).

2.3 Arcabouços *para Web*

As aplicações Web, em função de sua arquitetura cliente-servidor, podem utilizar mais de um arcabouço em sua construção. Esta seção tem como foco arcabouços que auxiliam a criação de interfaces gráficas com o usuário e a comunicação dos componentes das interfaces com a parte servidora da aplicação.

A seção 2.3.1 apresenta o MVC e mostra como este pode ser implementado em uma arquitetura Web. A seção 2.3.2 mostra algumas características desejáveis a arcabouços para Web. A seção 2.3.3 analisa alguns trabalhos existentes.

2.3.1. MVC e Web

MVC (Modelo/Visão/Controlador) é considerado por alguns como um arcabouço (Johnson, 1997) ou como um padrão de projeto por outros (Buschman et al., 1996), devido ao grande número de vezes que este já foi implementado.

A abordagem MVC, inicialmente proposta por Trygve Reenskaug para a plataforma Smalltalk-80 (Lalonde, 1994), é composta por três tipos de objetos. O modelo é o objeto de aplicação, a Visão é a apresentação na tela e o Controlador é o que define a maneira como a interface do usuário reage às entradas do mesmo. (Gamma et al, 2002). Esta separação destes objetos aumenta a flexibilidade e a reutilização (Gamma et al, 2002) e tem influenciado a maneira de pensar o desenho de interfaces (Fowler, 2003).

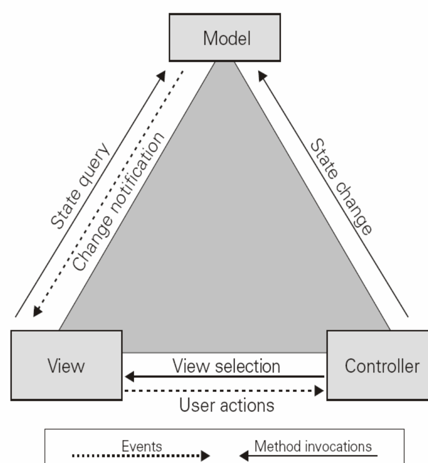

Figura 2-2 – MVC Clássico (Husted et al., 2003)

No MVC tradicional, uma visão deve garantir que sua aparência reflita o estado do modelo. Sempre que os dados do modelo mudam, este notifica as visões que dependem destes dados. Em resposta, cada visão tem a oportunidade de atualizar-se (Gamma et al, 2002). Normalmente, este comportamento é implementado de acordo com o padrão *Observer* (Gamma et al, 2002).

No entanto, como pode ser visto na seção 2.2.3.2, no desafio 5 relativo a elaboração de aplicações Web, o modelo de requisições / respostas fornecido pelo protocolo HTTP faz com que as visões não possam ser atualizadas pelos modelos automaticamente. Elas precisam fazer uma requisição para o servidor.

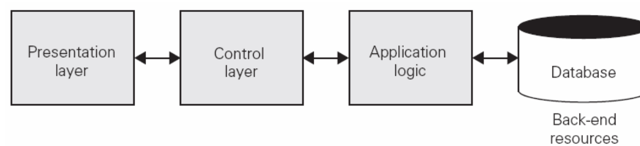

Figura 2-3 – Camadas em Aplicações Web (Husted et al., 2003)

Desta forma, uma aplicação Web convencional é dividida em camadas de acordo com um desenho mais plano do que o MVC convencional (veja Figura 2.3) (Husted et al., 2003). O controlador é colocado entre as camadas de apresentação (Visão) e de lógica da aplicação (Modelo). As responsabilidades principais de cada componente são as mesmas, porém, cada busca por estado ou notificação de mudança deve passar pelo controlador. Além disso, quando a visão desenha o conteúdo dinâmico, ela utiliza dados que são fornecidos pelo controlador, ao invés de dados retornados diretamente do modelo (Husted et al., 2003).

Esta versão do MVC, sem o padrão de notificação, pode ser chamada de MVC2 ou MVC Web (Husted et al., 2003).

2.3.2. Características Desejáveis

Qualquer arcabouço para desenvolvimento na Web deve considerar os desafios associados a este tipo de abordagem. A seção 2.2.3.2 relaciona os principais encontrados em nossa pesquisa.

Para ajudar a resolver grande parte desses desafios (especialmente os desafios 2, 3 e 4), uma aplicação Web deve utilizar uma abordagem MVC para separar interfaces de regras de negócio na aplicação (Fowler, 2003).

Em java, linguagem escolhida para implementação do arcabouço neste trabalho, uma boa forma de implementar esta abordagem é utilizar *Servlets*[6] como controladores, páginas HTML ou JSP[7] como visões e os objetos contendo as regras de negócio (modelo) sendo instanciados e chamados pelos controladores (Ramachandran, 2002).

Esta implementação do MVC difere um pouco da implementação original para o *Smalltalk*, pois um *Servlet* não pode atualizar uma página automaticamente, sem que haja uma requisição. Isto também é discutido na seção 2.3.1.

Outra característica desejável é que o arcabouço seja, predominantemente, de caixa preta, pois, conforme visto na seção 2.1.1.2, este modelo torna mais simples a sua utilização. Acreditamos que isso é ainda mais marcante no caso de arcabouços para suporte a aplicações Web. A complexidade associada à criação de interfaces com HTML é grande, conforme mostrado na seção 2.2.3.2. A utilização de componentes de tela deve criar um grau de abstração mais elevado para elaboração das interfaces, simplificando consideravelmente o desenvolvimento.

Além dos desafios associados ao domínio da Web, um arcabouço deve considerar os desafios associados à construção de arcabouços de forma geral, destacados por Bosch (Bosch et al., 2000) e explorados na seção 2.1.1.3.

Outro aspecto chave no projeto é a definição dos pontos de extensão – lugares no arcabouço onde o projetista de uma aplicação introduz as variações ou diferenças da aplicação (Schwabe et al., 2001). É importante que um arcabouço proveja pontos de extensão necessários para deixá-lo flexível para as aplicações do domínio (Schwabe et al., 2001) e que estes pontos estejam devidamente documentados (Bosch et al., 2000).

2.3.3. Principais Arcabouços para Web

Diversos arcabouços para Web têm sido criados nos últimos anos. Uma simples pesquisa por “web *frameworks*” feita no sítio sourceforge.net, hoje o maior sítio de projetos com código livre na internet, realizada no início de 2008, apontou para 22890 projetos (Sourceforge, 2008). Obviamente, esta simples busca não leva em consideração a

[6] Classes básicas para criação de servidores, distribuída pela API da linguagem Java (Servlet, 2008).
[7] Tecnologia para criação de páginas web dinâmicas. A especificação do JSP é liderada pela Sun Microsystems. (JSP, 2008)

qualidade e nem a aplicabilidade destes arcabouços, mas serve como um indicativo de que uma enorme quantidade de produtos está sendo criada. Se considerarmos os arcabouços proprietários este número certamente seria bem maior, pois estes já são necessidade estratégica para organizações em todo o mundo (Fayad, 2000).

Entre os principais trabalhos publicados nesta área encontrados em nossa pesquisa, destaca-se um arcabouço para desenvolvimento para Web independente de plataforma proposto por Chang, Kim e Agha (Chang et al., 2004). Neste trabalho, é utilizado um tradutor para geração de código específico para cada cliente. Tanto o código da parte cliente quanto o código da parte servidora são desenvolvidos em uma linguagem base e traduzidos de acordo com um conjunto de regras de implantação passado para o tradutor.

Além dos trabalhos acadêmicos, vários arcabouços com código aberto e escritos em Java foram analisados. Foram escolhidos três arcabouços para Web para uma análise mais profunda. O *Struts* é o mais antigo dos arcabouços analisados e o atual líder de mercado mundial (Husted et al., 2003). O JSF é um arcabouço muito importante, por estar em vias de fazer parte da especificação JEE da Sun (Mann, 2005), além de possuir um modelo interessante de organização dos elementos de interface com o usuário em componentes (caixa preta). O GWT é o mais novo dos três, mas foi escolhido por apresentar um modelo interessante, de modo que ele é utilizado como base para criação de algumas das mais bem sucedidas e utilizadas aplicações Web atuais, como o *gmail*, o *google maps* e várias outras aplicações da *google* (GWT, 2008). O GWT é utilizado diretamente na solução proposta no capítulo 3. As seções 2.3.3.1, 2.3.3.2 e 2.3.3.3 detalham esses três arcabouços mencionados.

2.3.3.1. Struts

O *Struts* é um arcabouço para Web mantido pela *Apache Software Foundation* (ASF), como parte de seu projeto Jakarta (Husted et al., 2003). Sua primeira versão foi desenvolvida entre maio de 2000 e junho de 2001 e é o arcabouço Web mais conhecido e utilizado atualmente (Husted et al., 2003).

O *Struts* utiliza uma arquitetura que segue o modelo MVC2, descrito na seção 2.3.1, e é mostrada na Figura 2.4. O controlador é implementado pela classe *ActionServlet* (subclasse de *Servlet*). Uma outra classe chamada *Action* é utilizada para acessar as classes de negócio. Quando o *ActionServlet* recebe uma requisição ele utiliza a URL (ou

"caminho") para determinar qual *Action* ele deve utilizar para tratar a requisição. Uma *Action* pode validar os dados vindos na requisição e acessar a camada de negócio para recuperar informações de bancos de dados ou de outros serviços (Husted et al., 2003).

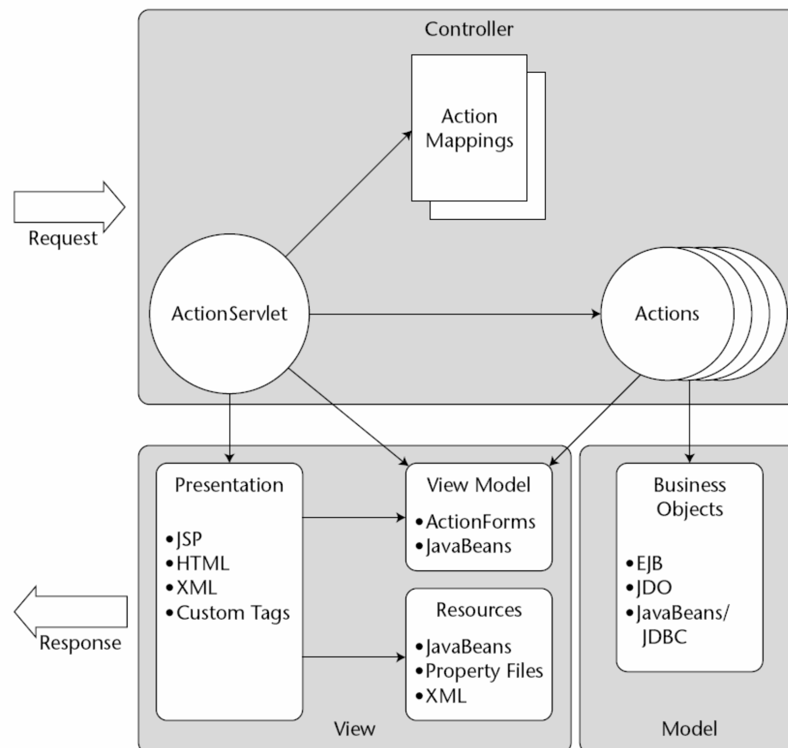

Figura 2-4 – MVC2 no *Struts* (Dudney et al., 2004).

Para validar os dados fornecidos e utilizá-los no modelo, uma *Action* precisa saber quais valores foram submetidos. O *ActionServlet* popula um *JavaBean*[8] com os valores recebidos na requisição. Estes *JavaBeans* são subclasses da classe *ActionForm*, fornecida pelo *Struts.* O *ActionServlet* pode determinar qual *ActionForm* utilizar através do caminho da requisição, da mesma forma como seleciona qual *Action* utilizar (Husted et al., 2003).

Cada requisição HTTP deve ser respondida com uma resposta. Normalmente, uma *Action* do *Struts* não escreve a resposta, mas encaminha a requisição para outro recurso, como uma página JSP. O *Struts* fornece uma classe chamada *ActionForward* que pode ser utilizada para indicar o caminho para uma página. Quando a regra de negócio é completada, a *Action* seleciona e retorna um *ActionForward* para o servlet. Este utiliza o

[8] *JavaBean* é um componente reutilizável escrito em Java. Para ser qualificada como *JavaBean*, uma classe deve ser concreta, pública e possuir um construtor sem parâmetros. *JavaBeans* expõem campos internos como propriedades através de métodos públicos que seguem um padrão de nomenclatura. Conhecendo as propriedades e seguindo este padrão, outras classes Java podem descobrir e manipular as propriedades do *JavaBean* através de introspecção (Husted et al., 2003).

caminho armazenado no *ActionForward* para chamar a página e completar a resposta (Husted et al., 2003).

O *Struts* encapsula esses detalhes juntos em um objeto do tipo *ActionMapping*. Cada *ActionMapping* está relacionado a um caminho específico. Quando este caminho é requisitado, o sevlet recupera o objeto *ActionMapping*. Este mapeamento informa ao servlet qual *Action, ActionForm* e *ActionForward* utilizar (Husted et al., 2003).

Todos estes detalhes, as *Actions*, *ActionForms*, *ActionForwards*, *ActionMappings,* dentre outros, são declarados no arquivo struts-config.xml. O *ActionServlet* lê este arquivo ao iniciar a aplicação e cria uma base de dados destes objetos de configuração (Husted et al., 2003).

A Figura 2.4 mostra como o *Struts* implementa o MVC2, destacando os componentes descritos.

O Controlador do *Struts* (*ActionServlet*), é um *Servlet* que, como tal, tem seu ciclo de vida controlado por um container Web, como *Tomcat*, *Resin* ou *WebLogic*. Quando o container inicia, ele lê o descritor da aplicação (web.xml) que informa, entre outras coisas, quais *Servlets* carregar (Husted et al., 2003). A Listagem 2.1 mostra como este controlador do *Struts* deve ser configurado na aplicação.

```
1  <web-app>
2      <servlet>
3          <servlet-name>action</servlet-name>
4          <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
5      </servlet>
6      <servlet-mapping>
7          <servlet-name>action</servlet-name>
8          <url-pattern>*.do</url-pattern>
9      </servlet-mapping>
10 </web-app>
```

Listagem 2-1 – Configuração do Controlador do *Struts* (Husted et al., 2003).

Quando uma requisição que case com o padrão de URL configurado no mapeamento do *Servlet* do *Struts* (*.do, no exemplo) chegar, o container a encaminhará para o *ActionServlet* (Husted et al., 2003).

O *ActionServlet* utiliza o arquivo struts-config.xml para mapear quais *Actions*, *ActionForms*, etc. estão associados à requisição recebida (Husted et al., 2003). A Listagem 2.2 exibe um exemplo de configuração deste arquivo.

```
1  <struts-config>
2      <form-beans>
3          <form-bean name="helloForm"
4              type="br.ufmg.HelloForm" />
5      </form-beans>
6      <action-mappings>
7          <action path="/hello"
8              type="br.ufmg.HelloAction"
9              name="helloForm" scope="request" validate="true"
10             input="/test/hello.jsp">
11             <forward name="success"
12                 path="/test/hello.jsp" />
13             <forward name="failure"
14                 path="/test/failure.jsp" />
15         </action>
16     </action-mappings>
17 </struts-config>
```

Listagem 2-2 – Exemplo do Arquivo struts-config.xml.

A configuração mostrada na Listagem 2.2 informa ao controlador que a classe (*Action*) "*br.ufmg.HelloAction*" deve ser utilizada para tratar uma requisição cuja parte variável da URL seja "*hello*" (ex: http://servidor:<porta>/contextodaaplicacao/hello.do). O *ActionForm* "*br.ufmg.HelloForm*" é o *JavaBean* que é utilizado para encapsular os parâmetros recebidos na requisição para que a *Action* possa acessá-los. A Listagem 2.3 mostra um exemplo de uma *Action* e a Listagem 2.5 mostra um exemplo de *ActionForm*.

```
1  public class HelloAction extends Action
2  {
3      public ActionForward execute(ActionMapping mapping,
4              ActionForm form, HttpServletRequest request,
5              HttpServletResponse response) throws ServletException
6      {
7          HelloForm formBean = (HelloForm) form;
8          String name = formBean.getName();
9
10         ActionMessages messages = new ActionMessages();
11         messages.add(ActionMessages.GLOBAL_MESSAGE,
12                 new ActionMessage("saudacao",new Object[]{name}));
13         addMessages(request, messages);
14
15         return mapping.findForward("success");
16     }
17 }
```

Listagem 2-3 – Exemplo de *Action* do *Struts*.

No exemplo mostrado na Listagem 2.3, é criada uma mensagem para ser enviada para tela. Esta mensagem é criada a partir de uma chave e de um conjunto de parâmetros. A partir desta chave e dos parâmetros passados, o *Struts* monta a mensagem, de acordo como um *resourceBundle*[9] associado. A Listagem 2.4 mostra um arquivo de mensagens de exemplo utilizado como *resourceBundle*.

```
1  saudacao=Hello {0}!
2
```

Listagem 2-4 – Exemplo de Arquivo de Mensagens no *Struts*.

O *ActionForm* mostrado na Listagem 2.5, é uma subclasse de *ValidatorForm*. Esta é uma subclasse de *ActionForm* que possibilita o uso de validações declarativas no *Struts* (Husted et al., 2003). Repare que o *HelloForm* é um JavaBean simples, sem precisar implementar nenhum método especial.

```
1  public class HelloForm extends ValidatorForm
2  {
3      private String name;
4
5      public String getName()
6      {
7          return name;
8      }
9      public void setName(String name)
10     {
11         this.name = name;
12     }
13 }
```

Listagem 2-5 – Exemplo de *ActionForm* do *Struts*.

A distribuição do *Struts* inclui um pacote para validação, contendo diversas classes para integrar o *Commons Validator* – subprojeto do projeto Jakarta da Apache – com o *Struts*. Este pacote, assim como o *Commons Validator* que ele estende, constituem o *Struts Validator* (Husted et al., 2003).

Usando este esquema de validação do *Struts*, são executadas validações tanto no lado do cliente quanto no lado do servidor, através de regras declaradas em um arquivo

[9] Parte da API do Java. ResourceBundle (`java.util.ResourceBundle`) é uma coleção de objetos Properties (`java.util.Properties`). Cada objeto Properties refere-se a uma localidade. Localidades são identificadas por sua região e idioma, utilizando um objeto Locale (`java.util.Locale`). Utilizados para fornecer suporte a internacionalização para aplicações (Husted et al., 2003).

descritor chamado validation.xml. A validação no lado do cliente é feita por *Javascript* e evita que requisições desnecessárias sejam enviadas. É realizada, também, a validação no lado do servidor, para garantir a integridade dos dados. A Listagem 2.6 exibe um exemplo de arquivo validation.xml.

```
1 <form-validation>
2     <formset>
3         <form name="helloForm">
4             <field property="name" depends="required">
5                 <arg0 key="Name" resource="false" />
6             </field>
7         </form>
8     </formset>
9 </form-validation>
```

Listagem 2-6 – Exemplo do Arquivo validation.xml.

No exemplo mostrado na Listagem 2.6, é criada uma regra para informar que o campo "*name*" é de preenchimento obrigatório.

O *Struts* também fornece um conjunto de *taglibs*[10] usadas nas páginas JSP que compõem a visão. Estas bibliotecas contêm marcações (*tags*) que possibilitam o vínculo de campos de um formulário com atributos dos *ActionForms,* além de marcações especializadas em mostrar erros de validação. A Listagem 2.7 mostra um exemplo de página utilizando as marcações do *Struts*.

[10] Taglibs são marcações especiais, misturadas nas marcações HTML. Elas podem ser usadas como se fossem marcações HTML comuns. Uma marcação (ou tag) JSP simples pode representar dúzias de comandos Java, mas tudo o que o desenvolvedor precisa saber é como inserir a marcação. O código de programação fica escondido em um arquivo java (Husted et al., 2003).

```
<%@ taglib uri="/tags/struts-html" prefix="html"%>
<HTML>
<HEAD>
<TITLE>Sign in, Please!</TITLE>
</HEAD>
<BODY>
<html:errors />
<html:messages id="message" message="true">
    <bean:write name="message" />
    <BR>
</html:messages>
<html:form action="/hello" focus="name">
    <TABLE border="0" width="100%">
        <TR>
            <TH align="right">Nome:</TH>
            <TD align="left"><html:text property="name" /></TD>
        </TR>
        <TR>
            <TD align="right"><html:submit /></TD>
            <TD align="left"><html:reset /></TD>
        </TR>
    </TABLE>
</html:form>
</BODY>
</HTML>
```

Listagem 2-7 – Exemplo de Página com marcações do *Struts*.

Padrões de Projeto no Struts

A arquitetura do Struts, trabalhando de acordo com as descrições do catálogo "Core J2EE Patterns" (Deepak et al., 2001), implementa vários padrões fundamentais, incluindo *Service to Worker, Front Controller, Singleton, Dispatcher, View Helper, Value Object, Composite View* e *Synchronizer Token*, entre outros. A Tabela 2.1 mostra vários padrões implementados pelos componentes do *Struts* (Husted et al., 2003).

Padrões	Componente do Struts
Service to Worker	ActionServlet, Action
Command (Gamma et al, 2002), Command and Controller, Front Controller, Singleton, Service Locator	ActionServlet, Action
Dispatcher, Navigator	ActionMapping, ActionServlet, Action, ActionForward
View Helper, Session Façade, Singleton	Action
Transfer Objects (fka Value Objects), Value Object Assembler	ActionForm, ActionErrors, ActionMessages
View Helper	ActionForm, ContextHelper, taglibs
Composite View, Value Object Assembler	Template taglib, Tiles taglib
Synchronizer Token	Action

Decorator (Gamma et al, 2002)	ActionMapping

Tabela 2-1 – Padrões de Projeto no *Struts* (Husted et al., 2003).

O *ActionServlet* do *Struts* fornece um ponto de acesso centralizado para tratar as requisições, de acordo com o padrão *Front Controller* (Deepak et al., 2001). Assim como muitos *Front Controllers*, ele implementa um componente de disparo (a classe *Action*) para tratar detalhes como recuperação de conteúdo e gerenciamento da visão (Husted et al., 2003). O *ActionServlet* executa um método conhecido na *Action*, passando os dados da requisição e delegando a responsabilidade de responder a ela. Isto é uma implementação do padrão *Command and Controller* (Deepak et al., 2001), que é baseado no padrão *Command* (Gamma et al, 2002).

O *ActionServlet* pode, também, manter uma lista de *DataSources*[11] referenciados por um nome, que pode ser especificado na configuração do *Struts* (Husted et al., 2003). Outros objetos podem recuperar, a partir do nome, um objeto *DataSource* – de acordo com o padrão *Service Locator* (Deepak et al., 2001).

O *Struts* transfere os parâmetros contidos em uma requisição HTTP para um *ActionForm*. Utilizar *JavaBeans* para transferir dados do modelo para os componentes de visão é seguir o padrão *View Helper* (Deepak et al., 2001).

As *Actions* no *Struts* são criadas de acordo com o padrão *Singleton* (Gamma et al, 2002), ou seja, apenas uma instância de cada classe *Action* é criada em toda a aplicação, deixando para o desenvolvedor a responsabilidade de controlar a concorrência (Husted et al., 2003).

O *Struts* utiliza também o padrão *Session Façade* (Deepak et al., 2001), em dois pontos principais. A classe *Action* abstrai as interações entre os objetos de negócio e fornece uma camada que expõe apenas as interfaces necessárias (Husted et al., 2003). Outro ponto onde este padrão é utilizado é no desenho do componente para acessar mensagens. A complexidade de se escolher a localidade do usuário é escondida. Apenas se solicita uma mensagem por sua chave e o componente de mensagens fornece a mensagem correta de acordo com a localidade (Husted et al., 2003).

A informação que queremos coletar em um único pacote é chamada de *Value Object*. Um *Value Object* encapsula dados de negócio, de modo que uma única chamada pode ser feita para enviar e receber estes dados (Deepak et al., 2001). O *ActionForm* do *Struts* é um

[11] DataSources são classes fornecidas pela API do Java (javax.sql.DataSource) utilizadas para abstrair o acesso a algum sistema de armazenamento. A maioria dos sistemas de armazenamento se refere a um banco de dados JDBC, mas podem ser utilizados para conectar a qualquer tipo de sistema (Husted et al., 2003).

exemplo de *Value Object*. Ele coleta os campos que precisaremos em um formulário HTML todos de uma vez, de modo que essa informação possa ser validada e enviada para processamento na *Action* (Husted et al., 2003). O *ActionForm* também suporta o padrão *Value Object Assembler* (Deepak et al., 2001), ou seja, pode-se criar um *Value Object* composto, usando outros *Value Objects* (Husted et al., 2003).

O *Struts* permite a criação de páginas JSP a partir de várias outras páginas que podem ser incluídas através de uma marcação *<template>* (Husted et al., 2003). Isto é uma implementação do padrão *Composite View* (Gamma et al, 2002).

O *Struts* implementa também o padrão *Synchronizer Token* (Deepak et al., 2001), que, através de uma *token* na sessão do usuário, impede que processamentos desnecessários sejam executados em função do usuário ter acionado um comando diversas vezes, sem que a resposta da requisição anterior tenha chegado (Husted et al., 2003).

2.3.3.2. JSF

O *JavaServer Faces* (JSF) é um arcabouço para criação de interfaces com usuário para aplicações Web em java, com sua especificação solicitada na JSR[12] 252 (McClanahan et al., 2004).

Assim como o *Struts*, o JSF também utiliza uma arquitetura que segue o modelo MVC2, descrito na seção 2.3.1. A principal diferença entre eles é que a visão no JSF é composta de uma árvore de componentes[13], enquanto no *Struts* a visão é centrada na página (normalmente JSP) (Dudney et al., 2004).

A Figura 2.5 mostra como o JSF implementa o MVC2, destacando seus principais componentes.

[12] *Java Community Process* (JCP) é o processo público utilizado para estender a linguagem Java com novas APIs e outras melhorias na plataforma. Novas propostas são chamadas de *Java Specification Requests* (JSRs) (*Mann, 2005).*

[13] O Termo 'árvore de componente' se refere à hierarquia de componentes de interface utilizados na camada de apresentação. Esta árvore de componentes segue o padrão *Composite* (Gamma et al., 2002) (Dudney et al., 2004).

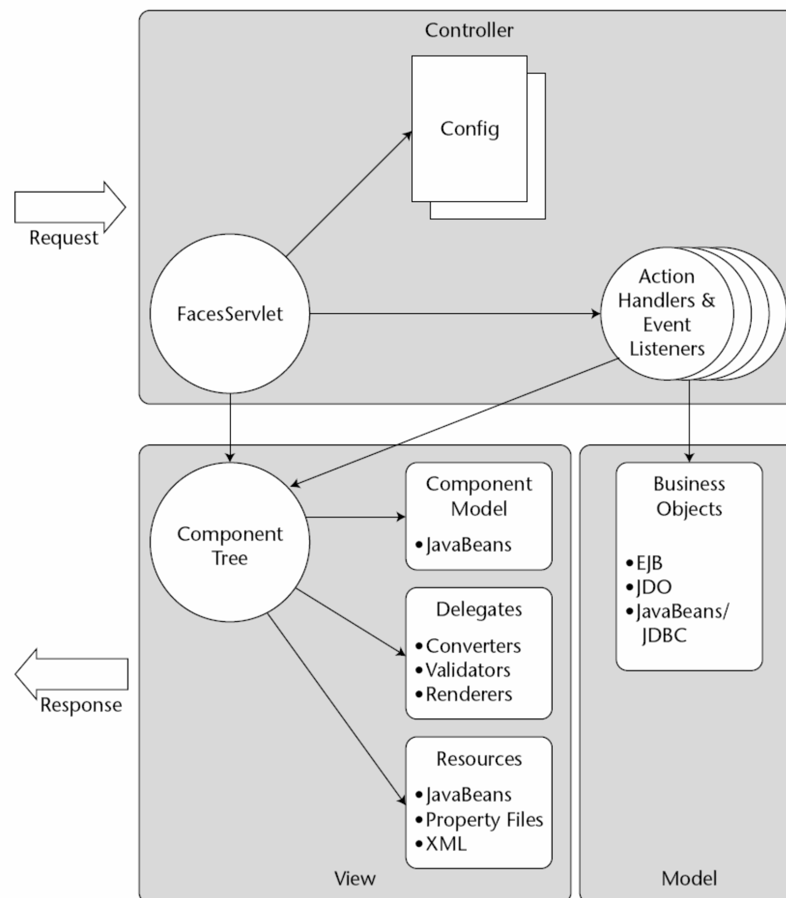

Figura 2-5 – MVC2 no JSF (Dudney et al., 2004)

O controlador do JSF consiste em um *Servlet* central chamado *FacesServlet* – segundo o padrão *Front Controller* – um ou mais arquivos de configuração e um conjunto de tratadores de ações (*action handlers*), que, de certa forma, se assemelham a *Actions* do *Struts* (Dudney et al., 2004).

O *FacesServlet* é responsável por receber requisições dos clientes e depois realiza um conjunto de passos para preparar e disparar uma resposta (Dudney et al., 2004). Os passos podem ser resumidos como:

- **Restaurar a Visão:** Recria-se a árvore de componentes para a página requisitada. Se a página tiver sido exibida previamente na mesma sessão do usuário e os dados salvos, o *FacesServlet* constrói a árvore a partir desses dados. Salvar o estado dos componentes entre requisições é uma funcionalidade importante do JSF (Dudney et al., 2004).
- **Aplicar Valores da requisição:** Uma vez que a árvore de componentes tiver sido recuperada, ela é atualizada com as novas informações da requisição. Isso inclui eventuais conversões de tipos, além de qualquer evento de algum

componente de interface que precise ser executado imediatamente ou enfileirado para processamento nos passos posteriores (Dudney et al., 2004).

- **Processar Validações:** Cada componente de interface pode ter lógicas internas de validação ou uma quantidade de validadores (*Validator*) registrados, que realizam uma verificação de correção dos dados (Dudney et al., 2004).
- **Atualizar Valores no Modelo:** Neste passo, é perguntado a cada componente de interface se ele precisa atualizar seu modelo (Dudney et al., 2004).
- **Chamar Aplicação:** Depois de atualizar o modelo dos componentes de interface, o *ActionListener* padrão irá responder a todo *ActionEvent* que esteja enfileirado, que normalmente são associados a componentes *UICommand* (ex: *button*) e requerem interação com os objetos de negócio da aplicação (Dudney et al., 2004).
- **Escrever a Resposta:** Com base no resultado do processamento da requisição que acabou de ser resumido, a resposta é gerada a partir da árvore de componentes corrente (que pode incluir modificações) ou de uma nova árvore de componentes. Quando são utilizadas páginas JSP para visão, cada árvore de componentes é associada a uma página JSP particular (Dudney et al., 2004).

A Figura 2.6 mostra o ciclo de vida de uma requisição no JSF.

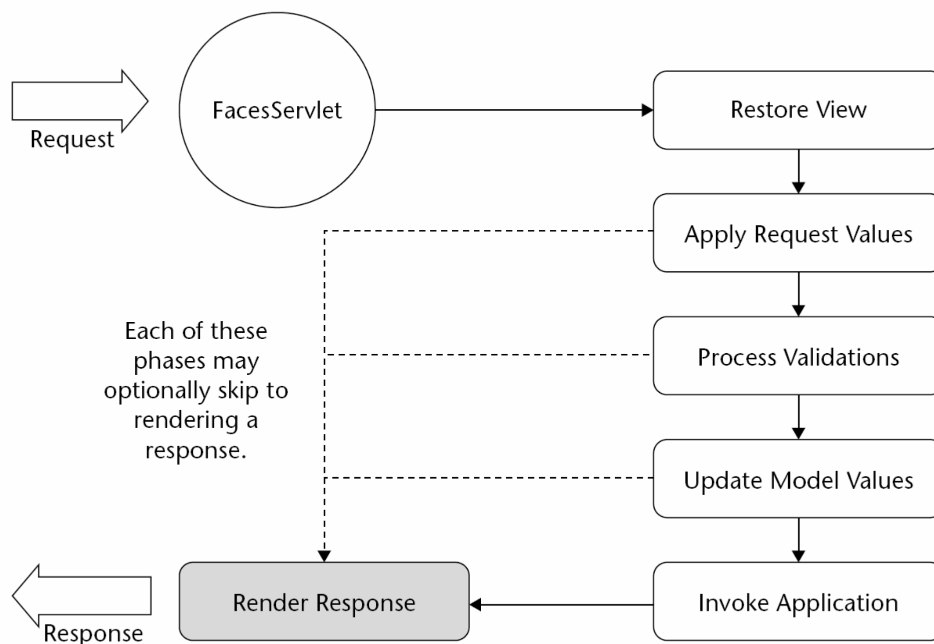

Figura 2-6 – Ciclo de Vida de uma requisição no JSF (Dudney et al., 2004).

O Controlador do JSF, de forma similar ao *Struts*, também é um *Servlet* e precisa ser configurado no descritor web.xml. A Listagem 2.8 mostra um exemplo desta configuração.

```
1  <web-app>
2      <servlet>
3          <servlet-name>Faces Servlet</servlet-name>
4          <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
5      </servlet>
6      <servlet-mapping>
7          <servlet-name>Faces Servlet</servlet-name>
8          <url-pattern>*.jsf</url-pattern>
9      </servlet-mapping>
10 </web-app>
```

Listagem 2-8 – Configuração do Controlador do JSF (Dudney et al., 2004).

A Listagem 2.9 mostra um tratador de uma ação no JSF. A primeira coisa que se pode perceber é que não há uma assinatura fixa no JSF como há em uma *Action* do *Struts*. O tratador aqui é um método simples em um JavaBean, sem parâmetros e que retorna apenas uma String. Neste exemplo, o método para tratar a ação é colocado em um *Managed Bean*, que pode ser comparado a um *ActionForm* no *Struts*. O método tratador retorna um resultado lógico ("*success*" ou "*failure*", neste caso), enquanto uma *Action* do *Struts* utiliza o resultado retornado para saber para qual página será redirecionada a requisição (Dudney et al., 2004).

Como parte do ciclo de vida da requisição, a próxima árvore de componentes (ou página JSP, neste exemplo) será determinada pelo resultado lógico retornado por este tratador. Um arquivo de configuração (assim como no *Struts*) fornece um mecanismo para definir o fluxo na interface do usuário (Dudney et al., 2004). A Listagem 2.10 mostra um exemplo de configuração deste fluxo.

```java
package jsf;

public final class LoginBean {
    private String password;
    private String userId;

    public String getPassword() {
        return (this.password);
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getUserId() {
        return (this.userId);
    }

    public void setUserId(String userId) {
        this.userId = userId;
    }

    public String login() {
        if ((userId == null) || (userId.length() < 1))
            return "failure";
        if ((password == null) || (password.length() < 1))
            return "failure";
        User user = null;
        UserSecurityService security = new UserSecurityService();
        user = service.login(userId, password);
        if(user == null)
            return "failure";
        else
            return "success";
    }
}
```

Listagem 2-9 – Exemplo de um *Managed Bean* no JSF (Dudney et al., 2004)

A regra de navegação mostrada na Listagem 2.10 especifica que caso o resultado da operação seja *"failure"*, o usuário irá permanecer na página "login.jsp". Se ele for bem sucedido (*"success"*) o usuário irá para a página "home.jsp" da aplicação.

```
1  <navigation-rule>
2      <from-view-id>/login.jsp</from-view-id>
3      <navigation-case>
4          <from-outcome>success</from-outcome>
5          <to-view-id>/home.jsp</to-view-id>
6      </navigation-case>
7      <navigation-case>
8          <from-outcome>failure</from-outcome>
9          <to-view-id>/login.jsp</to-view-id>
10     </navigation-case>
11 </navigation-rule>
```

Listagem 2-10 – Exemplo de Regras de Navegação no JSF (Dudney et al., 2004).

No JSF, de forma similar ao *ActionForm* do *Struts*, utiliza-se um *JavaBean* para armazenar e comunicar dados entre as camadas de Visão e Modelo da aplicação. Utiliza-se no JSF o termo "Modelo de Componentes" para referir-se a estes *JavaBeans*. Deve-se cuidar para não confundir este termo com o utilizado para referenciar objetos de negócio (Dudney et al., 2004).

Para que um *JavaBean* possa ser usado como um objeto do Modelo de Componentes, precisamos declará-lo como um *JavaBean* Gerenciável (*Managed Bean*), o que fará com que ele fique disponível para ser acessado pelos demais componentes do JSF. A Listagem 2.11 mostra como declarar um *JavaBean* como um *Managed Bean*.

```
1  <managed-bean>
2      <managed-bean-name>login</managed-bean-name>
3      <managed-bean-class>jsf.LoginBean</managed-bean-class>
4      <managed-bean-scope>request</managed-bean-scope>
5      <managed-property>
6          <property-name>userId</property-name>
7          <null-value />
8      </managed-property>
9      <managed-property>
10         <property-name>password</property-name>
11         <null-value />
12     </managed-property>
13 </managed-bean>
```

Listagem 2-11 – Declarando um *Managed Bean* no JSF (Dudney et al., 2004)

Ao utilizar um *Managed Bean*, deve-se garantir que cada propriedade declarada exista na classe definida, seguindo as convenções de nome de propriedades em JavaBeans. Declarados desta forma, cada *Managed Bean* pode ser acessado e vinculado a componentes de interface (Dudney et al., 2004).

Como pode ser percebido, a camada de Visão no JSF consiste principalmente da árvore de componentes. Um benefício do JSF é que componentes individuais ou a árvore de componentes como um todo podem ser desenhados de formas diferentes, para suportar múltiplos tipos de interfaces com usuário. Na maioria dos casos, é utilizada uma linguagem de marcação como o HTML (Dudney et al., 2004).

Outro benefício do JSF é que os componentes trazem um estilo mais orientado a eventos para as aplicações Web (Dudney et al., 2004). A Listagem 2.12 mostra um exemplo de uma página JSP utilizando os componentes do JSF.

```
1  <!doctype html public "-//w3c//dtd html 4.0 transitional//en">
2  <html>
3  <head>
4  <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
5  <title>Login</title>
6  <%@ taglib uri="http://java.sun.com/jsf/core/" prefix="f"%>
7  <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
8  </head>
9  <body>
10 Log in with your User ID and Password.
11 <br />
12 <br />
13 <f:view>
14     <h:form>
15         <h:outputLabel for="userId">
16             <h:outputText value="User ID"/>
17         </h:outputLabel>
18         <h:inputText id="userId" value="#{login.userId}" />
19         <br />
20         <h:outputLabel for="password">
21             <h:outputText value="Password" />
22         </h:outputLabel>
23         <h:inputSecret id="password" value="#{login.password}" />
24         <br />
25         <h:commandButton action="#{login.login}" value="Login" />
26     </h:form>
27 </f:view>
28 </body>
29 </html>
```

Listagem 2-12 – Exemplo de Página com Componentes do JSF (Dudney et al., 2004).

Pode-se perceber que cada campo de entrada (*input*) possui um atributo *value*. Este atributo permite a ligação entre os campos de entrada e as propriedades de um objeto. Neste exemplo, os campos de entrada estão ligados a propriedades do *JavaBean LoginBean*. Como esta classe foi declarada como um *Managed Bean*, ela é criada automaticamente, iniciada e colocada como atributo com escopo de requisição, quando a

página é processada. A chave para esta ligação é a primeira parte da expressão. O texto *"login"* deve coincidir com o nome dado na marcação <managed-bean-name>, no arquivo de configuração (ver Listagem 2.11) (Dudney et al., 2004).

Este mecanismo simples de ligação entre os componentes de interface e os *Managed Beans* permite capturar entradas do usuário e validá-las através do tratador *login()*. As regras de navegação garantem que o usuário será direcionado para a página apropriada após cada ação (Dudney et al., 2004).

Validações no JSF podem ser feitas através de classes que implementem a interface *Validator*. Uma coleção de validadores padrão já é distribuída e novos podem ser registrados e utilizados (Dudney et al., 2004). A Listagem 2.13 mostra um exemplo de uso de um *Validador*.

```
1 <h:inputText id="ccno" size="16" converter="#{creditCardConverter}"
2     required="true">
3     <f:validator type="demo.FormatValidator" />
4     <f:attribute name="formatPatterns"
5                 value="9999999999999999|9999 9999 9999 9999|9999-9999-9999-9999"/>
6 </h:inputText>
```

Listagem 2-13 – Exemplo de *Validador* no JSF (Dudney et al., 2004).

Como visto na seção 2.2.3.2, as requisições HTTP carregam apenas parâmetros do tipo String. Para convertê-los para os tipos adequados, o JSF utiliza um conversor. Este deve implementar a interface *Converter* e pode ser associado a um componente de forma similar aos *Validators* (Dudney et al., 2004).

O JSF fornece, também, suporte para internacionalização[14] similar ao do *Struts*, baseado em *ResourceBundles*. A Listagem 2.14 mostra como uma mensagem pode ser obtida em uma classe Java.

```
1 FacesMessage errMsg = MessageFactory.getMessage(
2         CONVERSION_ERROR_MESSAGE_ID,
3         (new Object[] { value, inputVal }));
```

Listagem 2-14 – Exemplo Recuperação de Mensagem no JSF.

[14] Capacidade do *software* de suportar diferentes idiomas.

Padrões de Projeto no JSF

Pode-se identificar, na arquitetura do JSF, diversos padrões de projeto fundamentais. A Tabela 2.2 mostra vários padrões de projeto implementados pelos componentes do JSF (Dudney et al., 2004) (Mann, 2005), sendo a maior parte deles descritos no catálogo "Core J2EE Patterns" (Deepak et al., 2001).

Padrões	Componente do JSF
Service to Worker	FacesServlet, Action Handlers, Event Listeners
Command (Gamma et al, 2002), Command and Controller, Front Controller, Singleton	FacesServlet, Action Handlers, Event Listeners
Dispatcher, Navigator	NavigationHandler, FacesServlet, Action Handlers, Event Listeners
View Helper, Session Façade	Action Handlers, Managed Beans
Transfer Objects (fka Value Objects), Value Object Assembler	Managed Beans, FacesMessage
View Helper	Managed Beans, FacesContext, taglibs
Strategy (Gamma et al, 2002)	ViewHandler, Renderer, RenderKit, Validator, Converter
Observer (Gamma et al, 2002)	Event Listeners
Composite View	UIComponent, UIComponentBase, UIViewRoot
Synchronizer Token	FacesServlet
Decorator (Gamma et al, 2002)	ViewHandler

Tabela 2-2 – Padrões de Projeto no JSF.

Comparando a Tabela 2.2 com a Tabela 2.1 (Padrões de Projeto no Struts), pode-se perceber uma grande interseção entre os conjuntos de padrões implementados por estes dois arcabouços.

O *FacesServlet* é o *Front Controller* no JSF, fazendo o papel do *ActionServlet* no *Struts*. Também seguindo o padrão *Command and Controller,* o *FacesServlet* executa métodos nos *Action Handlers* (Dudney et al., 2004).

Uma diferença com relação ao *Struts* é que os tratadores no JSF não implementam o padrão *Singleton*, como acontece com as *Actions* do *Struts*. Além disso, o modelo de eventos do JSF é mais elaborado e segue o padrão *Observer* (Gamma et al., 2002) (Dudney et al., 2004).

A Visão no JSF, assim como no *Struts*, implementa o padrão *Composite* (Gamma et al., 2002). No entanto, no *Struts* esta implementação é feita compondo várias páginas JSP através da marcação *template*, utilizando uma extensão (*plugin*) do *Struts* chamada *Tiles*. O JSF, por sua vez utiliza uma hierarquia de componentes bem mais próxima da estrutura

definida no padrão *Composite*, através de sua árvore de componentes. A Figura 2.7 mostra a implementação do *Composite* no JSF (Dudney et al., 2004).

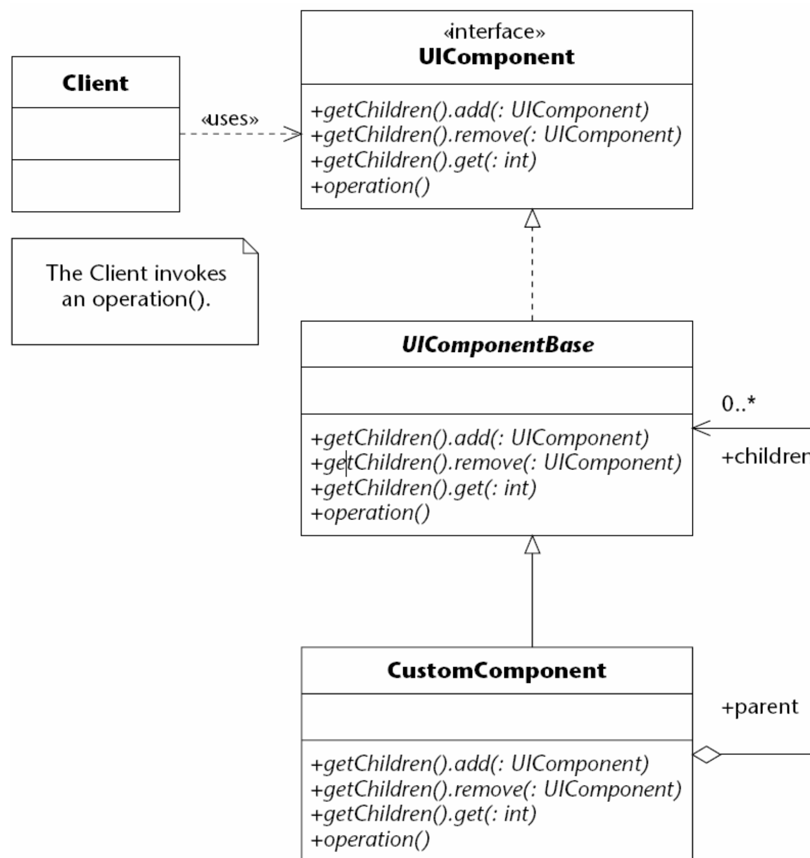

Figura 2-7 – Implementação do *Composite* no JSF (Dudney et al., 2004).

Além dos diversos padrões em comum, que podem ser percebidos pela comparação das Tabelas 2.1 e 2.2, o JSF implementa também o padrão *Strategy* (Gamma et al., 2002), para suportar diversos tipos de Validadores (*Validator*) e Conversores (*Converter*) para os dados da tela (Dudney et al., 2004).

2.3.3.3. GWT

O GWT (*Google Web Toolkit*) é um conjunto de ferramentas para desenvolvimento de aplicações Web. Contido neste conjunto está um arcabouço baseado em AJAX (vide seção 2.2.4.9) (GWT, 2008).

Esta seção descreve as características principais deste arcabouço e de algumas das demais ferramentas contidas neste conjunto de ferramentas.

No núcleo do GWT está um compilador de Java para *Javascript*, que produz código capaz de executar nos principais navegadores (Internet Explorer, Firefox, Mozilla, Safári e Ópera). Este compilador converte código em Java para *Javascript* utilizando versões em *Javascript* de classes Java comumente usadas (Hanson-Tacy, 2007).

O compilador do GWT recebe a localização do módulo a ser compilado como parâmetro. Um módulo é um conjunto de classes Java e arquivos relacionados acompanhados por um arquivo de configuração. Este arquivo de configuração contém a definição do módulo e, normalmente, inclui um ponto de entrada, que é uma classe que é chamada quando a aplicação inicia.

O compilador inicia com a classe de entrada, seguindo as dependências necessárias para compilar o código Java. No entanto, ele difere de compiladores tradicionais, pois ele não compila tudo o que está contido no módulo, mas inclui apenas os métodos e classes que estão sendo usados (Hanson-Tacy, 2007).

Além da informação sobre o ponto de entrada do lado cliente da aplicação, o arquivo de configuração do módulo – que deve ter o nome *"<nome_do_modulo>.gwt.xml"* – trás diversas informações, como definição de outros módulos utilizados, propriedades e definição de *Servlets* utilizados para comunicação com a parte servidora da aplicação (GWT, 2008). A Listagem 2.15 mostra um exemplo de um arquivo de configuração de um módulo.

```
1 <module>
2     <inherits name='com.google.gwt.user.User' />
3     <inherits name='com.google.gwt.json.JSON' />
4     <entry-point class='org.gwtbook.client.JSON' />
5 </module>
```

Listagem 2-15 – Exemplo de Configuração de um Módulo GWT (Hanson-Tacy, 2007).

Outro ponto importante é que o GWT não suporta o uso de reflexão, tipicamente utilizada para construção de arcabouços em Java. Esta limitação é o preço a se pagar pelas grandes otimizações feitas pelo compilador do GWT. Para que ele possa remover métodos e classes não utilizadas no código, é preciso que todas as amarrações entre as classes utilizadas possam ser resolvidas em tempo de compilação (GWT, 2008).

No entanto, o mecanismo de reflexão é uma grande ferramenta para o desenvolvimento, de modo que o GWT oferece uma alternativa para suprir a falta deste recurso. Existe um recurso denominado de "amarração adiada" (*deferred binding*), que

permite ao desenvolvedor criar classes dinamicamente por meio de geradores. Estes geradores podem utilizar propriedades para determinar a forma como construir uma determinada classe alvo, sem, no entanto, prejudicar o compilador em suas otimizações (Hanson-Tacy, 2007). Pode-se associar um gerador a uma classe qualquer, através de uma configuração feita no arquivo de configurações do módulo.

Além do compilador para *Javascript*, o GWT fornece um conjunto de ferramentas para auxiliar no desenvolvimento de aplicações Web. A Figura 2.8 mostra as principais, dividindo-as em dois conjuntos: as que estão vinculadas ao compilador e as que compõem as demais APIs do GWT (Hanson-Tacy, 2007).

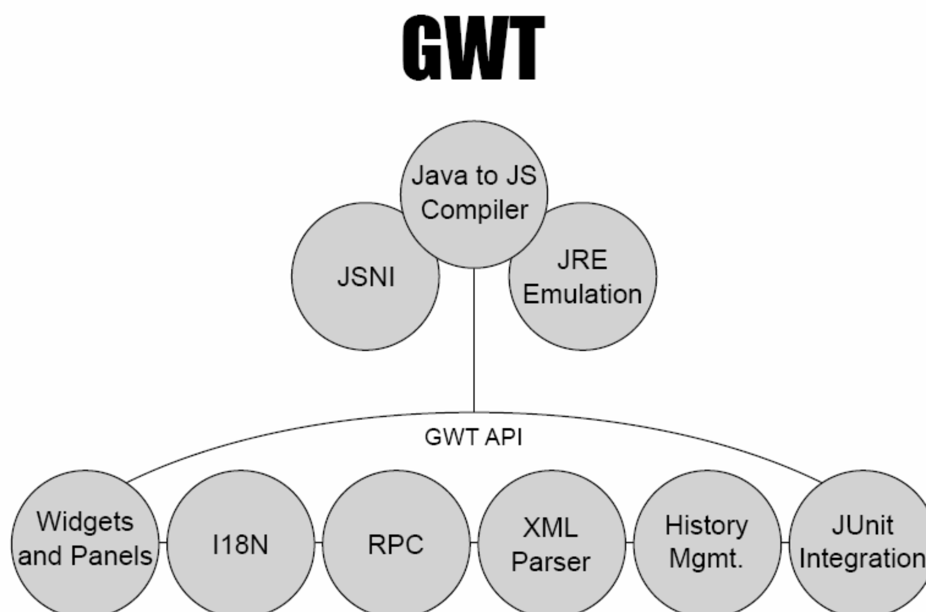

Figura 2-8 – Principais Componentes do GWT (Hanson-Tacy, 2007).

O compilador possui três modos de saída, que determinam o aspecto do código *Javascript*. O modo padrão é *"obfuscated"*, que faz com que o código final seja comprimido e praticamente impossível de ser lido. Isto ajuda a manter o código tão pequeno quanto possível, o que é muito importante para aplicações grandes. Os outros modos são *"pretty"* e *"detailed"*, que adicionam informações sobre o código para facilitar a leitura e análise de mensagens no navegador, devendo, portanto, ser usados apenas durante o desenvolvimento da aplicação (Hanson-Tacy, 2007). A Tabela 2.3 mostra o código *Javascript* gerado para um mesmo método em cada um dos três modos de saída.

Modo	Código Gerado
Obfuscated	function b(){return this.c+'@'+this.d();}

Pretty	function _toString(){
	return this._typeName + '@' + this._hashCode();
	}

Detailed	function java_lang_Object_toString__(){
	return this.java_lang_Object_typeName + '@' + this.hashCode__();
	}

Tabela 2-3 – Modos de Saída do Compilador GWT.

Outro aspecto importante do compilador GWT é que ele gera o código a partir do arquivo fonte em Java e não do arquivo compilado (*.class*), o que faz com que seja necessário incluir o código fonte dos arquivos ao se distribuir componentes reutilizáveis. Além disso, ele gera um arquivo para cada tipo de navegador alvo. Um código de iniciação na página principal determina qual destes arquivos gerados deve ser incluído, com base no navegador e localidade do cliente que acessou a aplicação. Isto faz com que o cliente não precise receber código que ele não vai usar. (Hanson-Tacy, 2007).

O GWT fornece uma forma de integração do código Java com códigos *Javascript* nativos. A JSNI (*Javascript Native Interface*) permite a execução de códigos *Javascript* a partir de um código Java, assim como executar códigos Java a partir do *Javascript*. Isto é possível porque o compilador GWT mescla códigos nativos em *Javascript* com o código gerado a partir do Java (Hanson-Tacy, 2007). A Listagem 2.16 mostra a inserção de código *Javascript* nativo em uma classe Java.

```
1  package gwt;
2  
3  public class Teste {
4  
5      public native int addTwoNumbers (int x, int y)
6      /*-{
7      var result = x + y;
8      return result;
9      }-*/;
10 
11     public native void fillData (List data)
12     /*-{
13     data.@java.util.List::add(Ljava/lang/Object;)('item1');
14     data.@java.util.List::add(Ljava/lang/Object;)('item2');
15     }-*/;
16 }
17 
```

Listagem 2-16 – Exemplo de JSNI do GWT.

Em Java, pode-se declarar um método como nativo (*native*), alertando o compilador que a implementação do método será escrita em alguma outra linguagem. Segundo a especificação da linguagem Java, quando se declara um método como nativo, não se pode especificar um bloco de código para o método. Pode-se perceber na Listagem 2.16 que o bloco de código *Javascript* aparece dentro de um comentário Java. Esse código será executado quando o método for chamado. Este mecanismo satisfaz a exigência sintática do Java e ao mesmo tempo informa ao compilador GWT o código *Javascript* que deve ser utilizado (Hanson-Tacy, 2007).

Na Listagem 2.16 pode-se perceber, também, como chamar um método de um objeto Java a partir do código *Javascript*. O método "*fillData*" chama o método *"add"* do objeto data que é do tipo *java.util.List*.

O GWT fornece, também, uma biblioteca para emular as classes Java distribuídas com a máquina virtual Java (*JRE Emulation*). Esta biblioteca contém as partes mais importantes da JRE (Hanson-Tacy, 2007).

Para criação das interfaces, o GWT utiliza componentes de tela de uma forma semelhante ao *Swing*[15]. *Widget* e *Panel* são utilizadas como classes base para construção de outros componentes. Um *Widget* é um controle utilizado pelo usuário (um botão, por exemplo) e um *Panel* é um container onde outros controles podem ser adicionados. Para tratamento de eventos, subclasses de *EventListener* podem ser adicionadas aos componentes.

A Listagem 2.17 mostra um exemplo de um jogo da velha simples implementado no GWT. Neste exemplo é criado um *EntryPoint*, que é chamado quando a aplicação inicia. Um tratador para o evento de clique dos botões é definido e uma tabela de 3 x 3 posições é criada, adicionando um botão em cada uma das posições da tabela (Hanson-Tacy, 2007).

[15] Biblioteca para construção de interfaces gráficas para aplicações desktop em Java, distribuída juntamente com a JDK.

```java
package gwt;

public class App implements EntryPoint {
    private boolean playerToggle = true;
    private ClickListener listener = new ClickListener(){
        public void onClick(Widget sender) {
            Button button = (Button) sender;
            if (button.getText().equals("")) {
                if (playerToggle) {
                    button.setText("X");
                }
                else {
                    button.setText("O");
                }
                playerToggle = !playerToggle;
            }
            else {
                Window.alert("That square is already taken ");
            }
        }
    };
    public void onModuleLoad() {
        Grid grid = new Grid(3, 3);
        for (int col = 0; col < 3; col++) {
            for (int row = 0; row < 3; row++) {
                Button button = new Button();
                button.setPixelSize(30, 30);
                button.addClickListener(listener);
                grid.setWidget(col, row, button);
            }
        }
        RootPanel.get().add(grid);
    }
}
```

Listagem 2-17 – Exemplo de Tratamento de Eventos no GWT (Hanson-Tacy, 2007).

Outra funcionalidade importante é o suporte para internacionalização. O GWT oferece um suporte interessante para definição de constantes e de mensagens internacionalizadas. Através de um arquivo de propriedades[16] o programador define todas as mensagens que serão utilizadas. Pode-se definir um arquivo para cada idioma suportado. Um exemplo deste arquivo pode ser visto na Listagem 2.18 (Hanson-Tacy, 2007).

```
welcomeMessage = Welcome to my book {0} {1}
logoImage = /images/logo.jpg
```

Listagem 2-18 – Exemplo de Arquivo de Mensagens no GWT (Hanson-Tacy, 2007).

[16] Arquivo texto contendo várias linhas da forma chave=valor.

É criada, então uma interface contendo métodos cujos nomes coincidem com as chaves utilizadas no arquivo de propriedades onde estão as mensagens. A Listagem 2.19 mostra um exemplo de uma interface destas (Hanson-Tacy, 2007).

```
1  package gwt;
2
3  public interface I18n extends Messages
4  {
5      String welcomeMessage(String fname, String lname);
6      String logoImage();
7  }
```

Listagem 2-19 – Exemplo de Interface de Mensagens no GWT (Hanson-Tacy, 2007).

Utilizando o mecanismo de amarração adiada, o compilador constrói a classe que de fato é utilizada pela aplicação, com base na interface definida para as mensagens e em uma propriedade informando a localização do cliente (Hanson-Tacy, 2007). A Listagem 2.20 mostra um exemplo de criação de uma classe de mensagens pela aplicação.

```
1      I18n messages = (I18n) GWT.create(I18n.class);
2      GWT.log (messages.logoImage())
```

Listagem 2-20 – Exemplo de Uso de Arquivo de Mensagens no GWT

Para comunicação com o servidor, o GWT oferece algumas alternativas interessantes. Existe uma implementação para o objeto *XMLHttpRequest* do navegador, que permite realizar requisições para o servidor. Além disso, é fornecida uma API para implementar chamadas remotas de métodos (RPC) de forma similar ao que o Java fornece em sua API RMI. Define-se a interface da classe que irá residir no servidor e que será chamada pelo cliente. Implementa-se uma classe no servidor estendendo uma subclasse de *Servlet* (disponibilizada pelo GWT) e implementando a interface remota definida. O cliente utiliza o método GWT.create, passando a interface definida (de forma semelhante à utilizada para criação da classe de mensagens) e o GWT, através de amarração adiada, cria uma classe que sabe se comunicar com a classe de implementação no servidor. Pode-se comparar a classe criada com um *stub* de comunicação no RMI (Hanson-Tacy, 2007), que segue o padrão *Proxy* (Gamma et al, 2002). A Listagem 2.21 mostra um exemplo de interface remota de serviço.

```
1  package gwt;
2
3  public interface PasswordService extends RemoteService {
4      Boolean changePassword (String user, String oldPass, String newPass);
5  }
```

Listagem 2-21 – Exemplo de Interface Remota (Hanson-Tacy, 2007).

Na Listagem 2.22, é mostrado um exemplo de classe de implementação para uma interface remota.

```
1  package gwt;
2
3  public class PasswordServiceImpl extends RemoteServiceServlet
4                                  implements PasswordService {
5      public Boolean changePassword (String user,
6              String old, String new) {
7          return true;
8      }
9  }
10 
```

Listagem 2-22 – Exemplo de Classe Remota de Serviço (Hanson-Tacy, 2007).

O resultado de uma chamada RPC é tratado por subclasses de *AsyncCallback*. A Listagem 2.23 mostra um exemplo de chamada a um método remoto, tratando o resultado retornado.

```
1      service.changePassword("jdoe", "abc123", "m@tr1x",
2              new AsyncCallback()
3      {
4          public void onSuccess (Object result) {
5              Window.alert("password changed");
6          }
7          public void onFailure (Throwable ex) {
8              Window.alert("uh oh!");
9          }
10     });
11
```

Listagem 2-23 – Exemplo de Chamada RPC no GWT (Hanson-Tacy, 2007).

O GWT fornece, também, classes para seriação de objetos no formato JSON[17], para processamento de documentos XML e para realização de testes de unidade com o

[17] JSON (Javascript Object Notation) é um formato para intercâmbio de informações, baseado em um subconjunto da linguagem Javascript (JSON, 2008).

arcabouço jUnit[18]. Além disso, acompanham o arcabouço várias ferramentas para aumentar a produtividade. São distribuídas ferramentas para criação do esqueleto do projeto e para criação de classes de mensagens, além de um navegador embutido, que permite a execução e depuração do código Java, sem que ele tenha sido traduzido para *Javascript* ainda (Hanson-Tacy, 2007).

Uma das grandes reclamações sobre as aplicações que utilizam um modelo de interfaces ricas, como as baseadas no modelo AJAX, é a quebra da funcionalidade "Voltar" do navegador (Nielsen, 1998) (Hanson-Tacy, 2007). Quando se substitui um pedaço do conteúdo da página com *Javascript*, o navegador não considera isso como uma troca de página.

Uma solução popular para este problema é utilizar um frame oculto na página e utilizar uma boa quantidade de código *Javascript* para manipular este frame quando o botão "Voltar" é acionado. Esta solução é complexa e o GWT já faz todo este trabalho para o programador. Basta que se escreva um tratador para o evento de mudança no histórico e o adicione ao objeto que representa o histórico no GWT (Hanson-Tacy, 2007). A Listagem 2.24 mostra um exemplo.

```
1  package gwt;
2
3  public class ExemploHistorico implements EntryPoint {
4      public void onModuleLoad() {
5          History.addHistoryListener(new HistoryListener()
6          {
7              public void onHistoryChanged (String historyToken) {
8                  if (historyToken.equals("overview")) {
9                      // display overview panel
10                 }
11                 else if (historyToken.equals("reports")) {
12                     // display reports panel
13                 }
14             }
15         }
16         );
17     }
18 }
```

Listagem 2-24 – Exemplo de Controle do Histórico no GWT (Hanson-Tacy, 2007).

Com o tratador de eventos registrado, pode-se simular uma mudança de página através da criação de uma "ficha" (*token*) de histórico. Uma ficha é uma chave que define uma

[18] Arcabouço com código aberto para testes de unidade na linguagem Java, desenvolvido por Eric Gamma e Kent Beck (jUnit, 2008).

mudança de conteúdo. Este conteúdo pode estar associado a uma mudança em um painel de abas ou a um item de menu (Hanson-Tacy, 2007). A Listagem 2.25 mostra como se adicionar uma ficha ao histórico.

```
5     public void overviewClick()
6     {
7         History.newItem("overview");
8         ...
9     }
10
11    public void reportsClick()
12    {
13        History.newItem("reports");
14        ...
15    }
```

Listagem 2-25 – Exemplo de Criação de Ficha de Histórico no GWT.

A criação de uma ficha faz duas coisas. Primeiramente, ela carrega invisivelmente uma nova página em um frame oculto, utilizado para controle do histórico. Como o frame oculto é carregado com uma nova página, o navegador conta isso como uma mudança de página e adiciona isso ao histórico do navegador. Em segundo lugar, ela chama o método *onHistoryChanged()* do *HistoryListener* registrado para indicar a mudança de conteúdo (Hanson-Tacy, 2007).

Padrões de Projeto no GWT

Pode-se identificar na arquitetura do GWT, diversos padrões de projeto fundamentais. A Tabela 2.4 mostra vários padrões de projeto implementados pelos componentes do GWT (Hanson-Tacy, 2007), sendo a maior parte deles descritos no catálogo "Core J2EE Patterns" (Deepak et al., 2001).

Padrões	Componente do GWT
Command (Gamma et al, 2002)	Command, Event Listeners, AsyncCallback
Listener	Event Listeners, SourcesEvent, History
Factory	GWT
Proxy, Façade	RemoteService, RemoteServiceServlet
Transfer Objects (fka Value Objects)	Event
View Helper	JSONValue
Strategy (Gamma et al, 2002)	DOMImpl, GWT Compiler
Observer (Gamma et al, 2002)	Event Listeners, SourcesEvent
Composite View	Widget, Panel

Tabela 2-4 – Padrões de Projeto no GWT

Ao contrário do Struts e do JSF, o GWT não utiliza um controlador centralizado, embora possa fazê-lo. Os objetos no servidor podem ser acessados através de *proxies* criadas pelo método de fábrica – segundo o padrão *Factory* (Gamma et al., 2002) – GWT.create(), que recebe como parâmetro uma interface remota, que também deve ser implementada pela classe de serviço (Hanson-Tacy, 2007).

O padrão *Strategy* é utilizado para garantir a compatibilidade com diversos navegadores. O compilador do GWT utiliza classes diferentes para cada tipo de navegador, assim como a implementação do DOM é diferenciada (Hanson-Tacy, 2007).

O GWT também implementa o padrão *Composite View* para construção da tela, através de suas classes *Widget* e *Panel*. Para o tratamento de eventos dos componentes de tela são utilizados os padrões *Observer* e *Listener* (Hanson-Tacy, 2007).

O padrão *Command* também é utilizado para abstrair as ações que devem ser realizadas em resposta a uma execução no servidor ou para abstrair ações de botões na interface e pode ser implementado através das interfaces *AsyncCallback* e *Command* (Hanson-Tacy, 2007).

2.3.3.4. Análise das Soluções

Os arcabouços analisados neste trabalho têm como escopo auxiliar a construção de interfaces com o usuário nas aplicações Web. Desta forma, esta seção os compara, considerando dois aspectos que consideramos fundamentais para arcabouços com este escopo e que direcionaram todo o desenho da solução que apresentamos nesta dissertação (vide capítulo 3): (1) a forma como são organizados os elementos de tela (árvore de componentes) e (2) como ocorre a comunicação entre a interface e a camada de negócio da aplicação.

Árvore de Componentes

Tanto o JSF quanto o GWT são arcabouços de caixa preta, que organizam a camada de visão em forma de uma árvore de componentes. Já o *Struts,* o mais antigo dos arcabouços analisados, não possui uma estrutura de componentes para construir a sua camada de visão. O suporte para construção de interfaces é muito primitivo. O desenvolvedor acaba

tendo que lidar com toda a complexidade envolvida na criação de interfaces utilizando HTML.

A árvore de componentes do JSF, no entanto, existe apenas no lado servidor da aplicação. Os componentes são desenhados por marcações próprias e um HTML final é produzido. Toda a abstração de componente, que existe no servidor, desaparece no lado cliente da aplicação. Não existem mais componentes, mas apenas aquilo que foi desenhado pelas marcações associadas aos mesmos.

Caso um evento precise executar no cliente, ele não tem como acessar a estrutura original, que só existe no servidor. Além disso, não há um suporte para criação destes eventos, visto que a linguagem usada no cliente é diferente da usada no servidor.

O GWT, por sua vez, monta uma árvore de componentes no lado cliente, mas que não existe no servidor. Além disso, o modelo de construção da árvore no GWT é muito parecido com o do *Swing*, que, apesar de poderoso, não é nada prático. Enquanto no JSF, pode-se definir a posição de um componente, associar um validador de entradas, ou um evento de forma declarativa (na própria página JSP), no GWT é preciso criar os objetos programaticamente, definindo, via código Java, posição e dimensões dos componentes. Isto faz com que telas relativamente simples obriguem a escrita de centenas de linhas de código, que além de tudo são trabalhosas de serem escritas.

Consideramos que o ideal seria juntar os benefícios vistos em ambos os arcabouços: criar a árvore de componentes de forma declarativa, disponibilizar essa árvore tanto no servidor como no cliente e permitir que códigos cliente e servidor pudessem ser escritos em uma mesma linguagem.

Modelo de Comunicação Entre Cliente e Servidor

Tanto o *Struts* quanto o JSF não possuem um suporte nativo para o modelo AJAX, embora seja possível utilizá-lo com estes arcabouços. Além disso, é utilizado um controlador centralizado que recebe as requisições e delega o controle para uma classe de tratamento da aplicação. Uma coleção de arquivos em XML é utilizada para mapear URLs para classes de tratamento, assim como parâmetros da requisição para objetos passados aos controladores. A navegação entre as páginas também é configurada através de arquivos XML.

No GWT, por outro lado, o modelo AJAX é suportado nativamente. Não é necessário mapear a navegação, ou as classes controladoras em arquivos XML. As requisições não são centralizadas. Para comunicar com o servidor, o que o GWT fornece de mais automático é a criação de *proxies* para objetos remotos, que são implementados como *Servlets*.

Desta forma, em todos os três arcabouços mencionados acima, é necessário realizar uma ou mais das seguintes tarefas:

1. Criar classes de controle que devem estender uma classe básica do arcabouço (gerando mais dependência).
2. Amarrar as classes de tratamento e de representação dos parâmetros em arquivos XML.

O GWT (onde os objetos remotos são *servlets* que estendem uma classe base do GWT) e o *Struts* (onde os controladores são subclasses de *Action*) obrigam a execução da primeira tarefa. O JSF e o *Struts* obrigam a execução da segunda tarefa.

Consideramos que o ideal seria permitir a construção de tratadores que fossem classes Java simples, sem exigir a herança de qualquer tipo de classe ou interface, além de não necessitar, também, de qualquer tipo de mapeamento em arquivos XML para as classes tratadoras e para as classes de representação dos parâmetros. Outro ponto importante seria suportar o AJAX de modo nativo.

CAPÍTULO 3 - O Arcabouço Crux

Este capítulo apresenta o Crux, o nosso arcabouço criado para atacar o problema de construção de aplicações Web. Neste capítulo, são apresentadas as principais decisões de projeto tomadas durante a sua concepção. O capítulo 4 apresenta aspectos mais detalhados da codificação feita para este arcabouço.

A seção 3.1 apresenta características gerais da solução, detalha o objetivo do arcabouço, faz observações a respeito da plataforma escolhida como base de desenvolvimento e delimita o escopo da solução.

A seção 3.2 mostra a arquitetura do arcabouço elaborado, destacando os padrões de projeto utilizados.

A seção 3.3 faz uma análise da solução proposta e discute como ela ataca os desafios relacionados ao uso de arcabouços (vide seção 2.1.1.3) e relacionados à utilização da Web como plataforma (vide seção 2.2.3.2), além de apresentar uma comparação com as demais soluções mostradas na seção 2.3.3.

3.1 Características Gerais

O Crux é um arcabouço de caixa preta voltado para o desenvolvimento de interfaces em aplicações Web de uso geral (não necessariamente ligadas a nenhum domínio específico).

As aplicações desenvolvidas com base no Crux suportam o modelo AJAX (vide seção 2.2.4.9) sem, no entanto, exigir do desenvolvedor da aplicação um profundo conhecimento a respeito deste modelo.

3.1.1. Objetivo

O objetivo do Crux é simplificar o desenvolvimento de aplicações Web, assim como permitir que sejam criadas aplicações com interfaces mais elaboradas, que proporcionem uma melhor qualidade de interação com o usuário.

Durante a sua construção tentamos cercar os desafios relacionados ao desenvolvimento para Web que ainda não são completamente tratados por nenhuma das soluções analisadas em nossa pesquisa, conforme discutido na seção 2.3.3.4.

3.1.2. Plataforma

O Crux foi escrito em Java e, internamente, utiliza o GWT, versão 1.5, para criação de código para execução no navegador do cliente (*Javascript*).

O cliente pode utilizar qualquer um dos seguintes navegadores para acessar uma aplicação desenvolvida com o Crux: Internet Explorer, Firefox, Mozilla, Safári e Ópera. Estes são os navegadores para o qual o GWT é capaz de gerar código compatível (Hanson-Tacy, 2007).

As aplicações desenvolvidas devem ser compatíveis com a versão 5.0 da máquina virtual Java e estar de acordo com a especificação JEE 1.4 ou superior (Sun, 2008).

3.1.3. Escopo

O Crux pode ser classificado como um arcabouço de infra-estrutura de sistemas (vide seção 2.1.1.2). Ele auxilia a criação de interfaces gráficas em aplicações Web e a comunicação destas interfaces com o restante da aplicação, executada no servidor.

Não está no escopo deste trabalho determinar uma metodologia para o desenvolvimento nem, tampouco, analisar todos os demais arcabouços que podem ser utilizados em outras camadas das aplicações Web, como acesso a dados, controle de transações ou estruturação das regras de negócio.

3.2 Arquitetura

Esta seção descreve a arquitetura definida para o Crux. A seção 3.2.1 apresenta uma visão geral do arcabouço, introduzindo sua estrutura e mostrando como uma aplicação pode ser desenvolvida com o seu suporte.

A seção 3.2.2 detalha o funcionamento dos seus componentes internos, mostrando como o arcabouço possibilita que as aplicações sejam executadas.

Neste capítulo, são mostrados alguns exemplos de código escritos para uma aplicação hipotética baseada no Crux. Estes são escritos em Java e em HTML.

3.2.1. Visão Geral

Esta seção introduz o arcabouço Crux e está dividida em duas seções.

A seção 3.2.1.1 mostra uma visão de contexto, destacando, em alto nível, os principais componentes da arquitetura.

A seção 3.2.1.2 apresenta uma visão geral do ponto de vista de desenvolvimento de aplicações com o arcabouço, destacando os tipos de artefatos que são produzidos e os pontos de extensão existentes.

3.2.1.1 Contexto

O Crux utiliza DHTML (vide seção 2.2.4.8) para construção das interfaces com o usuário no lado cliente da aplicação (pelo navegador) e um modelo de comunicação de dados com o servidor baseado no AJAX (vide seção 2.2.4.9), de modo que apenas dados são transmitidos e não documentos inteiros (dados + estrutura HTML).

A Figura 3.1 apresenta a estrutura de uma aplicação baseada no Crux. Este possui duas partes principais: um motor que é adicionado às páginas HTML e um motor localizado em um servidor Web. O motor adicionado ao cliente cuida de desenhar os elementos de tela no navegador do usuário, enquanto o motor localizado no servidor cuida de processar requisições e encaminhar para os tratadores de eventos, que deverão realizar processamentos necessários, como, por exemplo, acessar informações em um banco de dados, acessar um *web service* ou chamar qualquer outra classe de negócio da aplicação.

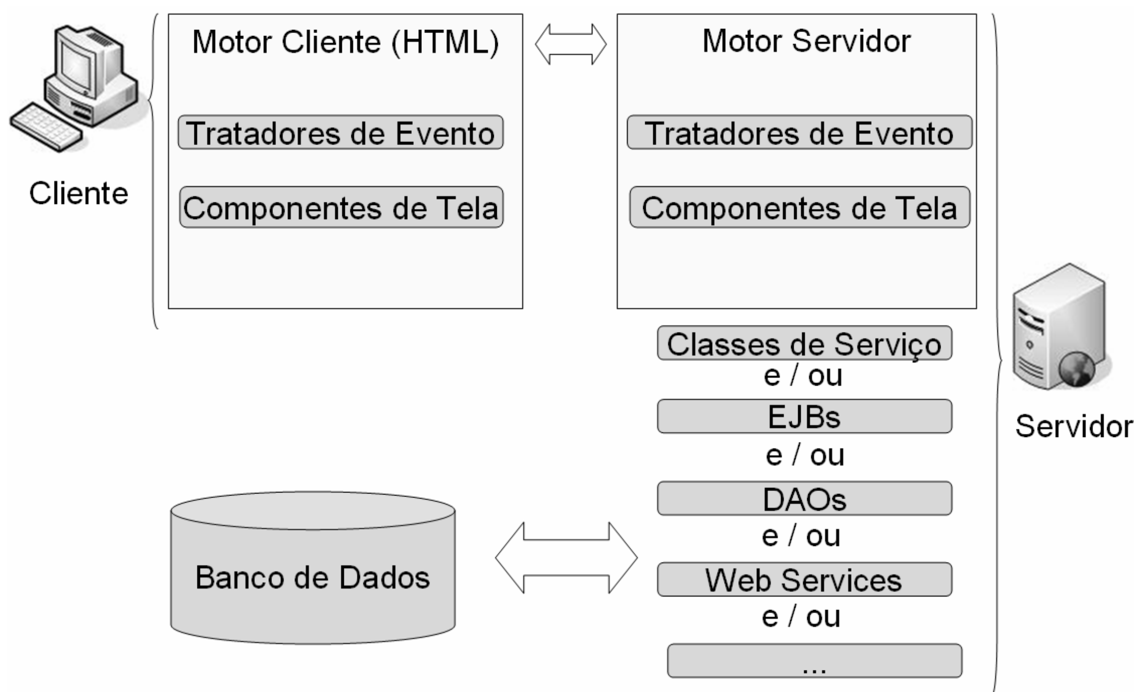

Figura 3-1 – Visão Geral da Arquitetura do Crux.

O Motor Cliente, mostrado na Figura 3.1, cuida de desenhar os elementos (componentes) de tela no navegador do usuário, além de coordenar os eventos associados a estes elementos, como cliques de *mouse* ou mudanças de estado. Estes eventos são interceptados pelo Motor Cliente que delega o tratamento dos mesmos para classes tratadoras específicas da aplicação. Os eventos podem ser de diversos tipos, podendo ser processados no próprio navegador (como, por exemplo, para exibir uma mensagem de alerta), ou no servidor (como, por exemplo, para recuperar uma informação do banco de dados).

Caso o evento deva ser processado no navegador (lado cliente), o Crux cuida de traduzir a classe tratadora para *Javascript* (utilizando para isso o compilador do GWT, antes da página ser enviada para o cliente), permitindo que a mesma possa ser executada sem problemas.

Caso o evento seja processado no lado servidor, o Motor Cliente cuida de comunicar ao Motor Servidor a sua ocorrência. O Motor Servidor, então, se encarrega de delegar o processamento do evento para uma classe tratadora da aplicação. Após o término do tratamento, o Motor Servidor retorna o controle para o Motor Cliente que, se necessário, atualiza o estado da página com as mudanças provocadas pelo evento.

As classes tratadoras de evento no servidor podem utilizar qualquer outra tecnologia ou arcabouço para realizar as regras de negócio da aplicação, como é ilustrado na Figura 3.1.

Os elementos de uma tela são representados no Crux por componentes que, de forma semelhante à do JSF e do GWT, estão estruturados em árvore. Como pode ser visto na Figura 3.1, estes componentes de tela podem ser acessados tanto por tratadores de eventos que executem no lado cliente como por tratadores que executem no lado servidor da aplicação. Os Motores Cliente e Servidor cuidam de manter sincronizadas as estruturas entre os dois lados da aplicação. A seção 3.3.5 discute os impactos no desempenho do arcabouço em função da adoção desta estratégia.

Este modelo também segue o padrão de projetos MVC2 (vide seção 2.3.1), onde a tela e os componentes de tela constituem a camada de visão, os Motores Cliente e Servidor constituem a camada de controle e os tratadores de evento são responsáveis por invocar os objetos da camada de negócio (modelo).

3.2.1.2 Desenvolvimento

Para criar uma tela em uma aplicação com o Crux, basta criar uma página HTML convencional. Dentro desta página, o programador deve incluir o Motor Cliente e pode utilizar as marcações <span> do HTML para indicar que deseja adicionar um componente do Crux.

A Listagem 3.1 mostra um exemplo de uma tela simples, contendo um botão e uma caixa de texto, dentro de um painel que, se necessário, adiciona barras de rolagem. Pode-se perceber, também, a inclusão de um script no topo da página. Este script é gerado por um módulo do GWT (vide seção 2.3.3.3) e este módulo carrega o Motor Cliente do Crux.

```
<html>
    <head>
        <script language='javascript'
            src='br.ufmg.thiago.teste.CruxTeste.nochache.js'</script>
    </head>
    <body>
        <table class='FormTable'>
            <tr>
                <td align='left' valign='middle'>
                    <span id='panel' _type='scrollPanel'
                                    _style='width:100%' >
                        <span id='panel.testeButton1' _type='button'
                              _value='Teste1'
                              _onclick='serverHandler.metodoTeste'></span>
                        <span id='panel.testeTextbox' _type='textBox'
                              _value='Valor Teste'
                              _onchange='clientHandler.metodoTeste|client'></span>
                    </span>
                </td>
            </tr>
        </table>
    </body>
</html>
```

Listagem 3-1 – Exemplo de Tela no Crux.

No exemplo acima, pode-se perceber que os componentes de tela são inseridos pelas marcações <span>, com um atributo "*_type*" sendo utilizado para informar qual o tipo do componente que se deseja adicionar. Outros atributos podem ser adicionados para informar ao Motor Cliente como desenhar o componente, ou a quais eventos o componente deverá responder. No exemplo mostrado, existem dois eventos informados, um deverá ser executado no servidor (através de uma chamada AJAX) e outro no cliente. A seção 3.2.2.1 apresenta o motivo que levou à utilização destas marcações para representar os componentes.

A Figura 3.2 ilustra como o Crux funciona. São mostrados os artefatos que o programador da aplicação deve produzir e como estes são interpretados pelo Crux até o envio da página para o usuário e no processamento de eventos no servidor.

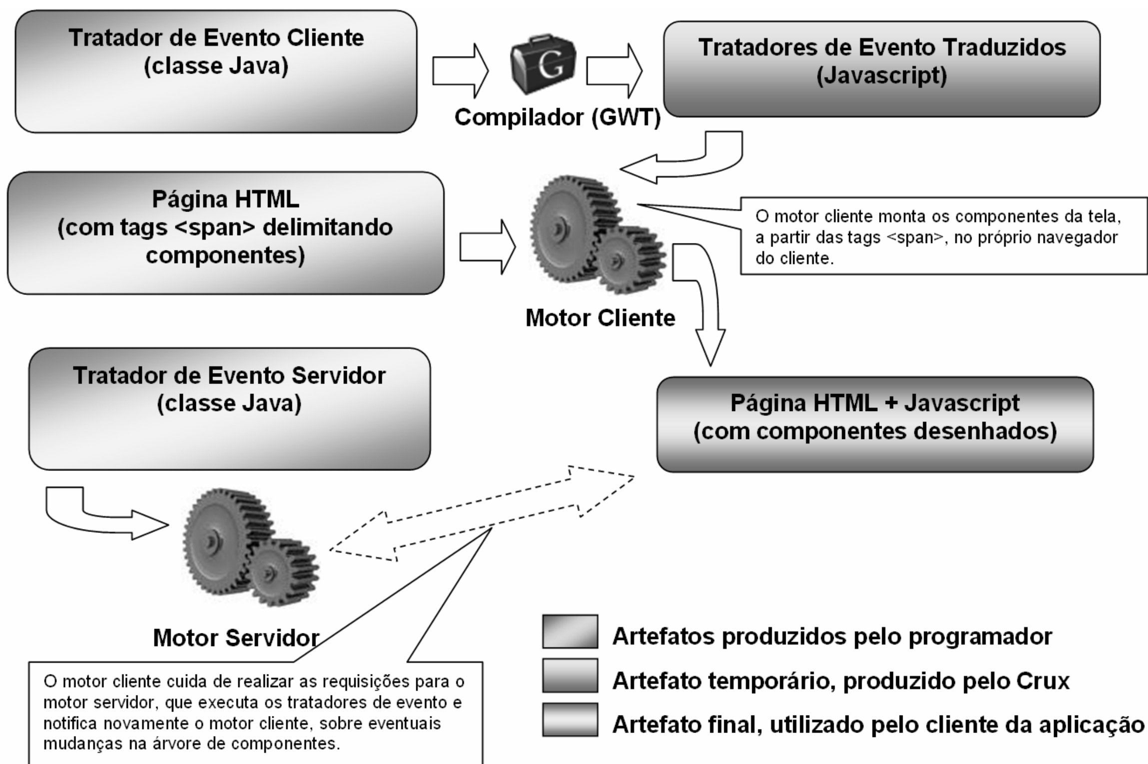

Figura 3-2 – Desenvolvimento com o Crux.

Os tratadores de evento são classes Java simples. Para que uma classe seja registrada como tratadora de um evento a ser executado no lado servidor da aplicação, basta adicionar a anotação "*Controller*" à classe. Através desta anotação, informa-se um nome, que identificará este tratador de eventos junto ao Crux. Na Listagem 3.1, é adicionado um evento de clique ao botão "panel.testeButton1" que executará o método "metodoTeste" da classe mostrada na Listagem 3.2.

```
1  @Controller("serverHandler")
2  public class TestController
3  {
4      private Screen screen;
5
6      public Screen getScreen() {
7          return screen;
8      }
9
10     public void setScreen(Screen screen) {
11         this.screen = screen;
12     }
13
14     public void metodoTeste()
15     {
16         screen.getComponent("panel.testeTextbox").setVisible(false);
17     }
18 }
```

Listagem 3-2 – Exemplo de Tratador de Eventos no Crux.

Pode-se perceber, na Listagem 3.2, que o método invocado ("metodoTeste") altera o componente de tela "panel.testeTextbox", modificando a sua visibilidade. O Motor Servidor do Crux se encarregará de notificar, ao término do evento, o Motor Cliente sobre esta modificação, para que o último possa refleti-la na tela.

O objeto "*screen*", acessado no exemplo da Listagem 3.2, representa a árvore de componentes da tela.

Este exemplo ilustra uma das formas de comunicação entre interface e servidor. Por motivos de desempenho, outras são disponibilizadas, através de eventos que não sincronizam o estado da árvore de componentes, conforme mostrado nas seções 3.2.2.1 e 3.2.2.2.

Para uma classe ser registrada como um tratador para um evento a ser executado no lado cliente da aplicação, o processo é similar ao realizado para tratadores do lado servidor. Um tratador de eventos cliente pode ser escrito como mostrado na Listagem 3.3, utilizando a anotação *ClientController*.

```java
@ClientController("clientHandler")
public class Teste{
    private String idSender;

    public String getIdSender() {
        return idSender;
    }

    public void setIdSender(String idSender) {
        this.idSender = idSender;
    }

    public void metodoTeste()
    {
        Window.alert("Funcionou!!! O componente "+idSender+" disparou esse evento!");
    }
}
```

Listagem 3-3 – Exemplo de Tratador de Evento Cliente no Crux.

O código acima, mesmo sendo executado no cliente, pode ser depurado normalmente dentro da mesma IDE utilizada para desenvolver e depurar classes executadas no servidor. Isto é feito utilizando a emulação da JRE oferecida pelo GWT (vide seção 2.3.3.3).

3.2.2. Detalhamento da Arquitetura

Como visto na seção 3.2.1, o Crux possui os seguintes componentes principais: o Motor Cliente, o Motor Servidor e a Árvore de Componentes. A Figura 3.3 mostra estes componentes e suas principais interfaces.

As seções 3.2.2.1, 3.2.2.2 e 3.2.2.3 mostram o funcionamento de cada um desses componentes.

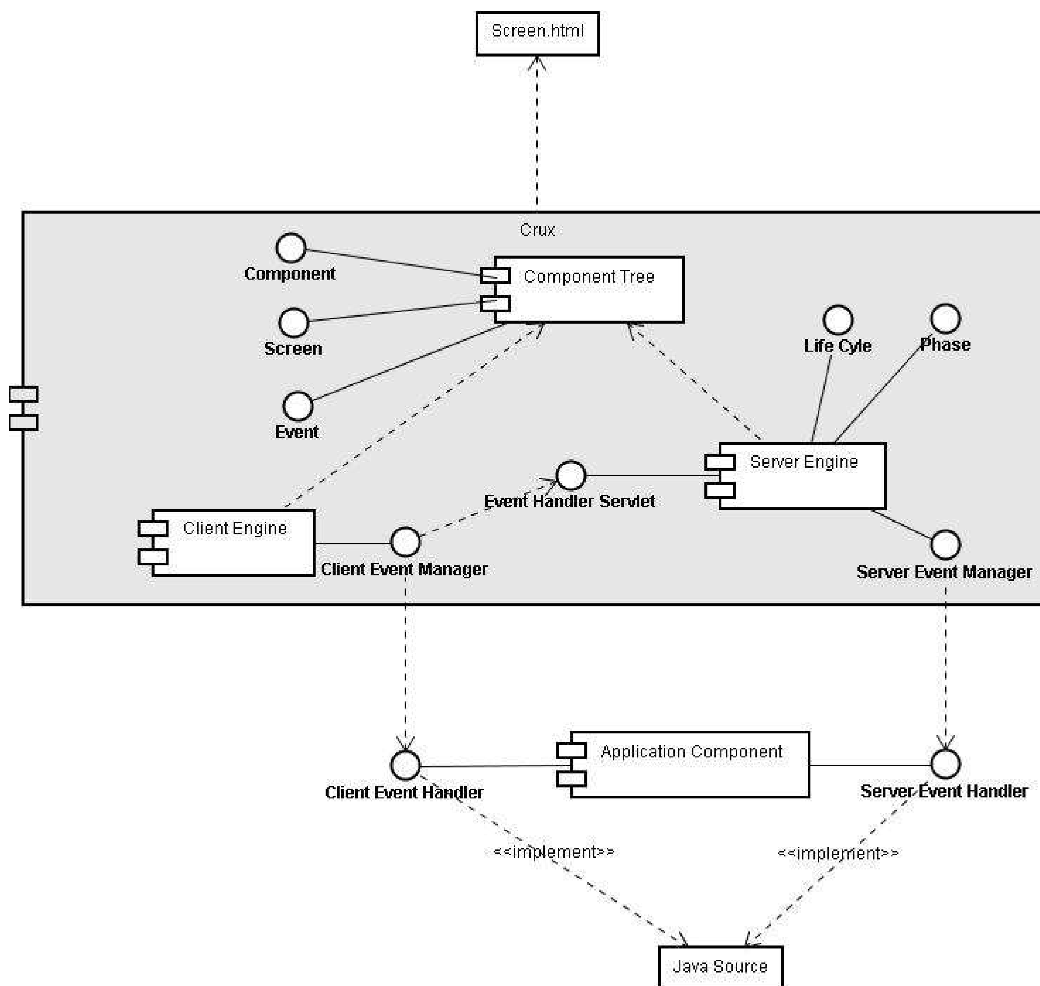

Figura 3-3 – Principais Componentes do Arcabouço Crux.

Observando a Figura 3.3, pode-se perceber que tanto o Motor Cliente quanto o Motor Servidor utilizam a Árvore de Componentes. Esta última é criada a partir de uma análise da página HTML onde o Motor Cliente está inserido (e que realiza as requisições para o Motor Servidor). Esta árvore existe e pode ser alterada por ambos os lados da aplicação.

O Motor Cliente possui um gerenciador de eventos, que pode, em função do tipo de evento, chamar uma classe tratadora da aplicação que tenha sido convertida pelo compilador GWT ou, então, comunicar ao *Servlet* tratador de eventos do Motor Servidor. Este último chama uma classe tratadora da aplicação e retorna uma resposta ao Motor Cliente, que, eventualmente, atualiza o estado da tela no cliente.

3.2.2.1 Árvore de Componentes

O Crux fornece uma estrutura de componentes de tela que, de forma semelhante à do JSF e do GWT, pode ser vista como uma árvore. O termo “Tela” é utilizado para referenciar esta árvore de componentes.

A Tela é composta por componentes e containeres. Os containeres são componentes que podem conter outros componentes. Este modelo também segue o padrão de projetos *Composite View* (Gamma et al, 2002). A Figura 3.4 ilustra a estrutura de uma Tela.

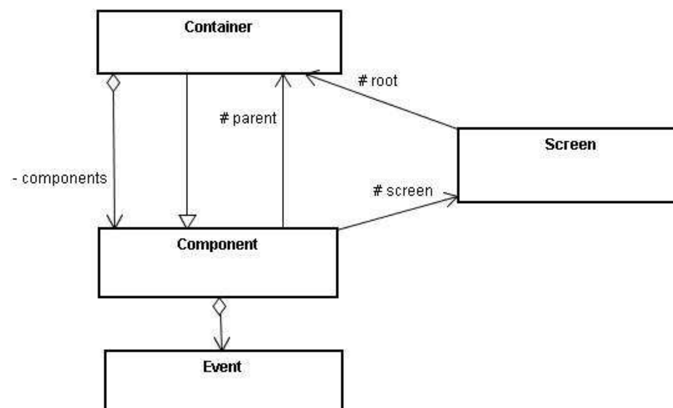

Figura 3-4 – Estrutura de uma Tela no Crux.

Componente

O componente é uma figura central na arquitetura do Crux. Cada componente representa um elemento de tela. Além da classe utilizada para representá-lo em memória (mostrada na Figura 3.4), é associada, ao componente, uma classe para montá-lo a partir de sua configuração na página.

Os componentes possuem uma série de atributos em comum, de forma semelhante a elementos HTML convencionais (ex. atributos para informar valor ou para associá-lo a uma classe de uma folha de estilo). Cada componente pode, também, conter atributos específicos para informar algum aspecto particular (ex. um componente de tabela pode conter um atributo para informar se as suas linhas devem estar ordenadas com base em alguma coluna).

Além dos atributos, eventos podem ser associados aos componentes. O funcionamento do mecanismo de disparo de eventos e como estes são tratados são explicados nas seções 3.2.2.2 e 3.2.2.3, onde são mostrados os Motores Cliente e Servidor do Crux.

Na Listagem 3.1 pode-se perceber a utilização de alguns atributos e eventos nos componentes adicionados.

Eventos e Componentes

O Crux possui um modelo de eventos que suporta, nativamente, três tipos de evento. Este modelo pode, entretanto, ser estendido. As seções 3.2.2.2 e 3.2.2.3 mostram como estendê-lo.

Os tipos de evento inicialmente suportados são:

1. **Client**: Este é um evento processado no cliente. O Motor Cliente chama o método da classe registrada como tratadora de eventos cliente, que é previamente traduzida para *Javascript* pelo compilador GWT.
2. **Server-RPC**: Este evento é processado segundo o modelo do AJAX (seção 2.2.4.9). Neste tipo de evento, é feita uma chamada ao Motor Servidor, que chama um método de um tratador de eventos. O Motor Servidor transmite o retorno deste método para o Motor Cliente, que o repassa para um método de uma classe no cliente, registrada como classe de resposta para o evento (*callback*).
3. **Server-Auto**: Este evento é similar ao evento Server-RPC. No entanto, o método no servidor não retorna nenhum valor e pode acessar toda a árvore de componentes da tela. Ao término do evento o Motor Servidor envia para o Motor Cliente todas as alterações feitas na árvore e este atualiza a Tela para refletir as alterações.

O evento Server-RPC permite que requisições curtas que não precisam manipular a árvore de componentes no servidor sejam feitas. Ele é mais rápido, pois não precisa sincronizar o estado da árvore com o servidor. Por outro lado, o evento Server-Auto é mais flexível, permitindo ao programador ter acesso a toda estrutura de componentes e cuidando, automaticamente, da sincronização do estado da árvore.

Os eventos podem ser disparados de forma síncrona ou assíncrona, definindo, assim, se a interface fica ou não bloqueada durante o seu processamento. O apêndice B.1 mostra um exemplo de declaração de cada um dos tipos de evento nativos do Crux.

Adicionando Componentes ao Crux

O Crux possui um *Listener*[19] que executa uma série de ações logo que a aplicação é iniciada. Entre estas ações, é feita uma busca, dentro do contexto da aplicação (em arquivos “jar”[20], inclusive) por arquivos com o nome “crux.xml”. Este arquivo é utilizado para registrar componentes no Crux. A Listagem 3.4 mostra um exemplo de um arquivo crux.xml.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <crux xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3        xsi:noNamespaceSchemaLocation="../../Crux/xsd/crux.xsd">
4    <component id="button"
5              clientClass="br.ufmg.thiago.crux.core.client.component.ComponentBase"
6              clientConstructorParams="new com.google.gwt.user.client.ui.Button()"
7              serverClass="br.ufmg.thiago.crux.core.server.screen.Component" />
8    <component id="textBox"
9              clientClass="br.ufmg.thiago.crux.core.client.component.ComponentBase"
10             clientConstructorParams="new com.google.gwt.user.client.ui.TextBox()"
11             serverClass="br.ufmg.thiago.crux.core.server.screen.Component" />
12 </crux>
```

Listagem 3-4 – Exemplo de Registro de Componentes no Crux.

Este arquivo, no entanto, não precisa ser conhecido pelo programador da aplicação, mas apenas por desenvolvedores de componentes do Crux. Através do mecanismo descrito acima, um desenvolvedor de componentes pode criar um conjunto de componentes, agrupá-los em um arquivo “jar” e distribuí-lo, de modo que o programador da aplicação só precisa copiar este arquivo para dentro do contexto da aplicação e o Crux registra estes componentes automaticamente.

Foi criado, também, um componente que pode ser associado a um *Widget* do GWT (vide seção 2.3.3.3), de modo que este atua como um *Adapter* (Gamma et al, 2002) para inserir componentes deste arcabouço no Crux. Na Listagem 3.4 vemos alguns componentes do GWT sendo registrados.

Montagem da Árvore de Componentes

A árvore de componentes (ou Tela) é montada na memória tanto pelo Motor Cliente quanto pelo Motor Servidor do Crux e pode ser acessada por tratadores de eventos dos dois tipos.

---

[19] Classe que implementa a interface *ServletContextListener*, da API JEE. Um ServletContextListener pode ser registrado para responder a eventos de inicio e término da aplicação, dentro do servidor de aplicações.
[20] Um arquivo “jar” é um arquivo compactado que contém um conjunto de classes ou outros recursos. Estes são agrupados para facilitar a distribuição dos mesmos.

Cada tela no Crux é descrita por uma página HTML, contendo marcações <span> com atributos "*_type*" para referenciar componentes, conforme pode ser visto na Listagem 3.1. Para montagem de cada tela, é feita uma análise de sua página HTML.

Existe uma classe de análise localizada no Motor Servidor e outra no Motor Cliente. Estas classes são detalhadas nas seções 3.2.2.2 e 3.2.2.3, respectivamente. Cada uma dessas classes de análise, uma no processamento servidor e outra no cliente, percorre a página HTML em busca das marcações <span> contendo o atributo "*_type*". Caso encontre uma marcação nesse formato, recupera o componente associado ao tipo fornecido.

É solicitado, então, que o próprio componente recuperado, através de uma classe de análise associada a ele, realize a análise do elemento do DOM da página associado à marcação identificada. Isto permite que cada componente veja quais atributos ou mesmo sub-elementos estão contidos na marcação <span> e os interprete da maneira mais conveniente. As seções 4.1.2 e 4.2.2 detalham as classes envolvidas neste processo.

A partir das informações contidas nesta marcação, o componente pode ser montado em memória e desenhado na tela do cliente, pelos Motores Servidor (vide seção 3.2.2.2) e Cliente (vide seção 3.2.2.3).

Cabe, também, explicar porque os componentes são representados pela marcação <span> e não por outra qualquer, como <div>, ou mesmo por marcações fictícias, como <textbox>, <grid>, ou nomes que expressariam mais adequadamente o componente, sem a necessidade de se utilizar o atributo "*_type*".

Um primeiro ponto é que os analisadores das páginas contidos nos navegadores ignoram qualquer marcação que não pertença ao conjunto de marcações do HTML (que, como visto na seção 2.2.3.2, no item 7, não é expansível), de modo que eles não poderiam ser acessados via DOM. Um outro ponto é que a marcação <span>, ao contrário de marcações como <div>, representa um container apenas lógico, sem qualquer influência no desenho da página pelo navegador. O uso de outra marcação, como o <div>, poderia alterar a forma como o restante do HTML seria desenhado pelo navegador.

A explicação para se utilizar o caractere "_" antes dos nomes dos atributos também está relacionada à forma como o analisador de HTML dos navegadores interpretaria os atributos. Caso encontrasse um atributo com o nome "onclick", por exemplo, ele tentaria associar um tratador de eventos para responder ao evento de cliques de mouse no elemento (span) que deveria apenas informar a descrição do componente e marcar a sua posição na página, mas não ser ele próprio o elemento final desenhado pelo componente.

A exceção a esta regra de nomenclatura dos atributos é o atributo "*id*". Este não possui o caractere "_", pois ele é utilizado para identificar o componente junto ao DOM.

3.2.2.2 Motor Servidor

O Motor Servidor é responsável por controlar toda a lógica da aplicação que deve ser executada no servidor Web. É ele quem carrega todas as configurações do arcabouço, registra os componentes que podem ser utilizados, assim como os tratadores de eventos, tanto os que executam no lado cliente como no lado servidor. É ele, também, quem recebe as requisições feitas pelo Motor Cliente e delega o controle para os tratadores de evento, cuidando, sempre que necessário, de manter sincronizada a árvore de componentes com o Motor Cliente ao término dos eventos.

Início da Aplicação

Conforme mencionado na seção anterior, o Motor Servidor possui um *Listener* que escuta o evento de criação do contexto da aplicação. Desta forma, esta classe executa, no momento em que o servidor de aplicações cria o contexto para a aplicação, as seguintes tarefas:

1. Carrega as configurações do arcabouço.
2. Registra os componentes que estarão disponíveis para construção das telas.
3. Inicia a classe de fábrica para os tratadores de evento do cliente e do servidor (que seguem o padrão *Factory*) (Gamma et al, 2002).

As configurações do Crux são carregadas a partir de um arquivo chamado "Crux.properties". Este permite personalizar o comportamento do arcabouço em vários aspectos. A partir deste arquivo, é possível, por exemplo, modificar a classe responsável pela construção dos objetos tratadores de eventos, ou alterar a forma como a pesquisa por novos componentes é feita no contexto da aplicação. A Listagem 3.5 mostra um exemplo deste arquivo.

```
1  controllerFactory=br.ufmg.thiago.crux.core.server.lifecycle.phase.dispatch.ControllerFactoryImpl
2  debug=false
3  initializeControllersAtStartup=true
4  lookupWebInfOnly=false
5  enableHotDeployForScreens=true
6  pagesHome=/
7  developmentPublicDir=public
```

Listagem 3-5 – Exemplo de Arquivo Crux.properties.

Dentro do arquivo de distribuição do Crux (crux-core.jar), existe um arquivo Crux.properties, contendo os valores padrão para todas as opções de configuração do arcabouço. Caso o programador de uma aplicação deseje modificar qualquer uma dessas opções, basta adicionar um arquivo Crux.properties em algum diretório que esteja no contexto de sua aplicação que o *Listener* do Crux o carregará e sobrescreverá qualquer opção que esteja declarada em seu arquivo.

A segunda tarefa realizada pelo Motor Servidor é o registro dos componentes. Na seção 3.2.2.1, é mostrado como os componentes são registrados junto ao Crux. Este processo é implementado pela classe *ComponentConfig*, que realiza a busca por arquivos de registro de componentes (crux.xml) e os processa, registrando os componentes declarados.

A terceira tarefa realizada pelo Motor Servidor, iniciar as classes de fábrica dos controladores, é implementada pela classe *ControllerFactoryInitializer*. Esta classe instancia uma classe de criação (*Factory*) para os objetos tratadores de eventos no lado servidor da aplicação. O Crux fornece uma implementação padrão para esta classe de criação que registra todas as classes que possuam a anotação *Controller*. Este esquema padrão, comentado na seção 3.2.1.2, permite que classes como a mostrada na Listagem 3.2 sejam registradas como controladores.

O Crux permite, no entanto, que a classe de criação seja substituída, a fim de possibilitar que outros arcabouços possam ser utilizados na camada servidora da aplicação. Poder-se-ia, por exemplo, substituir a classe que cria os controladores por uma que buscasse por eles no arcabouço *Spring* (Spring, 2008), onde poderiam ser configurados, usufruindo do controle de transações declarativo ou das injeções de aspectos (AOP) proporcionados por este arcabouço.

Para alterar a classe de fábrica criada pelo *ControllerFactoryInitializer*, basta alterar o valor da propriedade "*controllerFactory*" no arquivo Crux.properties (veja na Listagem 3.5).

A Figura 3.5 mostra as classes envolvidas no esquema de carga do arcabouço, responsáveis por implementar as operações citadas. A seção 4.1.1 detalha o funcionamento destas classes.

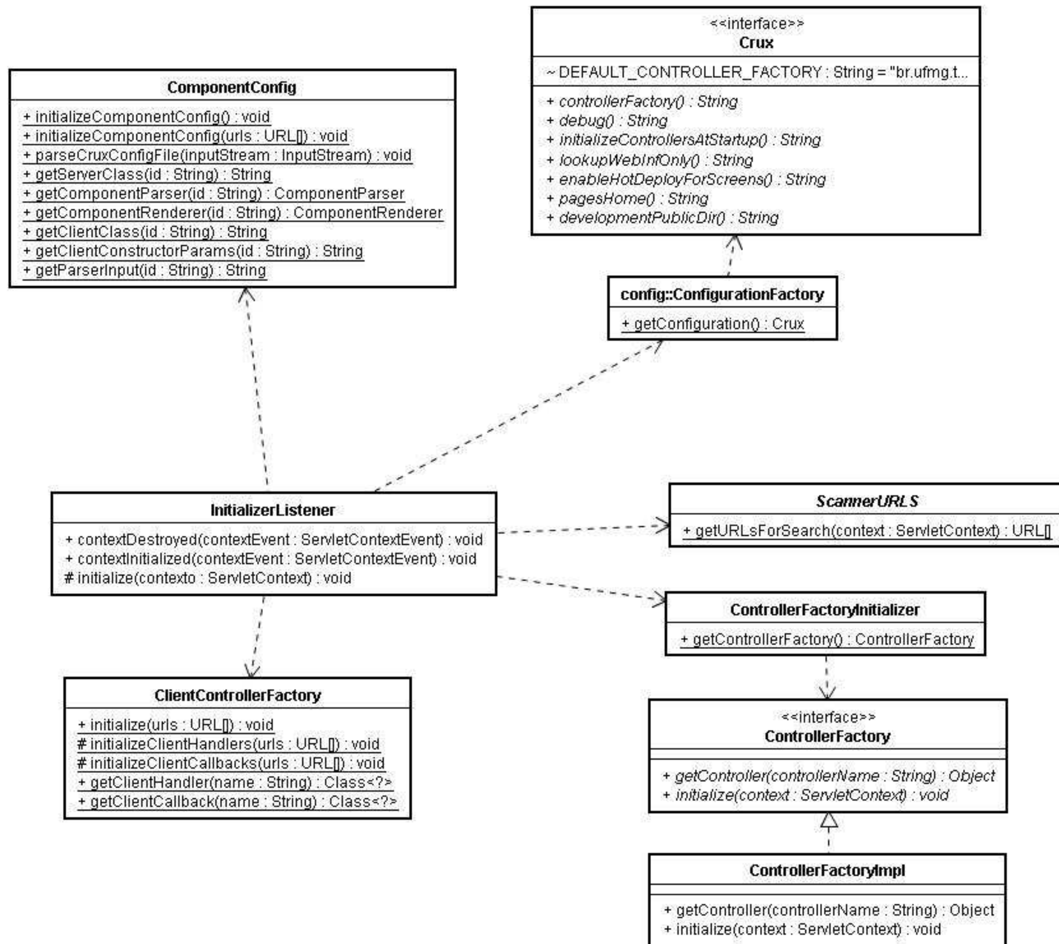

Figura 3-5 – Carregando uma Aplicação com o Crux.

De forma análoga à feita pela classe de implementação padrão da interface *ControllerFactory,* a classe *ClientControllerFactory* registra os tratadores de evento do lado cliente da aplicação. Esta classe registra como tratadores de evento classes que possuam a anotação *ClientController*, como a mostrada na Listagem 3.3. São registradas, também, classes utilizadas para responder ao resultado de algum processamento no servidor (*callback*), através da anotação *ClientCallback*.

A classe *ScannerURLS* retorna as URLs para os recursos que serão percorridos pelas demais classes de Registro (componentes e controladores). Com base na propriedade "*lookupWebInfOnly"*, presente no arquivo Crux.properties, esta classe decide se deve

pesquisar em todos os arquivos no caminho do processo Java ou apenas dentro da pasta WEB-INF[21] da aplicação.

Tratamento de Eventos

No Crux, os eventos que devem ser processados no lado servidor são tratados por *Servlets* que devem herdar a classe *BaseServlet*. A Figura 3.6 mostra as classes relacionadas ao processamento de um evento servidor.

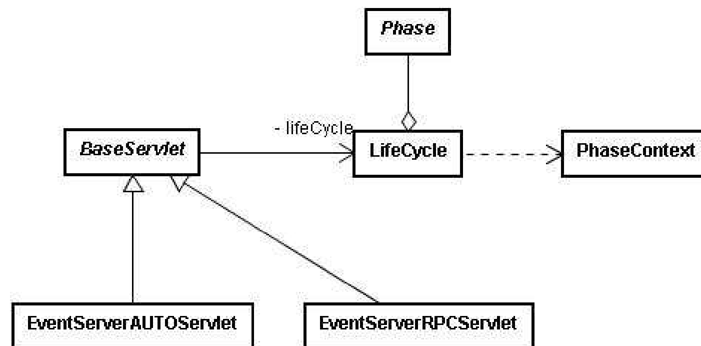

Figura 3-6 – Tratamento de Eventos no Motor Servidor.

Cada um destes *Servlets* é responsável por tratar um tipo de evento diferente. Conforme é mostrado na seção 3.2.2.3, o Crux possui um mecanismo de eventos extensível. Existem nativamente três tipos de evento suportados e novos podem ser criados, bastando, pelo lado do servidor, que se adicione uma nova subclasse de *BaseServlet*.

A classe *BaseServlet* delega para o objeto do tipo *LifeCycle* a responsabilidade de tratar uma requisição recebida pelo servidor. Esta última classe é a responsável por gerenciar o ciclo de vida das requisições no Crux.

O ciclo de vida de uma requisição é composto por Fases (classe *Phase*). Cada Fase representa um passo do processamento completo. Elas podem ser compostas de forma a montar uma cadeia de processamento, seguindo o padrão *Chain of Responsability* (Gamma et al, 2002). As Fases possuem acesso a um objeto de contexto que pode ser utilizado para intercâmbio de informações a respeito da requisição.

Existem três fases principais definidas no ciclo de vida de uma requisição no Crux:

---

[21] Toda aplicação Web, para executar dentro de um container Java, deve seguir uma estrutura padrão de diretórios. É obrigatório possuir um diretório WEB-INF, que possui um subdiretório chamado lib, onde ficam contidas todas as bibliotecas utilizadas pela aplicação, e outro chamado classes, onde ficam os arquivos compilados da aplicação (.class).

1. **ParametersBindPhase**: Nesta fase é feita a associação dos parâmetros passados na requisição com os objetos que representarão estes valores durante o processamento. Nesta fase, caso seja necessário, é recuperada a árvore de componentes, que é atualizada com alguma modificação feita do lado cliente.
2. **DispatchPhase**: Nesta fase é feito o disparo do método tratador da aplicação. Neste ponto o arcabouço delega o controle do fluxo de execução para a aplicação (IoC, vide seção 2.1.1.3).
3. **RenderResponsePhase**: Nesta fase a resposta à requisição é criada e enviada ao cliente. Nesta fase pode ser enviada a resposta de um determinado método, ou a lista de alterações sofridas pela árvore de componentes, para que o Motor Cliente faça as atualizações.

Cada *Servlet* de tratamento de eventos pode personalizar o seu próprio ciclo de vida. Isso é possível porque a classe *LifeCycle* monta a cadeia de Fases, que irão compor seu ciclo de vida, com base em anotações que são adicionadas ao *Servlet*.

É possível informar qual classe de implementação deverá ser usada em cada uma das fases mostradas, assim como adicionar novos passos a este ciclo. O apêndice B.2 ilustra como isso pode ser feito.

Componentes e Dados de Tela

Os componentes de tela, frequentemente, irão exibir alguma informação para o usuário ou receber deste algum tipo de entrada. O Crux permite que estas informações sejam armazenadas em um objeto java simples, que siga a notação de *javabean*. Esta estratégia segue o padrão *Data Transfer* (Deepak et al., 2001) e é semelhante à forma como o JSF passa valores passa o seu *Managed Bean* (vide seção 2.3.3.2)

Para informar a um componente que o mesmo deverá exibir informações contidas em um objeto java associado ao controlador, ou enviar informação para popular alguma propriedade desse objeto, basta informar isto através do atributo "_serverBind", utilizado na página HTML, junto à declaração do componente.

Para ilustrar este processo, examine a Listagem 3.6.

```
1  <html>
2      <head>
3          <script language='javascript'
4              src='br.ufmg.thiago.teste.CruxTeste.nochache.js'</script>
5      </head>
6      <body>
7          <table class='FormTable'>
8              <tr>
9                  <td align='left' valign='middle'>
10                     <span id='panel' _type='scrollPanel'
11                                      _style='width:100%' >
12                         <span id='panel.testeButton1' _type='button'
13                           _value='Teste1'
14                           _onclick='serverHandler.metodoTeste'>
15                         </span>
16                         <span id='panel.testeTextbox' _type='textBox'
17                           _value='Valor Teste'
18                           _serverBind='pessoa.nome'
19                           _onchange='serverHandler.metodoTeste|server-rpc|clientCallback.metodoTeste|synchronous'>
20                         </span>
21                     </span>
22                 </td>
23             </tr>
24         </table>
25     </body>
26 </html>
```

Listagem 3-6 – Exemplo de Transferência de Dados Entre Componente e Servidor.

Repare que, no componente "panel.testeTextbox", foi adicionado o atributo "*_serverBind*". No exemplo mostrado, este atributo está informando que o valor deste componente deve ser associado à propriedade "nome" de um objeto pessoa. Caso exista uma propriedade chamada "pessoa" no tratador deste evento e esta possua uma propriedade chamada "nome", esta será preenchida com o valor contido no componente "panel.testeTextbox". A Listagem 3.7 mostra um exemplo de tratador que receberia esta informação.

```
1  @Controller("serverHandler")
2  public class TestController
3  {
4      private Pessoa pessoa = new Pessoa();
5
6      public Pessoa getPessoa() {
7          return pessoa;
8      }
9
10     public void setPessoa(Pessoa pessoa) {
11         this.pessoa = pessoa;
12     }
13
14     public String metodoTeste()
15     {
16         return "O Nome da Pessoa é:"+pessoa.getNome();
17     }
18 }
```

Listagem 3-7 – Exemplo de Tratador de Evento Server-RPC no Crux.

Comparando o mecanismo apresentado com a forma como o JSF transfere dados para o seu controlador, percebe-se uma grande semelhança. No entanto, no Crux eliminou-se a necessidade de declarar cada controlador (ou *managed bean*, para o JSF), em um arquivo XML, onde cada propriedade do controlador também precisa ser informada (vide Listagem 2.11).

Além de retorno de métodos processados, ou lista de modificações na árvore de componentes, o Motor Servidor envia também os valores de propriedades do tratador que tiverem sido alteradas durante o processamento e que estejam ligadas, através da propriedade "*_serverBind*", a algum componente da tela. Estes valores são usados, pelo Motor Cliente, para atualizar os componentes na tela.

Eventos Server-RPC

Para o processamento dos eventos do tipo Server-RPC, o Crux utiliza um *Servlet* tratador de eventos cuja fase de disparo realiza as seguintes ações:

1. O objeto tratador de eventos (controlador) é criado pela classe de fábrica configurada no arquivo Crux.properties, conforme mostrado na seção "Início da Aplicação".
2. Os valores recebidos na fase de associação dos parâmetros são copiados para o controlador, desde que este possua um método de escrita para a propriedade, segundo a nomenclatura dos *JavaBeans*.
3. O método de tratamento do evento do controlador é chamado e o seu retorno é armazenado no contexto associado ao ciclo de vida.

A Listagem 3.6 mostra uma declaração de evento do tipo Server-RPC, disparado quando o componente "panel.testeTextbox" é modificado e a Listagem 3.7 mostra um tratador para este evento.

Depois do processamento do método, a fase de escrita da resposta é chamada. Esta pega o valor de retorno do método (armazenado no contexto do ciclo de vida) e o envia para o Motor Cliente, juntamente com as propriedades ligadas a componentes que tiverem sido alteradas. O Motor cliente, então chama o método de tratamento da resposta informado no evento. A Listagem 3.8 mostra um exemplo de como poderia ser este método.

```
1  @ClientCallback("clientCallback")
2  public class TesteCallback
3  {
4      String idSender;
5
6      public String getIdSender(){
7          return idSender;
8      }
9
10     public void setIdSender(String idSender){
11         this.idSender = idSender;
12     }
13
14     public void metodoTeste(JSONValue result){
15         Window.alert("Resultado recebido: " + result.toString());
16     }
17 }
```

Listagem 3-8 – Exemplo de Método Tratador de Resposta de um Evento Server-RPC.

Todas essas informações são enviadas no formato JSON (json, 2008), para facilitar o seu tratamento pelo método tratador de resposta, uma vez que este formato é bastante simples e amplamente usado.

Eventos Server-Auto

Para o processamento de um evento do tipo Server-Auto, o processo é bastante semelhante. A diferença principal é que não existe método de tratamento de resposta e que caso o método chamado no controlador retorne algum valor, este é ignorado.

Outra diferença importante é que a árvore de componentes é enviada. Na verdade, não é enviada, necessariamente, inteira, mas tudo o que for necessário para remontá-la no servidor.

A Fase de escrita da resposta em um evento Server-Auto também é um pouco diferente. Ela envia como resposta, além das propriedades ligadas a componentes que tiverem sido alteradas, a lista de modificações que foram feitas na árvore de componentes. O Motor Cliente as recebe e atualiza a tela do cliente.

A Listagem 3.9 ilustra um exemplo de chamada de evento Server-Auto.

```
1  <html>
2      <head>
3          <script language='javascript'
4              src='br.ufmg.thiago.teste.CruxTeste.nochache.js'</script>
5      </head>
6      <body>
7          <table class='FormTable'>
8              <tr>
9                  <td align='left' valign='middle'>
10                     <span id='nome'
11                           _type='textBox'
12                           _serverBind='pessoa.nome'
13                           _width='670px'></span>
14                     <span id='testeButton'
15                           _type='button'
16                           _value='Teste AUTO'
17                           _width='135px'
18                           _onclick='serverHandler.metodoTeste'></span>
19                 </td>
20             </tr>
21         </table>
22     </body>
23 </html>
```

Listagem 3-9 – Exemplo de Chamada de Evento Server-Auto no Crux.

O exemplo acima é muito semelhante ao mostrado na Listagem 3.6. A principal diferença é que não existe nenhum tratador de resposta. A Listagem 3.10 mostra um exemplo de tratador para o evento.

```
1  @Controller("serverHandler")
2  public class TestController
3  {
4      private Screen screen;
5      private Pessoa pessoa = new Pessoa();
6
7      public Pessoa getPessoa(){
8          return pessoa;
9      }
10
11     public void setPessoa(Pessoa pessoa){
12         this.pessoa = pessoa;
13     }
14
15     public Screen getScreen() {
16         return screen;
17     }
18
19     public void setScreen(Screen screen) {
20         this.screen = screen;
21     }
22
23     public void metodoTeste()
24     {
25         screen.getComponent("panel.testeTextbox").setVisible(false);
26         this.pessoa.setNome("Nome Alterado");
27     }
28 }
```

Listagem 3-10 – Exemplo de Tratador de Evento Server-Auto no Crux.

Para informar as mudanças ocorridas na árvore de componentes, é utilizada, em cada componente, uma variável de controle chamada *dirty*. Se alguma informação é alterada, o valor desta variável é modificado. Desta forma, a fase de escrita da resposta de um evento Server-Auto, percorre toda a árvore verificando quais componentes foram modificados.

Cada componente possui associado a si uma classe de escrita. Esta classe é responsável por escrever as modificações, de modo que o Motor Cliente saiba redesenhá-lo na tela.

No arquivo crux.xml (vide Listagem 3.4), pode-se associar uma classe de escrita ao componente (caso não se informe, será utilizada uma classe padrão do Crux). Esta associação é feita pela propriedade *ServerRenderClass*. O Apêndice B.3 mostra um exemplo desta configuração e o Apêndice B.4 mostra a resposta escrita pelo Crux para o evento, no exemplo apresentado.

O Motor Cliente recebe esta resposta e atualiza os componentes na tela.

Criação da Árvore de Componentes

A seção 3.2.2.1 apresenta a árvore de componentes no Crux e como ela é definida pelo programador da aplicação, através de uma página HTML contendo marcações <span> com atributos especiais. Nesta seção é descrito o processo de criação desta árvore pelo Motor Servidor do Crux.

A classe utilizada para construção da árvore de componentes no servidor é chamada *ScreenFactory*. Esta classe analisa a página HTML que descreve a tela, procurando marcações <span> com atributos *_type*. Para analisar a página, é utilizada a biblioteca *Jericho* (jericho, 2008). Esta biblioteca analisa uma página HTML de forma similar a um DOM, considerando, no entanto, o fato de o HTML não ser um XML bem formado.

Sempre que a *ScreenFactory* encontra uma marcação <span>, ela delega para uma classe de análise a criação do componente. Cada componente possui associado a si uma classe de análise. Assim como a classe de escrita (vide seção anterior), a classe de análise é configurada no arquivo crux.xml (caso não seja informada, uma classe padrão é utilizada). Esta associação é feita pela propriedade *ServerParserClass*. Veja o Apêndice B.5 para um exemplo de associação de classe de análise.

A classe de análise do componente recebe, então, o elemento representando a sua marcação <span>, de modo que ele mesmo possa interpretar o seu conteúdo. A forma

como esse elemento é passado para a classe de análise pode ser configurada no arquivo crux.xml. Pode-se representar a marcação como um elemento da biblioteca *Jericho*, como um elemento DOM ou como string. Esta escolha pode ser informada pelo atributo *parserInput* (vide Apêndice B.5).

A *ScreenFactory* realiza a análise de cada tela solicitada apenas uma vez e a armazena em memória, para melhoria de desempenho. Esta tela armazenada é, então, "clonada" e devolvida para o solicitante. Caso a opção *enableHotDeployForScreens*, presente no arquivo Crux.porperties, possua o valor *true*, a *ScreenFactory* verifica a data da última alteração no arquivo pare decidir se re-analisa a página ou não. Esta configuração é interessante, pois acelera o desenvolvimento. Sem ela, o programador precisaria reiniciar o servidor para enxergar cada modificação na tela.

Esta implementação da classe *ScreenFactory* segue os padrões *Factory*, *Builder* e *Prototype* (Gamma et al, 2002).

A classe *ScreenFactory* monta a árvore de componentes de acordo com o seu estado inicial, configurado na página HTML. No entanto, uma tela pode sofrer modificações entre requisições processadas. Componentes podem se tornar invisíveis, mudarem suas dimensões ou outros atributos na tela. Desta forma, é necessário que exista um mecanismo para persistir e recuperar o estado da árvore de componentes entre as requisições.

De forma semelhante ao mecanismo usado para seleção da classe de criação dos tratadores de evento no servidor, a classe que gerencia o estado das árvores de componentes pode ser configurada no arquivo Crux.properties. O Apêndice B.6 mostra como selecionar qual classe de gerência de estado utilizar.

Classes de gerência de estado devem implementar a interface *ScreenStateManager*. O Crux fornece duas implementações para esta interface: uma chamada *ScreenStateManagerClientImpl e ScreenStateManagerServerImpl.* A primeira delas, utilizada por padrão, informa ao Crux que as alterações na árvore de componentes devem ser resubmetidas a cada requisição. A segunda classe armazena o estado da árvore na sessão do usuário. A gerência de estado no Crux segue, desta forma, o padrão *Strategy* (Gamma et al, 2002).

Quando se utiliza a primeira abordagem, a classe de gerência de estado, após solicitar à *ScreenFactory* o objeto representando a árvore, aplica os parâmetros relacionados ao estado da mesma para reconstruí-la corretamente.

Quando se utiliza a segunda abordagem, a classe de gerência de estado solicita à *ScreenFactory* a árvore de componentes apenas se ela não existir na sessão do usuário.

Além disso, o cliente não necessita submeter sempre todas as mudanças feitas na mesma, mas apenas as que ele ainda não tiver submetido nenhuma vez.

Armazenar o estado na sessão do usuário é mais eficiente do que no cliente. No entanto, a aplicação pode apresentar mais dificuldades para ser escalada entre mais de um servidor de aplicações.

A seção 3.3.5 discute o desempenho da aplicação em cada uma das situações abordadas nesta seção.

Internacionalização

O GWT (vide seção 2.3.3.3) disponibiliza um mecanismo de internacionalização inteligente e simples de utilizar. Como o Crux utiliza o GWT internamente para criação de código para execução no cliente, é possível utilizar o mecanismo do GWT para recuperar mensagens nas classes usadas no cliente.

Para o servidor, o Crux implementa este mesmo mecanismo, de modo que as mensagens internacionalizadas possam ser recuperadas da mesma forma no cliente e no servidor.

A classe *MessagesFactory* cria as mensagens da mesma forma como é feito no GWT, a partir de uma interface onde os métodos declarados são usados como chave em um arquivo de propriedades.

A Listagem 3.19 mostra um exemplo de criação de mensagem no servidor.

```
1 ServerMessages messages = (ServerMessages)MessagesFactory.getMessages(ServerMessages.class);
2 if (logger.isInfoEnabled()) logger.info(messages.screenStateManagerInitializerUsingDefaultFactory());
```

Listagem 3-11 – Exemplo de Uso de Mensagens Internacionalizadas.

A interface *ServerMessages* é mostrada na Listagem 3.20

```
1 public interface ServerMessages
2 {
3     String screenStateManagerErrorCloningScreen(String screenId, String errMsg);
4     String screenStateManagerInitializerUsingDefaultFactory();
5 }
```

Listagem 3-12 – Exemplo de Interface Usada para Internacionalização.

O arquivo de propriedades com as mensagens é mostrado na Listagem 3.21

```
1 screenStateManagerErrorCloningScreen=[screenStateManager - 001] - Error retrieving screen {0} from its template: {1}.
2 screenStateManagerInitializerUsingDefaultFactory=[request 011] - Using default screen state manager.
```

Listagem 3-13 – Exemplo de Arquivo de Propriedades Usado para Internacionalização.

A classe *MessagesFactory* cria um objeto que implementa a interface passada como parâmetro. Este objeto busca por um arquivo de propriedades que possua o mesmo nome da interface. Este arquivo é utilizado para retornar as mensagens, de acordo com o nome de cada método da interface.

Para criação do objeto de implementação da interface, é utilizada uma *proxy* dinâmica da API do Java. Detalhes desta implementação são mostrados na seção 4.1.4.

3.2.2.3 Motor Cliente

O Motor Cliente é responsável por controlar toda a lógica da aplicação que deve ser executada no navegador do usuário. É ele quem desenha a tela, a partir das configurações dos componentes na página. É ele, também, que gerencia todos os disparos de eventos, delegando o controle para tratadores da aplicação ou notificando o Motor Servidor.

O Motor Cliente foi implementado como um módulo do GWT. A classe *JSEngine* é o ponto de entrada (*EntryPoint*) deste módulo, sendo chamada quando a página é carregada no navegador do usuário.

Registro das Classes Utilizadas

Conforme pode ser visto na seção 2.3.3.3, o GWT, por motivos de desempenho, não suporta reflexão e oferece o mecanismo de “amarração tardia” para suprir a falta deste recurso.

Através deste mecanismo, geradores podem ser associados para criação de objetos de determinado tipo. Estes geradores são executados no momento da compilação do código para *Javascript* e produzem código java para as classes geradas, que depois também são traduzidas pelo compilador. Os objetos são criados, então, segundo os tipos gerados.

Vale ressaltar, também, que estes geradores são chamados pelo compilador GWT, sendo, portanto, executados no servidor (no momento da compilação para *Javascript*) e não no cliente.

Este mecanismo é utilizado para criar algumas classes no Motor Cliente. Em primeiro lugar é preciso criar uma classe que contenha os componentes que são utilizados na tela, com os seus respectivos nomes de registro. Isto é necessário para evitar que todas as telas tenham que possuir referência para todos os tipos de componentes registrados no Crux, o que faria com que o compilador GWT precisasse gerar código para todos estes componentes, aumentando significativamente o tamanho do arquivo *Javascript* gerado.

Para resolver este problema foi definida uma interface chamada *RegisteredComponents*. Esta possui um método que é chamado quando se necessita construir um componente. Um gerador chamado *RegisteredComponentsGenerator* constrói uma classe, dinamicamente, que implementa esta interface, de modo que ela só saiba construir objetos dos tipos utilizados na tela que está sendo compilada. A Figura 3.7 mostra estas classes.

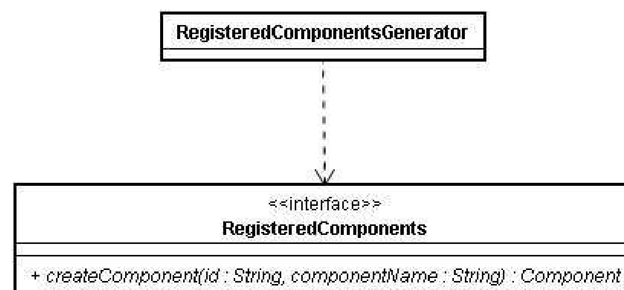

Figura 3-7 – Classes Envolvidas no Registro de Componentes no Cliente.

Para construir esta classe, o *RegisteredComponentsGenerator* recupera uma instância da tela que está sendo compilada (pedindo para classe *ScreenFactory*). Ela, então, percorre a tela e verifica os tipos dos componentes que estão contidos nela. Para cada componente encontrado, ela verifica na lista carregada pela classe *ComponentConfig* (vide seção 3.2.2.2) as informações do componente.

Cada componente registrado no Crux deve informar, também, qual a classe que o representa no cliente, assim como eventuais parâmetros necessários para sua construção. Estes parâmetros são informados no arquivo crux.xml. O Apêndice B.5 mostra um exemplo desta configuração e a Listagem 3.14 mostra um exemplo de classe gerada pelo gerador *RegisteredComponentsGenerator*.

```java
public class RegisteredComponentsImpl implements RegisteredComponents
{
    public Component createComponent(String id, String componentName) throws InterfaceConfigException
    {
        if ("textBox".equals(componentName)){
            return new br.ufmg.thiago.crux.core.client.component.ComponentBase(
                                        id, new com.google.gwt.user.client.ui.TextBox());
        } else if ("button".equals(componentName)){
            return new br.ufmg.thiago.crux.core.client.component.ComponentBase(
                                        id, new com.google.gwt.user.client.ui.Button());
        }
        throw new InterfaceConfigException("Component not found:"+componentName);
    }
}
```

Listagem 3-14 – Classe Gerada para Registro dos Componentes

Outra classe que necessita ser gerada dinamicamente é uma que contenha os tratadores de evento registrados para uso nesta página. Seguindo processo similar ao descrito para a geração da classe contendo os componentes registrados, a classe *RegisteredClientEventHandlerGenerator* gera uma classe que implemente a interface *RegisteredClientEventHandlers*, que sabe construir objetos para invocação de todos os eventos que podem ser chamados pela página corrente. A Figura 3.8 mostra estas classes.

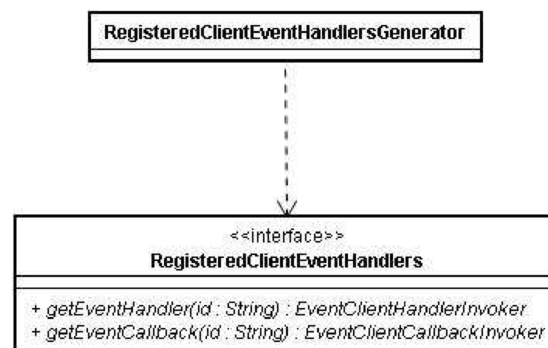

Figura 3-8 – Classes Envolvidas no Registro de Tratadores de Evento no Cliente.

Para construção desta classe, a *RegisteredClientEventHandlerGenerator* também acessa a tela (a partir da *ScreenFactory*) e consulta os tratadores a partir da classe *ClientControllerFactory*, carregada no início da aplicação (vide seção 3.2.2.2). A Listagem 3.15 mostra um exemplo de classe gerada pelo gerador *RegisteredClientEventHandlerGenerator.*

```java
public class RegisteredClientEventHandlersImpl implements RegisteredClientEventHandlers
{
    private java.util.Map clientHandlers = new java.util.HashMap();
    private java.util.Map clientCallbacks = new java.util.HashMap();

    public RegisteredClientEventHandlersImpl(){
        class TesteWrapper extends br.ufmg.thiago.teste.client.Teste
                           implements br.ufmg.thiago.crux.core.client.event.EventClientHandlerInvoker {
            public void invoke(String metodo, br.ufmg.thiago.crux.core.client.component.Screen screen, String idSender)
                            throws Exception {
                TesteWrapper wrapper = new TesteWrapper();
                if ("metodoTeste".equals(metodo)){
                    wrapper.metodoTeste();
                } else if ("metodoTeste2".equals(metodo)){
                    wrapper.metodoTeste2();
                } // else Reporta método não encontrado
            }
        }
        clientHandlers.put("clientHandler", new TesteWrapper());
        class TesteCallbackWrapper extends br.ufmg.thiago.teste.client.TesteCallback
                           implements br.ufmg.thiago.crux.core.client.event.EventClientCallbackInvoker {
            public void invoke(String metodo, br.ufmg.thiago.crux.core.client.component.Screen screen,
                               String idSender, com.google.gwt.json.client.JSONValue result) throws Exception {
                TesteCallbackWrapper wrapper = new TesteCallbackWrapper();
                if ("metodoTeste".equals(metodo)) {
                    wrapper.metodoTeste(result);
                }
                else if ("metodoTeste2".equals(metodo)) {
                    wrapper.metodoTeste2(result);
                } // else Reporta método não encontrado
            }
        }
        clientCallbacks.put("clientCallback", new TesteCallbackWrapper());
    }

    public EventClientHandlerInvoker getEventHandler(String id){
        return (EventClientHandlerInvoker) clientHandlers.get(id);
    }

    public EventClientCallbackInvoker getEventCallback(String id){
         return (EventClientCallbackInvoker) clientCallbacks.get(id);
    }
}
```

Listagem 3-15 – Classe Gerada para Registro dos Tratadores de Evento

O mecanismo mostrado para geração das classes, ajuda a manter o código pequeno (apenas o que é usado é referenciado) e permite que os componentes e tratadores sejam acessados pelo Motor Cliente a partir dos nomes informados na página, o que seria impossível de outra forma, pois não se pode usar reflexão. Pode-se perceber, na Listagem 3.15, que o tipo do objeto de tratamento de eventos retornado é *EventClientHandlerInvoker* e o do tratador de resposta *EventClientCallbackInvoker.* Estas duas interfaces são utilizadas pelo Motor Cliente para chamada dos eventos.

Como resultado do processo explicado, o mecanismo de tratamento de eventos do Motor Cliente, detalhado na próxima seção, pode solicitar um controlador pelo nome e invocar o método desejado também pelo nome. Outra conseqüência deste processo é que apenas o que é utilizado na página é inserido no código *Javascript* final que é enviado para o cliente, o que é importante para eficiência da aplicação.

O Apêndice B.7 mostra como os geradores apresentados nesta seção são associados aos tipos que precisam gerar.

Tratamento de Eventos

O Motor Cliente associa, a cada evento de componente, uma chamada para o seu gerenciador de eventos, que decide como processar o evento ocorrido.

Para processar um evento, o gerenciador de eventos cria um objeto que implementa a interface *EventProcessor*. Um objeto que implementa esta interface deve possuir o método *processEvent*, que cuida de processar o evento. A Figura 3.9 mostra as classes envolvidas neste processo.

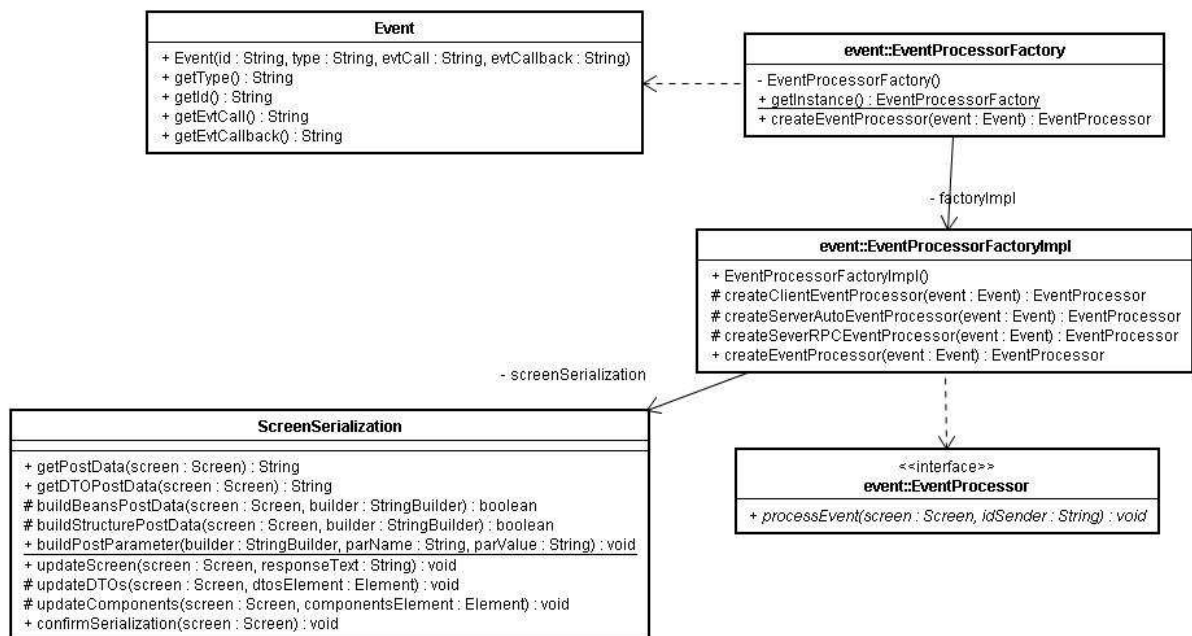

Figura 3-9 – Classes Envolvidas no Tratamento de Evento no Cliente.

Para criar o processador do evento, é utilizada a classe *EventProcessorFactory* (que segue o padrão *Factory*). Esta classe recebe um objeto representando um evento, com o tipo do evento a ser criado, a chamada a ser executada no controlador e eventuais métodos de tratamento da resposta para o evento.

A classe *EventProcessorFactory* é, na realidade, uma classe abstrata. O Crux possui uma implementação para esta classe que sabe criar objetos processadores de evento para os três tipos de evento mostrados na seção 3.2.2.1 (Client, Server-RPC e Server-Auto).

Esta, no entanto, pode ser substituída, adicionando uma instrução no arquivo de configuração do módulo do Motor Cliente. O Apêndice B.8 mostra como isso poderia ser feito.

Este mecanismo de escolha da classe de fábrica para os processadores de evento segue o padrão *Strategy* e o mecanismo de delegar o processamento para objetos que implementam a interface *EventProcessor* segue o padrão *Command* (Gamma et al, 2002).

As três seções seguintes descrevem como a classe de fábrica cria processadores de evento para os três tipos de evento nativamente suportados pelo Crux.

Eventos Client

Para os eventos do tipo *Client*, é criado um processador de eventos que primeiramente solicita à classe *RegisteredClientEventHandlers*, gerada segundo o processo mostrado no início da seção 3.2.2.3, a classe de tratamento do evento. Esta classe implementa a interface *EventClientHandlerInvoker* e, deste modo, possui o método *invoke,* que executa o método passado como parâmetro.

Todas estas classes, incluindo o próprio gerenciador de eventos, executam no navegador do usuário como *Javascript,* traduzidas pelo compilador GWT.

Eventos Server-RPC

Para os eventos do tipo *Server-RPC*, é criado um processador de eventos que realiza uma requisição para o *servlet* de tratamento no Motor Servidor. Nesta requisição o processador de eventos envia como parâmetros os valores de todos os componentes que estejam associados a alguma propriedade no servidor, através do atributo "*_serverBind*" (vide seção 3.2.2.2).

O Motor Servidor processa a requisição e envia a resposta seriada segundo o protocolo JSON. O processador de eventos do cliente, então, caso exista algum método de tratamento de resposta associado ao evento, invoca o método citado, seguindo o mesmo processo de invocação descrito para um evento cliente, a partir da classe *RegisteredClientEventHandlers.* O objeto retornado pelo servidor é enviado como parâmetro para o método de tratamento de resposta.

A comunicação com o Motor Servidor é feita de forma assíncrona. Isto significa que o processador de eventos envia a requisição e não fica esperando pelo Motor Servidor. Quando a resposta chegar, ele dispara um método de tratamento do processador de eventos.

No entanto, como visto na seção 3.2.2.1, um evento no Crux pode ser disparado de forma síncrona ou assíncrona. Este sincronismo se refere ao comportamento da interface e não a forma como a comunicação é feita. O envio da mensagem, por seguir o modelo AJAX, é feito de forma assíncrona, mas a interface pode se comportar de maneira síncrona, aguardando que a resposta para o evento seja recebida para permitir outra interação com o usuário.

Eventos Server-Auto

Para os eventos do tipo *Server-Auto*, é criado um processador de eventos que, de forma semelhante ao que é feito nos eventos *Server-RPC*, realiza uma requisição para o *servlet* de tratamento do Motor Servidor. As diferenças para este tipo de evento são os parâmetros a serem enviados e o tratamento do retorno da requisição.

No evento *Server-Auto*, toda a árvore de componentes é disponibilizada para o controlador no servidor. Deste modo, todas as informações necessárias para reconstruí-la devem ser submetidas. Quais informações devem ser submetidas depende do tipo de estratégia utilizada para persistência do estado da árvore de componentes (reenvio na requisição ou armazenamento na sessão). A classe *ScreenSerialization* (veja Figura 3.9) cuida de enviar todas as informações necessárias sobre a árvore de componentes.

Ao receber a resposta do servidor (também passada de forma assíncrona), o processador de eventos recebe um XML com as alterações na árvore de componentes (vide Apêndice B.6). É feita uma análise desta resposta e, para cada alteração detectada, é chamado um método de atualização do próprio componente, que recebe como parâmetro o elemento XML associado a ele. A Figura 3.10 mostra as classes envolvidas neste processo.

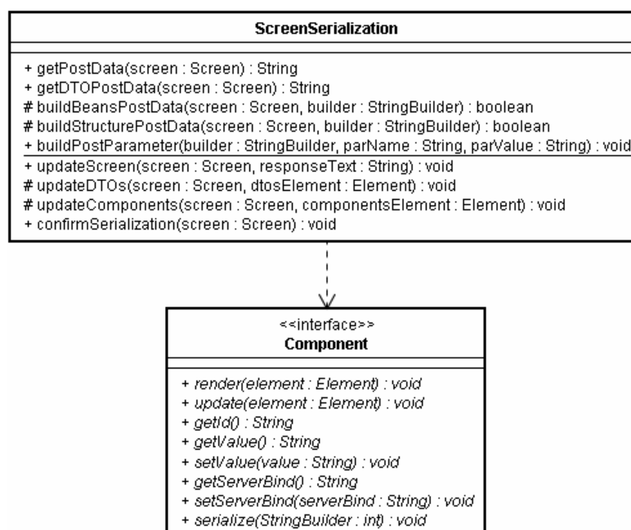

Figura 3-10 – Classes Envolvidas na Análise da Resposta de um Evento Server-Auto.

Criação da Árvore de Componentes

A construção da árvore de componentes no Motor Cliente é feita de forma semelhante à construção realizada no Motor Servidor (vide seção 3.2.2.2). Uma classe *ScreenFactory* também existe no cliente e realiza a análise da mesma forma como no servidor. A diferença é que a biblioteca de análise do HTML utilizada faz parte da própria API do GWT. Desta forma, o elemento representando a marcação <span> passado para o componente é um elemento do DOM da API do GWT.

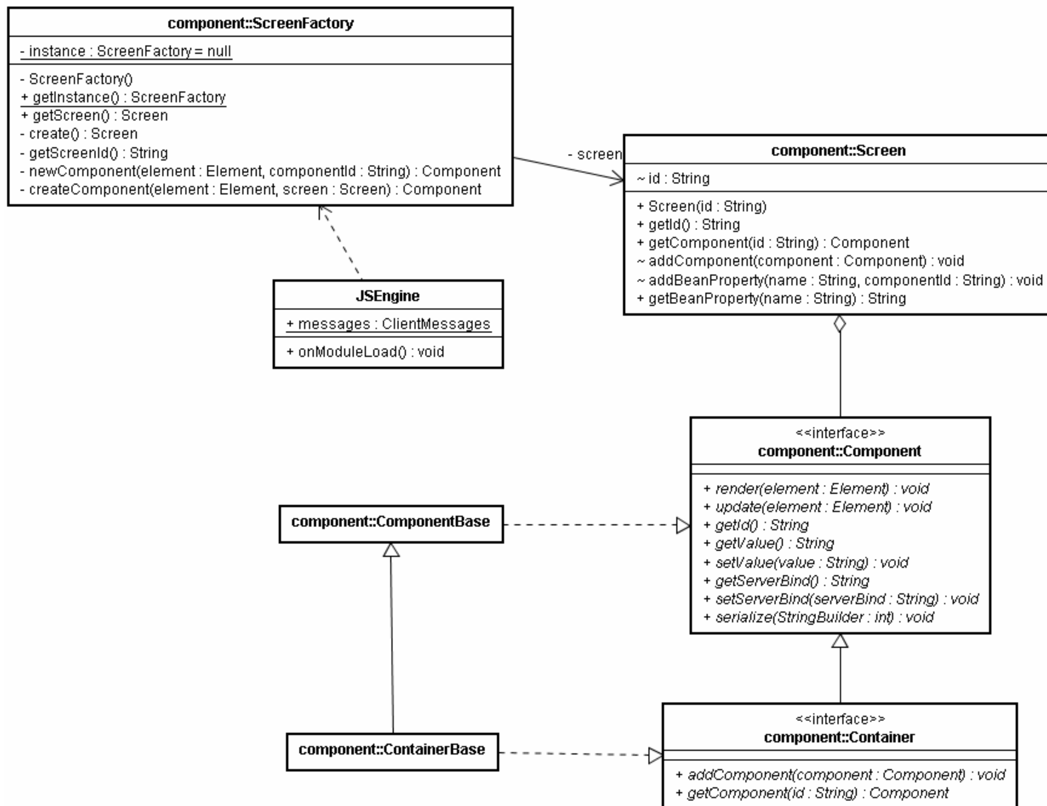

Figura 3-11 – Classes Envolvidas na Criação da Árvore de Componentes.

Outra diferença em relação ao modelo do servidor é que, ao invés de existirem classes de análise e de escrita associadas aos componentes, a própria classe de componente possui métodos que realizam estas tarefas. Foi escolhido este desenho porque o código final será um *Javascript* e foi feito todo o possível para deixá-lo pequeno e mais simples que o do servidor. Como não é possível utilizar reflexão, este desenho deixa o modelo consideravelmente mais simples. Para ter classes de análise e de escrita separadas por componente, o Crux precisaria adicionar mais uma coleção de geradores e de classes ao *Javascript* final gerado.

As classes *ComponentBase* e *ContainerBase*, mostradas na Figura 3.11, são classes de implementação das interfaces *Component* e *Container*. Estas classes são fornecidas para permitir o uso de componentes e painéis do GWT no Crux, seguindo o padrão *Adapter* (Gamma et al, 2002).

3.3 Análise do Arcabouço

Esta seção apresenta uma análise do arcabouço produzido, destacando como este ataca os desafios citados nas seções 2.1.1.3 e 2.2.3.2. São apresentadas, também, algumas considerações gerais a respeito da solução proposta.

3.3.1 O Crux e os Desafios Relacionados aos Arcabouços

Na seção 2.1.1.3, foram apresentados alguns desafios descritos por Bosch relacionados ao uso, desenvolvimento, manutenção e composição de arcabouços. Nesta seção, destacamos como o Crux aborda cada um desses desafios.

3.3.1.1 Desenvolvimento

- **Modelos de negócio:** Como este é um trabalho acadêmico, não será considerado aqui um modelo para uma eventual exploração financeira do arcabouço como produto. O objetivo do trabalho é estudar arcabouços, construindo o Crux para orientar o estudo.
- **Verificação de comportamentos abstratos:** Este é um desafio que foi facilmente percebido durante o desenvolvimento. Não conhecemos uma solução trivial. Tivemos que instanciar um projeto de testes para permitir o desenvolvimento e teste do arcabouço. O GWT facilitou uma tarefa que seria ainda mais difícil para este arcabouço específico, que seria testar código feito em *Javascript*.
- **Gerenciamento de versões:** Não existem versões deste arcabouço sendo utilizadas, de forma que não chegamos a experimentar aqui este problema.

3.3.1.2 Utilização

- **Aplicabilidade do arcabouço e Estimativas de esforço de desenvolvimento:** Estes desafios serão proporcionais à complexidade e à dificuldade para aprender a utilizar e estender o arcabouço. Não foram realizados testes para medir o grau de complexidade e a forma da curva de aprendizagem necessária para aprender a utilizar o arcabouço. No entanto, acreditamos que ele apresenta um modelo de

desenvolvimento mais simples do que o dos outros arcabouços analisados em nossa pesquisa, conforme discutido na seção 3.3.3.

- **Depuração de aplicações:** Este desafio cresce quando se utiliza inversão de controle, presente no arcabouço. Além disso, o Crux é um arcabouço predominantemente de caixa preta, outro fator que dificulta a depuração das aplicações. O Crux, no entanto, possui algumas características que ajudam a amenizar os seus principais problemas de depuração. A primeira delas é que o código *Javascript* pode ser depurado antes da conversão, quando ainda é um código Java (graças ao GWT). Outra característica é que o Crux distribui todo o seu código fonte junto com o *jar* do arcabouço. Isto já seria exigido para o código cliente (por causa do GWT), mas também foi aplicado às classes com código do lado servidor.

3.3.1.3 Composição

- **Diferenças de arquitetura e Composição de controle:** Uma preocupação constante no desenho do Crux foi deixá-lo o mais flexível possível, para que pudesse se adaptar a diferentes arquiteturas. Desta forma, sempre que precisamos tomar alguma decisão importante, procuramos encapsular esta decisão em um ponto que pudesse ser alterado, caso o arcabouço precisasse se adaptar a alguma nova arquitetura. Isto ocorre com todas as classes de Fábrica (por exemplo, na criação de classes de controle, telas, componentes, processadores de evento e arquivos de mensagens) e com os tratadores das requisições no servidor, que podem ser substituídos por outros que possuam ciclos de vida totalmente modificados e personalizados.
- **Sobreposição de entidades:** Para minimizar este problema, o Crux não faz nenhum tipo de exigência para as classes que tratarão eventos. Qualquer classe java simples pode tratar um evento. O problema, no entanto, pode aparecer à medida que algum desenvolvedor de componentes crie um componente, para atender algum domínio específico, que mapeie alguma entidade desde domínio e depois tente utilizar outro arcabouço em conjunto.
- **Composição com componentes legados:** Para composição com componentes legados, ou mesmo implementados para outro arcabouço, é possível criar

adaptadores, como o criado para permitir a integração com os componentes do GWT.

3.3.1.4 Manutenção

- **Mudanças no domínio:** Para fugir de problemas relacionados a mudanças de domínio, o Crux adota uma postura simples: restringir seu escopo, de modo a não se prender em um domínio específico. Isto não significa que o Crux não possa fornecer suporte específico de um domínio a uma aplicação. A questão é que o Crux se desconecta de qualquer domínio específico, sendo distribuído em um módulo genérico e permitindo que novos componentes (que podem ser voltados para um domínio qualquer) sejam registrados de forma automática. Desta forma, uma mudança de domínio impactaria apenas um determinado conjunto de componentes que poderiam ser alterados sem a necessidade de redistribuir o cerne do arcabouço.

3.3.2 O Crux e os Desafios Relacionados à Web

Em resposta aos desafios apontados na seção 2.2.3.2, pode-se dizer sobre cada um dos desafios:

1. O GWT fornece suporte para controlar o histórico de navegação de modo que "voltar" e "próximo" funcionam perfeitamente. Como o GWT é usado internamente no Motor Cliente, pode-se utilizar este suporte.
2. As páginas HTML realmente não são apropriadas para criação de interfaces gráficas, como as que temos em aplicações tradicionais. No entanto, a tela orientada a componentes, combinada com o modelo de comunicação de dados do AJAX ajuda a melhorar a qualidade de interação consideravelmente.
3. O Crux restringe um pouco mais a gama de dispositivos que podem acessar a aplicação, exigindo que seu navegador possua suporte a *Javascript*.
4. O GWT garante que o Crux possa executar nos principais navegadores existentes hoje, facilitando a tarefa do programador.
5. Embora seja possível emular o comportamento de notificação para a interface no Crux, utilizando o mecanismo descrito por Hanson-Tacy, (Hanson-Tacy, 2007), não

vemos necessidade de este comportamento ser utilizado. O modelo MVC2 atende a grande maioria dos casos reais.

6. A conversão de dados dos parâmetros é feita automaticamente pela fase de amarração dos parâmetros no Crux.

7. Para contornar o fato de o HTML não poder ser estendido, o Crux utiliza marcações <span> com atributos *_type*, permitindo estender os tipos de elementos que podem ser desenhados (embora, no fundo, todos os componentes são desenhados usando uma combinação dos elementos HTML existentes).

8. O GWT resolve, para o Crux, o problema de teste e depuração de códigos no navegador do usuário perfeitamente bem.

9. Ainda não foi implementado um mecanismo de validação para os dados fornecidos pelo usuário. A seção 3.3.3 discute como isso poderia ser feito.

3.3.3 Considerações Gerais

Acreditamos que o Crux torna o desenvolvimento de uma aplicação Web mais simples. Se comparado aos outros arcabouços pesquisados, o Crux:

- Não possui nenhum arquivo XML de configuração que seja de conhecimento obrigatório do programador.
- Possui suas telas implementadas por páginas que são feitas em HTML puro, de modo que podem ser abertas e alteradas em qualquer editor de HTML existente, ao invés de utilizar páginas JSP, repletas de marcações próprias.
- Possui tratadores de eventos que são escritos sempre em Java, independente de se serão executados no servidor ou no cliente (navegador).
- Possui uma árvore de componentes que existe e pode ser alterada tanto pelo cliente quanto pelo servidor.
- Pode ser estendido de várias formas, para que possa trabalhar em conjunto com outros arcabouços.

Um ponto importante, no entanto, não chegou a ser implementado no Crux: a validação e formatação de dados na tela. Esta parte ainda precisa ser mais bem definida, mas acreditamos que um mecanismo semelhante ao utilizado para o registro de controladores (utilizando anotações) poderia ser utilizado para se registrar classes de formatação (e validação).

Um formatador poderia ser referenciado na página HTML através de um atributo como "*_formatter*" adicionado à marcação <span>. Esta classe seria chamada para exibir o valor contido no componente associado ao formatador (segundo o padrão *Decorator*) (Gamma et al, 2002). Da mesma forma, poderia ser feita a validação de formato dos dados fornecidos pelo cliente.

Poderia, também, ser criado um mecanismo para informar, junto às chamadas de evento, se um determinado evento deve realizar algum tipo de validação antes da sua execução.

Para validação no lado servidor, uma fase poderia ser adicionada ao ciclo de vida dos tratadores de evento, antes da fase de disparo.

Outro ponto importante seria a criação de ferramentas para acelerar o desenvolvimento com o arcabouço. A seção 5.1.2 aborda essa questão.

3.3.4 O Crux e Outras Soluções

Esta seção posiciona o Crux ao lado das outras soluções apresentadas na seção 2.3.3. A seção 3.3.4.1 mostra uma relação dos padrões de projeto utilizados no Crux e a seção 3.3.4.2 compara a abordagem do Crux com a dos demais arcabouços citados.

3.3.4.1 Padrões de Projeto no Crux

O Crux foi criado a partir de diversos padrões de projeto. Muitos destes são utilizados, também, em outros arcabouços que analisamos. Os principais padrões utilizados são mostrados na Tabela 3.1.

Padrões	Componente do Crux
Command (Gamma et al, 2002)	EventClientHandler, EventClientCallback, Controller, EventProcessor
Session Façade	Controller
Factory, Singleton (Gamma et al, 2002)	ControllerFactory, ScreenFactory, MessagesFactory, EventProcessorFactory, ConfigurationFactory
Prototype (Gamma et al, 2002)	Screen, ScreenFactory
Proxy (Gamma et al, 2002)	MessagesFactory
Transfer Objects (fka Value Objects)	Event, DTOs[22]

[22] *Data Transfer Object*. Trata-se de um *JavaBean* utilizado para agrupar dados relacioanados. Segue o padrão *Transfer Objects* (Deepak et al., 2001).

Front Controller	Servlets de Tratamento de Eventos
View Helper, Value Object Assembler	DTOs, JSONValue
Strategy (Gamma et al, 2002)	ControllerFactoryImpl, EventProcessorFactoryImpl, ScreenStateManager
Builder (Gamma et al, 2002)	ScreenFactory
Chain of Responsibility (Gamma et al, 2002)	LifeCycle, Phase
Composite View	Component, Container, Screen

Tabela 3-1 – Padrões de Projeto no Crux.

Analisando a Tabela 3.1, pode-se perceber que ela apresenta diversos padrões que são utilizados nos três outros arcabouços apresentados na seção 2.3.3. Algumas novidades, no entanto, aparecem nesta tabela.

A classe *ScreenFactory* utiliza o padrão *Builder* para construção das telas. Ela controla todo o processo de criação da Árvore de Componentes, varrendo a página HTML em busca de marcações que descrevem componentes. Quando uma marcação é encontrada, uma classe associada ao componente cuida de criar o componente propriamente dito.

Além disso, a *ScreenFactory* utiliza, também, o padrão *Prototype*. Desta forma, ela evita ter de re-analisar a página HTML toda vez que uma tela precisa ser criada. Um único objeto é criado e cópias dele são feitas para atender as próximas demandas.

O padrão *Strategy* também é bastante utilizado no Crux, para permitir que todos os pontos mais delicados possam ser personalizados, caso necessário.

O padrão *Chain Of Responsability* foi usado para dividir a responsabilidade do processamento de uma requisição no servidor. A classe *LifeCycle* permite que se personalize o ciclo de vida, adicionando ou sobrescrevendo fases, que devem estender a classe abstrata *Phase*.

3.3.4.2 Comparação do Crux com as Outras Soluções

Esta seção compara o Crux, com os demais arcabouços mostrados na seção 2.3.3. Conforme discutido na seção 2.3.3.4, consideramos dois aspectos chave para esta análise: a forma como são organizados os elementos de tela (árvore de componentes) e como ocorre a comunicação entre a interface e a camada de negócio da aplicação.

Árvore de Componentes

O Crux junta os benefícios vistos em todos os arcabouços analisados na seção 2.3.3.4 (JSF, Struts e GWT). Pode-se criar a árvore de componentes de forma declarativa, de forma ainda mais fácil do que no JSF. Não são necessárias definições em arquivos XML e não são utilizadas páginas JSP com marcações próprias. É utilizado HTML comum. Além disso, a árvore de componentes existe e pode ser manipulada tanto no servidor como no cliente. A sincronização de estado entre cliente e servidor é transparente para o desenvolvedor da aplicação. Como o Crux utiliza o GWT, o suporte para codificação no cliente também é aproveitado. Uma classe Java pode ser escrita e executada tanto no cliente como no servidor.

Modelo de Comunicação Entre Cliente e Servidor

O Crux possui tratadores que são classes Java simples, sem exigir a herança de qualquer tipo de classe ou interface. Não é necessário realizar, também, qualquer tipo de mapeamento em arquivos XML para as classes tratadoras e para as classes de representação dos parâmetros. Além disso, pode-se modificar a forma como os controladores são criados, substituindo a sua classe de fábrica.

Outro ponto é que o Crux suporta o AJAX de modo nativo e possui um modelo de eventos que pode ser estendido, de forma a adicionar novos tipos.

Desta forma, o Crux não apresenta os problemas de comunicação apontados na seção 2.3.3.4.

3.3.5 O Desempenho do Crux

Esta seção aborda os impactos no desempenho do Crux em função das decisões de desenho apresentadas neste capítulo.

Um primeiro ponto está relacionado à decisão de transferir parte do processamento para o cliente, ao invés de usar o mesmo para apenas visualizar HTML, como trabalham as aplicações Web tradicionais.

O Crux adiciona um Motor no cliente que tem como responsabilidade montar os componentes de tela e comunicar com o servidor. Desta forma, enquanto nas aplicações tradicionais, uma tabela seria escrita inteira em HTML (linha por linha), no Crux, apenas uma marcação indicando que uma tabela deve ser desenhada na tela é escrita. O Motor Cliente, executando no navegador do usuário, monta essa tabela no cliente. Posteriormente, esta tabela poderia ter seus dados alterados, com eventos que transmitem os dados que devem ser exibidos pela tabela (apenas dados, ao invés de toda uma página HTML).

Este modelo descrito segue a linha do que o GWT faz. Os componentes também são montados no cliente e é disponibilizado um modelo de comunicação que segue o AJAX (vide seção 2.2.4.9). Os benefícios de desempenho em se utilizar AJAX e DHTML são comentados por Garret (Garret, 2005).

Outro ponto importante está relacionado à tradução do código a ser executado no cliente. O Crux utiliza o compilador do GWT para permitir que o programador possa escrever código em Java que é traduzido para *Javascript* e depois enviado ao cliente. Uma questão importante é: Quão boa é esta tradução? Esse código é tão bom quanto um código escrito diretamente em *Javascript*?

O GWT responde a essa pergunta em sua página principal na internet com a frase: "*Faster AJAX than you'd write by hand*". O compilador do GWT faz várias otimizações no código. Entre as principais medidas, podemos citar a remoção de quaisquer métodos e classes não utilizadas, gerando códigos compactos. Além disso, são geradas várias versões do código, uma versão ótima para cada tipo de navegador. Outro ponto importante é a substituição do mecanismo de reflexão do Java pelo mecanismo de amarração tardia (vide seção 2.3.3.3). Aplicações como o *gmail* (gmail, 2008) e o *google maps* (maps, 2008), feitas com o GWT, são exemplos de sistemas com bom desempenho que apresentam interfaces muito ricas e atendem um altíssimo volume de usuários.

Um terceiro ponto a ser considerado é a forma como uma aplicação Web mantém o estado de seus elementos de tela entre diferentes requisições. Conforme pode ser visto na seção 2.2.4.3, o protocolo HTTP não mantém estado entre as requisições. Deste modo, caso uma aplicação Web possua uma tela com um campo qualquer que, em dado momento, tem alguma propriedade, como por exemplo a visibilidade, alterada, ela precisará guardar essa informação de alguma forma para que em uma próxima requisição ela não se perca. Isto independe de esta aplicação organizar seus elementos em uma árvore de componentes (como o Crux e o JSF), ou não (como o *Struts*).

Existem algumas formas de uma aplicação manter o estado entre as requisições:

1. Salvar esse estado no servidor (em memória, ou disco);
2. Salvar em um ponto central, de onde todas as máquinas acessam (um banco de dados, por exemplo).
3. Salvar no cliente, de modo que este re-envie as informações em todas as requisições para uma mesma tela.

O problema em se colocar estes objetos em memória no servidor é a maior dificuldade em colocar uma aplicação em *cluster*[23]. Para que qualquer máquina do *cluster* possa responder a requisições de qualquer usuário, é necessário que as informações sejam replicadas para todos os componentes do *cluster*. Vários servidores de aplicação dão suporte automático a isso, mas quanto mais informações para serem replicadas, maior o trabalho do servidor e pior o desempenho.

Colocar as informações em um banco de dados, faria com que toda operação em qualquer componente de tela tivesse que ser gravada e lida do banco, o que também geraria um impacto de desempenho.

A terceira opção aumenta a quantidade de informação que é enviada em uma requisição e faz com que o servidor atualize os elementos de tela antes de disponibilizá-los para as classes de tratamento da aplicação.

O *Struts*, analisado na seção 2.3.3.1, adota apenas a primeira alternativa mostrada acima e fornece a opção de colocar todo o seu *formBean* (objeto Java que transporta os dados da tela no *Struts*) na sessão do usuário. Desta forma, em uma requisição futura, os dados não são perdidos. Caso não se queira manter esta informação em sessão, para evitar os problemas mostrados, deve-se, a cada requisição, refazer a lógica que desabilitou o campo, para manter os componentes com um estado correto.

O JSF possui um mecanismo de configuração que permite ao programador escolher entre a primeira e a terceira soluções. Por padrão, ele utiliza a primeira abordagem e mantém a sua árvore de componentes na sessão do usuário. Caso se queira colocar a aplicação em *cluster*, recomenda-se trocar para a terceira abordagem, fazendo com que o JSF percorra a sua árvore pedindo a cada componente que retorne toda a sua informação de estado. Essa informação é concatenada e colocada em um campo oculto na página HTML enviada ao cliente, para que ela possa ser re-enviada na próxima requisição.

[23] Várias máquinas executando uma mesma aplicação, mas aparecendo para o mundo como uma única máquina. Um *cluster* serve para aumentar a quantidade de usuários que a aplicação está atendendo, dividindo a carga entre mais de um servidor, e para proporcionar uma maior tolerância a falhas.Caso um servidor pare de responder, os outros continuam mantendo a aplicação no ar.

O GWT não fornece acesso a sua árvore de componentes no lado servidor da aplicação, de modo que sempre realiza todas as mudanças em componentes no próprio navegador do usuário, sem submeter estado para o servidor. O problema com esta abordagem é que existem situações em que é muito mais cômodo ter a árvore no servidor, principalmente quando a aplicação tem muitas regras que influenciam a forma como os componentes devem se comportar.

O Crux possui um modelo configurável, mais flexível que o do JSF. Existe uma interface para a classe de gerência de estado (vide seção 4.1.2). Juntamente com o Crux, já são distribuídas implementações para a primeira e para a terceira abordagem. Caso se queira utilizar a segunda, basta criar mais uma implementação. Além disso, é possível trabalhar com o Crux da mesma forma como o GWT faz, realizando todas as mudanças de estado apenas no lado cliente, sem nunca enviar estado para o servidor. Para isso, bastaria usar apenas eventos Server-RPC. Além disso, a forma como o Crux implementa a terceira abordagem é mais eficiente que a forma do JSF. Apenas o que foi alterado na árvore de componentes é re-enviado. O Crux mantém o rastreamento destas alterações para diminuir a quantidade de informação trafegada.

Além dos pontos de impacto em desempenho destacados, o Crux se preocupa com o desempenho de algumas das tarefas que são críticas para ele. A criação de arquivos de mensagens ou de configurações utiliza *caches* internos, para garantir que executem apenas uma vez. A análise de suas telas, a partir das páginas HTML, também ocorre apenas uma vez e o padrão *Prototype* (Gamma et al, 2002) é utilizado para *clonar* as telas já em memória. Geradores para classes de acesso a componentes e tratadores utilizados na página também ajudam a manter apenas o extremamente necessário no arquivo *Javascript* enviado ao cliente (vide seção 3.2.2.3).

CAPÍTULO 4 - Detalhamento da Solução

Esta seção detalha a solução apresentada no capítulo 3, mostrando trechos de classes dos Motores Cliente e Servidor e de sua Árvore de Componentes, que não foram apresentados para que não ofuscassem a idéia principal que estava sendo exposta.

A seção 4.1 detalha o funcionamento do Motor Servidor, enquanto a seção 4.2 detalha o funcionamento do Motor Cliente.

4.1 Motor Servidor

Esta seção apresenta aspectos relacionados ao Motor Servidor, não apresentados no capítulo anterior.

A seção 4.1.1 detalha os processos realizados ao início da aplicação. A seção 4.1.2 detalha a construção da árvore de componentes. A seção 4.1.3 detalha o tratamento de eventos. A seção 4.1.4 detalha o processo de criação de classes de mensagens internacionalizadas.

4.1.1 Início da Aplicação

Conforme apresentado na seção 3.2.2.2, um *Listener* se encarrega de iniciar as configurações necessárias para o funcionamento da aplicação. Esta classe, descrita no capítulo anterior, se chama *InitializerListener* e, conforme mostrado na seção 3.2.2.2, realiza três tarefas importantes.

A carga do arquivo de configurações do Crux – primeira destas tarefas – é um processo simples, feito de maneira semelhante ao processo realizado pela classe *MessagesFactory*, descrito na seção 4.1.4. As configurações são acessadas a partir da interface *Crux* e escritas em um arquivo de propriedades com o mesmo nome.

O registro dos componentes – a segunda das tarefas – é feito pela classe *ComponentConfig*. Esta recebe uma lista de URLs, contendo as localidades que deverão ser consultadas em busca de componentes.

Estas duas primeiras tarefas são suficientemente detalhadas no capítulo anterior. A terceira tarefa – início das classes de fábrica dos controladores – possui alguns aspectos não mencionados ainda, que são descritos a seguir.

Esta tarefa é realizada pelas classes *ClientControllerFactory* e *ControllerFactory*, apresentadas na seção 3.2.2.2. Estas classes devem percorrer a aplicação, procurando por classes que devam ser registradas como controladores. Como visto na seção 3.1.2, os controladores, tanto os executados no servidor como os executados no cliente, utilizam anotações específicas.

O desafio no registro desses controladores está em conseguir inspecionar todas as classes que estejam no contexto da busca sem que se precise carregá-las todas para memória, o que teria um custo elevadíssimo.

Para realizar esta tarefa, foi utilizada a biblioteca *Scannotation* (Scannotation, 2008). Esta foi criada para pesquisa de classes a partir de uma determinada anotação. Ela recebe um conjunto de URLs, apontando para os locais onde deve realizar sua busca, e varre todas as classes encontradas neste local, sem, no entanto, carregar as classes para memória. Ela monta, então, uma tabela com os nomes das classes encontradas e quais anotações cada classe possui.

Para inspecionar uma classe sem que se precise carregá-la, a *Scannotation* utiliza a biblioteca *Javassist* (Javassist, 2008), que permite a inspeção de classes diretamente a partir de seu *bytecode*[24].

Desta forma, foi criada uma classe chamada *ClassScanner*, que utiliza a *Scannnotation* internamente. Esta monta uma tabela com todas as classes que utilizam cada uma das anotações encontradas na busca. O Apêndice B.9 mostra a implementação da classe *ClassScanner*.

Desta forma, tanto a classe *ClientControllerFactory* quanto a classe de implementação padrão do Crux para a interface *ControllerFactory*, utilizam a classe *ClassScanner* para descobrir as classes que utilizam as anotações de registro de tratador de evento.

[24] Nome dado ao código Java compilado, contido em um arquivo com a extensão "*class*". É interpretado pela máquina virtual do Java.

4.1.2 Árvore de Componentes

Conforme visto na seção 3.2.2.2, a árvore de componentes é montada, no servidor, pela classe de fábrica *ScreenFactory*. Esta classe, através dos dados carregados pela classe *ComponentConfig*, decide quais são as classes que devem ser usadas para representar o componente e para analisar o elemento HTML na página, para construí-lo.

A Listagem 4.1 mostra o método da classe *ScreenFactory* usado para criação de um componente. Este é chamado quando a mesma encontra um elemento que descreve um componente na página HTML da tela.

```
1  private Component newComponent(Element element, String componentId) throws ScreenConfigException {
2      String type = element.getAttributeValue("_type");
3      String className = ComponentConfig.getServerClass(type);
4      ComponentParser parser = ComponentConfig.getComponentParser(type);
5      String parserInput = ComponentConfig.getParserInput(type);
6      if (className == null || parser == null || parserInput == null) {
7          throw new ScreenConfigException(messages.screenFactoryErrorCreateComponent(componentId));
8      }
9      try {
10         Component component = (Component) Class.forName(className).newInstance();
11         component.setId(componentId);
12         component.setType(type);
13         if (ComponentConfigData.PARSER_INPUT_DOM.equals(parserInput)) {
14             parser.parse(component, toDomElment(element));
15         } else if (ComponentConfigData.PARSER_INPUT_STRING.equals(parserInput)) {
16             parser.parse(component, element.toString());
17         } else {
18             parser.parse(component, element);
19         }
20         return component;
21     } catch (Throwable e) {
22         throw new ScreenConfigException(messages.screenFactoryErrorCreateComponent(componentId), e);
23     }
24 }
```

Listagem 4-1 – Método Chamado para Criação de um Componente na *ScreenFactory*.

A Interface *ComponentParser*, deve ser implementada pela classe de análise associada ao componente (configurada no arquivo crux.xml) e é mostrada na Listagem 4.2, abaixo.

```
1  public interface ComponentParser
2  {
3      void parse(Component component, Object element);
4  }
```

Listagem 4-2 – Interface *ComponentParser*.

O Crux fornece uma classe padrão de implementação desta interface que popula o componente com eventos a partir de atributos que possuam nome de acordo com o padrão

“_on<nome_evento>” e propriedades de acordo com o padrão “_<nome_propriedade>”. O Apêndice B.10 mostra o método de análise do elemento desta classe padrão. Este aceita, como tipo de entrada, um elemento *Jericho*.

Na seção 3.2.2.2 é mostrado, também, o mecanismo utilizado para conservação do estado da árvore de componentes. Gerenciadores de estado podem ser criados, implementando a interface *ScreenStateManager*, mostrada na Listagem 4.3.

```
1 public interface ScreenStateManager
2 {
3     Screen getScreen(String screenName, HttpServletRequest request)
4             throws ScreenConfigException;
5     boolean clientMustPreserveState();
6 }
```

Listagem 4-3 – Interface *ScreenStateManager*.

O método *getScreen* é utilizado para recuperar a Árvore de Componentes e o método *clientMustPreserveState* é utilizado para informar ao Motor Cliente se este precisa guardar as alterações sofridas nos componentes da tela e as re-enviar a cada requisição.

A fase de amarração dos parâmetros para os eventos Server-Auto – que necessita recuperar a Árvore de Componentes – solicita para a classe de gerência de estado que a recupere. Esta decide quando chamar a classe *ScreenFactory*, caso seja necessário construí-la.

Conforme mostrado na seção 3.2.2.2, o Crux já possui duas implementações para esta interface. O Apêndice B.11 mostra a implementação destas duas classes.

4.1.3 Tratamento de Eventos

Conforme apresentado na seção 3.2.2.2, os tratadores de eventos processados no servidor estendem a classe *BaseServlet*, mostrada no Apêndice B.2. Esta utiliza a classe *LifeCycle* para processar a requisição recebida que, como visto na seção 3.2.2.2, segue o padrão *Chain of Responsability* (Gamma et al, 2002).

Nesta seção, são detalhadas as fases envolvidas no processamento dos eventos processados no servidor (Server-RPC e Server-Auto). A Figura 4.1 mostra as classes envolvidas no processamento das fases de disparo e de escrita da resposta para os eventos mencionados.

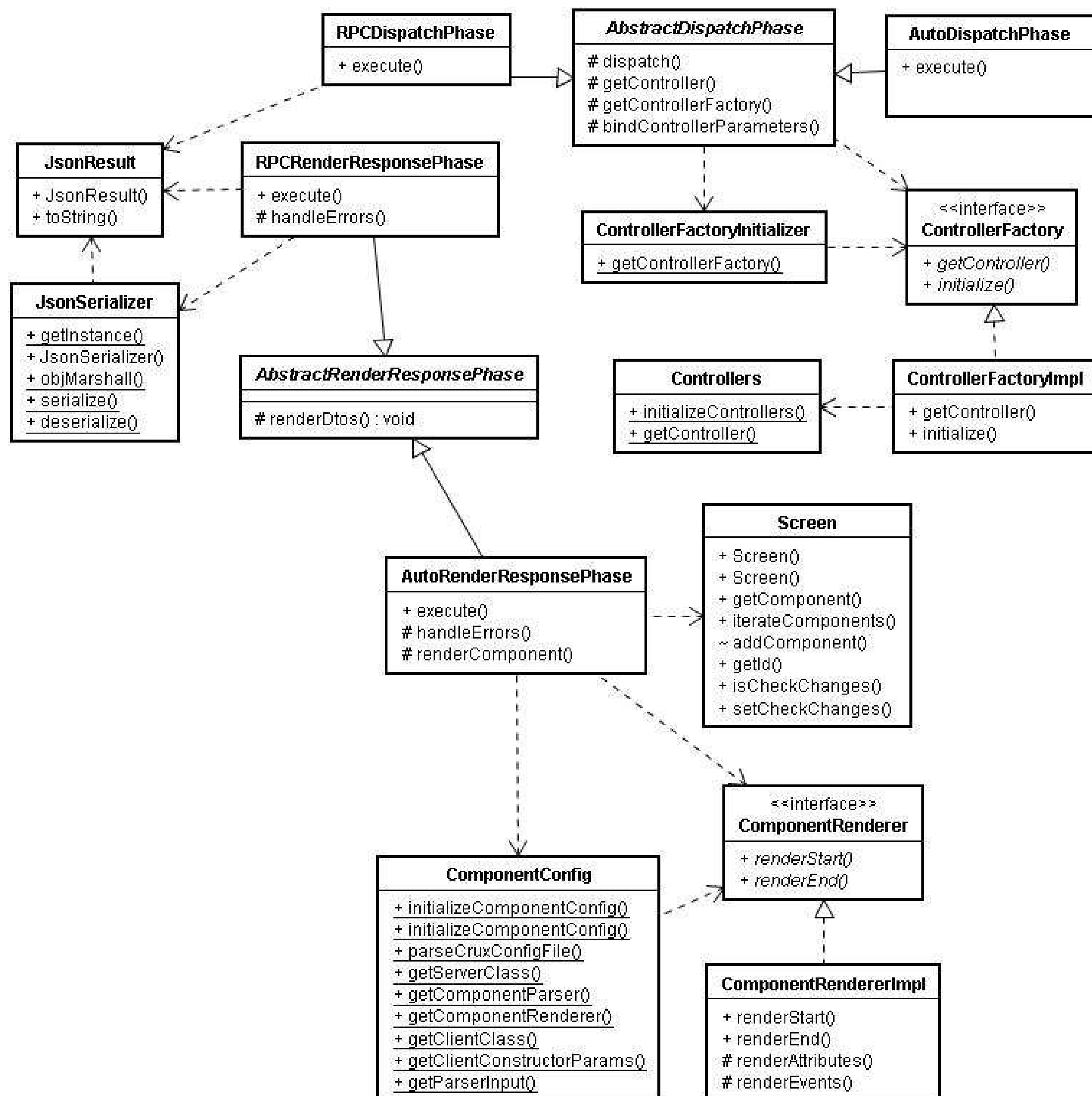

Figura 4-1 – Classes Envolvidas com as Fases de Eventos Processados no Servidor.

4.1.3.1 Evento Server-RPC

A classe *RPCDispatchPhase* é a classe responsável pela fase de disparo em um evento Server-RPC, descrita na seção 3.2.2.2. Após realizar o disparo do método de tratamento do evento, ela salva o resultado do disparo dentro de um objeto *JsonResult*.

A classe *RPCRenderResponsePhase* é a classe responsável pela fase de escrita da resposta para o Motor Cliente, também descrita na seção 3.2.2.2. Ela recupera o resultado do disparo do evento (gravado pela fase de disparo) e o escreve para o cliente. O apêndice B.12 mostra trechos do código destas classes.

4.1.3.2 Evento Server-Auto

A classe *AutoDispatchPhase* é a classe responsável pela fase de disparo em um evento Server-Auto, descrita na seção 3.2.2.2. Após realizar o disparo do método de tratamento do evento, ela salva a árvore de componentes como resultado do processamento do evento.

A classe *AutoRenderResponsePhase* é a classe responsável pela fase de escrita da resposta para o Motor Cliente. Ela recupera a Árvore de Componentes e verifica quais componentes foram alterados. Para cada componente alterado, é chamada a sua classe de escrita (vide seção 3.2.2.2), que deve implementar a interface *ComponentRenderer*. O Apêndice B.13 mostra trechos de código destas classes.

A interface *ComponentRenderer*, utilizada para escrita dos componentes alterados é mostrada na Listagem 4.4.

```
1  public interface ComponentRenderer
2  {
3      void renderStart(Component component, PrintWriter writer);
4      void renderEnd(Component component, PrintWriter writer);
5  }
```

Listagem 4-4 – Interface *ComponenteRenderer*.

O Crux distribui, também, uma classe de implementação padrão para esta interface. Esta produz um bloco de XML semelhante ao mostrado no Apêndice B.4, contendo as propriedades comuns aos componentes e todos os eventos associados ao mesmo.

4.1.4 Internacionalização

Conforme visto na seção 3.2.2.2, o Crux fornece um mecanismo de internacionalização semelhante ao do GWT, que permite que interfaces – como a mostrada na Listagem 3.12 – sejam usadas para acessar arquivos de mensagens – como o mostrado na Listagem 3.13. A classe *MessagesFactory* é utilizada para construir objetos a partir das interfaces – como mostrado na Listagem 3.11.

A Listagem 4.5 mostra o método utilizado para criação destes objetos de implementação para as interfaces. Os objetos retornados são *Proxies* dinâmicas, criadas utilizando uma API padrão do Java (segundo o padrão *Proxy*) (Gamma et al, 2002).

```java
1  private static Object initProxy (Class<?> targetInterface, final Locale locale) {
2      Object proxy = cachedProxies.get(targetInterface);
3      if (proxy != null) {
4          return proxy;
5      }
6      final PropertyResourceBundle properties = loadProperties(targetInterface, locale);
7      if (properties == null) {
8          throw new NullPointerException("resource bundle not found");
9      }
10     InvocationHandler invocationHandler = new InvocationHandler() {
11         public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
12             try {
13                 return MessageFormat.format(properties.getString(method.getName()),args);
14             } catch (Throwable e) {
15                 return null;
16             }
17         }
18     };
19     Class<?> proxyClass = Proxy.getProxyClass(targetInterface.getClassLoader(), new Class<?>[] {targetInterface});
20
21     try {
22         proxy = proxyClass.getConstructor(new Class<?>[] { InvocationHandler.class }).
23                                         newInstance(new Object[] { invocationHandler });
24         cachedProxies.put(targetInterface, proxy);
25         return proxy;
26     } catch (Exception e) {
27         throw new MessageException(e.getMessage(), e);
28     }
29 }
```

Listagem 4-5 – Método *initProxy*, da Classe *MessagesFactory*.

O Método *invoke* do objeto *invocationHandler*, criado na Listagem 4.5, é chamado para tratar toda chamada feita a qualquer método do objeto (*proxy*) retornado.

Também pode ser visto na Listagem 4.5, que um *cache* é utilizado para evitar que mais de um objeto seja criado para a mesma interface. O método de fábrica dos objetos de mensagens sempre verifica neste *cache* se um objeto já foi criado. Caso não tenha sido, o método mostrado acima é chamado. A Listagem 4.6 mostra o método de fábrica da classe *MessagesFactory*.

```
1  public static Object getMessages(final Class<?> targetInterface, final Locale locale)
2  {
3      Object proxy = cachedProxies.get(targetInterface);
4      if (proxy != null) {
5          return proxy;
6      }
7      try {
8          lock.lock();
9          return initProxy(targetInterface, locale);
10     } finally {
11         lock.unlock();
12     }
13 }
```

Listagem 4-6 – Método de Fábrica da Classe *MessagesFactory*.

4.2 Motor Cliente

Esta seção apresenta aspectos relacionados ao Motor Cliente, não apresentados no capítulo anterior.

A seção 4.2.1 detalha os processos realizados ao se carregar uma página no navegador do cliente. A seção 4.2.2 detalha a construção da árvore de componentes.

4.2.1 Início da Aplicação

Nesta seção, é detalhado o mecanismo utilizado para carga de configurações necessárias para o Motor Cliente do Crux.

Utilizando o mecanismo de amarração tardia do GWT (vide seção 2.3.3.3), é criada uma classe de implementação para a interface *CruxConfig*. Esta contém um método para cada configuração necessária para o Motor Cliente e tem o seu código mostrado na Listagem 4.7.

```
1  public interface CruxConfig
2  {
3      boolean clientMustPreserveState();
4  }
```

Listagem 4-7 – Interface *CruxConfig*.

A classe de implementação para esta interface é gerada, através da amarração tardia, pelo gerador *CruxConfigGenerator*. Esta utiliza a classe de gerência de estado do Motor

Servidor para gerar o método *clientMustPreserveState* da interface *CruxConfig*, mostrado na Listagem 4.7. A Listagem 4.8 mostra como este método é gerado.

```
1 private void generatePreserveStateMethod(SourceWriter sourceWriter, JClassType classType)
2         throws InstantiationException, IllegalAccessException, ClassNotFoundException
3 {
4     sourceWriter.println("public boolean clientMustPreserveState(){");
5     sourceWriter.println("return "+ScreenStateManagerInitializer.
6                             getScreenStateManager().clientMustPreserveState()+";");
7     sourceWriter.println("}");
8 }
```

Listagem 4-8 – Método *generatePreserveStateMethod*, da Classe *CruxConfigGenerator*.

Caso outras opções de configuração para o Motor Cliente precisem ser adicionadas, basta acrescentá-las na interface *CruxConfig* e atualizar o gerador para criar os métodos que retornem os seus valores.

4.2.2 Árvore de Componentes

Conforme mostrado na seção 3.2.2.3, a classe *ScreenFactory* realiza a análise da página HTML e constrói a Árvore de Componentes, além de desenhar os componentes na tela do cliente.

O método da classe *ScreenFactory* varre a página HTML em busca de elementos de descrição de componentes. Uma vez encontrado um elemento desses, ela cria o componente, utilizando o método mostrado na Listagem 4.9, abaixo:

```
1 private Component newComponent(Element element, String componentId) throws InterfaceConfigException
2 {
3     Component component = registeredComponents.createComponent(componentId, element.getAttribute("_type"));
4     component.render(element);
5     return component;
6 }
```

Listagem 4-9 – Método *newComponent*, da Classe *ScreenFactory*.

Como mostrado na seção 3.2.2.3, a classe *RegisteredComponents*, utilizada na Listagem 4.9 para criação do componente, é criada a partir de um gerador, para que saiba criar um objeto a partir do tipo passado como parâmetro. A Listagem 3.14 mostra um exemplo de geração dessa classe.

Pode-se perceber na Listagem 4.9 que, depois do componente ser criado, é chamado o seu método *render*. Este método irá, a partir das informações contidas no elemento (sua marcação <span>), desenhar o componente na tela e salvar as propriedades e eventos informados no HTML. O Apêndice B.14 mostra a implementação do método *render* da classe *ComponentBase*, usada para representar e desenhar componentes do GWT (vide seção 3.2.2.3).

Outro aspecto importante, relacionado à Árvore de Componentes, é a conservação do estado dos componentes entre requisições. A seção 4.2.3, mostra como este estado é mantido, ao se processar eventos que são executados no servidor.

4.2.3 Tratamento de Eventos

Quando um componente dispara um evento no Crux, o Motor Cliente disponibiliza, na classe *EventFactory*, um método chamado *callEvent*, para processar o disparo do evento. O Apêndice B.14 mostra um exemplo de classe de componente que utiliza este mecanismo para tratar eventos e a Listagem 4.10 mostra o código deste método.

```
1  public static void callEvent(Event event, String idSender)
2  {
3      try
4      {
5          EventProcessor processor = EventProcessorFactory.getInstance().createEventProcessor(event);
6          processor.processEvent(ScreenFactory.getInstance().getScreen(), idSender);
7      }
8      catch (InterfaceConfigException e)
9      {
10         GWT.log(e.getLocalizedMessage(), e);
11     }
12 }
```

Listagem 4-10 – Método *callEvent*, da Classe *EventFactory*.

Pode-se perceber pelo código mostrado acima que um objeto do tipo *EventProcessor* é criado para o processamento do evento. A seção 3.2.2.3 mostra este mecanismo. A Listagem 4.11 mostra o método *createEventProcessor* da classe de implementação padrão para a interface *EventProcessorFactory*.

```
1  public class EventProcessorFactoryImpl {
2      private RegisteredClientEventHandlers registeredClientEventHandlers;
3      private ScreenSerialization screenSerialization;
4  
5      public EventProcessorFactoryImpl()
6      {
7          this.registeredClientEventHandlers = (RegisteredClientEventHandlers)GWT.create(RegisteredClientEventHandlers.class);
8          this.screenSerialization = (ScreenSerialization)GWT.create(ScreenSerialization.class);
9      }
10 
11     public EventProcessor createEventProcessor(final Event event) throws InterfaceConfigException
12     {
13         if (EventFactory.TYPE_CLIENT.equals(event.getType()))
14         {
15             return createClientEventProcessor(event);
16         }
17         else if (EventFactory.TYPE_SERVER_AUTO.equals(event.getType()))
18         {
19             return createServerAutoEventProcessor(event);
20         }
21         else if (EventFactory.TYPE_SERVER_RPC.equals(event.getType()))
22         {
23             return createSeverRPCEventProcessor(event);
24         }
25 
26         throw new InterfaceConfigException(JSEngine.messages.eventProcessorFactoryInvalidEventType(
27                                 event.getId(), event.getEvtCall(), event.getType()));
28     }
29 }
```

Listagem 4-11 – Método *createEventProcessor*, da Classe *EventProcessorFactoryImpl*.

O processo realizado para criação de cada um dos três tipos de evento existentes é explicado na seção 3.2.2.3. A Listagem 4.12 detalha o processador de eventos para o tipo Server-Auto, pois este necessita garantir a sincronização da Árvore de Componentes com o Motor Servidor e merece uma explicação mais detalhada.

```
1  protected EventProcessor createServerAutoEventProcessor(final Event event){
2      return new EventProcessor() {
3          public void processEvent(final Screen screen, String idSender){
4              if (event.isSync()) screen.blockToUser();
5              String moduleRelativeURL = GWT.getModuleBaseURL() + "auto";
6              String postData = "idSender="+URL.encodeComponent(idSender!=null?idSender:"")+
7                                "&evtCall="+URL.encodeComponent(event.getEvtCall())+
8                                "&screenId="+URL.encodeComponent(screen.getId())+
9                                "&"+screenSerialization.getPostData(screen);
10
11             RequestCallback callback = new RequestCallback(){
12                 public void onError(Request request, Throwable exception) {
13                     if (event.isSync()) screen.unblockToUser();
14                     Window.alert(JSEngine.messages.eventProcessorServerAutoError());
15                 }
16
17                 public void onResponseReceived(Request request, Response response) {
18                     if (response.getStatusCode() != 200) {
19                         if (event.isSync()) screen.unblockToUser();
20                         Window.alert(JSEngine.messages.eventProcessorServerAutoError());
21                         return;
22                     }
23                     screenSerialization.confirmSerialization(screen);
24                     screenSerialization.updateScreen(screen, response.getText());
25                     if (event.isSync()) screen.unblockToUser();
26                 }
27             };
28             try{
29                 RequestBuilder builder = new RequestBuilder(RequestBuilder.POST, moduleRelativeURL);
30                 builder.setHeader("Content-type", "application/x-www-form-urlencoded; charset=utf-8");
31                 builder.sendRequest(postData, callback);
32             } catch (Exception e) {
33                 if (event.isSync()) screen.unblockToUser();
34                 Window.alert(JSEngine.messages.eventProcessorServerAutoError());
35             }
36         }
37     };
38 }
```

Listagem 4-12 – Método *createServerAutoEventProcessor*, da Classe *EventProcessorFactoryImpl*.

Na Listagem 4.12, pode-se perceber que um objeto do tipo *ScreenSerialization* é utilizado para montar a lista de parâmetros, relacionados à Árvore de Componentes, que devem ser adicionados à requisição. Este mesmo objeto é usado para atualizar a árvore a partir do retorno do Motor Servidor.

A classe *ScreenSerialization* percorre a Árvore de Componentes, solicitando que cada um informe quais parâmetros deve enviar para que ele possa ser montado novamente no servidor.

Um processo semelhante é feito para atualizar o estado a partir do retorno do servidor. Para cada bloco de alteração encontrado na resposta (vide Apêndice B.4), é solicitado ao componente referenciado que atualize o seu estado. A Listagem 4.136 mostra os métodos de montagem dos parâmetros e de atualização da tela, ambos da classe *ScreenSerialization*.

```
1  public String getPostData(Screen screen){
2      StringBuilder builder = new StringBuilder();
3      boolean notFirst = buildStructurePostData(screen, builder);
4      if (notFirst)
5          builder.append("&");
6      buildBeansPostData(screen, builder);
7      return builder.toString();
8  }
9
10 public void updateScreen(Screen screen, String responseText){
11     try {
12         Document document = XMLParser.parse(responseText);
13         NodeList children = document.getDocumentElement().getChildNodes();
14
15         for (int i = 0; i < children.getLength(); i++){
16             if (children.item(i).getNodeType() == Node.ELEMENT_NODE){
17                 Element element = (Element)children.item(i);
18                 String componentId = element.getAttribute("id");
19                 if ("_server_error_".equals(componentId)){
20                     Window.alert(JSEngine.messages.screenSerializationServerError(element.getAttribute("_value")));
21                     break;
22                 } else if ("_components_".equals(componentId)) {
23                     updateComponents(screen, element);
24                 } else if ("_dtos_".equals(componentId)) {
25                     updateDTOs(screen, element);
26                 }
27             }
28         }
29     } catch (DOMParseException e) {
30         GWT.log(e.getLocalizedMessage(), e);
31         Window.alert(JSEngine.messages.eventProcessorServerAutoResponseParserError());
32     }
33 }
```

Listagem 4-13 – Métodos *getPostData* e *updateScreen*, da Classe *ScreenSerialization*.

O método *update*, da interface *Component* realiza um trabalho semelhante ao do método render, que desenha a tela (vide Apêndice B.14). A diferença é que ele substitui os valores existentes pelos novos, tanto para as propriedades como para os eventos.

O Apêndice B.15 mostra os demais métodos utilizados na Listagem 4.13 para seriar os componentes e propriedades.

O método *buildBeansPostData* envia todos os valores de componentes associados a propriedades "*_serverBind*". O método *buildStructurePostData* percorre toda a árvore de componentes, solicitando a cada um que envie, através do método *serialize,* todas as informações necessárias para remontar o componente no servidor. Um exemplo de implementação deste método é mostrado na Listagem 4.14 (extraído da classe *ComponentBase*).

```java
1  public void serialize(StringBuilder builder)
2  {
3      if (modifiedProperties.size()>0)
4      {
5          boolean first = true;
6          for (String property : modifiedProperties.keySet())
7          {
8              if (!first)
9              {
10                 builder.append("&");
11             }
12             first = false;
13             ScreenSerialization.buildPostParameter(builder, "c("+getId()+")."+property, modifiedProperties.get(property));
14         }
15     }
16 }
```

Listagem 4-14 – Método *serialize*, da Classe *ComponentBase*.

Sempre que uma propriedade do componente é alterada no cliente, ela é adicionada à lista *modifiedProperties*, mostrada na Listagem acima. Quando um evento do tipo Server-Auto é respondido pelo servidor, o gerenciador de eventos chama o método *confirmSerialization*, da classe *ScreenSerialization* (vide Listagem 4.12).

Este método verifica o valor da propriedade de configuração "*clientMustPreserveState*", mostrada na seção 4.1.2. Esta configuração informa ao Motor Cliente se o mesmo deve submeter todo o estado da Árvore de Componentes, ou apenas o estado dos componentes que foram alterados desde a última requisição.

Desta forma, o método *confirmSerialization* foi escrito conforme é mostrado na Listagem 4.15.

```java
1  public void confirmSerialization(Screen screen)
2  {
3      if (!JSEngine.cruxConfig.clientMustPreserveState())
4      {
5          for (Component component : screen.components.values())
6          {
7              component.confirmSerialization();
8          }
9      }
10 }
```

Listagem 4-15 – Método confirmSerialization, da Classe *ScreenSerialization*.

O método *confirmSerialization* do objeto *component*, mostrado na Listagem acima, apenas limpa a sua lista de modificações (*modifiedProperties*).

CAPÍTULO 5 - Conclusão

Ao longo deste trabalho, ficou claro para nós que os arcabouços são de vital importância para o sucesso de grandes aplicações Web. Conforme constatamos, estas aplicações são sistemas distribuídos complexos e os arcabouços podem simplificar consideravelmente o seu desenvolvimento, apesar dos grandes desafios associados a eles.

A criação de interfaces com o usuário através de páginas HTML é muito trabalhosa. A utilização de componentes aumenta significativamente o grau de abstração usado pelo programador, que precisa se prender menos aos detalhes e dificuldades deste modelo. O arcabouço Crux, apresentado nesta dissertação, auxilia o desenvolvimento de novos componentes, montando um contexto para a criação dos mesmos, assim como para a troca de informações entre eles.

5.1 Lições Aprendidas

A construção do Crux nos proporcionou vivenciar os problemas relacionados a esta técnica de reuso. Esta, apesar de muito poderosa, é complexa e exige um profundo conhecimento a respeito do domínio envolvido, assim como dos padrões de projeto aplicáveis.

A percepção do forte relacionamento existente entre padrões e arcabouços foi outro grande aprendizado obtido. Um arcabouço implementa diversos padrões de projeto, relacionando-os de forma a montar um esqueleto para construções de aplicações. Estudá-los foi fundamental para nos auxiliar a desenhar a nossa solução.

5.2 Trabalhos Futuros

Na nossa visão, o Crux fornece um modelo de desenvolvimento que torna mais simples a criação de aplicações Web, se comparado às demais soluções analisadas. No entanto, esta comparação de soluções pode ser mais aprofundada.

Como primeiro trabalho futuro, consideramos a implementação de uma aplicação real e complexa, onde possamos validar, de forma mais concreta, a solução proposta, tentando quantificar os ganhos obtidos com a sua utilização.

É necessário, também, realizar experimentos para mensurar o grau de complexidade e a forma da curva de aprendizagem necessária para aprender a utilizar o Crux. Estes fatores são fundamentais, uma vez que um bom entendimento do arcabouço é necessário para auxiliar a determinar a aplicabilidade do mesmo e a estimar esforço de desenvolvimento para aplicações derivadas.

Outro ponto importante é a criação de um ambiente de desenvolvimento para aplicações baseadas no arcabouço. Este ambiente deve permitir a criação de telas de forma visual, sem a necessidade de se codificar as páginas HTML diretamente.

Deve ser criada, também, uma especificação para "tempo de desenho" dos componentes, a fim de determinar uma forma padrão para os mesmos se associarem ao ambiente e para informá-lo como devem ser tratados por ele, detalhando as propriedades e eventos que possuem, assim como outras informações eventuais.

Isto poderia ser feito criando-se uma interface para o componente, que seria usada pela ferramenta de criação de telas do ambiente de desenvolvimento. Assim como acontece para as classes de análise e de escrita da resposta, apresentadas no capítulo 3, a classe para tempo de desenho poderia ser informada no arquivo crux.xml.

Uma outra melhoria, que já estamos tratando, é dotar o Crux de um esquema para validação e formatação de dados.

Apêndice A – Padrões de Projeto.

Adapter	Converte uma interface de uma classe para outra interface esperada pelo cliente. Permite que classes com diferentes interfaces possam funcionar em conjunto.
Builder	Separa a construção de um objeto complexo da sua representação. Normalmente utilizado em conjunto com o padrão *Composite*.
Chain of Responsability	Evita o acoplamento entre o remetente e o destinatário de uma requisição, dando a mais de um objeto a chance de tratar a requisição. Os objetos são encadeados e a requisição é passada pela cadeia até que seja tratada por um (ou mais) objeto(s).
Command	Encapsula uma ação como um objeto. Este objeto encapsula a ação e seus parâmetros.
Composite	Compõe objetos em estruturas em árvore, para representar uma hierarquia parte-todo. O *Composite* permite a um cliente tratar objetos individuais e objetos compostos de uma mesma maneira.
Decorator	Associa responsabilidades adicionais a um objeto dinamicamente. *Decorators* fornecem uma alternativa flexível ao mecanismo de herança para extensão de funcionalidades.
Factory	Define uma interface para criação de objetos, mas deixa as suas subclasses decidirem qual classe instanciar.
Façade	Fornece uma interface única para um conjunto de interfaces em um subsistema. Fachadas definem uma interface de alto nível que tornam um subsistema mais fácil de utilizar.
Observer	Define uma dependência do tipo um-para-muitos entre objetos, de modo que quando um objeto muda de estado, todos os dependentes são notificados.
Prototype	Especifica como um objeto deve ser criado utilizando uma instância de protótipo e criando novos objetos através de cópias deste protótipo.
Proxy	Fornece um substituto para um objeto, de forma a controlar o acesso a ele.
Strategy	Define uma família de algoritmos, encapsula cada um e permite que qualquer um deles possa ser utilizado. *Strategy* permite que um algoritmo mude de forma transparente para os clientes que o utilizam.
Singleton	Garante que uma classe só terá uma instância e fornece um ponto de acesso único para ela.

Tabela A-1 – Padrões de Projeto (Gamma et al, 2002).

Composite View	Baseado no padrão *Composite*, é utilizado para montar a camada de visão a partir de várias sub-visões que podem ser dinamicamente inseridas na visão toda.
Front Controller	Define um controlador central, como ponto de entrada para tratar todas as requisições.
Service Locator	Encapsula o mecanismo de busca por componentes e serviços. Esconde os detalhes de implementação associados com estes mecanismos de busca.
Service To Worker	Centraliza o controle e o tratamento de uma requisição para recuperar o modelo da apresentação antes de passar o controle para visão. A visão gera uma resposta dinamicamente, baseada no modelo da apresentação.
Transfer Objects	Utilizar um objeto para carregar múltiplos dados entre camadas da aplicação. Estes normalmente são *Javabeans*.
Transfer Objects Assembler	Constrói modelos da aplicação como composições de *Transfer Objects*. Agrega múltiplos *Transfer Objects*.
View Helper	Utiliza as *Views* para encapsular código de formatação e *Helpers* para encapsular lógicas de processamento para apresentação. As classes *Helper* podem ser *Javabeans* ou marcações JSP.

Tabela A-2 – Padrões de Projeto (Deepak et al, 2001).

Apêndice B – Detalhes de Códigos.

Apêndice B.1 – Eventos e Componentes .

A Listagem B.1 ilustra um exemplo de declaração de cada um dos tipos de eventos nativos do Crux.

```
1  <span id="testeButton"
2      _type="button"
3      _value="AUTO"
4      _width="135px"
5      _onclick="serverHandler.metodoTeste|server-auto||synchronous"></span>
6  <span id="testeButton2"
7        _type="button"
8        _onclick="serverHandler.metodoTeste2|server-rpc|clientCallback.updateButtonClick|synchronous"
9        _onchange=""
10       _width="135px"
11       _value="RPC"></span>
12 <span id="testeButton4"
13       _type="button"
14       _value="CLIENT"
15       _width="135px"
16       _onclick="clientHandler.metodoTeste|client||assynchronous" ></span>
```

Listagem B-1 – Exemplo de Eventos no Crux.

A sintaxe para definição de um evento no Crux é a seguinte:

<chamada>|<tipo do evento>|<método de resposta>|<forma de sincronismo>

A chamada é formada pelo nome do tratador + "." + nome do método. Caso haja algum método de tratamento da resposta (válido para eventos Server-RPC), este segue a mesma sintaxe da chamada do tratador. Caso não seja informado o tipo do evento, o Crux assume o tipo Server-Auto como padrão.

A forma de sincronismo pode assumir um dos valores "*synchronous*" ou "as*synchronous*". Este valor informa se a interface deve aguardar pela resposta do evento (bloqueando ações do usuário) enviado, ou não.

Apêndice B.2 – Tratamento de Eventos no Servidor.

Para ilustrar este mecanismo, veja a Listagem B.2 abaixo que contém um trecho de código extraído da classe *BaseServlet*.

```
@ParametersBindPhase
@DispatchPhase
@RenderResponsePhase
public abstract class BaseServlet extends HttpServlet
{
    private LifeCycle lifeCycle;

    @Override
    public final void init(ServletConfig config) throws ServletException
    {
        super.init(config);
        lifeCycle = LifeCycle.getLifeCycle(getClass(), config.getServletContext());
    }

    @Override
    protected final void doPost(HttpServletRequest request, HttpServletResponse response)
                throws ServletException, IOException
    {
        lifeCycle.processRequest(request, response);
    }

    @Override
    protected final void doGet(HttpServletRequest request, HttpServletResponse response)
                throws ServletException, IOException
    {
        doPost(request, response);
    }
}
```

Listagem B-2 – *BaseServlet.*

Na Listagem B.2, pode-se perceber a chamada do método *LifeCycle.getLifeCycle()* logo no carregamento do *Servlet*. Este método constrói um objeto *LifeCycle*. Para isso, ele verifica as anotações contidas na classe recebida como parâmetro.

Na Listagem B.3, é mostrado um tratador de eventos para o evento do tipo Server-Auto. Este tratador substitui as fases de "disparo" e de "escrita da resposta" da classe *BaseServlet*.

```
@DispatchPhase(AutoDispatchPhase.class)
@RenderResponsePhase(AutoRenderResponsePhase.class)
public class EventServerAUTOServlet extends BaseServlet
{
    private static final long serialVersionUID = -7786236115297341839L;
}
```

Listagem B-3 – EventServerAutoServlet

É possível, também, adicionar mais fases diferentes ao ciclo de vida. Através das anotações *PreParametersBindPhase*, *PreDispatchPhase* e *PreRenderResponsePhase* é possível adicionar novas fases em qualquer ponto do ciclo de vida. Uma aplicação poderia modificar o tratador de eventos para o evento Server-Auto para, por exemplo, adicionar

uma fase de verificação de permissão e outra de auditoria, ambas devendo ser executadas, nesta ordem, antes da fase de disparo. Para isso, bastaria substituir o tratador pelo *Servlet* mostrado na Listagem B.4.

```
1 @PreDispatchPhase(values={MinhaFaseDeVerificacao.class,MinhaFaseDeAuditoria.class})
2 public class MeuTratadorDeEventos extends EventServerAutoServlet{
3     private static final long serialVersionUID = 5746563514570274342L;
4 }
```

Listagem B-4 – Exemplo de Tratador de Evento com Ciclo de Vida Personalizado.

Apêndice B.3 – Associação de uma Classe de Escrita a um Componente.

A Listagem B.5 ilustra como uma classe de escrita pode ser associada a um componente. Esta configuração é feita no arquivo crux.xml.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <crux xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3       xsi:noNamespaceSchemaLocation="../../Crux/xsd/crux.xsd">
4   <component id="textArea"
5              clientClass="br.ufmg.thiago.crux.ext.client.component.TextArea"
6              clientConstructorParams="new com.google.gwt.user.client.ui.TextArea()"
7              serverClass="br.ufmg.thiago.crux.ext.server.component.TextArea"
8              serverRendererClass="br.ufmg.thiago.crux.ext.server.component.TextAreaRenderer" />
9 </crux>
```

Listagem B-5 – Associação de uma Classe de Escrita a um Componente.

Apêndice B.4 – Exemplo de Resposta Enviada ao Motor Cliente.

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <span>
3      <span id="_components_">
4          <span id="nome"
5                _type="textBox"
6                _serverBind="pessoa.nome"
7                _width="670px"
8                _visible="false">
9          </span>
10     </span>
11     <span id="_dtos_">
12         <span id="pessoa.nome" value="Nome Alterado" ></span>
13     </span>
14 </span>
```

Listagem B-6 – Exemplo de Alteração na Árvore de Componentes Enviada pelo Crux.

Apêndice B.5 – Associação de uma Classe de Análise a um Componente.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <crux xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3        xsi:noNamespaceSchemaLocation="../../Crux/xsd/crux.xsd">
4    <component id="textArea"
5               clientClass="br.ufmg.thiago.crux.ext.client.component.TextArea"
6               clientConstructorParams="new com.google.gwt.user.client.ui.TextArea()"
7               serverClass="br.ufmg.thiago.crux.ext.server.component.TextArea"
8               serverParserClass="br.ufmg.thiago.crux.ext.server.component.TextAreaParser"
9               parserInput='jericho'
10              serverRendererClass="br.ufmg.thiago.crux.ext.server.component.TextAreaRenderer" />
11 </crux>
```

Listagem B-7 – Associação de uma Classe de Análise a um Componente.

Apêndice B.6 – Configuração do Mecanismo de Gerência de Estado das Árvores de Componentes.

```
1 controllerFactory=br.ufmg.thiago.crux.core.server.lifecycle.phase.dispatch.ControllerFactoryImpl
2 screenStateManager=br.ufmg.thiago.crux.core.server.screen.ScreenStateManagerClientImpl
3 debug=false
4 initializeControllersAtStartup=true
5 lookupWebInfOnly=false
6 enableHotDeployForScreens=true
7 pagesHome=/
8 developmentPublicDir=public
```

Listagem B-8 – Mecanismo de Gerência de Estado das Árvores de Componentes.

Apêndice B.7 – Associação de Geradores no Crux.

A Listagem B.9 mostra como os geradores de classes de registro de componentes e de tratadores de evento no cliente são associados aos tipos que precisam gerar. Esta configuração é feita no arquivo que descreve o módulo GWT do Motor Cliente (JSEngine.gwt.xml).

```
1  <module>
2      <!-- Specify the generator for registered client handlers.-->
3      <generate-with class="br.ufmg.thiago.crux.core.rebind.RegisteredClientEventHandlersGenerator">
4          <when-type-assignable class="br.ufmg.thiago.crux.core.client.event.RegisteredClientEventHandlers" />
5      </generate-with>
6
7      <!-- Specify the generator for registered components.     -->
8      <generate-with class="br.ufmg.thiago.crux.core.rebind.RegisteredComponentsGenerator">
9          <when-type-assignable class="br.ufmg.thiago.crux.core.client.component.RegisteredComponents" />
10     </generate-with>
11 </module>
```

Listagem B-9 – Associação dos Geradores.

Apêndice B.8 – Configuração da Classe de Fábrica dos Processadores de Evento.

```
<module>
    <replace-with class="br.ufmg.thiago.crux.core.client.event.EventProcessorFactoryImpl">
        <when-type-assignable class="br.ufmg.thiago.crux.core.client.event.EventProcessorFactory" />
    </replace-with>
</module>
```

Listagem B-10 – Configuração da Classe de Fábrica dos Processadores de Evento.

Apêndice B.9 – Busca de Classes por Anotação.

```java
public class ClassScanner{
    private static final Log logger = LogFactory.getLog(ClassScanner.class);
    private ServerMessages messages = (ServerMessages)MessagesFactory.getMessages(ServerMessages.class);
    private static final Lock lock = new ReentrantLock();
    private AnnotationDB scannerDB;
    private boolean indexBuilt = false;
    private static final ClassScanner instance = new ClassScanner();

    private ClassScanner() {
        scannerDB = new AnnotationDB();
        scannerDB.setIgnoredPackages(new String[]{"javax", "java", "sun", "com.sun",
                                                  "org.apache", "com.google", "javassist",
                                                  "br.com.sysmap.crux.core", "org.json",
                                                  "com.metaparadigm", "junit"});
        scannerDB.setScanFieldAnnotations(false);
        scannerDB.setScanMethodAnnotations(false);
        scannerDB.setScanParameterAnnotations(false);
        scannerDB.setScanClassAnnotations(true);
    }
    private void buildIndex(URL[] urls) throws ClassScannerException {
        if (indexBuilt) {
            return;
        }
        lock.lock();
        try {
            if (indexBuilt) {
                return;
            }
            if (logger.isInfoEnabled())logger.info(messages.annotationScannerBuildIndex());
            scannerDB.scanArchives(urls);
            indexBuilt = true;
        } catch (IOException e) {
            throw new ClassScannerException(messages.annotationScannerBuildIndexError(e.getLocalizedMessage()), e);
        }finally{
            lock.unlock();
        }
    }
    public Set<String> searchClassesByAnnotation(Class<? extends Annotation> annotationClass) {
        if (!indexBuilt) {
            throw new ClassScannerException(messages.annotationScannerIndexNotFound());
        }

        return scannerDB.getAnnotationIndex().get(annotationClass.getName());
    }
    public static ClassScanner getInstance(URL[] urls) {
        if (!instance.indexBuilt) {
            instance.buildIndex(urls);
        }
        return instance;
    }
}
```

Listagem B-11 – Classe *ClassScanner.*

Apêndice B.10 – Método de Análise Padrão para Componentes no Crux.

```
1  @Override
2  public void parse(Component component, Object element) {
3      Element elem = (Element) element;
4
5      Attributes attrs =  elem.getAttributes();
6      for (Object object : attrs) {
7          Attribute attr = (Attribute)object;
8          String attrName = attr.getName();
9
10         if (attrName.equals("id") || attrName.equals("_type")) {
11             continue;
12         }
13
14         if (attrName.startsWith("_on")) {
15             setEvent(component, attrName, attr.getValue());
16         } else if (attrName.equals("_class")) {
17             setProperty(component, "className", attr.getValue());
18         } else {
19             setProperty(component, attrName.substring(1), attr.getValue());
20         }
21     }
22 }
```

Listagem B-12 – Método de Análise da Classe *ComponentParserImpl*.

O atributo "*class*", utilizado em elementos HTML para referenciar uma definição feita em uma folha de estilos, é chamado de "*className*". Isto porque a propriedade "*class"* já existe em qualquer objeto Java, com uma semântica completamente diferente.

Apêndice B.11 – Gerenciadores de Estado da Árvore de Componentes.

```
1  public class ScreenStateManagerServerImpl extends AbstractScreenStateManager implements ScreenStateManager
2  {
3      private static final String SCREEN_KEY = "_SCREEN_KEY_";
4  
5      @Override
6      public Screen getScreen(String screenName, HttpServletRequest request) throws ScreenConfigException {
7  
8          Screen result = (Screen)request.getSession().getAttribute(SCREEN_KEY);
9          if (result == null)
10         {
11             result = getScreen(screenName);
12             request.getSession().setAttribute(SCREEN_KEY, result);
13         }
14 
15         return result;
16     }
17 
18     @Override
19     public boolean clientMustPreserveState()
20     {
21         return false;
22     }
23 }
```

Listagem B-13 – Classe ***ScreenStateManagerServerImpl*****.**

A seguir, é mostrado o código da classe *ScreenStateManagerClientImpl*.

```
1  public class ScreenStateManagerClientImpl extends AbstractScreenStateManager implements ScreenStateManager
2  {
3      @Override
4      public Screen getScreen(String screenName, HttpServletRequest request) throws ScreenConfigException
5      {
6          return getScreen(screenName);
7      }
8  
9      @Override
10     public boolean clientMustPreserveState()
11     {
12         return true;
13     }
14 }
```

Listagem B-14 – Classe ***ScreenStateManagerClientImpl*****.**

O método *getScreen(String)*, utilizado em ambas as classes, pertence à classe *AbstractScreenStateManager* e é mostrado na Listagem B.15.

```java
1  public abstract class AbstractScreenStateManager implements ScreenStateManager
2  {
3      protected ServerMessages messages = (ServerMessages)MessagesFactory.getMessages(ServerMessages.class);
4
5      public Screen getScreen(String id) throws ScreenConfigException
6      {
7          try
8          {
9              return (Screen)ScreenFactory.getInstance().getScreen(id).clone();
10         }
11         catch (CloneNotSupportedException e)
12         {
13             throw new ScreenConfigException(messages.screenStateManagerErrorCloningScreen(id, e.getLocalizedMessage()), e);
14         }
15     }
16 }
```

Listagem B-15 – Classe *AbstractScreenStateManager.*

Apêndice B.12 – Classes do Ciclo de Vida do Servlet para Tratamento de Eventos Server-RPC

```java
1  public class RPCDispatchPhase extends AbstractDispatchPhase
2  {
3      @Override
4      public void execute(PhaseContext context) throws PhaseException
5      {
6          try
7          {
8              DispatchData dispatchData = context.getDispatchData();
9              String evtCall = dispatchData.getEvtCall();
10             String[] call = RegexpPatterns.REGEXP_DOT.split(evtCall);
11
12             Object controller = getController(call[0]);
13             bindControllerParameters(controller, dispatchData);
14
15             Class<?>[] parametersTypes = {};
16             Object[] parametersValues = {};
17             Object result = dispatch(controller, call[1], parametersTypes, parametersValues);
18             context.setCycleResult(new JsonResult(JsonResult.CODE_SUCCESS,JsonSerializer.objMarshall(result)));
19             context.setDto(controller);
20         }
21         catch (Throwable e)
22         {
23             context.setCycleResult(null);
24             throw new PhaseException(e);
25         }
26     }
27 }
```

Listagem B-16 – Classe *RPCDispatchPhase*.

```
1  public class RPCRenderResponsePhase extends AbstractRenderResponsePhase implements Phase {
2      private static final Log logger = LogFactory.getLog(RPCRenderResponsePhase.class);
3      private ServerMessages messages = (ServerMessages)MessagesFactory.getMessages(ServerMessages.class);
4      @Override
5      public void execute(PhaseContext context) throws PhaseException {
6          if (logger.isDebugEnabled()) logger.debug("RPCResponseRenderPhase => rendering response");
7          try {
8              JsonResult result = handleErrors(context.getPhaseException());
9              if (result == null) {
10                 result = (JsonResult)context.getCycleResult();
11             }
12             if (result != null) {
13                 result.setDtoChanges(getDtoChanges(context.getDto(), context.getDispatchData()));
14                 context.getResponse().setContentType("text/plain;charset=utf-8");
15                 OutputStream out = context.getResponse().getOutputStream();
16                 byte[] bout = result.toString().getBytes("UTF-8");
17                 context.getResponse().setIntHeader("Content-Length", bout.length);
18                 context.getResponse().setHeader("Connection", "keep-alive");
19                 context.getResponse().setHeader("Cache-Control", "no-store, no-cache, must-revalidate");
20                 context.getResponse().addHeader("Cache-Control", "post-check=0, pre-check=0");
21                 context.getResponse().setHeader("Pragma", "no-cache");
22                 context.getResponse().setHeader("Expires", "-1");
23
24                 out.write(bout);
25                 out.flush();
26                 out.close();
27             }
28         } catch (Exception e) {
29             throw new PhaseException(messages.rpcResponseRenderPhaseError(e.getLocalizedMessage()), e);
30         }
31     }
32     protected JsonResult handleErrors(PhaseException phaseException) {
33         if (phaseException != null) {
34             return new JsonResult(JsonResult.CODE_ERR_METHOD, phaseException.getLocalizedMessage());
35         }
36         return null;
37     }
38     protected String getDtoChanges(Object dto, DispatchData dispatchData) {
39         StringWriter strWriter = new StringWriter();
40         PrintWriter writer = new PrintWriter(strWriter);
41         renderDtos(dto, dispatchData, writer);
42         String updateScreen = strWriter.toString();
43         if (updateScreen.length() > 0) {
44             return "<?xml version=\"1.0\" encoding=\"UTF-8\"?><span><span id=\"_dtos_\">"+updateScreen+"</span></span>";
45         }
46         return null;
47     }
48 }
```

Listagem B-17 - Classe *RPCRenderResponsePhase.*

O método *renderDtos* pertence à classe abstrata *AbstractRenderResponsePhase*, mostrada na Listagem B.18.

```java
public abstract class AbstractRenderResponsePhase implements Phase
{
    private static final Log logger = LogFactory.getLog(AbstractRenderResponsePhase.class);
    protected void renderDtos(Object dto, DispatchData dispatchData, PrintWriter writer)
    {
        for (String key : dispatchData.getParameters())
        {
            try
            {
                Object propValueAnt = dispatchData.getParameter(key);
                String propValue = BeanUtils.getProperty(dto, key);
                if (propValue == null && propValueAnt != null)
                {
                    writer.println("<span id=\""+key+"\" value=\"\" ></span>");
                }
                else if (propValue != null && (propValueAnt == null || !propValueAnt.equals(propValue)))
                {
                    writer.println("<span id=\""+key+"\" value=\""+HtmlUtils.filterValue(propValue)+"\" ></span>");
                }
            }
            catch (Throwable e)
            {
                if (logger.isDebugEnabled()) logger.debug("DTO não possui a propriedade "+key);
            }
        }
    }
}
```

Listagem B-18 – Classe *AbstractRenderResponsePhase.*

Apêndice B.13 – Classes do Ciclo de Vida do Servlet para Tratamento de Eventos Server-Auto

```
1  public class AutoDispatchPhase extends AbstractDispatchPhase
2  {
3      private static final Log logger = LogFactory.getLog(AutoDispatchPhase.class);
4      private ServerMessages messages = (ServerMessages)MessagesFactory.getMessages(ServerMessages.class);
5  
6      @Override
7      public void execute(PhaseContext context) throws PhaseException
8      {
9          try
10         {
11             DispatchData dispatchData = context.getDispatchData();
12             String evtCall = dispatchData.getEvtCall();
13 
14             String[] call = RegexpPatterns.REGEXP_DOT.split(evtCall);
15 
16             Object controller = getController(call[0]);
17             bindControllerParameters(controller, dispatchData);
18             try
19             {
20                 BeanUtils.copyProperty(controller, "screen", context.getScreen());
21             }
22             catch (Throwable e)
23             {
24                 if (logger.isInfoEnabled()) logger.info(messages.dispatchPhasePropertyNotBound("screen"));
25             }
26 
27             Class<?>[] parametersTypes = {};
28             Object[] parametersValues = {};
29             dispatch(controller, call[1], parametersTypes, parametersValues);
30             context.setCycleResult(context.getScreen());
31             context.setDto(controller);
32         }
33         catch (Throwable e)
34         {
35             throw new PhaseException(e);
36         }
37     }
38 }
```

Listagem B-19 – Classe *AutoDispatchPhase.*

```java
public class AutoRenderResponsePhase extends AbstractRenderResponsePhase implements Phase {
    private static final Log logger = LogFactory.getLog(AutoRenderResponsePhase.class);
    private ServerMessages messages = (ServerMessages)MessagesFactory.getMessages(ServerMessages.class);
    public void execute(PhaseContext context) throws PhaseException {
        if (logger.isDebugEnabled()) logger.debug("AutoResponseRenderPhase => rendering response");
        try {
            context.getResponse().setContentType("text/plain;charset=iso-8859-1");
            PrintWriter writer = new PrintWriter(context.getResponse().getOutputStream());
            writer.println("<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?><span>");
            if (handleErrors(writer, context.getPhaseException())) {
                writer.println("<span id=\"_components_\">");
                renderComponents(context.getScreen(), writer);
                writer.println("</span>");
                writer.println("<span id=\"_dtos_\">");
                renderDtos(context.getDto(), context.getDispatchData(),writer);
                writer.println("</span>");
            }
            writer.println("</span>");
            writer.close();
        } catch (IOException e) {
            throw new PhaseException(messages.autoResponseRenderPhaseError(e.getLocalizedMessage()), e);
        }
    }
    protected void renderComponents(Screen screen, PrintWriter writer) {
        Iterator<Component> components = screen.iterateComponents();
        while (components.hasNext()) {
            try {
                renderComponent(components.next(), writer);
            }catch (Throwable e) {
                logger.error(messages.autoResponseRenderComponentError(e.getLocalizedMessage()), e);
            }
        }
    }
    protected void renderComponent(Component component, PrintWriter writer) {
        ComponentRenderer renderer = null;
        if (component.isDirty()) {
            renderer = ComponentConfig.getComponentRenderer(component.getType());
            renderer.renderStart(component, writer);
        }
        if (component instanceof Container) {
            Container parent = (Container)component;
            Iterator<Component> components = parent.iterateComponents();
            while (components.hasNext()) {
                renderComponent(components.next(), writer);
            }
        }
        if (component.isDirty()) {
            renderer.renderEnd(component, writer);
        }
    }
}
```

Listagem B-20 – Classe *AutoRenderResponsePhase.*

Apêndice B.14* – Adapter para *Desenho de Componentes do GWT no Crux.

```
1  protected Event getComponentEvent(Element element, String evtId){
2      String evt = element.getAttribute(evtId);
3      return EventFactory.getEvent(evtId, evt);
4  }
5  public void render(Element element) {
6      renderAttributes(element);
7      attachEvents(element);
8  
9  }
10 protected void renderAttributes(Element element){
11     String width = element.getAttribute("_width");
12     if (width != null && width.trim().length() > 0)
13         widget.setWidth(width);
14 
15     String height = element.getAttribute("_height");
16     if (height != null && height.trim().length() > 0)
17         widget.setHeight(height);
18 
19     if (widget instanceof HasHTML){
20         String innerHtml = element.getInnerHTML();
21         if (innerHtml != null && innerHtml.trim().length() > 0){
22             ((HasHTML)widget).setHTML(innerHtml);
23             element.setInnerHTML("");
24         }
25     }
26     if (widget instanceof HasText){
27         String text = element.getAttribute("_value");
28         if (text != null && text.trim().length() > 0)
29             ((HasText)widget).setText(text);
30     }
31     // Outras propriedades
32 }
33 protected void attachEvents(Element element)
34 {
35     if (widget instanceof SourcesClickEvents){
36         final Event event = getComponentEvent(element, EventFactory.EVENT_CLICK);
37         if (event != null){
38             ClickListener listener = new ClickListener(){
39                 public void onClick(Widget sender) {
40                     EventFactory.callEvent(event, getId());
41                 }
42             };
43             ((SourcesClickEvents)widget).addClickListener(listener);
44             eventListeners.put(EventFactory.EVENT_CLICK, listener);
45         }
46     }
47     // Outros eventos
48 }
```

Listagem B-21 – Método *render*, da Classe *ComponentBase*.

Apêndice B.15 – Métodos para Seriação dos Parâmetros da Tela.

```
1  protected boolean buildBeansPostData(Screen screen, StringBuilder builder) {
2      boolean first = true;
3      for (String beanProperty : screen.beansProperties.keySet()) {
4          if (!first)
5              builder.append("&");
6          first = false;
7          buildPostParameter(builder, beanProperty, (String)screen.getBeanProperty(beanProperty));
8      }
9      return !first;
10 }
11 protected boolean buildStructurePostData(Screen screen, StringBuilder builder){
12     boolean first = true;
13     for (Component component : screen.components.values()) {
14         if (!first)
15             builder.append("&");
16         first = false;
17         component.serialize(builder);
18     }
19     return !first;
20 }
21 protected void updateDTOs(Screen screen, Element dtosElement) {
22     NodeList children = dtosElement.getChildNodes();
23     for (int i = 0; i < children.getLength(); i++){
24         if (children.item(i).getNodeType() == Node.ELEMENT_NODE){
25             Element element = (Element)children.item(i);
26             String key = element.getAttribute("id");
27             String value = element.getAttribute("value");
28             screen.setBeanProperty(key, value);
29         }
30     }
31 }
32 protected void updateComponents(Screen screen, Element componentsElement) {
33     NodeList children = componentsElement.getChildNodes();
34     for (int i = 0; i < children.getLength(); i++) {
35         if (children.item(i).getNodeType() == Node.ELEMENT_NODE) {
36             Element element = (Element)children.item(i);
37             String componentId = element.getAttribute("id");
38             Component component = screen.getComponent(componentId);
39             if (component == null){
40                 // Report error.
41             } else{
42                 component.update(element);
43             }
44         }
45     }
46 }
```

Listagem B-22 – Métodos para Seriação dos Parâmetros da Tela, da classe *ScreenSerialization.*

Referências

Alexander, C., A Pattern Language, Oxford University Press, 1977.

Baresi, L., Morasca, S., Three Empirical Studies on Estimating the Design Effort of Web Applications, ACM Transactions on Software Engineering and Methodology, Vol. 16, No. 4, Article 15, September 2007.

Baresi, L., Garzotto, F., Paolini, P., From Web Sites to Web Applications: New Issues for Conceptual Modeling. In Proc. of the ER'00 Workshop on World Wide Web and Conceptual Modeling, Salt Lake City (USA), 2000.

Bosch, J., Molin, P., Mattsson, M., and Bengtsson, P. 2000. Object-oriented framework-based software development: problems and experiences. *ACM Comput. Surv.* 32, 1es (Mar. 2000), 3. DOI= http://doi.acm.org/10.1145/351936.351939

Brugali, D., Sycara, K., Frameworks and Pattern Languages: an Intriguing Relationship, ACM Computing Surveys (CSUR), 2000

Buschmann, F., Meunier, R., Rohnert, H., Sommerland, P. & Stal, M (1996): Pattern-Oriented Software Architectur A System of Patterns, John Wiley & Sons, 1996

Ceri, S., Fraternali, P., Bongio, A., Web Modeling Language (WebML): a Modeling Language for Designing Web Sites, WWW9 Conference, Amsterdam, 2000.

Chang, P., Kim, W., Agha, G., An Adaptive Programming Framework for Web Applications, Symposium on Applications and the Internet, pp. 152-159, ISBN: 0-7695-2068-5, 2004

Conallen, J., Building Web Applications with UML, second edition Addison-Wesley, ISBN: 0-20-173038-3, 2002.

Conte, T., Mendes, E., Travassos, G. H., Processos de Desenvolvimento para Aplicações Web: Uma Revisão Sistemática, Brazilian Symposium on Multimedia and the Web (WebMedia), December, 2005

Deepak, A., Crupi, J., Malks, D., Core J2EE Patterns: Best Practices and Design Strategies, 2 ed., (Upper Saddle River, NJ: Prentice Hall PTR, 2001; ISBN: 0130648841)

Diaz, P., Montero, S., Aedo, I, Modelling hypermedia and web applications: the Ariadne Development Method, Information Systems, Article in Press, 2004.

Dudney, B., Lehr, J., Willis, B., Mattingly, L., Mastering JavaServer™ Faces, Wiley Publishing Inc., Indianapolis, Indiana, 2004, ISBN: 0-471-46207-1.

Eclipse, www.eclipse.org, 2008

Fayad, M. E, Introduction to the Computing Surveys' Electronic Symposium on Object-Oriented Application Frameworks, ACM Computing Surveys, Vol.32, No. 1, March 2000.

Fayad, M. E, Schmidth, D. C, Special Issue on Object-Oriented Application Frameworks, Communications of the ACM, Vol. 40, No. 10, October 1997.

Fingar, P. 1996. Blueprint for Business Objects. Multimedia, New York, NY.

Firefox, http://mozillalinks.org/wp/2007/01/planned-features-for-firefox-3/, visitado em 06/01/2008.

Flash, http://www.adobe.com/products/flash/, 2008

Fons, J., Pelechano, V., Albert, M., Pastor, O., Development of Web Applications from Web Enhanced Conceptual Schemas, Conference on Conceptual Modeling (ER), Is International, 22nd, Il-Yeol Song, Stephen W. Liddle, Tok Wan Ling, Peter Scheuermann, Springer-Verlag, Lecture Notes in Computer Science, 3-540-20299-4, 2813, 2003.

Fowler, D. Rice, M. Foemmel, E. Hieatt, R. Mee, and R. Stafford, Patterns of Enterprise Application Architecture. Reading, Massachusetts: Addison-Wesley, 2003

Gamma, E., Helm, R., Johnson & R., Vlissides, J., Padrões de Projeto: Soluções reutilizáveis de software orientados a objetos. Bookman, 2002.

Garret, J. J., 2005, Ajax: A New Approach to Web Applications, http://www.adaptivepath.com/publications/essays/archives/000385.php

Gellersen, H., Wicke, R., Gaedke, M., WebComposition: An Object-Oriented Support System for the WebEngineering Lifecycle," Computer Networks and ISDN Systems, vol. 29, no. 8-13, pp. 1429-1437, 1997.

Ginige, A., Murugesan, S., Guest editors' introduction: Web engineering—an introduction. IEEE MultiMedia 8, 1, 14–18, 2001.

Gmail, http://mail.google.com, 2008.

GWT, Google Web Toolkit – Build AJAX apps in the Java language, http://code.google.com/webtoolkit/, 2008

Hanson, R., Tacy, A., GWT in Action: Easy Ajax With Google Web Toolkit, Manning Publications, Greenwich, 2007, ISBN 1-933988-23-1.

Husted, T., Dumoulin, C., Franciscus, G., Winterfeldt, D., Struts in Action – Building web applications with the leading Java framework, Manning Publications, Greenwich, 2003.

Javassist, http://www.csg.is.titech.ac.jp/~chiba/javassist/, 2008

Jazayeri, M., Some Trends in Web Application Development, Future of Software Engineering (FOSE'07), 2007.

Jericho, http://jerichohtml.sourceforge.net/doc/index.html, 2008.

Johnson, R., Expert One-on-One J2EE Design and Development. Reading: Wiley Publishing Inc., 2002.

Johnson, R. E, Documenting Frameworks using Patterns, OOSPSLA, 1992, pp. 63-76.

Johnson, R. E, Frameworks = (Components + Patterns), Communications of the ACM, October 1997/ Vol. 40, No. 10.

JSF, JavaServer Faces Specification, http://java.sun.com/javaee/javaserverfaces/download.html, 2004

JSON, http://www.json.org/, 2008

JSP, http://java.sun.com/products/jsp/, 2008

JUnit, http://www.junit.org, 2008

Koch, N., Kraus, A., The expressive Power of UML-based Web Engineering. Second Int. Worskhop on Web-oriented Software Technology (IWWOST´02), 2002.

Lalonde, W (1994): Discovering Smalltalk. The Benjamin/Cummings Publishing Company, Inc, 1994.

Lowe, D., Henderson-Sellers, B, Characteristics of Web Development Processes, SSGRR-2001: INFRASTRUCTURE FOR E-BUSINESS, E-EDUCATION, AND E-SCIENCE

Maps, http://maps.google.com.br , 2008.

Mann, K. D., Javaserver Faces In Action, Manning Publications, Greenwich, 2005, ISBN 1-932394-11-7

Markiewicz, M. E. and de Lucena, C. J. 2001. Object oriented framework development. Crossroads 7, 4 (Jul. 2001), 3-9

Maurer, P.M, 1999, Varanasi, M.; Katkoori, S.; Wai-Kai Mak; Component-Level Programming: A Revolution in Software Technology

McLellan, D., 2005, Very Dynamic Web Interfaces, http://www.xml.com/pub/a/2005/02/09/xml-http-request.html

McClanahan, C., Burns, E., Kitain, R., JavaServer Faces Specification, http://java.sun.com/javaee/javaserverfaces/, 2004

Nielsen, J., User interface directions for the Web, Communications of the ACM, Vol. 42, No. 1, January 1999.

Nielsen, J., 1998, Does Internet=Web?, http://www.useit.com/alertbox/980920.html

Nielsen, J., 1997, The Difference Between Web Design and GUI Design, http://www.useit.com/alertbox/9705a.html

Nielsen, J., 2002, Flash and Web-Based Applications, http://www.useit.com/alertbox/

Nielsen, J., 2000, Flash: 99% Bad, http://www.useit.com/alertbox/20001029.html

Nielsen, J., 1997, When to Open Web-Based Applications in a New Window, http://www.useit.com/alertbox/9710b.html

Pree, W., Design Patterns for Object-Oriented Software Development, Addison-Wesley, 1995.

Ramachandran, V., Design Patterns for Building Flexible and Maintainable J2EE Applications. http://java.sun.com/developer/technicalArticles/J2EE/despat/index.html, 2002.

Scannotation, http://scannotation.sourceforge.net/, 2008.

Schmidt, D. C. and Buschmann, F. 2003. Patterns, frameworks, and middleware: their synergistic relationships. In *Proceedings of the 25th international Conference on Software Engineering* (Portland, Oregon, May 03 - 10, 2003). International Conference on Software Engineering. IEEE Computer Society, Washington, DC, 694-704.

Schwabe, D., Esmeraldo, L., Rossi, G., Lyardet, F., Engineering Web Applications for Reuse, 2001.

Schwabe, D., Rossi, G., Barbosa, S., Systematic Hypermedia Design with OOHDM, ACM Conference on Hypertext, 1996.

Servlet, http://java.sun.com/products/servlet/, 2008

Sourceforge, http://sourceforge.net/, 2008

Sparks, S., Benner, K., Faris, C., Managing object-oriented framework reuse. IEEE Computer, 53-61, 1996.

Spring, http://www.springframework.org/, 2008

Struts, http://struts.apache.org/, 2008

Sun, http://java.sun.com/, 2008

Szyperski, C. (1997): Component Software: Beyond Object-Oriented Programming, Addison-Wesley, ISBN 0-201-17888-5

W3C, 2004, Document Object Model (DOM) Level 3 Core Specification, http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/introduction.html

W3C, 2005, Document Object Model (DOM), http://www.w3.org/DOM/

W3C, 2003, Document Object Model (DOM) Level 2 HTML Specification, http://www.w3.org/TR/DOM-Level-2-HTML/

W3C, 1999a, HTML 4.01 Specification, http://www.w3.org/TR/html4/

W3C, 1998, Cascading Style Sheets, level 2 CSS2 Specification, http://www.w3.org/TR/CSS2/

W3C, 1999b, Hypertext Transfer Protocol -- HTTP/1.1, http://www.w3.org/Protocols/rfc2616/rfc2616.html

WEISS, A., The web designer's dilemma: when standards and practice diverge. netWorker, New York, volume 10, issue 1, p. 18- 25, mar. 2006

Xu, L., Xu, B., A Framework for Web Applications Testing, International Conference on Cyberworlds (CW'04), 2004.

Zhang, J., Buy, U., 2003, A Framework for the Efficient Production of Web Applications, Eighth IEEE International Symposium on Computers and Communication (ISCC).

Zhao, W., Chen, J., 1999, CoOWA: A Component Oriented Web Application Model, TOOLS 31, Proceedings of the Technology, of Object-Oriented Languages and Systems.