Universidade Federal de Pernambuco
Centro de Informática

Pós-graduação em Ciência da Computação

Desenvolvimento de Sistemas *Web* em Java: *Frameworks*, Padrões de Projeto e Diretrizes para a Camada de Apresentação

por
Gibeon Soares de Aquino Júnior

Dissertação de Mestrado

Recife, 21 de maio de 2002

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA

Gibeon Soares de Aquino Júnior

Desenvolvimento de Sistemas *Web* em Java: *Frameworks*, Padrões de Projeto e Diretrizes para a Camada de Apresentação

Este trabalho foi apresentado à Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito para a conclusão do mestrado na mesma.

ORIENTADOR:
Prof. Paulo Henrique Monteiro Borba

Agradecimentos

A Deus, por me ajudar e me dar forças para realizar este trabalho. Aos meus pais, pela educação recebida e investimento nos meus estudos. A minha esposa, Silvia, pelo companheirismo nos momentos mais difíceis.

Ao meu orientador, Paulo Borba, por me guiar nas pesquisas da tese, realizar revisões detalhadas na dissertação, me acompanhar desde a graduação e sempre procurar transferir sua experiência e conhecimento. Aos professores da banca de defesa André Santos e Rossana Andrade pelas sugestões e críticas construtivas que melhoraram sensivelmente este trabalho.

Ao Centro de Estudos e Sistemas Avançados do Recife – CESAR, pelo apoio técnico, pessoal e financeiro. Principalmente a Valença, Denise e Cuca que sempre apostaram na minha capacidade e me incentivaram do início ao fim do meu mestrado. Aos amigos do CESAR, por aplicarem minhas idéias e sugestões no desenvolvimento dos sistemas *Web* da empresa e, principalmente, pelas críticas e elogios recebidos, que me ajudaram a refinar e melhorar as soluções propostas.

Ao Centro de Informática, pelo suporte técnico e acadêmico, e a todos os professores deste Centro pela transferência de conhecimento.

A todos os meus amigos e familiares pela participação na minha vida e por sempre torcerem por mim.

Especialmente, ao meu filho, João Victor, por ser a principal razão desta conquista.

Obrigado.

Resumo

Com o crescimento e popularização da Internet estão sendo desenvolvidos, cada vez mais, sistemas para *Web*. Ao contrário do que se pensa, construir aplicações *Web* é uma tarefa difícil devido às características especiais impostas por este tipo de ambiente, tais como Concorrência, Escalabilidade, Segurança e Disponibilidade. O uso da plataforma Java para desenvolvimento de sistemas *Web* tem se mostrado bastante adequado, por isso empresas em todo o mundo estão adotando esta tecnologia para este fim.

O objetivo deste trabalho é propôr e descrever ferramentas que auxiliem o desenvolvimento de sistemas *Web* em Java, melhorando a produtividade dos desenvolvedores e a qualidade do *software* produzido. Este objetivo foi alcançado através da documentação de um conjunto de padrões de projeto para sistemas *Web*, definição de diretrizes de desenvolvimento e criação de um *framework* para estruturação de componentes *Web*.

Os padrões de projeto, catalogados neste trabalho, documentam soluções recorrentes para problemas específicos e inerentes a sistemas *Web*. As diretrizes guiam a implementação de algumas características intrínsecas a sistemas *Web*, possibilitando que as pessoas envolvidas no desenvolvimento destes tipos de sistemas tomem decisões que melhor se adeqüem aos requisitos do seu tipo de aplicação. O *framework* facilita o desenvolvimento da camada de apresentação das aplicações *Web* desenvolvidas em Java [3], aumentando a produtividade, o reuso e o desacoplamento entre o código de processamento das requisições e o código de montagem de páginas, sem acrescentar complexidade de entendimento, desenvolvimento e distribuição ao sistema.

Abstract

With the widespread use of the Internet, an increasing number of web-based information systems are being developed. On the contrary of what is thought, the development of Web applications is a difficult task because there are specials characteristics imposed by this type of environment, such as Concurrency, Scalability, Security and Avaiability. The use of the Java plataform for Web systems development has been very adequate, therefore companies are mostly adopting this technology for this use.

The objective of this work is to propose and describe tools for assisting Java Web systems development, in this way improving the productivity of developers and the quality of the produced software. This objective is reached by means of design patterns for Web systems, development guidelines, and a framework of Web components.

The design patterns catalogued here registers repetitive solutions to particular Web systems problems. The guidelines direct the implementation of some specialized characteristics of Web systems, making possible that people involved with development of this kind of systems choose the best way to resolve their applications problems. The framework supports the development of Web systems presentation layers [3], increasing productivity, reuse and uncoupling between the request process and the Web page formatting code.

Índice

1 Introdução **1**
- 1.1 Camadas da plataforma J2EE . . . 3
- 1.2 Tecnologias para desenvolvimento de sistemas *Web* em Java . . . 4
  - 1.2.1 *Servlets* . . . 5
  - 1.2.2 JSP (*Java Server Pages*) . . . 8
- 1.3 Objetivos . . . 9
- 1.4 Organização da Dissertação . . . 10

2 Padrões de Projeto para sistemas *Web* **11**
- 2.1 *Web Interceptor* . . . 12
- 2.2 *Web Handlers* . . . 20
- 2.3 *Web Compiler* . . . 33
- 2.4 *Super Component* . . . 40

3 *Framework* para implementação de sistemas *Web* **46**
- 3.1 Introdução . . . 47
- 3.2 Características do *framework* . . . 48
  - 3.2.1 Ciclo de Vida . . . 49
  - 3.2.2 Implementação do *Framework* . . . 50
- 3.3 Guia de utilização do *framework* . . . 63
  - 3.3.1 Definindo os *handlers* de processamento . . . 63
  - 3.3.2 Definindo os *handlers* de apresentação . . . 67
  - 3.3.3 Definindo os serviços do sistema . . . 70
  - 3.3.4 Tratando erros . . . 71
  - 3.3.5 Estendendo o Controlador de *handlers* . . . 72
  - 3.3.6 Customizando a definição de serviços . . . 73
- 3.4 Usos conhecidos . . . 74

4 Diretrizes para desenvolvimento da camada de apresentação de sistemas *Web* **75**
- 4.1 Formatação da apresentação . . . 76
  - 4.1.1 *Servlet* puro . . . 77
  - 4.1.2 Processamento de *templates* . . . 78
  - 4.1.3 Tratamento de elementos HTML como objetos Java . . . 79
  - 4.1.4 JSP puro . . . 81
  - 4.1.5 JSP com *View Helpers* . . . 82
  - 4.1.6 Uso de tecnologias baseadas em XML . . . 83

4.1.7 Análise comparativa entre as abordagens . . . 85
4.2 Validação de dados da requisição . . . 86
4.2.1 Abordagem Simplificada . . . 86
4.2.2 Abordagem Estruturada . . . 88
4.2.3 Abordagem Estruturada Customizável . . . 93
4.2.4 Análise comparativa entre as abordagens de validação . . . 95
4.3 Arquitetura dos componentes *Web* . . . 96
4.3.1 Modelo monolítico . . . 97
4.3.2 Modelo orientado a operação . . . 97
4.3.3 Modelo orientado a página . . . 99
4.3.4 Modelo baseado em eventos . . . 100
4.3.5 Análise comparativa entre as abordagens . . . 100
4.4 Persistência do estado do cliente . . . 102
4.4.1 Usando hiddens . . . 103
4.4.2 Usando cookies . . . 106
4.4.3 Usando sessões . . . 108
4.4.4 Análise comparativa entre as abordagens . . . 111

5 Conclusões **114**
5.1 Contribuições e Conclusões . . . 115
5.1.1 Padrões de Projeto para sistemas *Web* . . . 115
5.1.2 *Framework* para implementação de sistemas *Web* . . . 116
5.1.3 Diretrizes e Técnicas para desenvolvimento de sistemas *Web* . . . 116
5.2 Trabalhos Relacionados . . . 117
5.2.1 *Framework Struts* . . . 118
5.2.2 J2EE *Patterns* . . . 118
5.2.3 *Técnicas de validação em Java* . . . 120
5.2.4 Geração e Execução de Testes de Aceitação de Sistemas *Web* . . . 120
5.3 Trabalhos Futuros . . . 120
5.3.1 Diretrizes para desenvolvimento de sistemas *Web* . . . 120
5.3.2 Padrões de Projetos . . . 121
5.3.3 *Framework* para sistemas *Web* . . . 121

A Esquema do Arquivo de Configuração do *Framework Web Handlers* 124

Lista de Figuras

1.1 Camadas da plataforma J2EE. 3
1.2 Total de linhas de código associado a cada camada do sistema Notitia. . . 5
1.3 Ciclo de vida dos *Servlets*. 7
1.4 Característica *Multithreading* dos *Servlets*. 7
1.5 Processo de transformação, compilação, carga e execução dos JSPs. . . . 9

2.1 Uso do *Web Interceptor* no exemplo do sistema bancário. 14
2.2 Exemplo de uma cadeia de Interceptadores. 15
2.3 Estrutura do padrão *Web Interceptor*. 16
2.4 Diagrama de seqüência dos componentes do padrão. 16
2.5 Mapa navegacional simples do sistema bancário. 20
2.6 Mapa navegacional completo do sistema bancário. 21
2.7 Composições possíveis de *handlers* de apresentação e processamento. . . . 23
2.8 Diagrama de classes do padrão *Web Handlers*. 24
2.9 Diagrama de seqüência dos componentes do padrão *Web Handlers*. . . . 25
2.10 Diagrama de classes refinado dos componentes do *Web Handlers*. 27
2.11 Estrutura do padrão no ambiente de *servlets*. 28
2.12 Estrutura do padrão refinada e no ambiente de *servlets*. 29
2.13 Uso do padrão *Web Compiler*. 35
2.14 Estrutura do padrão *Web Compiler*. 36
2.15 Diagrama de seqüência dos componentes do padrão *Web Compiler*. . . . 37
2.16 Exemplo de implementação do padrão *Web Compiler*. 38
2.17 Diagrama de classes do padrão Super Component. 43
2.18 Diagrama de seqüência dos componentes do padrão no momento da inicialização de um componente. 44

3.1 Ciclo de vida dos *Web Handlers*. 49
3.2 Diagrama UML dos componentes do *framework*. 50
3.3 Diagrama das classes básicas do *framework* relacionadas com os *handlers* de processamento. 53
3.4 Diagrama das classes básicas do *framework* relacionadas com os *handlers* de apresentação. 55
3.5 Especificação do serviço de login. 56
3.6 Diagrama UML da classe Servico. 57
3.7 Diagrama UML da interface ConfiguracaoHandlers. 57
3.8 Diagrama de classes das entidades que implementam o montador de serviço. 57
3.9 Diagrama de seqüência do processo de recuperação das configurações de serviços. 59

3.10 Diagrama de seqüência da execução de cada requisição *Web*. 60
3.11 Diagrama de classes do *Handler* de erro. 62

4.1 Definição da classe `ValidationRule` em UML. 89
4.2 Hierarquia das classes básicas que representam as regras de validação. . . 91
4.3 Diagrama UML da classe abstrata Validator. 92
4.4 Modelo monolítico de estruturação de sistemas *Web*. 97
4.5 Modelo de estruturação orientado a operações. 98
4.6 Exemplo de um sistema bancário. 98
4.7 Modelo de estruturação orientado a páginas. 99
4.8 Exemplo do sistema bancário simplificado. 102
4.9 O mecanismo de persistir o estado do cliente através de *hiddens*. 104

Lista de Tabelas

4.1 Comparações entre as abordagens de formatação da apresentação. 85
4.2 Comparações entre as abordagens de validação de dados de entrada. . . . 95
4.3 Comparações entre as abordagens de arquitetura para sistemas *Web*. . . 101
4.4 Comparações entre as abordagens de persistência do estado do cliente *Web*.112

Capítulo 1

Introdução

Neste capítulo motivamos a necessidade de técnicas, diretrizes, padrões e frameworks que auxiliem o desenvolvimento de sistemas Web. Também apresentamos uma descrição sucinta das tecnologias para desenvolvimento de sistemas Web em Java.

Com o crescimento e popularização da Internet estão sendo desenvolvidos, cada vez mais, sistemas para *Web*. Atualmente existem muitos sistemas, de todas as formas e com objetivos distintos. Ao contrário do que se pensa, construir aplicações na *Web* é uma tarefa difícil, pois estes tipos de sistemas possuem características especiais que tornam o seu desenvolvimento mais complexo do que o de aplicações *desktop*, tais como:

- **Concorrência** – Os sistemas *Web* podem ser acessados por muitos usuários simultaneamente. Isto exige um maior controle por parte da aplicação para garantir que o sistema se comporte de forma eficiente e segura. A definição de métodos que guiem o tratamento da concorrência em sistemas *Web* é essencial para execução de programas em ambiente concorrente [60];

- **Escalabilidade** – Na maioria dos casos é difícil fazer uma estimativa do número de usuários de um sistema *Web*. Muitas vezes este número pode aumentar ou variar bastante durante o tempo de vida do sistema. Por este motivo é necessário que estes tipos de sistema sejam escaláveis, para atender os usuários de forma satisfatória;

- **Segurança** – Sistemas *Web* podem ser acessados por usuários remotos em todo o mundo. Isto exige um maior comprometimento com o controle da segurança e integridade da aplicação [18];

- **Disponibilidade** – Geralmente os sistemas *Web* são acessados por pessoas de diferentes perfis, em várias partes do mundo e em diferentes horários. Por este motivo é necessário que os mesmos tenham períodos de indisponibilidade muito curtos.

Outro ponto a considerar é a popularização da linguagem de programação Java. Hoje, muitos sistemas estão sendo desenvolvidos na plataforma Java. O motivo é que ela oferece diversas vantagens sobre as outras linguagens e vem sendo uma tecnologia fortemente aceita no mercado [20]. A quantidade de produtos e soluções desenvolvidas em Java [38] tem aumentado muito e isto está fazendo com que ela se torne uma linguagem de programação padrão.

Particularmente, o uso de Java para desenvolvimento de sistemas *Web* tem se mostrado muito eficaz, por isso empresas em todo o mundo estão adotando esta tecnologia no desenvolvimento de seus sistemas. O motivo principal desta adoção em massa é que a plataforma de desenvolvimento de sistemas *Web* em Java é um padrão aberto [58]. Isto significa que vários fabricantes podem produzir APIs, ferramentas de auxílio ao desenvolvimento, infra-estruturas de execução de sistemas *Web* Java, produtos, etc., evitando assim o monopólio da tecnologia por uma única empresa. O fato da comunidade Java poder participar dos rumos das especificações e enviar solicitações de mudança a mesma é outra característica que influencia o sucesso da linguagem [40].

Sistemas *Web* desenvolvidos em Java, mais especificamente, na plataforma J2EE – *Java 2 Enterprise Edition* [59] são naturalmente multicamadas. Esta plataforma por si só é um modelo de aplicação distribuído e multicamada [3]. A Seção 1.1 detalha melhor cada uma destas camadas.

1.1 Camadas da plataforma J2EE

A plataforma J2EE especifica quatro camadas para desenvolvimento de sistemas *Web*. A Figura 1.1 dá uma idéia destas camadas, da localização física e tecnologias envolvidas em cada uma delas.

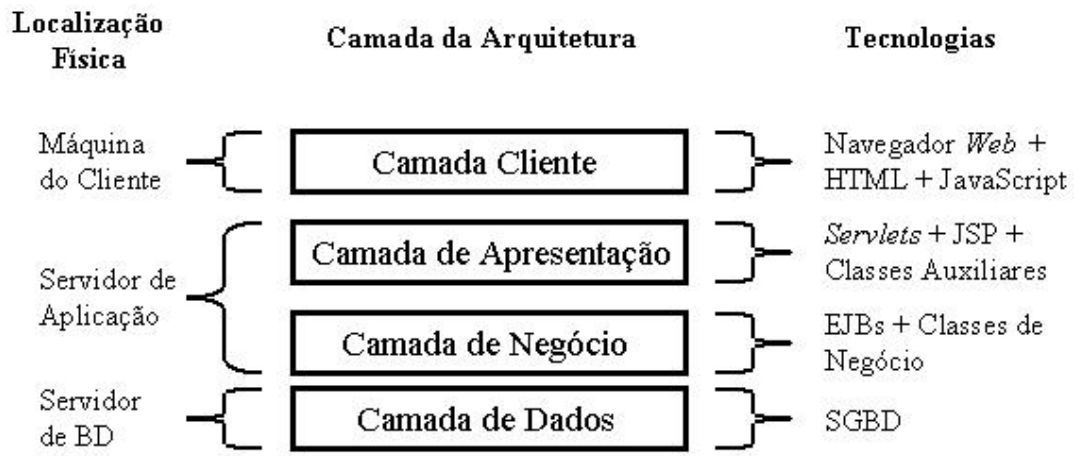

Figura 1.1: Camadas da plataforma J2EE.

- **Camada Cliente** – Tem o papel de uma interface de entrada e saída para interação do sistema com usuário e é executada na máquina do cliente. Esta camada, em aplicações *Web*, é implementada com apoio do *Web browser*, que tem basicamente o papel de interpretar e apresentar o conteúdo gerado pela **Camada de Apresentação** (geralmente HTML e JavaScript). Ela interage com a **Camada de Apresentação** utilizando protocolos como HTTP [15] e HTTPS[1];

- **Camada de Apresentação ou *Web*** – É a primeira camada do servidor de aplicação e tem o papel de disponibilizar os serviços da **Camada de Negócio** para o ambiente *Web* oferecendo conteúdo estático e conteúdo dinâmico gerado pelos componentes *Web*. Geralmente o conteúdo gerado por esta camada é HTML, mas a mesma pode gerar qualquer formato suportado pelo protocolo HTTP, tais como XHTML [10], WML [11], XML [4], etc.;

- **Camada de Negócio** – Representa o núcleo do sistema e é nela onde estão implementadas todas as regras de negócio da aplicação. A plataforma J2EE oferece a tecnologia de *Enterprise Java Beans* (EJB) [42] para implementar esta camada. No entanto, o modelo é flexível o suficiente para que seja possível usar tecnologias como CORBA [22] ou até mesmo componentes implementados usando apenas a API básica de Java (J2SE – Java 2 Standard Edition) [34];

[1]HTTPS (*Hypertext Transfer Protocol over Secure Socket Layer*) é um protocolo *Web* que encripta e decripta as requisições de clientes e respostas do servidor *Web*. HTTPS é o uso de SSL (Secure Socket Layer) como uma subcamada do padrão HTTP. SSL é o protocolo para gerenciamento de transmissão de mensagens na *Internet* [55].

- **Camada de Dados** – Esta camada é responsável pelo gerenciamento dos dados do sistema. Ela pode ser vista como a infra-estrutura necessária para gerenciamento dos recursos da aplicação. O SGBD (Sistema de Gerenciamento de Banco de Dados) é um exemplo de infra-estrutura localizada nesta camada.

Destas camadas, as únicas que envolvem esforço de programação são as de Apresentação e Negócio. A Camada Cliente compreende basicamente as telas do sistema[2] e o *Web Browser*, onde as telas do sistemas são geradas pela Camada de Apresentação. A Camada de Dados contempla basicamente o gerenciamento dos dados do sistema.

Geralmente a Camada de Apresentação é desenvolvida sem a preocupação com os bons princípios de qualidade de *software* [50]. Isto tem diversos impactos negativos para o desenvolvimento e manutenção desta camada do sistema, tais como pouco reuso e conseqüentemente muita replicação de código; problemas de legibilidade do código fonte dos componentes; uso inapropriado dos recursos disponíveis na plataforma; difícil manutenção nesta camada. Dois são os principais motivos deste pouco comprometimento, falta de suporte técnico para esta camada e mito que o desenvolvimento desta camada é muito simples.

De fato, existe uma carência de guias e diretrizes de desenvolvimento, componentes de *software*, *frameworks* e padrões de projetos que auxiliem programadores e arquitetos a solucionarem problemas corriqueiros e intrínsecos ao desenvolvimento da camada de apresentação de sistemas *Web*.

Apesar de ser dado pouco valor ao desenvolvimento da Camada de Apresentação de sistema *Web* e de acreditar-se que a mesma pode ser implementada de forma rápida e sem necessidade de métodos, diretrizes, componentes, etc., a experiência prática com desenvolvimento de sistemas *Web* tem mostrado que esta premissa é falsa e que o desenvolvimento da Camada de Apresentação está longe de ser uma atividade simples.

Para exemplificar o esforço envolvido com o desenvolvimento da Camada de Apresentação de sistemas *Web*, podemos citar um levantamento realizado pela Empresa de *software* Newstorm [44] que produz o sistema Notitia [45], ferramenta responsável pelo gerenciamento dinâmico de *sites Web*. Este levantamento teve como objetivo calcular a distribuição de código na versão 2.0 do sistema Notitia. Como mostra a Figura 1.2, a Newstorm identificou que 48% do código corresponde à Camada de Apresentação.

Inclusive percebe-se que o esforço em linhas de código para desenvolvimento da Camada de Apresentação é cerca de 12% maior do que a da Camada de Negócios. As demais partes são relativas ao código que poderia ter sido gerado automaticamente, com a ajuda de geradores de código, e ao código referente a execução de APIs desenvolvidas para melhorar o reuso no sistema.

1.2 Tecnologias para desenvolvimento de sistemas *Web* em Java

A plataforma J2EE oferece suporte ao desenvolvimento da Camada de Apresentação através das tecnologias Java *Servlet* [12] e Java ServerPages (JSP) [43]. Estas são descritas com mais detalhes nas Seções 1.2.1 e 1.2.2, respectivamente.

[2]Pode conter código JavaScript [16] para realizar validações sob os dados de entrada do usuário, mas geralmente este código é bastante simples.

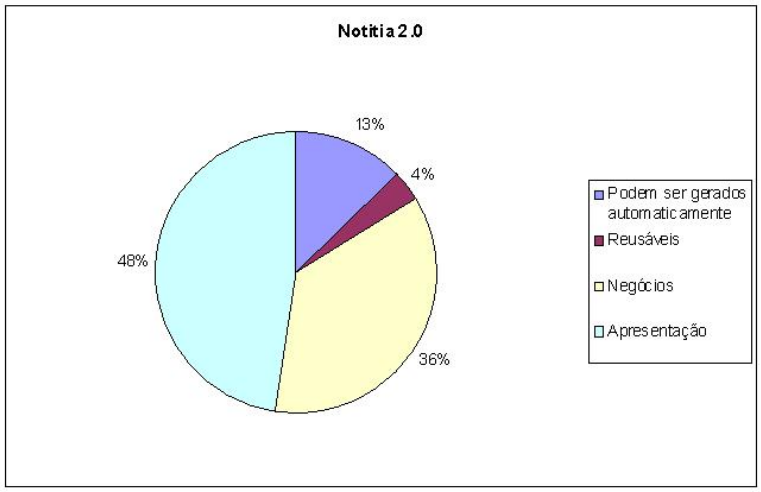

Figura 1.2: Total de linhas de código associado a cada camada do sistema Notitia.

1.2.1 *Servlets*

Os *Servlets* nada mais são do que programas escritos na linguagem Java e que são executados em resposta a requisições *Web*. Eles ficam residentes no servidor *Web* (mais especificamente no *Web Container*[3]) e podem ser entendidos como uma extensão do servidor *Web*, já que também recebem requisições HTTP e enviam respostas de volta, mas são capazes de gerar conteúdo dinâmico como resposta de suas execuções. Os *Servlets* são definidos como classes Java e possuem alguns métodos padrões que devem ser implementados para que se comportem como esperado [24].

Características

Apesar de ser uma tecnologia nova e de já existirem muitas outras para o mesmo fim, tais como ASP [67], PHP [48], CGI [23], Perl [66], existem inúmeras vantagens em escolher o uso de *Servlets* para geração de páginas dinâmicas:

- São classes Java e por isso possuem as mesmas vantagens de qualquer outro programa da linguagem, tais como independência de plataforma, concorrência, modularidade [20];
- Apesar de serem escritos em Java, uma linguagem interpretada, são geralmente mais rápidos do que a maioria dos programas baseados em CGI. Duas características influenciam neste ganho de performance: a primeira é que eles são carregados na memória uma única vez, durante a inicialização, e a segunda é que eles podem executar pedidos concorrentemente [26].
- Foram projetados para serem simples e extensível. De fato, a API de *Servlets* é bem simplificada contendo um conjunto conciso e completo de funcionalidades. A arquitetura dos *Servlets* também pode ser facilmente estendida, pois apesar de

[3]Parte do servidor *Web* ou servidor de aplicação responsável por executar e gerenciar o ciclo de vida dos *Servlets* [12].

existirem muitas funcionalidades bem definidas, alguns pontos são muito flexíveis, facilitando a definição de um novo tipo de comportamento ou customização do já existente.

Para definir um *Servlet* deve-se criar uma classe que estende de `HttpServlet`[4] e redefinir o método `service`[5] [24]. O trecho de código abaixo exemplifica o esqueleto básico do *Servlet* `HelloClient`.

```
... \\ imports necessários
public class HelloClient extends HttpServlet {
  ...  \\ Declaração dos atributos
  public void service(...)throws ServletException{
    ... \\ Execução da requisição e envio da resposta
  }
}
```

Os *Servlets* são executados pelo cliente *Web* da mesma forma como os programas baseados em CGI: através de acesso a URLs que indicam a localização dos mesmos. Por exemplo, a URL abaixo indica uma referência ao *Servlet* chamado `HelloWorld` que está localizado no servidor *Web* `www.algumlugar.br`.

```
http://www.algumlugar.br/servlet/HelloWorld
```

O formato destas URLs são bem dependentes do *Web Container* onde o *Servlet* está executando e, principalmente, das configurações dos mesmos.

Ciclo de Vida

Os *Servlets* possuem um ciclo de vida bem definido, com basicamente três estados: inicialização, finalização e serviço. Este ciclo de vida é gerenciado pelo *Web Container*, que é responsável por carregar e executar os *Servlets*. A Figura 1.3 demonstra o ciclo de vida destes componentes.

O primeiro estágio do ciclo de vida é o de inicialização. Este é o momento em que *Web Container* carrega o *Servlet* em memória e faz uma invocação ao seu método `init`. Este método não será executado novamente até que o *Servlet* seja destruído, e só após a inicialização o *Servlet* pode receber e executar requisições. É neste momento que o *Servlet* deve iniciar os recursos que serão compartilhados na execução de todas as requisições. Exemplo de recursos que devem ser iniciados neste momento são configurações do sistema, conexões com banco de dados, conexões RMI[6], instância do sistema (geralmente objetos que implementam o padrão de projeto Facade [17]).

Após a inicialização, o *Servlet* entra para a fase de serviço (Disponível). Neste momento ele está pronto para receber as requisições *Web*, processá-las e enviar a resposta.

[4]Esta classe já contém o comportamento básico que é comum para todos os *Servlets* de uma aplicação *Web*.

[5]O desenvolvedor pode redefinir outros métodos mais específicos ao invés deste. Estes métodos são invocados pela implementação padrão do `service`. Exemplos deles são `doGet` – executado quando a requisição *Web* é feita via GET, `doPost` – executado quando a requisição *Web* é feita via POST [15].

[6]*Remote Method Invocation* – Tecnologia Java que dá suporte a distribuição [36].

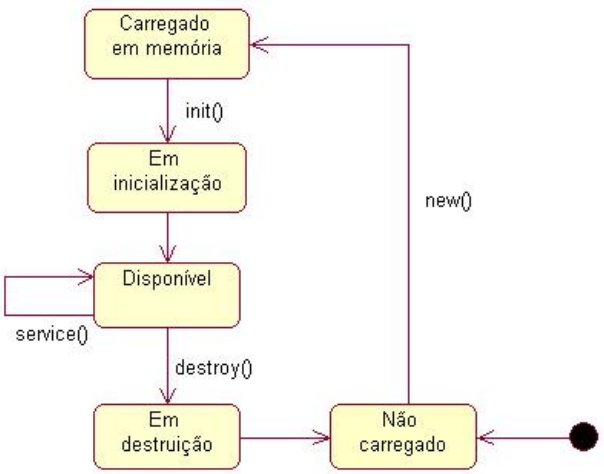

Figura 1.3: Ciclo de vida dos *Servlets*.

Geralmente, é nesta fase que o *Servlet* passa a maior parte do seu tempo de vida. Cada requisição *Web* resulta na execução de um novo *thread*, que basicamente faz uma chamada ao método `service` do *Servlet* em memória. A Figura 1.4 dá uma idéia do funcionamento do ambiente de execução dos *Servlets*.

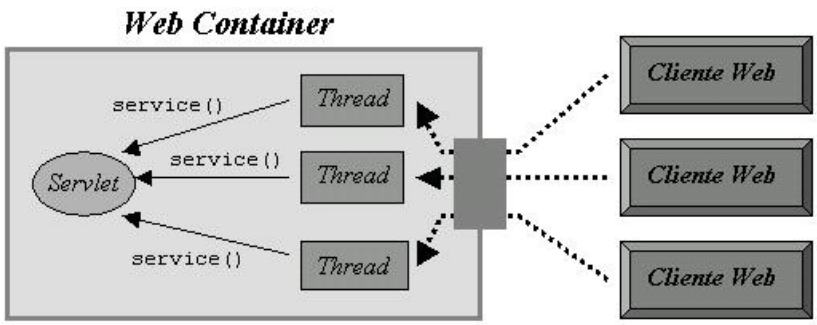

Figura 1.4: Característica *Multithreading* dos *Servlets*.

O comportamento padrão do método `service` é verificar qual o tipo da requisição *Web* (GET, POST, DELETE, etc.) e então chamar um dos métodos específicos associados ao tipo da requisição (`doGet`, `doPost`, `doDelete`, etc.). Por isso nem sempre é necessário redefinir o método `service`, o desenvolvedor pode redefinir um dos métodos mais específicos e obter a garantia que este só vai ser invocado quando o tipo de requisição for o que ele espera.

O método `service` e suas especializações recebem dois parâmetros: o primeiro do tipo `HttpServletRequest`, que contém todas as informações da requisição, tais como parâmetros do usuário, tipo da requisição, informações de cabeçalho; o segundo do tipo `HttpServletResponse`, é o canal por onde o *Servlet* envia as informações de volta ao cliente *Web*.

A última fase do ciclo de vida: destruição, é o momento em que o *Servlet* é removido da memória e todos os recursos associados ao mesmo liberados. Antes de remover o

Servlet da memória, o *Web Container* faz uma invocação ao seu método `destroy`. Por isso o desenvolvedor deve redefinir este método, inserindo código relativo a liberação dos recursos inicializados no método `init`. Esta chamada é feita uma única vez após cada inicialização do *Servlet* e é útil para permitir que *Servlet* se prepare para ser removido da memória.

1.2.2 JSP (*Java Server Pages*)

JavaServer Pages (JSP) é a tecnologia da plataforma J2EE para construção de aplicações que geram conteúdo *Web* dinâmico, tais como HTML, DHTML, XHTML, XML [43]. Uma página JSP é basicamente um documento baseado em texto que descreve como processar um pedido e gerar uma resposta. Esta descrição é uma mistura de trechos HTML com algumas ações dinâmicas. JSP não é uma linguagem que é executado no *browser* do cliente com a intenção de estender a linguagem HTML, mas sim uma linguagem de *script* que é executado no servidor, e o resultado desta execução é uma resposta que pode ser HTML ou qualquer outro tipo de dado suportado pelo protocolo HTTP. Um exemplo de JSP pode ser visto no trecho abaixo.

```
<html>
  ...
  Esta é uma página JSP,
  Gerada na data <%=(new Date()).toString()%>.
  ...
</html>
```

De fato percebe-se que o JSP é bastante similar a uma página HTML, mas contém trechos de código Java que são executados no servidor. No exemplo acima, o comando entre os operadores `<%=` e o `%>`) recupera a data atual. A semântica destes operadores é avaliar a expressão limitado por eles e colocar o valor da mesma neste ponto da página.

A tecnologia de JSP possui o mesmo poder de execução de *Servlets*, ou seja, tudo que é possível fazer com uma destas também é com a outra. A grande diferença entre as duas tecnologias está apenas no modelo de programação, ou melhor, na forma como o programador escreve os componentes *Web*. Com o uso de *Servlets* o programador basicamente precisa escrever uma classe em Java, que goza dos mesmos recursos de qualquer programa da linguagem. Neste modelo o desenvolvedor escreve um programa Java comum e em alguns pontos do código envia trechos de HTML para o cliente *Web*. Já o modelo de JSP é mais direcionado à página, o que quer dizer que o desenvolvedor basicamente escreve uma página HTML comum e em alguns trechos desta coloca código Java, que será processado no servidor.

De fato as duas tecnologias só diferem em relação ao modelo de programação, pois toda página JSP é convertida automaticamente para um *Servlet* e este é que executado pelo *Web Container* [43]. Esta tradução de JSP para *Servlet* é um mecanismo automático do *Web Container*, o desenvolvedor não precisa se preocupar com este processo. A Figura 1.5 dá uma idéia do processo de transformação, compilação, carga e execução automática dos JSPs.

Como já dito anteriormente, uma página JSP possui dois tipos de elementos: trechos de HTML e comandos JSP. Durante a execução de uma página JSP os trechos HTML

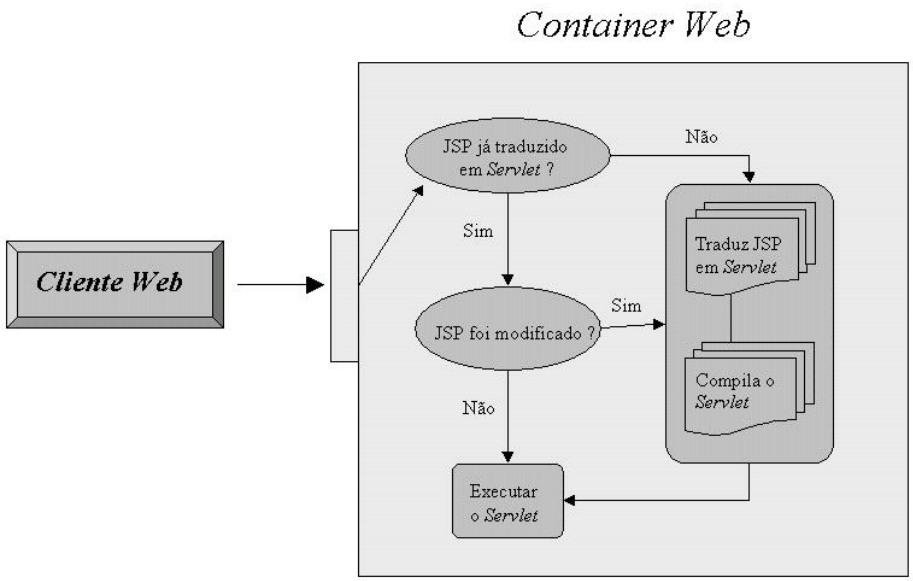

Figura 1.5: Processo de transformação, compilação, carga e execução dos JSPs.

são ignorados para processamento pelo *Web Container*, que simplesmente os envia para o cliente *Web*, enquanto que os elementos de JSP são executados e podem resultar em informação dinâmica para ser inserida na página.

Existem basicamente três tipos de elementos JSP: Diretivas, *Scripts* e Ações [43]. Os *Scripts* são os elementos de JSP que são mapeados diretamente em construtores da linguagem Java. O uso de *Scripts* permite a escrita de qualquer construtor da linguagem Java, tais como *loops*, expressões, comandos, declarações, dentro de uma página JSP. As Diretivas são informações usadas para o processo de transformação do JSP para o *Servlet* apenas. As ações são comandos JSP que permitem uma abstração maior para o programador. Eles são *tags* especiais que quando encontradas em uma página JSP são executadas e podem gerar algum resultado dinâmico. Um exemplo de uma ação JSP pode ser vista no trecho de código abaixo. Esta ação `forward` faz um redirecionamento da execução corrente para outra URL.

```
...
<jsp:forward page="www.jsp.com/HelloForward.jsp" />
...
```

Outra característica interessante de JSP é que, ao contrário de outras tecnologias de *scripting*, ela permite a definição de novas *tags* pelo desenvolvedor. O desenvolvedor pode criar bibliotecas de *tags* e usá-las na página JSP de modo similar as *tags* padrões de JSP.

1.3 Objetivos

O objetivo desta dissertação é aumentar a produtividade no desenvolvimento de sistemas *Web* em Java e melhorar a qualidade do *software* produzido. Mais especificamente, o

foco deste trabalho foi no desenvolvimento da Camada de Apresentação na linguagem Java. Para isso provemos um conjunto de ferramentas de auxilio a desenvolvedores e arquitetos de sistemas *Web* em Java na forma de padrões de projeto para sistemas *Web*, diretrizes para implementar características inerentes ao desenvolvimento *Web* e um *framework* para estruturação de componentes *Web* de uma aplicação.

1.4 Organização da Dissertação

Este trabalho está organizado em cinco capítulos, dos quais o primeiro é o capítulo corrente. O Capítulo 2 apresenta quatro padrões de projeto específicos para sistemas *Web* identificados no desenvolvimento de diversos sistemas de empresas tais como CESAR [9] e Mobile [56].

O Capítulo 3 descreve um *framework* para desenvolvimento de componentes *Web* que foi construído durante este trabalho de tese e já é usado em vários sistemas do CESAR e Mobile. Este *framework* também dá suporte a implementação de alguns dos padrões de projeto definidos no Capítulo 2.

No Capítulo 4 são apresentadas várias abordagens para implementação de algumas características intrínsecas à sistemas *Web*. Também são feitas comparações entre as diferentes abordagens para implementar uma mesma característica, com o objetivo de guiar os desenvolvedores a tomarem a decisão mais correta no contexto de seu sistema.

Por fim, no Capítulo 5, apresentamos as conclusões finais deste trabalho e suas contribuições, além dos trabalhos relacionados e sugestões de trabalhos futuros.

Capítulo 2

Padrões de Projeto para sistemas *Web*

Neste capítulo apresentamos um catálogo de padrões de projeto para sistemas Web. São ao todo quatro padrões e eles podem ser usados separadamente ou em conjunto, dependendo da necessidade do sistema.

A tecnologia de desenvolvimento de programas para *Web* é muito poderosa, no entanto é muito nova. Isto faz com que existam poucos padrões de projeto que possam ajudar os programadores e projetistas a solucionarem problemas corriqueiros e intrínsecos de sistemas *Web*. Por este motivo se faz necessário um esforço na identificação e documentação, no formato de padrões de projeto [17, 7], das soluções constantemente empregadas em aplicações *Web*.

Neste capítulo apresentamos quatro padrões de projetos para sistemas *Web*. O *Web Interceptor* (Seção 2.1) tem o objetivo de evitar a duplicação de código no início das operações de execução das requisições em sistemas *Web* estruturados de forma modular. O *Web Handlers* (Seção 2.2) evita a duplicação de código e complexidade na estruturação de sistemas *Web* com relacionamento M:N entre a apresentação e o processamento. O *Web Compiler* (Seção 2.3) soluciona o problema de como desenvolver uma aplicação *Web* de forma a evitar que o *layout* das páginas HTML estejam misturadas com a lógica de execução das operações do sistema. Por último, o *Super Component* (Seção 2.4) descreve a solução para o problema inerente ao desenvolvimento de sistemas *Web* em Java: como evitar a duplicação de código de inicialização e destruição nos diversos componentes *Web* de um sistema.

2.1 *Web Interceptor*

Contexto

Imagine um sistema bancário simples que possui alguns serviços disponíveis na Internet: crédito, débito, transferência e saldo. Como na maioria dos sistemas, a execução destes tipos de operações requer que algumas validações de segurança sejam satisfeitas, tais como as seguintes:

- O usuário precisa ter feito o login previamente;
- A página na qual foi submetida a requisição deve ter sido gerada por um determinado servidor *Web* (para evitar que um usuário tente fazer o acesso direto a um serviço sem respeitar as regras de navegação do sistema);
- A sessão do usuário não está expirada;
- O usuário deve possuir informações armazenadas em sua sessão, como, por exemplo, seu perfil.

Estes tipos de validações são muito comuns em sistemas reais que exigem um nível mínimo de segurança. Em aplicações que necessitam de um nível maior de segurança, como aplicações bancárias reais, comércio eletrônico, etc., a quantidade de validações deste tipo é bem maior.

Na tentativa de estruturar o sistema de forma modular, evitamos a estruturação monolítica (um único componente responsável por executar todas as requisições do sistema) por dificultar o desenvolvimento em equipe e diminuir a legibilidade do código (devido ao tamanho do componente). Por isso, escolhemos um modelo descentralizado, onde cada tipo de requisição é executada por um componente diferente.

No entanto esta escolha implica na replicação do código executado logo no início de cada operação. Pois será necessário que os componentes *Web* realizem uma série de validações comuns de segurança antes da execução de cada operação do sistema, que são comuns a todos. Além destas validações é muito comum haver a necessidade de escrever outros tipos de código antes da invocação dos métodos de negócio: chamadas a `setContentType`, que determina qual o tipo de resposta que vai ser dada ao cliente (por exemplo, gif, HTML, XML); adição de informações ao cabeçalho da resposta à requisição; operações de *log* para estatísticas de acesso, auditorias de segurança e outros.

Problema

Como evitar a duplicação de código no início das operações que executam as requisições em sistemas *Web* estruturados de forma modular?

Forças

- Permitir que qualquer tipo de operação (validações, *log* e código específico de *servlets*) possa ser escrito antes da invocação dos métodos de negócio;
- Evitar a replicação de código nos componentes *Web*;
- Diminuir o acoplamento entre o cliente e os componentes *Web*;
- Permitir a estruturação do sistema de forma modular (vários componentes *Web*).

Solução

Uma forma simples de evitar o problema descrito anteriormente, de forma a atender todas as forças, é através do uso de um *Web Interceptor*, como pode ser visto na Figura 2.1. Este é um intermediário na comunicação entre o cliente (*browser*) e os componentes *Web*. Todas as requisições passam por ele, onde a parte comum a todos os componentes é executada, e depois o mesmo delega a requisição para o componente responsável por ela.

A grande diferença nesta solução está no fato de que agora os clientes *Web* interagem com um único componente, diminuindo o acoplamento entre os clientes e os elementos do servidor, e aumentando a segurança do sistema. A inserção deste novo componente também evita a repetição de código nos componentes *Web* de maneira simples e bem compreensível. A centralização do recebimento de requisições não dificulta o desenvolvimento em equipe e a legibilidade do sistema pois geralmente o *Web Interceptor* não vai precisar sofrer mudanças. Isto acontece porque ele pode ser implementado de forma a ser reusado em diferentes sistemas ou, nos casos onde isto não seja possível, ele pode ser generalizado de forma que não seja necessária mudança quando houver a inserção ou remoção de novos componentes *Web* (Ver maiores detalhes na seção Implementação).

Aplicabilidade

Todas as características e problemas ao qual o este padrão se destina a resolver e melhorar são comuns a maioria dos sistemas *Web*, por isso o seu uso é recomendável em

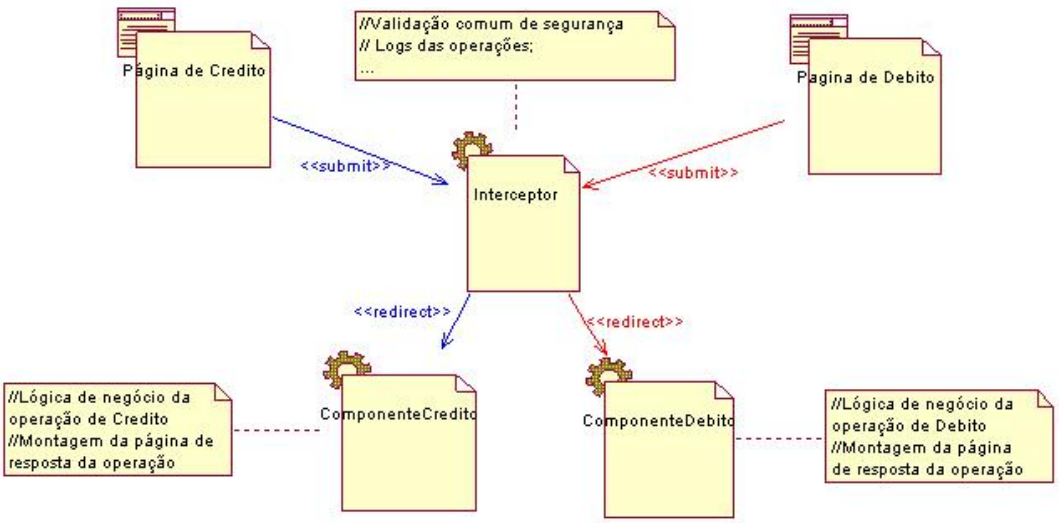

Figura 2.1: Uso do *Web Interceptor* no exemplo do sistema bancário.

qualquer situação onde há preocupação com o reuso de software. Em sistemas cuja segurança é um requisito muito importante, o uso deste padrão é essencial para a implementação de mecanismos eficazes contra ataques.

Nos casos onde é necessária a realização de filtros ou tratamentos sobre o conteúdo da entrada dos componentes *Web*, é interessante a adoção deste padrão, desde que o tratamento ou filtro seja comum a mais de um destes componentes. Outra situação onde o uso do padrão se torna útil é para a realização de *logs*, pois todas as requisições passam por um único ponto, o que faz com que seja mais simples efetuar diversos tipos de *logs*: de acesso, de parâmetros.

Um uso do padrão para *refactoring* pode ser identificar se existe código comum duplicado no início da execução da requisição dos componentes *Web*. Quando a resposta for positiva, deve-se transferir este código para um componente que vai fazer o papel do *Web Interceptor*.

Apesar de parecer que só deve existir um componente que implemente o padrão *Web Interceptor* por sistema *Web*, podem existir situações onde sejam necessários diversos deles. Esta utilização pode ser necessária quando existirem conjuntos diferentes de componentes que precisam executar operações em comum. Nestes casos deve existir um interceptador para cada um destes conjuntos, mais continua sendo recomendado que exista apenas um que interaja diretamente com os clientes *Web*. A Figura 2.2 exemplifica esta situação.

Estrutura

A Figura 2.3 dá uma idéia da estrutura do padrão, onde todas as requisições devem passar pelo `Interceptor`, para só então serem repassados para os componentes destino.

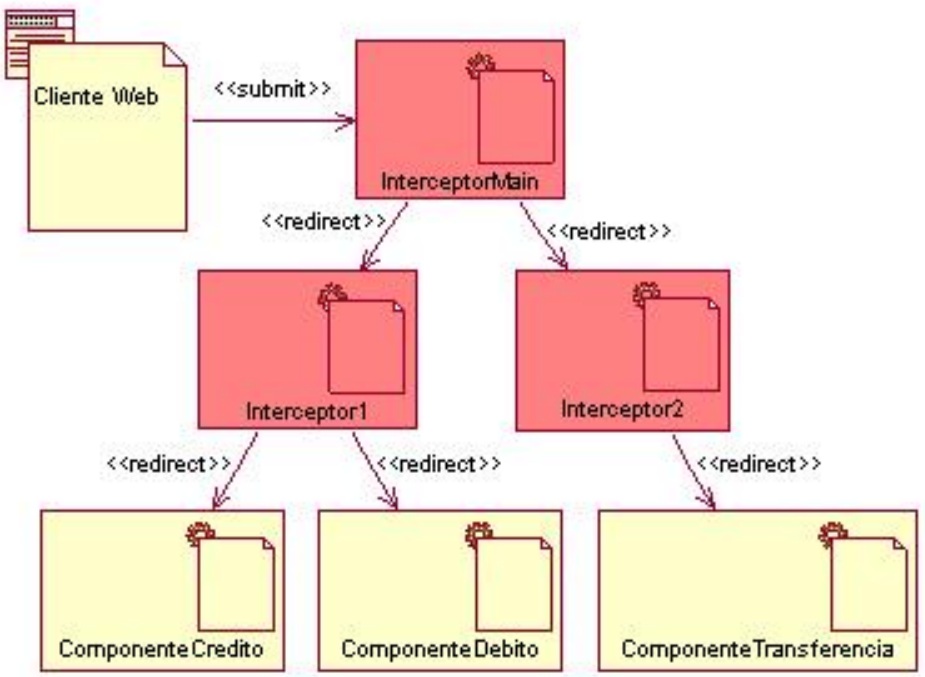

Figura 2.2: Exemplo de uma cadeia de Interceptadores.

Participantes

- `Cliente Web` - Envia requisição para o `Interceptor`, informando que operação deve ser executada e quais os argumentos;
- `Container` - Módulo do servidor *Web* que é responsável por localizar e executar os componentes residentes em seu contexto;
- `Interceptor` - Componente responsável por receber todas as requisições e com ajuda do `Container` delegá-las para o seu destino;
- `Destino` - Destino final da requisição. Responsável pela execução da requisição e resposta ao cliente.

Dinâmica

- O `Cliente Web` envia a requisição para o `Container` (mais especificamente para o `Interceptor`), informando que operação deve ser executada e quais os parâmetros;
- O `Container` localiza o `Interceptor`, colocando-o para executar a requisição;
- O `Interceptor` executa o conjunto de operações que são de sua responsabilidade (operações comuns aos componentes do sistema) e com ajuda do `Container` delega a requisição para o `Destino`;
- O `Destino` executa as operações específicas dele e, logo em seguida, envia a resposta para o `Cliente Web`.

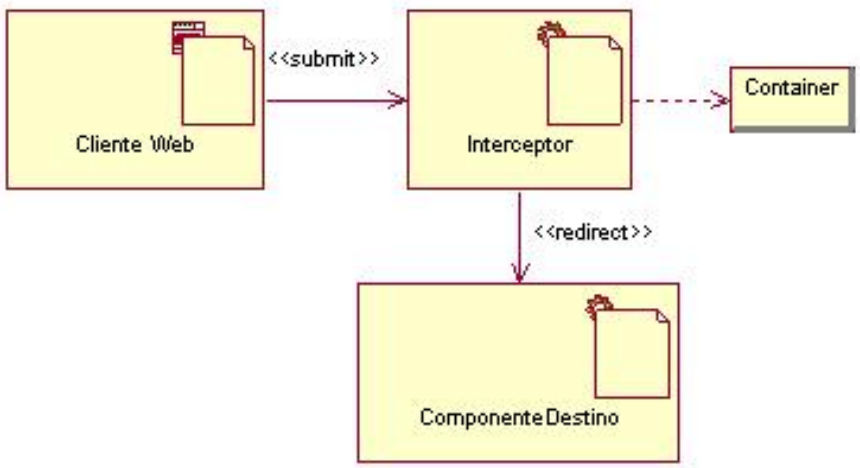

Figura 2.3: Estrutura do padrão *Web Interceptor*.

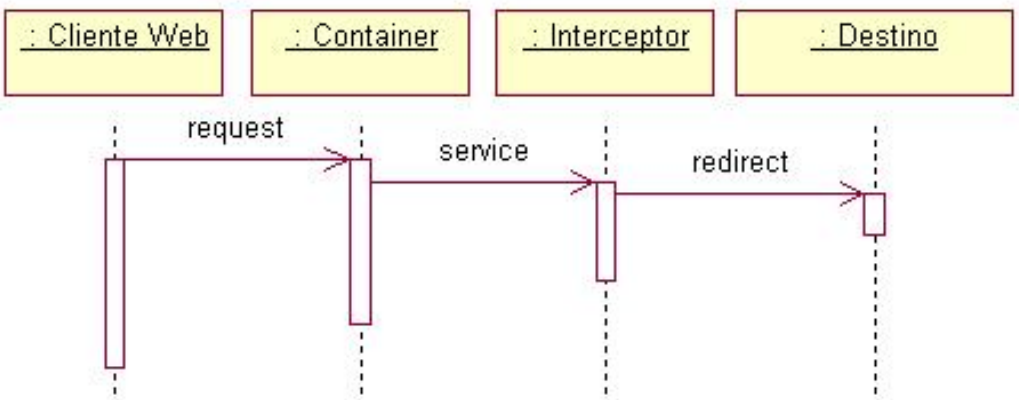

Figura 2.4: Diagrama de seqüência dos componentes do padrão.

Conseqüências

- Único ponto de acesso ao sistema — O que permite um maior controle da segurança da aplicação e facilita a implementação de operações de log do sistema;
- Evita repetição desnecessária de código - A parte inicial da operação que geralmente é comum para todos os componentes *Web* fica centralizada em um único ponto, evitando desta forma a duplicação de código;
- Diminui o acoplamento entre o cliente e os componentes *Web* - Esta característica permite que os componentes destinos possam ser substituídos por outros que tenham a mesma funcionalidade, ou até mesmo por qualquer outro tipo de tecnologia com o mesmo intuito, de forma que o cliente não sofra impacto algum;
- Diminui a performance - Todas as requisições são recebidas e pré-processadas pelo interceptador e depois redirecionadas para o componente destino responsável por

ela. Este redirecionamento causa um pequeno atraso na execução da requisição, o que não acontece em sistemas que não usam este padrão. A adoção de vários níveis na cadeia de interceptadores (exemplificado na Figura 2.2) piora progressivamente a performance do sistema.

Implementação

A forma como o interceptador identifica qual o destino da requisição pode ser implementada de várias formas. Em todos estes casos será necessário que o cliente *Web* adicione um novo parâmetro à requisição. Este parâmetro irá identificar que componente *Web* deve ser executado ao final. Algumas destas opções são as seguintes:

- **Associação Estática** – O nome do componente a ser executado está associado ao identificador de um atributo, diretamente no código Java. Esta forma é mais simples, no entanto requer que o interceptador seja recompilado sempre que houver qualquer adição de novos serviços no sistema ou mudança na associação dos serviços já existentes.

- **Associação automática** – A implementação deste mecanismo requer que seja padronizada a associação de nomes de operações aos nomes dos componentes *Web*. Um exemplo bem simples deste mecanismo é padronizar que o nome dos componentes devem ser `Servlet_OPERACAO`, onde `OPERACAO` é o nome da operação que o *servlet* é responsável por executar. Desta forma com o interceptador, sabendo que operação o cliente deseja executar, pode concluir que componente deve ser executado.

- **Associação Dinâmica** – Neste caso o mapeamento de identificadores aos nomes dos componentes está registrado em um arquivo de configurações (por exemplo, XML, *properties* ou txt.). Este arquivo pode ser lido uma única vez (durante a inicialização do sistema), sempre que for feita uma requisição ou apenas quando o arquivo for modificado.

Em Java, por exemplo, o *Web Interceptor* deve ser implementado como um *servlet*, ou seja, ele deve ser subclasse de `HttpServlet` [12]. O interceptador também pode ser implementado como uma página JSP, mas isto não é recomendado já que páginas JSP devem ser usadas apenas para tratamento da apresentação (mais detalhes podem ser obtidos na Seção 4.1). Apesar de Java permitir a delegação da execução da requisição sem intermédio do *Web Container*, esta prática não é recomendável por que o *Container* deve ser responsável por todo o gerenciamento da execução dos *servlets* [12].

Este padrão não define nenhum mecanismo para evitar que os destinos sejam executados diretamente sem o intermédio do interceptador. Mas uma opção é fazer com que os destinos verifiquem se a requisição passou pelo interceptador e descartá-las caso contrário. Em Java isto pode ser feito pelo interceptador, através da adição de atributos de controle à requisição, ou mais especificamente através da chamada ao método `setAttribute(String,Object)` da classe `HttpServletRequest` [33].

Código de exemplo

O trecho abaixo é um exemplo de código em Java que executa diversas operações que são comuns a todos os componentes *Web* do sistema bancário exemplificado na Seção Contexto.

```
01 :  response.setContentType("text/html");
02 :  Log.gravarEntrada(request);
03 :  HttpSession sessao = request.getSession();
04 :
05 :  if(sessao.isNew())
06 :    throw new SegurancaException("Sessão ainda não foi criada");
07 :  else if(sessao.isInvalidated())
08 :    throw new SegurancaException("Sessão expirada");
09 :  else if(sessao.getAttribute("perfil") == null)
10 :    throw new SegurancaException("Perfil invalido");
```

Na linha 1, o tipo de resposta que será enviada ao cliente é informado, neste caso será HTML. Na 2 é gravada uma entrada no *log* com uma série de informações sobre a requisição. Nas demais linhas são feitas validações de segurança, tais como verificar se o usuário já possuía sessão, verificar se não houve *timeout* e procurar pelo perfil do usuário, nas linhas 5, 7 e 9, respectivamente. Caso estas verificações sejam falsas é levantada uma exceção de segurança: `SegurancaException`.

Sem a utilização do padrão de projeto *Web Interceptor* este código teria que estar replicado nos diversos componentes *Web* do sistema. Com o uso do padrão estas operações podem ser removidas de todos os *servlets* e inseridas apenas no interceptador. Como pode ser visto no trecho abaixo, o *servlet* `Interceptor` é muito simples. Ele executa as operações comuns aos *servlets* do sistema na linha 09 e depois faz uma delegação ao *servlet* destino através da chamada ao método `include` da classe `RequestDispatcher` da API de *servlets* (linha 11).

```
01: ...
02: public class Interceptor extends HttpServlet{
03:   ...
04:   public void service(HttpServletRequest request,
         HttpServletResponse response) throws ServletException,
                                              IOException {
05:     RequestDispatcher dispatcher;
06:     String servletDestino;
07:     servletDestino = request.getParameter("destino");
08:     if(servletDestino != null)
09:       ... // Executa as operações comuns a todos os servlets
10:       dispatcher = request.getRequestDispatcher(servletDestino);
11:       dispatcher.include(request, response);
12:     }
13:   }
14: }
```

Usos conhecidos

- **Web handlers** - Um *framework* que usa este padrão (veja Capítulo 3) para implementar o seu componente Controlador. Este é responsável por receber todas as requisições do sistema, interpretá-las e delegá-las para os *Handlers* de Apresentação e Processamento responsáveis pela mesma.

- **Portal Encontre & Compre** [28] - Sistema de consultas dos anunciantes Listel, que também permite que o visitante faça transações de negócios on-line com os anunciantes. Todos os acessos ao sistema são feitos através de um único componente, que implementa este padrão de projeto;

- **FiS (Financial Services)** - O projeto contempla a migração dos Módulos de Contabilidade, Crédito, Lojistas e Serviços da HiperCard, para um novo ambiente tecnológico (J2EE). Estes módulos são integrados ao Sistema de Integrado de Crédito da HiperCard (SIC), ao R3/SAP e a outros sistemas legados. Também faz parte do projeto o desenvolvimento de um módulo de Controle de Acesso único e centralizado que poderá ser utilizado por qualquer aplicação da HiperCard disponível neste novo ambiente. Este sistema utilizam o *framework Web handlers* e por conseqüência todos os acessos são feitos diretamente através do Controlador (veja Seção 3);

- **Central de Regulação SUS** - Sistema de software que possibilita tratar o atendimento a pacientes em uma dada região de saúde, de forma a garantir o melhor atendimento à população. Dois dos seus módulos: marcação de consultas e exames especializados e controle de internação hospitalar, desenvolvidos pelo CESAR [9], usam o padrão de projeto *Web Interceptor* na implementação de seus componentes *Web*.

Padrões relacionados

- **Facade** [17] – O padrão de projeto *Web Interceptor* pode ser considerado uma customização do *Facade* para sistemas *Web*. De fato, eles possuem muitas características em comum e possuem intenções similares: prover uma interface de acesso única aos componentes do sistema;

- **Front Controller** [1] – Este padrão de projeto J2EE define um componente que funciona como ponto inicial de contato ao sistema *Web*. Mas a principal diferença em relação ao *Web Interceptor* é que o *Front Controller* possui lógica para decidir que componente deve receber a requisição, enquanto que o Interceptador apenas repassa a requisição de acordo com configurações por ele carregadas;

- **Intercepting Filter** [1] – O *Web Interceptor* tem um comportamento similar a este padrão de projeto J2EE, já que ambos tem como objetivo realizar algum processamento antes da execução da operação propriamente dita. A diferença é que o *Intercepting Filter* não funciona como ponto de entrada para todos os componentes do sistema, eles são apenas acoplados aos *servlets* para realizar um pré-processamento.

2.2 *Web Handlers*

Contexto

Em sistemas *Web* baseados no modelo de pedido-resposta é comum existirem diferentes requisições (pedidos) gerando dinamicamente a mesma página HTML como resposta. Esta situação pode ser percebida em um exemplo bem simplificado de sistema bancário que possui apenas as operações de crédito e débito em conta corrente, além de uma operação de *login* no sistema. O funcionamento deste é especificado através do mapa navegacional da Figura 2.5, que usa a notação de Diagrama de Estados de UML [25], onde as operações são desenhadas como eventos e as páginas HTML (dinâmicas ou estáticas) como estados.

De fato, analisando a figura percebe-se que as operações `Debito`, `Credito` e `Login`, apesar de serem operações distintas, geram como resposta de sua execução a mesma página (`Menu de Movimentações`). Este fato, apesar de exemplificado com um menu principal, ocorre em diversas outras situações.

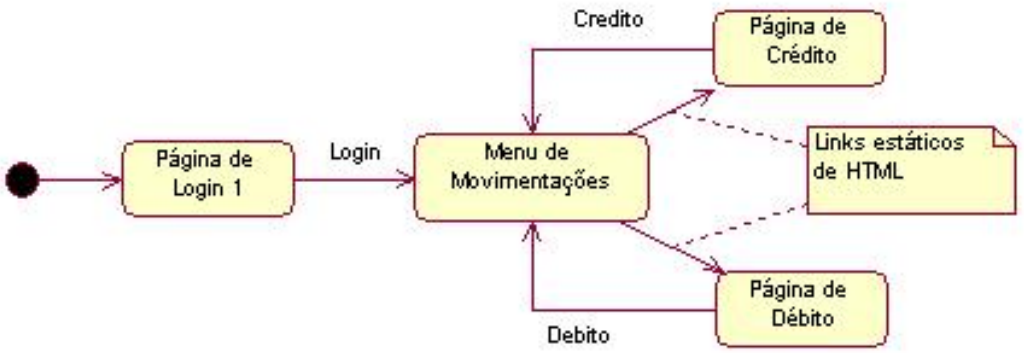

Figura 2.5: Mapa navegacional simples do sistema bancário.

Outra situação comum em sistemas desta natureza é termos uma mesma requisição gerando diferentes respostas, dependendo da origem da requisição ou do resultado do de seu processamento. Para exemplificar esta situação, considere que o cliente, além de realizar movimentações em conta (débito e crédito), pode fazer atualizações em seu cadastro. A Figura 2.6 exemplifica bem a situação em que a mesma operação, `Login`, é executada a partir de contextos diferentes (`Página de Login 1` e `Página de Login 2`) e deve gerar uma saída específica (`Menu de Atualização` e `Menu de Movimentações`) dependendo da origem da requisição. Estas duas páginas de *login* possuem o conteúdo bem diferente pois elas estão em contextos distintos e não possuem relação direta.

As situações ilustradas pelas Figuras 2.5 e 2.6 são bastante comuns em sistemas *Web* não triviais. Estas geralmente trazem problemas de implementação, como duplicação e complexidade do código, quando não é aplicada uma estruturação de componentes *Web* adequada ao problema (veja Seção 4.3). Em situações onde a mesma página pode ser gerada como resultado da execução de diferentes requisições (Ilustrado pela Figura 2.5) é comum ocorrer duplicação de código relativo à montagem desta página em cada um dos

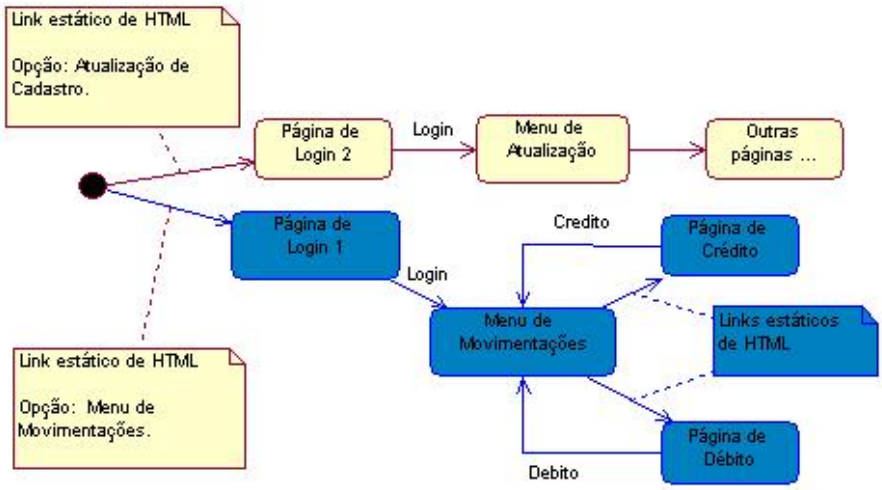

Figura 2.6: Mapa navegacional completo do sistema bancário.

componentes que tratam as requisições. Já em situações onde uma mesma operação pode gerar diferentes páginas como resultado de seu processamento (Ilustrado pela Figura 2.6) é comum haver um aumento na complexidade de implementação do componente que executa a requisição, já que este precisa decidir que resposta apresentar, além de possuir o código relativo a montagem de cada uma das páginas de resposta gerada por ele.

Problema

Evitar a duplicação de código e complexidade na estruturação de sistemas *Web* com relacionamento M:N entre a apresentação e o processamento.

Forças

- Evitar a duplicação do código referente à montagem da apresentação.
- Evitar a duplicação do código de processamento da requisição.
- Não tornar o código do componente *Web* mais complexo quando a quantidade de operações que geram a mesma página de saída aumenta.
- Não tornar o código do componente *Web* mais complexo quando aumenta o número de páginas de saída possíveis para a mesma operação.

Solução

A solução é baseada na construção de entidades denominadas *handlers*. Existem dois tipos destes: *Handlers* de Apresentação e *Handlers* de Processamento. Os primeiros

contêm apenas código relativo à montagem das páginas dinâmicas e os outros possuem código (ou chamadas) relativo à execução da lógica de negócio. Com esta estruturação é necessário criar um *handler* de processamento para cada operação do sistema e um de apresentação para cada página dinâmica.

Cada requisição *Web* dispara uma execução do lado do servidor que dinamicamente associa um par de *handlers* (um de apresentação e um de processamento) para responder ao cliente. Esta composição dinâmica é determinada por um parâmetro da requisição do cliente. A entidade responsável pela montagem deste par e delegação da requisição primeiro para o *handler* de processamento e depois para o de apresentação é o **Controlador de Handlers**, que é especificado com mais detalhes nas Seções Estrutura e Implementação.

Os *handlers* de processamento, além de conterem chamadas às operações do sistema, possuem código responsável pela validação dos dados vindos do cliente *Web*, e código responsável pela preparação dos dados para seu par de apresentação.

Os *handlers* de apresentação também possuem validação de dados, além de código de montagem das páginas dinâmicas. Esta validação é necessária porque estes precisam validar os dados gerados pelo seu pares, já que eles são entidades independentes e podem ser compostos de diferentes formas.

Exemplo da solução

Para o sistema bancário, as seguintes entidades são criadas com a utilização do padrão:

- *Handlers* de Apresentação:
  - `HA_MenuAtualizacao`, monta o `Menu de Atualização` dinamicamente ;
  - `HA_MenuMovimentacoes`, responsável por montar dinamicamente o `Menu de Movimentações`.
- *Handlers* de Processamento:
  - `HP_Login`, responsável por executar a operação `Login`;
  - `HP_Credito`, responsável por executar a operação `Crédito`;
  - `HP_Debito`, responsável por executar a operação `Débito`.

Estes *handlers* podem ser compostos de diferentes formas para responder aos diferentes tipos de requisições. A Figura 2.7 exemplifica algumas composições possíveis para o sistema bancário com a utilização desse tipo de estruturação. De fato, o uso de *handlers* é capaz de resolver os problemas de duplicação e complexidade de código em casos de relacionamentos M:N entre a apresentação e o processamento, podendo assim ser aplicado na maioria das situações comuns a sistemas *Web*. Na Figura 2.7 é possível ver como os *handlers* podem ser reusados em diferentes requisições, evitando assim a repetição de código.

Figura 2.7: Composições possíveis de *handlers* de apresentação e processamento.

Aplicabilidade

Use o padrão principalmente em sistemas onde aparecem relacionamentos M:N entre as partes de processamento e apresentação.

Desenvolver o sistema já usando do padrão mesmo quando não ocorrem estas situações é uma boa prática, pois desta forma o desenvolvedor já torna o sistema imune à problemas de repetição de código antes mesmo de identificá-los.

Uma forma simples de identificar se o seu sistema necessita do uso deste padrão é desenhando o mapa navegacional e verificando duas coisas:

1. Se a mesma operação aparece em duas transições diferentes (como na Figura 2.6), significa que exixtem relacionamentos 1:N do processamento para a apresentação;

2. Se há mais de uma seta apontando para a mesma página significa que existe relacionamentos 1:N da apresentação para o processamento.

Estrutura

A estrutura do padrão *Web Handlers* é especificada através do diagrama de classes de UML da Figura 2.8.

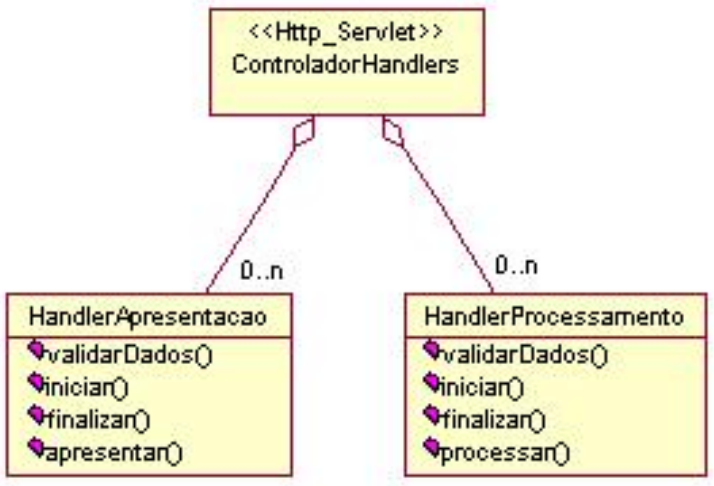

Figura 2.8: Diagrama de classes do padrão *Web Handlers*.

Participantes

- `ControladorHandlers` - Responsável pelo recebimento das requisições e controle da execução dos *handlers* responsáveis pela mesma;
- `HandlerApresentacao` - Responsável pela montagem de uma ou mais páginas semelhantes;
- `HandlerProcessamento` - Responsável pela invocação dos serviços, e também pela geração de dados para o *handler* de apresentação a ser executado em conjunto com ele;

Os *handlers* possuem algumas operações padrões que devem ser definidas pelo programador para que eles se comportem da maneira esperada, estas são:

- `iniciar` - Inicia o *handler* e é invocado pelo ambiente;
- `finalizar` - Finaliza o *handler* e é invocado pelo ambiente;
- `apresentar` - Contém a lógica referente à montagem das páginas dinâmicas;
- `processar` - Contém a lógica referente à chamada dos serviços;
- `validarDados` - Contém regras de validação dos dados de entrada.

Os *handlers* possuem ciclo de vida semelhante aos dos *servlets* [12]. Assim são oferecidos os métodos `iniciar` e `finalizar`, para que o programador possa definir operações que são executadas na sua inicialização e finalização, respectivamente. Todo *handler* pode implementar o método `validarDados`, que é executado antes do `processar` ou do `apresentar`. Para os *handlers* de processamento ele pode ser usado para implementar regras de validação dos dados da requisição *Web*, enquanto que nos de apresentação ele contém a validação dos dados gerados pelo processamento.

Dinâmica

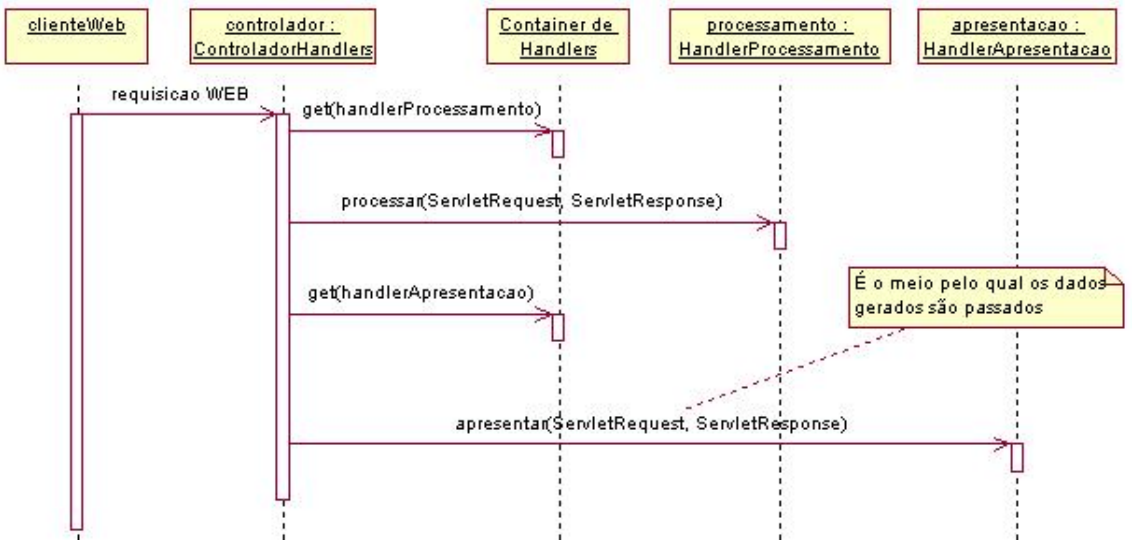

Figura 2.9: Diagrama de seqüência dos componentes do padrão *Web Handlers*.

- Toda requisição *Web* é recebida pelo `controlador`, que interpreta seus parâmetros, recupera o `processamento` e o executa. Após o término do `processamento`, o `controlador` recupera o `apresentacao`, baseado nas informações de configuração e no resultado do execução do `processamento`, colocando-o para executar logo em seguida.

- O `processamento` é o primeiro a ser executado através da chamada ao seu método `processar`. Nele vão estar a chamadas aos serviços do sistemas, implementados por objetos que encapsulam toda a regra de negócio da aplicação (EJBs [42], *Facade* [17] etc). Ele também é responsável por produzir os dados de entrada do *handler* de apresentação que será associado a ele.

- O `apresentacao` é executado após o término com sucesso do `processamento` através de uma chamada a seu método `apresentar`. Nele não deve haver lógica de processamento da requisição, apenas código referente à montagem da página. Este *handler* consulta os dados gerados pelo seu par processamento e constrói a página de resposta baseada nestes.

- O `Container de Handlers` é uma entidade que está neste contexto apenas para que se tenha um entendimento melhor do processo de acesso e recuperação dos *handlers*. Ele nada mais é do que o ambiente onde os *handlers* são executados, em aplicações *Web*, por exemplo, um servidor *Web* ou mais especificamente um *Container Web*.

Conseqüências

- Grande flexibilidade na composição das partes de apresentação e processamento – Os *handlers* de apresentação e processamento são independentes um dos outros e podem ser integrados de forma diferente para responder a diferentes tipos de requisições, desde que eles sejam compatíveis em relação aos dados produzidos e consumidos.
- Maior reuso de código - O padrão evita a repetição desnecessária de código, pois permite o compartilhamento de entidades para responder a diferentes requisições.
- Mudanças nos mecanismos de montagem da apresentação (JSP [24], FreeMarker [61], WebMacro [29] ou Velocity [52]) não causam efeito algum nas entidades de processamento.
- Facilita a implementação de sistemas que requerem diferentes formatos de saída (por exemplo, XML [4], HTML, WML [11], XHTML [10].) para a mesma operação – Com o uso do *Web Handlers* o desenvolvedor pode criar *handlers* de processamento que serão compostos com diferentes *handlers* de apresentação, onde estes últimos possuem implementações para cada formato de saída possível.
- Facilita a implementação de componentes (*handlers* de apresentação e processamento) que podem ser reusados em outros sistemas – Com o uso do padrão é mais fácil manter uma biblioteca de *handlers* onde o desenvolvedor pode consultar serviços já implementados e compô-los a fim de obter a implementação de um serviço desejado.
- O padrão ajuda a evitar repetição de código, no entanto ele possui a desvantagem de aumentar o número de classes necessárias para implementar um sistema. Por isso é preciso ter um certo cuidado em não tornar o sistema modular demais, pois quanto mais modular ele for, maior será o número de classes que precisam ser criadas;
- A divisão da responsabilidade pelo tratamento da requisição acrescenta uma complexidade à implementação dos componentes, pois é necessário a passagem de parâmetros do *handler* de processamento para o de apresentação a cada requisição, já que eles são entidades distintas.

Implementação

Todos os *handlers* possuem um comportamento bem similar e interfaces bem definidas, por isso é interessante que existam classes e interfaces que dêem apoio ao funcionamento

do padrão, provendo comportamento genérico e especificando a interface das operações dos *handlers*. Uma estruturação interessante deste padrão pode ser vista na Figura 2.10.

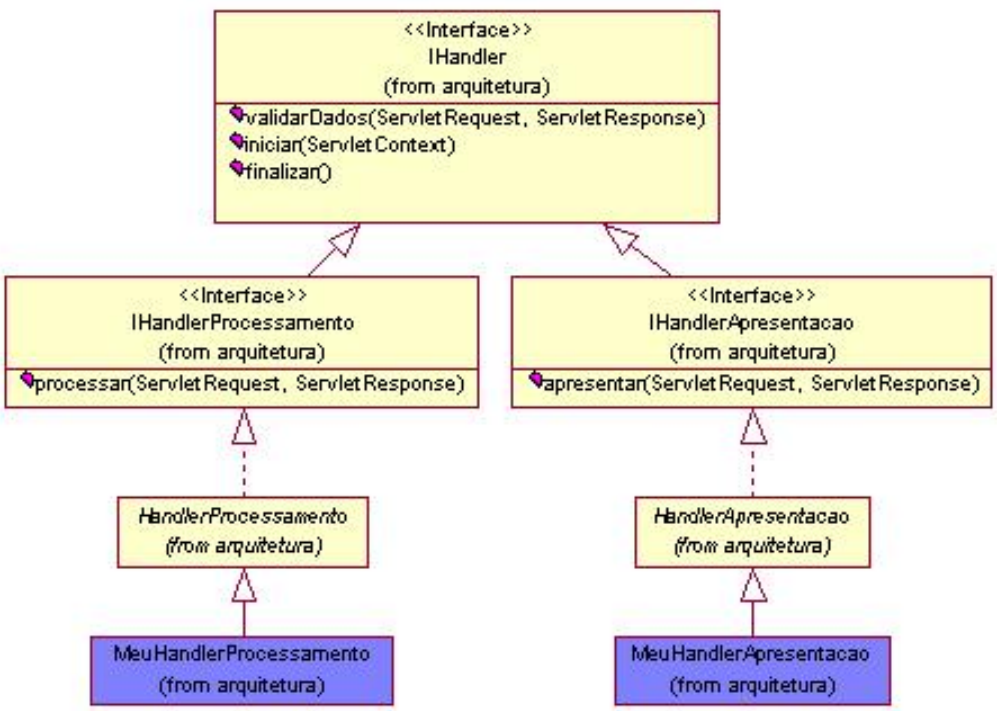

Figura 2.10: Diagrama de classes refinado dos componentes do *Web Handlers*.

- `IHandler` – Interface genérica que especifica as funcionalidades que todo *handler* deve oferecer.
- `IHandlerProcessamento` – Interface que define todas as operações que devem existir em um *handler* de processamento.
- `IHandlerApresentacao` – Interface que define todas as operações que devem ser oferecidas por um *handler* de apresentação.
- `HandlerProcessamento` – Classe abstrata que implementa os métodos definidos na interface `IHandlerProcessamento`. Ela provê uma implementação padrão para os *handlers* de processamento. Toda entidade de processamento deve estender esta classe para possuir o comportamento de um *handler* de processamento.
- `HandlerApresentacao` – Classe abstrata que implementa as funcionalidades definidas na interface `IHandlerApresentacao`. Toda entidade de apresentação deve estender esta classe para possuir o comportamento de um *handler* de apresentação.

Os tipos `ServletRequest`, `ServletResponse` e `ServletContext` são classes padrões da API de *servlets* [12]. A primeira é o meio pelo qual os parâmetros da requisição são passados. O segundo é usado para enviar resposta para o cliente *Web*. O último é uma

referência para o *Servlet Container*, através do qual pode-se recuperar parâmetros de configuração do ambiente, comunicar-se com outras entidades que estejam executando no mesmo contexto, etc. Os componentes aqui apresentados são específicos para uma implementação baseada em *servlets*, no entanto o padrão é genérico o bastante para que possa ser implementado em outras tecnologias.

Muitas características dos *handlers* são comuns as dos *servlets*, mas como o ambiente de *servlets* (*Servlet Container*) já provê todos estes serviços, é uma boa idéia usá-los para os *handlers*. Uma forma de usar estes recursos de forma efetiva é fazendo com que os *handlers* sejam executados dentro do *Servlet Container*. Para isso eles precisam ter a interface de um *servlet* e se comportar como tal. Uma pequena modificação na estrutura apresentada anteriormente pode ser feita de forma a atender estes requisitos e preservar a semântica dos *handlers* apresentada até o momento. A Figura 2.11 dá uma idéia da alteração necessária no modelo.

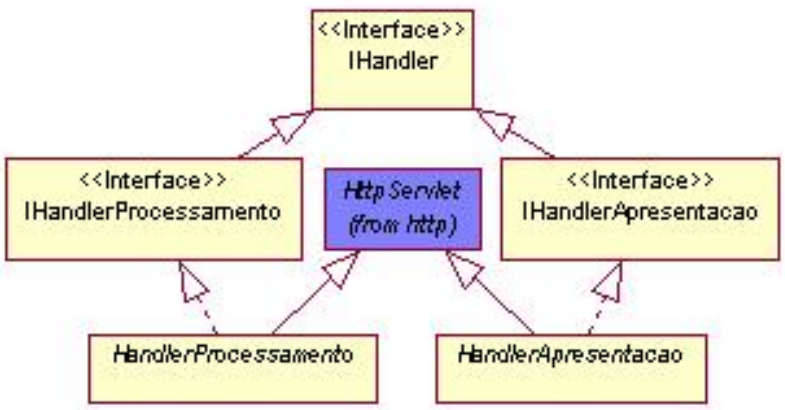

Figura 2.11: Estrutura do padrão no ambiente de *servlets*.

As implementações genérica `HandlerApresentacao` e `HandlerProcessamento` continuam implementando as mesmas interfaces, mas agora estendem a classe `HttpServlet` (pertencente a API de *servlets*) para herdarem o comportamento de *servlet* e poderem ser gerenciados pelo *Web Container*. Outro detalhe é que estas implementações genéricas declaram todos os métodos herdados da classe `HttpServlet` como "final", desta forma evitam que suas subclasses redefinam estes métodos. Enfim, para as classes que o programador precisa implementar, esta mudança na herança de `HttpServlet` não causa nenhum efeito direto.

Na Figura 2.12 pode-se ver o diagrama de classes do padrão completo. Foram acrescentadas as classes de exceção `ApresentacaoException` e `ProcessamentoException`, ambas herdando de `ServletException`, e cada uma destas podem ser lançadas pelos métodos dos *handlers* de Apresentação e Processamento, respectivamente.

A classe `ControladorHandlers` é um *servlet* que faz o papel do Controlador de *Handlers* mostrado na Seção Estrutura. Este é implementado como um *servlet* porque precisa receber todas as requisições *Web*, e só depois de interpretá-las, repassá-las para os *handlers* responsáveis pelo sua execução.

Uma variação da implementação para resolver o problema do aumento do número de classes do sistema é permitir que o desenvolvedor possa agrupar operações relacio-

nadas ou semelhantes em um único *handler*. Com isso seria necessário implementar um mecanismo de execução de *handlers* mais elaborado, onde além de informações a respeito dos *handlers* a serem executados numa determinada requisição, deveria haver parâmetros indicando que operação executar em cada um deles. Este mecanismo pode ser implementado com a utilização da API *Reflection* [41] de Java.

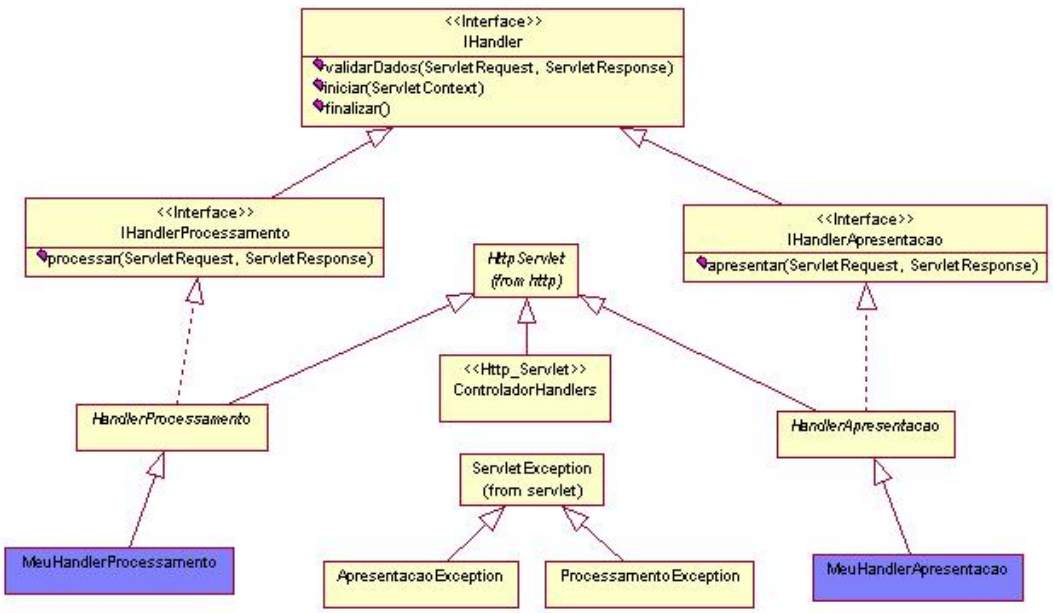

Figura 2.12: Estrutura do padrão refinada e no ambiente de *servlets*.

Código de exemplo

Para exemplificar o uso do padrão são mostradas as implementações de duas classes do sistema bancário, um *handler* de apresentação e outro de processamento. A Figura 2.12, mostrada anteriormente, dá uma idéia geral da estrutura destas classes e como elas são compostas para atender as requisições do sistema. No trecho abaixo é mostrada a implementação concreta destas classes em Java.

A classe `HP_Login` é um *handler* de processamento que executa a operação de *login* no sistema. Como pode ser visto na linha 1, este estende a classe `HandlerProcessamento`, herdando assim o comportamento de *handler* de processamento. Na linha 3 é declarada uma variável do tipo `Sistema`, que agrupa todos os serviços do sistema e é uma implementação dos padrões *Facade* e *Singleton*. Na inicialização deste *handler* (chamada ao método `iniciar`) é recuperada uma instância do sistema para que seja usado durante o processamento (linha 7).

```
1:   public class HP_Login extends HandlerProcessamento {
2:
3:      private Sistema sistema;
4:
```

```
5:      public void iniciar(ServletConfig config)
6:         throws ProcessamentoException{
7:         sistema = Sistema.getInstancia();
8:      }
```

O método `validarDados` verifica se os parâmetros da requisição necessários para execução estão presentes (linhas 13 e 14), caso não estejam é lançada uma exceção indicando que o processamento deve ser interrompido.

```
9:      public void validarDados(HttpServletRequest request,
10:                              HttpServletResponse response)
11:        throws ProcessamentoException{
12:
13:        if(request.getParameter("login") == null ||
14:           request.getParameter("senha") == null){
15:           throw new ProcessamentoException(
                   "Parâmetros imcompatíveis",null);
16:        }
17:     }
```

As linhas 24 e 25 são usadas para recuperar o valor do login e senha, passados como parâmetro da requisição. Estas são usadas para fazer a validação do usuário no sistema através de uma chamada ao método `validarUsuario` do objeto Facade (linha 27). Se o login ocorrer com sucesso, as informações do usuário são recuperadas e armazenadas em um objeto do tipo `Usuario` através de uma chamada ao método `recuperarUsuario` da Facade (linha 28). Após este processo, o objeto é armazenado no `request` para que possa ser recuperado pelo *handler* de apresentação associado a esta requisição.

```
18: public void processar(HttpServletRequest request,
19:                       HttpServletResponse response)
20:    throws ProcessamentoException{
21:    String login,senha;
22:    Usuario usuario;
23:
24:    login = request.getParameter("login");
25:    senha = request.getParameter("senha");
26:
27:    if(sistema.validarUsuario(login,senha)){
28:       usuario = sistema.recuperarUsuario(login);
29:       request.setAttribute("usuario",usuario);
30:    }
31:    else{
32:       throw new ProcessamentoException("Login Inválido",null);
33:    }
34: }
35:}
```

O `HA_MenuMovimentacao` é um *handler* de apresentação, por isso ele estende a classe `HandlerApresentacao`. Seu método `validarDados` verifica a existência do parâmetro `usuario` (linha 7).

```
1: public class HA_MenuMovimentacao extends HandlerApresentacao {
2:
3:  public void validarDados(HttpServletRequest request,
4:                           HttpServletResponse response)
5:    throws ProcessamentoException{
6:
7:    if(request.getAttribute("usuario") == null){
8:      throw new
9:        ApresentacaoException("Parâmetros imcompatíveis",null);
10:   }
11: }
```

O método apresentar recupera o objeto do tipo `Usuario` através do `request` (linha 23). A lógica principal de montagem da página está na linha 26, onde a página é montada através de uma chamada ao método `WebCompiler.processaPagina()`. Este método recebe com entrada um *array* de chaves, outro de valores e um nome de arquivo. Este arquivo é um *template* de uma página HTML contendo alguns identificadores especiais nos locais onde serão inseridas informações dinâmicas. A tarefa deste método é varrer o arquivo procurando ocorrências de algumas das palavras do *array* de chaves e substituí-las por palavras do *array* de valores. Na linha 27 a página de resposta é enviada para o cliente *Web*.

```
13: public void apresentar(HttpServletRequest request,
14:                        HttpServletResponse response)
15:    throws ApresentacaoException{
16:
17:    PrintWriter out;
18:    String pagina;
19:    Usuario u;
20:
21:    out = response.getWriter();
22:    try{
23:      u = (Usuario) request.getAttribute("usuario");
24:      String[] chaves = {"$USUARIO"};
25:      String[] valores = {u.getName()};
26:      pagina = WebCompiler.processaPagina(chaves, valores,
                                             "Menu_Movimentacao.html");
27:      out.println(pagina);
28:    }
29:    catch(Exception e){
30:       throw new ApresentacaoException("Erro de Apresentação",e);
31:    }
32: }
33:}
```

Usos conhecidos

- **Portal Encontre & Compre** [28] - Sistema de consultas dos anunciantes Listel, que também permite que o visitante faça transações de negócios on-line com os anunciantes;
- **O Sistema de Fomento Lattes** [27] - Sistema de informação para gestão de programas de fomento ao desenvolvimento científico e tecnológico. O módulo responsável pela emissão de parecer de consultor *Ad hoc* usa o padrão aqui descrito em sua implementação;
- **Prospectar** [54] - Sistema de prospecção tecnológica do Governo Federal;
- **Web2Billing** [57] – É uma solução completa de EBPP (*Eletronic Bill Presentation and Payment*), desenvolvida pela Wiser Tecnologies, e que permite a criação, geração, gerenciamento, apresentação, consulta e pagamento de faturas *online*;
- **FiS (Financial Services)** – O projeto contempla a migração dos Módulos de Contabilidade, Crédito, Lojistas e Serviços da HiperCard, para um novo ambiente tecnológico (J2EE). Estes módulos são integrados ao Sistema de Integrado de Crédito da HiperCard (SIC), ao R3/SAP e a outros sistemas legados. Também faz parte do projeto o desenvolvimento de um módulo de Controle de Acesso único e centralizado que poderá ser utilizado por qualquer aplicação da HiperCard disponível neste novo ambiente;
- **Fep (Call Center no FEP)** – Desenvolvimento de uma aplicação para a Central de Atendimento HiperCard que autorizará compras no autorizador FEP (*Front End Processor*) e no sistema legado SIC (Sistema de Integrado de Crédito da HiperCard) via *browser*;
- **Gin (Sistema de Gestão Interna)** [19] – Sistema de apoio a gestão interna do CESAR com cadastros e relatórios gerais, além de englobar os sistemas financeiro e avaliação de colaboradores.

Todos os sistemas descritos anteriormente usam o padrão *Web Handlers* na construção de seus serviços *Web*. A implementação do padrão proposta neste documento foi fruto de um trabalho de correção dos problemas identificados nas implementações anteriores, mas a estrutura do padrão continua a mesma.

Padrões relacionados

- Na construção dos *handlers* de apresentação pode ser usado o padrão *Web Compiler* (Seção 2.3) para implementar a lógica de montagem das páginas dinâmicas. O uso do padrão neste contexto traz uma série de benefícios ao desenvolvimento, pois permite a separação entre o código HTML e Java, facilitando o desenvolvimento e a manutenção do sistema.

- É interessante usar o padrão de projeto *Facade* [17] para agrupar a lógica de negócio do sistema em um único ponto e fazer com que os *handlers* de processamento chamem estas funcionalidades, ao invés de implementarem-na diretamente em seus corpos.

- O Controlador de *Handlers* (componente da estrutura do *Web Handlers*) deve implementar o padrão *Web Interceptor* (Seção 2.2) já que o Controlador deve ser o único ponto de acesso aos *handlers* do sistema.

- O padrão *Super Component* (Seção 2.4) pode ser usado na implementação dos *handlers* de apresentação e processamento a fim de evitar a duplicação de código nos métodos `iniciar` e `finalizar`.

2.3 *Web Compiler*

Contexto

Aplicações *Web* criam dinamicamente páginas HTML como resultado de suas execuções. Todo componente que precisa gerar páginas para clientes *Web* executa os seguintes passos:

- Recupera informações que são necessárias para montar o conteúdo da página (informações dinâmicas), normalmente através de algum componente de *software* que encapsula todas as regras de negócio e acesso ao banco de dados;

- Recupera o conteúdo da página estática, que é a parte que não muda para as diversas requisições (na maioria dos casos este conteúdo contém apenas informações do *layout* da página);

- Compõe as informações dinâmicas com as estáticas e envia o resultado da operação (página dinâmica) para o cliente.

Uma prática muito comum no desenvolvimento de sistemas *Web* é especificar o código HTML resultante diretamente no código fonte do componente *Web*. Esta forma é muito usada, devido a sua simplicidade, facilidade de entendimento e por ser bem intuitiva. Esta técnica nada mais faz do que uma concatenação de textos, e funciona da seguinte forma: cria-se vários textos (*strings*) contendo a informação estática da página e durante a execução do pedido concatena-se os *strings* que representam a parte estática da página com os que são gerados dinamicamente. Um exemplo deste tipo de implementação em Java pode ser visto no trecho de código abaixo:

```
...
mensagem = consultarMensagem();
texto= "<HTML><HEAD>Exemplo 1</HEAD><BODY>";
texto=  texto + mensagem;
texto= texto + "</BODY></HTML>";
out.println(texto);
...
```

O que o trecho acima faz é consultar alguma mensagem que vai ser gerada dinamicamente de acordo com o contexto do pedido; esta mensagem é armazenada na variável `mensagem`. Depois um *string* representando a página que vai ser enviada como resposta à requisição do cliente é criado, concatenando algum trecho que é estático para toda requisição com o conteúdo da variável `mensagem`.

Apesar deste tipo de implementação ser bastante simples, ela tem uma série de desvantagens:

- Impossibilidade de ver o *layout* da página sem executar o componente *Web* – Muitas vezes durante o desenvolvimento de sistemas para *Web* é necessário fazer a visualização da página a ser gerada, mesmo antes do final do desenvolvimento dos componentes;

- Conflito entre as responsabilidades do *designers* e engenheiros de *software* – Geralmente as páginas ficam prontas antes da parte Java do sistema começar a ser implementado e elas sofrem muitas modificações durante o desenvolvimento. Este fato torna esta abordagem muito limitante, pois qualquer modificação no formato da página tem que ser feita dentro do código Java e por engenheiros de software, quando mudanças no *layout* da página deveriam ser feitas por *designers*;

- Qualquer modificação no *layout* da página obriga o desenvolvedor a recompilar a aplicação – Como o código HTML está misturado com o código dos componentes *Web*, qualquer modificação no *layout* da página requer que o código fonte dos componentes seja modificado diretamente. Para que as modificações tenham efeito na aplicação é necessário que o sistema seja recompilado. Esta necessidade de recompilar o código fonte dos componentes *Web* para qualquer modificação, mesmo que esta alteração seja muito simples, diminui a produtividade dos desenvolvedores;

- Dificulta a legibilidade do código dos componentes *Web* – Como existe código HTML diretamente dentro do código fonte dos componentes, teremos que tratar duas linguagens dentro de um único arquivo. Os construtores da linguagem de programação se confundem com os de HTML e isto faz com que o código fonte se torne ilegível para sistemas muito grandes. Nestes casos, mesmo que o sistema seja simples, se a página HTML gerada tiver muita informação dinâmica, o código será grande e complicado. E esta característica não é desejável: a complexidade do componente *Web* não deve depender do *layout* ou tamanho da página que vai ser gerada.

Problema

Como desenvolver uma aplicação *Web* de forma a evitar que o *layout* das páginas HTML estejam misturadas com a lógica de execução das operações do sistema?

Forças

- Garantir a legibilidade do código dos componentes *Web*;

- Permitir que engenheiros de *software* e *designers* possam desenvolver partes diferentes do sistema;

- Garantir a produtividade no desenvolvimento.

Solução

Para evitar todos os problemas descritos anteriormente podemos separar o *layout* da apresentação do código de acesso aos serviços do sistema e a manipulação dos resultados, deixando-os em entidades diferentes e fazendo a ligação destes através de um componente específico, chamado aqui de *Web Compiler*. A Figura 2.13 dá uma idéia deste funcionamento.

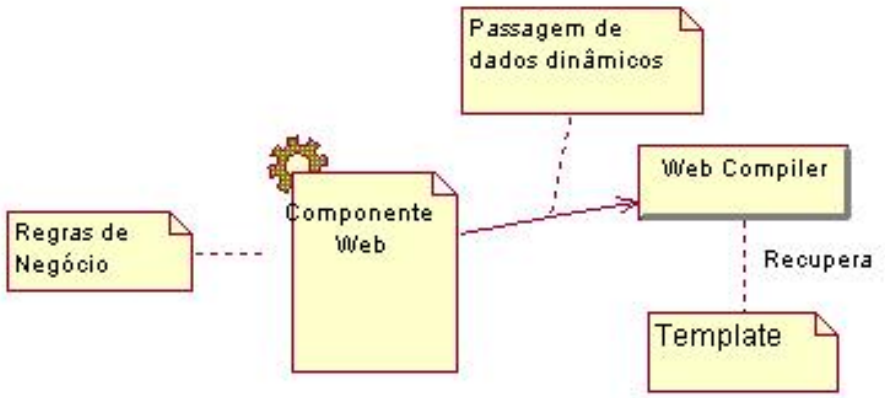

Figura 2.13: Uso do padrão *Web Compiler*.

O componente no qual reside a parte estática da apresentação é o `Template`, pois ele contém toda a parte estática da página HTML de resposta. Toda as invocações à operações do sistema e métodos de negócio continuam dentro do componente *Web*. O `Web Compiler` é responsável por compor o *layout* da página (residente no componente `Template`) com as informações dinâmicas resultado da execução das operações do sistema.

Ao invés de possuir todo o código de apresentação e fazer a composição das informações dinâmicas e estáticas, o componente *Web* só precisa informar ao `WebCompiler` qual o `Template` que deseja usar e que informações dinâmicas deseja inserir nele para ter o resultado desejado: a página de resposta para o cliente.

Aplicabilidade

Este padrão deve ser usado em qualquer sistema *Web* pois o *layout* das páginas HTML misturado com a lógica de execução das operações do sistema traz uma série de problemas, como já visto.

O *Web Compiler* pode ser estendido para outros tipos de aplicações que não possuam páginas HTML como interface gráfica, pois a funcionalidade dele é bem genérica: separar layout de apresentação das chamadas a operações do sistema e tratamento do resultado.

Estrutura

A estrutura do padrão *Web Compiler* é especificada através do diagrama de classes UML da Figura 2.14.

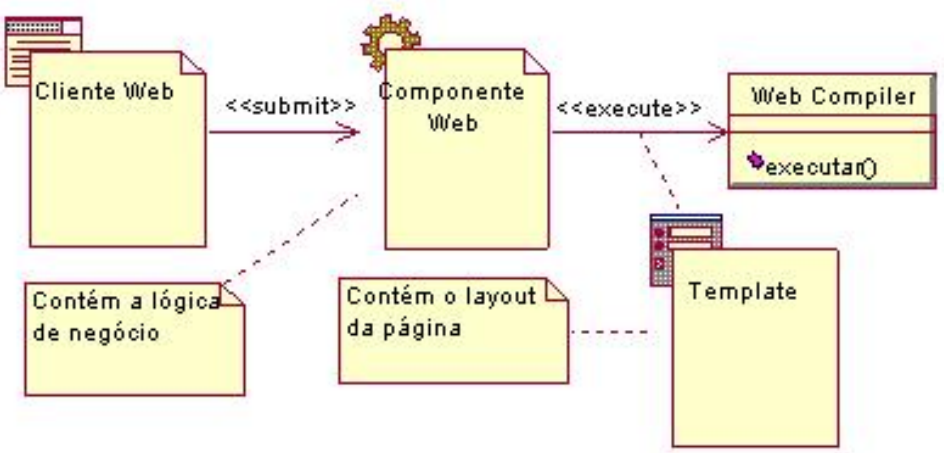

Figura 2.14: Estrutura do padrão *Web Compiler*.

Participantes

- `Cliente Web` – Responsável por enviar a requisição para o componente *Web* informando os dados necessários para a execução das operações do sistema;
- `Componente Web` – Responsável pela execução e controle do serviço solicitado pelo cliente;
- `Web Compiler` – Compõe um `Template` com as informações dinâmicas geradas pela aplicação;
- `Template` – Contém a parte estática da resposta ao `Cliente Web`.

Dinâmica

- `Cliente Web` – Faz uma requisição para o `Componente Web` passando informações necessários para execução do serviço;
- `Componente Web` – Recebe as informações, executa as operações do sistema, informa qual o `Template` que deve ser usado e controla a execução do `Web Compiler`;
- `Web Compiler` – Compõe o `Template` com as informações dinâmicas geradas pelo sistema;
- `Template` – *Layout* da página de resposta ao `Cliente Web`.

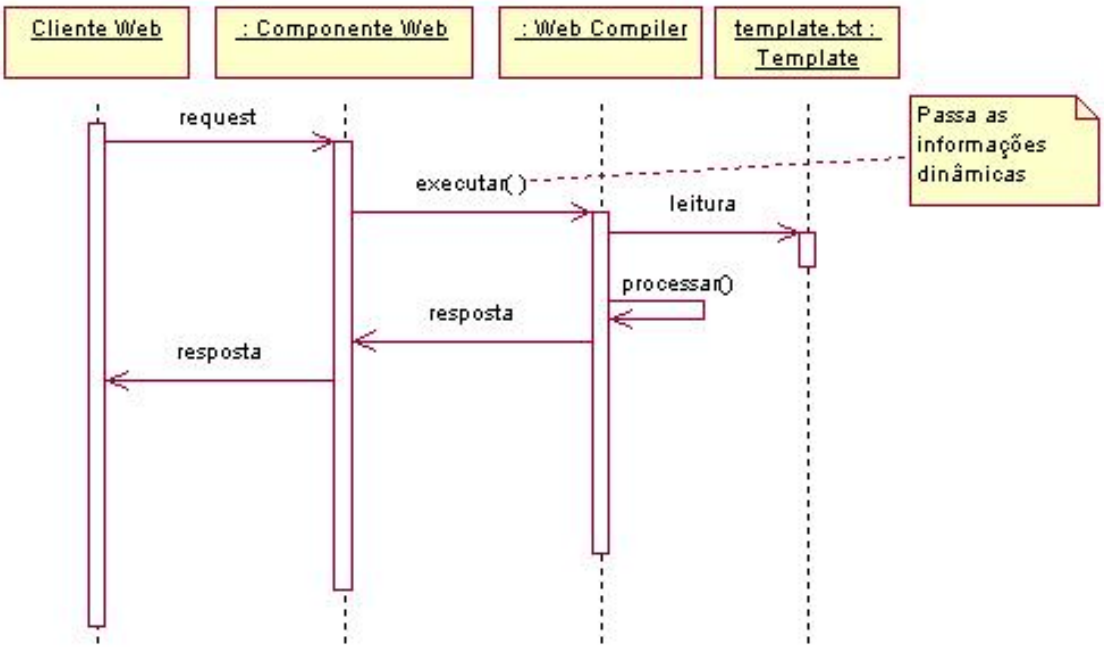

Figura 2.15: Diagrama de seqüência dos componentes do padrão *Web Compiler*.

Conseqüências

- Separação completa entre o código de apresentação (geralmente HTML) e o código de processamento da requisição (Java, por exemplo) – Ambas as partes podem ser modificados de forma independente e conseqüentemente por pessoas diferentes (como *designers* e engenheiros de software). Além disto, o código dos componentes *Web* torna-se mais legível;

- *Layout* da página pode ser visto independentemente da execução da aplicação – Se o `Template` for um arquivo HTML, ele pode ser visualizado em qualquer ferramenta de visualização HTML independente da aplicação estar pronta;

- Alterações no *layout* da página não requerem que a aplicação seja recompilada - De fato, mudanças no arquivo HTML podem ser feitas a qualquer momento sem a necessidade de recompilação dos componentes *Web* que a usam;

- A utilização deste componente causa um pequeno atraso no processamento da requisição, já que é necessário realizar um processamento a mais para inserir as informações dinâmicas ao conteúdo do *template* da página de resposta.

Implementação

Este padrão de projeto é genérico o suficiente para ser implementado de diversas formas, inclusive em outras linguagens de programação. O objetivo principal do mecanismo é substituir a ocorrência de determinadas palavras do *template* por outras palavras que representam a informação dinâmica. Para isso é necessário informar ao *Web Compiler*

qual o *template* a ser processado e que palavras devem ser substituídas pelos valores dinâmicos.

A forma mais simples de implementar este padrão é através da inserção de identificadores especiais no arquivo (Template) para marcar os locais onde deve ser inserida uma determinada informação dinâmica. A partir deste arquivo e do mapeamento de identificadores especiais em valores dinâmicos, o *Web Compiler* pode realizar o seu processamento substituindo cada ocorrência de um identificador especial no arquivo pelo valor dinâmico associado a ele.

Um exemplo da execução deste mecanismo pode ser visto na Figura 2.16. O *Web Compiler* recebe o *template* e um mapeamento de identificadores em valores, o resultado de sua execução é um texto onde as informações dinâmicas são substituídas pelas ocorrências dos identificadores no *template*.

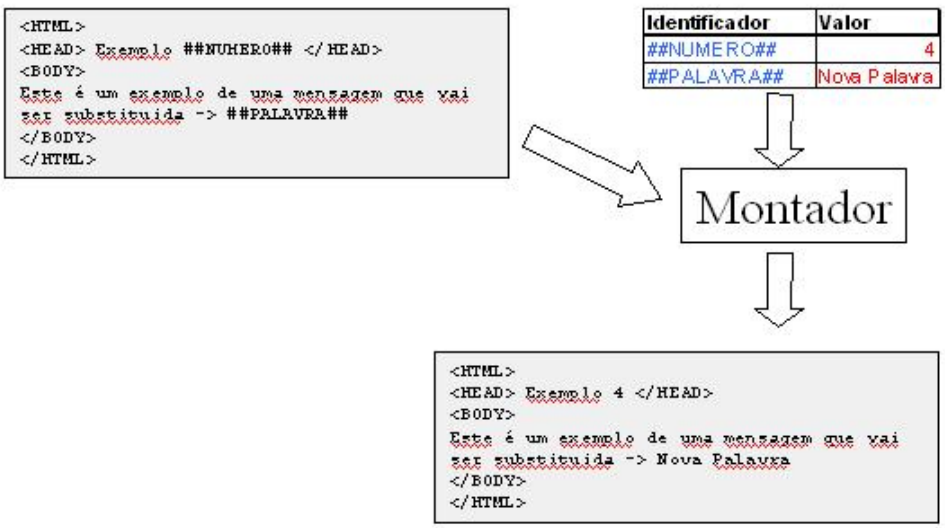

Figura 2.16: Exemplo de implementação do padrão *Web Compiler*.

Afim de melhorar a performance com o uso deste padrão pode-se definir um mecanismo mais eficiente para manipulação dos arquivos de *templates*, que ao invés de ler o arquivo a cada execução fosse capaz de manter o seu conteúdo em *cache*. A implementação destes mecanismos de *cache* exige a preocupação em realizar verificações períodicas para recarregar mudanças realizadas por processos externos nos arquivos de *template*.

Outras implementações mais elaboradas permitem que sejam inseridos pequenos trechos de códigos que, ao executarem, produzem alguma informação dinâmica. Exemplos destas tecnologias são JSP [43], Web Macro [29], FreeMarker [61] e Velocity [52].

Código de exemplo

O `ServletExemploCompiler` usa o `WebCompiler` para fazer a composição das informações dinâmicas com o *template* da página. O conteúdo do *template* está gravado em um arquivo chamado `template.txt`. O nome do arquivo e mais dois *arrays* de *strings*, um com os identificadores (linha 6) e outro com os valores dinâmicos (linha 7), são

passados como parâmetro para o método `executar` da classe `WebCompiler` (linha 9), que retornará um texto que é o resultado da composição de informações dinâmicas e estáticas. Os valores dinâmicos são normalmente gerados como resultado da invocação de métodos de negócio, apesar do exemplo abaixo não ilustrar este fato.

```
1 : public class ServletExemploCompiler extends HttpServlet{
2 :    public void doPost(ServletRequest request,
                          ServletResponse response){
3 :       String template = "template.txt";
4 :       String resposta;
5 :
6 :       String[] identificadores = {"##PALAVRA##","##NUMERO##"};
7 :       String[] valores = {"Nova Palavra","4"};
8 :
9 :       resposta = WebCompiler.executar(template,identificadores,
                                          valores);
10:       out.println(resposta);
11:    }
12: }
```

O código do `WebCompiler` é muito simples. Ele lê o conteúdo do arquivo e executa as substituições em memória, retornando o texto de resposta ao final do seu processamento.

Usos conhecidos

- **FreeMarker** [61] – Soluciona o problema de misturar código HTML com Java encapsulando o código HTML em *templates*, que são interpretados e executados por um componente que faz o papel do *Web Compiler*. O código fonte para *templates* é um documento HTML que contém instruções para incluir dados gerados dinamicamente. Estas instruções são simples (de modo que os *Web designers* ainda possam fazer seu trabalho), mas poderosas o bastante para permitir que o uso de estruturas de dados de complexidade arbitrária;

- **WebMacro** [29] – Funciona de forma semelhante a FreeMarker. A única diferença é em relação a sintaxe das instruções que inserem dados dinâmicos no template HTML;

- **Velocity** [52] – É um *framework* para camada de apresentação e tem um funcionamento similar a Freemarker e WebMacro, mas com uma sintaxe diferente;

- **API do CESAR** [9] – Implementa o *Web Compiler* através da criação de uma classe que tem métodos para carregar arquivos HTML e executar a substituição léxica de palavras. Estes métodos recebem um mapeamento de nome de identificadores em valores e o nome do *template*, lê o conteúdo do arquivo e por fim percorre o texto em memória substituindo todas as ocorrências dos identificadores existentes no mapeamento em seus respectivos valores.

Padrões relacionados

- **Skin** [49] – O *Web Compiler* é muito similar ao *Skin*. De fato, ambos resolvem o mesmo problema: evitar que o *layout* das páginas HTML estejam misturadas com a lógica de execução das operações do sistema. O padrão *Skin* é mais aplicado em situações onde o processamento do *template* é apenas uma substituição de *tokens*, enquanto que o *Web Compiler* pode ser usado em situações onde o *template* possui elementos tão complexos quanto os construtores das linguagens de programação: *loops*, escolhas, declarações de funções e atributos, chamadas à funções.

2.4 *Super Component*

Contexto

Muitos componentes *Web* possuem um ciclo de vida bem definido, que é gerenciado pelo servidor *Web*, ou mais especificamente por módulos acoplados a este. O ciclo de vida envolve basicamente três estados: inicialização, serviço e destruição. A fase de inicialização é o momento em que o componente *Web* deve carregar os recursos mais custosos e que serão compartilhados por todas as requisições. Exemplos de operações executadas nesta fase são abertura de conexão com banco de dados, abertura e leitura de arquivos, criação ou instanciação de objetos que encapsulam as regras de negócio do sistema. Exemplos de tecnologias *Web* que possuem este ciclo de vida são *servlets*, JSP e *Web Handlers* (veja a Seção 2.2).

O trecho de código abaixo dá uma idéia da implementação de um método `init`, da fase de inicialização, bem típico de aplicações com *servlets*. Na linha 4 é feita a leitura de um arquivo de inicialização que será usado ao longo da fase de serviço. A linha 5 é um exemplo de instanciação de um objeto *Singleton* [17]; é através deste que o *servlet* deverá invocar os serviços do sistema.

```
1    ...
2 :  public void init(ServletConfig config){
3 :    try{
4 :      conteudoArquivo = Biblioteca.lerArquivo("ServletTeste.init");
5 :      sistema = Sistema.getInstancia();
6 :    }
7 :    catch(IOException e){
8 :   // Tratamento do erro na leitura do arquivo
9 :    }
10:    catch(SistemaException e){
11:    // Tratamento do erro de inicialização do sistema
12:    }
13:  }
14: ...
```

Em geral estes recursos ou pelo menos a forma de inicializá-los são compartilhadas por todos os componentes *Web* do sistema. Isto acontece por causa da própria natureza do recurso, ou seja, por serem caros em relação ao tempo de iniciá-los (ex: conexões

na rede ou leitura de arquivos) ou por serem limitados (ex: conexões com Banco de dados). Por este motivo é percebido que o corpo dos métodos de inicialização dos diversos componentes *Web* de um determinado sistema são similares, e em muitos casos, idênticos.

De fato, é comum existirem sistemas com dezenas ou centenas de componentes com o corpo do método de inicialização repetido em todos. O problema se torna ainda pior pois a fase de destruição é o momento em que se deve liberar todos os recursos alocados durante a inicialização, e por conseqüência da similaridade dos métodos de inicialização, os métodos de destruição também são muito similares, já que eles fazem exatamente o oposto, ou seja, liberar ou finalizar os recursos.

Problema

Como evitar a duplicação de código de inicialização e destruição nos diversos componentes *Web* de um sistema?

Solução

Esta duplicação de código na inicialização e destruição é realmente desnecessária e pode ser facilmente evitada criando um componente que contém apenas os métodos de inicialização e destruição, com o código comum a todos os componentes *Web* do sistema. Os demais componentes *Web* devem estender este último e apenas especificar a operação para execução da requisição. A classe `SuperServlet` implementa o padrão *Super Component* para um sistema baseado em *servlets*. Como pode ser visto, ela contém apenas a lógica de inicialização.

```
1: public class SuperServlet extends HttpServlet {
2:   private Sistema sistema;
3:   private String conteudoArquivo;
4:
5:   public void init(ServletConfig config)
           throws ServletException{
6:     try{
7:       conteudoArquivo=Biblioteca.lerArquivo("XXX.ini");
8:       sistema = Sistema.getInstancia();
9:     }
10:    catch(IOException e){
11:    //Tratamento do erro na leitura do arquivo
12:    }
13:    catch(sistemaException e){
14:    //Tratamento do erro de inicialização do sistema
15:    }
16:  }
17:}
```

Os demais *servlets* do sistema irão se preocupar apenas em definir o comportamento do método de serviço, enquanto herdam o comportamento da inicialização e finalização.

De fato a classe `ServletXXX` estende o `SuperServlet` e define apenas o método `service` (responsável pelo tratamento da requisição).

```
1: public class ServletXXX extends SuperServlet {
2:   ...
3:   public void service(ServletRequest request,
                         ServletResponse response){
4:   ... // Invocação dos métodos de negócio do sistema
5:   ... // Formatação da página de resposta
6:   }
7: }
```

Aplicabilidade

O fato dos componentes *Web* possuírem métodos de inicialização e finalização similares é uma característica muito comum em qualquer sistema baseado neste tipo de tecnologia. Por este motivo o padrão aqui descrito é bem geral e poderá ser usado na maioria das situações. O padrão deve ser aplicado principalmente em sistemas que possuem recursos ou configurações que são compartilhados pelos componentes *Web* do sistema.

Uma ocasião mais concreta, de *refactoring*, que justifique o uso do *Super Component* é se for verificada a ocorrência de código referente a inicialização e destruição duplicado em mais de um componente *Web* do sistema.

Este padrão de projeto pode ser aplicado apenas a uma parte do sistema, desde que esta parte possua métodos de inicialização e destruição parecidos. Outra opção é usar mais de um *Super Component* no sistema, cada um sendo pai de um conjunto disjunto de componentes *Web*.

Estrutura

A Figura 2.17 especifica a estrutura do padrão através do diagrama de classes de UML.

Participantes

- `Super Component` - Possui apenas os métodos de inicialização e finalização, contendo o código comum aos componentes que herdam dele;
- `Destino` - Componente responsável pela execução final da requisição e deve estender a classe `Super Component` que atenda a seus requisitos de inicialização e destruição.

Dinâmica

Como pode ser visto na Figura 2.18, durante a inicialização de um componente *Web* do sistema, através de uma chamada ao seu método `init`, é feita uma chamada ao método de inicialização do `Super Component`, onde reside o código de inicialização comum aos componentes *Web* do sistema.

O comportamento dos componentes do padrão no momento da finalização é similar a inicialização. Quando o componente *Web* for destruído, através de uma chamada ao

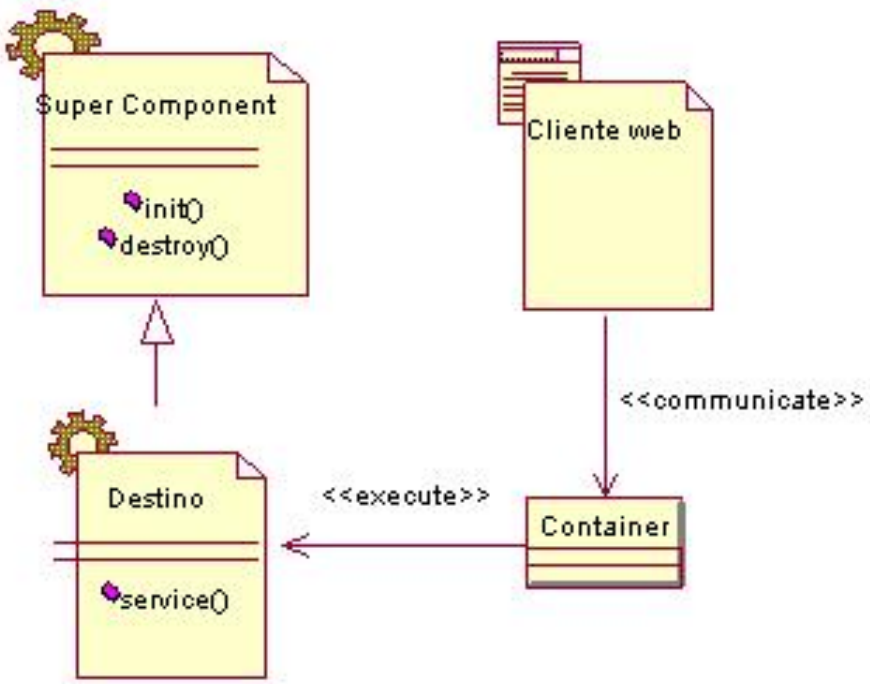

Figura 2.17: Diagrama de classes do padrão Super Component.

seu método `destroy`, é feita uma chamada ao método de destruição, que foi herdado do `Super Component` e onde reside a lógica de liberação dos recursos alocados na fase de inicialização.

Conseqüências

- **Código mais limpo** - Com o uso deste padrão os componentes *Web* do sistema precisam ter apenas a implementação do método de serviço, já que toda a complexidade de inicialização e destruição é resolvida no *Super Component*.
- **Evita a duplicação de código** - O comportamento comum dos componentes *Web* é herdado do *Super Component*.

Implementação

É comum existirem situações onde um conjunto de componentes *Web* do sistema não possuem os métodos de inicialização e destruição exatamente iguais, mas possuem pelo menos parte deles em comum. Uma forma de implementar estas situações evitando a duplicação de código é criar o *Super Component* contendo a parte da inicialização e destruição que é comum a todos. Nestes casos os demais componentes do sistema continuarão a herdar do *Super Component* e deverão ter seus próprios métodos de inicialização e destruição, mas estes deverão conter apenas o trecho de código que é específico dele e uma chamada ao método do *Super Component*. Este comportamento em orientação a objetos é conhecido como redefinição de método [32], pois o método herdado é sobrescrito na classe filha.

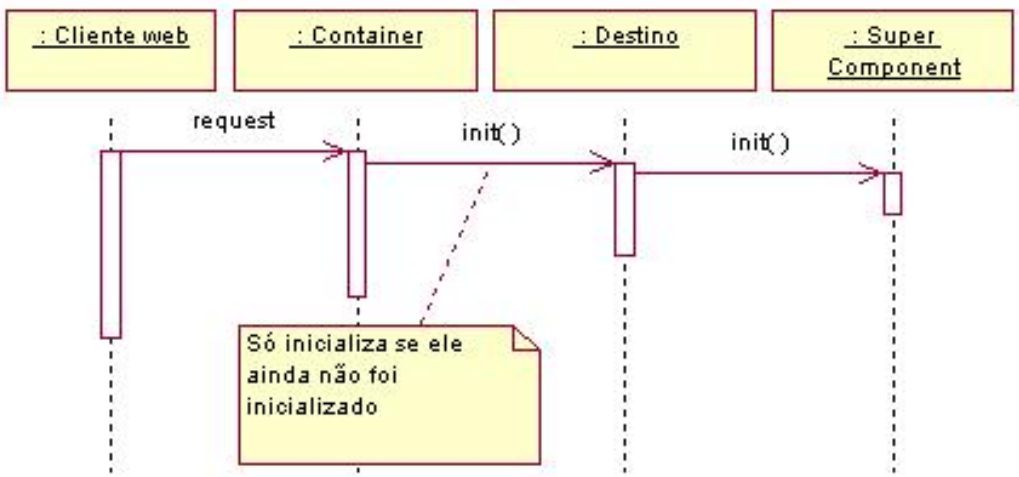

Figura 2.18: Diagrama de seqüência dos componentes do padrão no momento da inicialização de um componente.

Um exemplo deste uso em Java pode ser visto no `ThinServlet`. Na linha 4 é feita uma chamada ao método de inicialização da classe pai. A linha 5 demonstra que podem ser executadas outras operações, além das predefinidas na superclasse.

```
1: public class ThinServlet extends SuperServlet {
2:    ...
3:    public void init(ServletConfig config)
        throws ServletException{
4:       super.init(config);
5:       ... // Abre conexao com o BD
6:    }
7:    public void service(ServletRequest request,
                          ServletResponse response){
8:      ... // Execução da requisição
9:  }
10:}
```

Em casos onde é usada tecnologia orientada a objetos, como *servlets* e *Web Handlers*, a implementação do padrão é bastante natural, com o uso de herança. Mas em tecnologias que não possuem este conceito não é possível usar o padrão como ele foi definido. Uma alternativa nestes casos é fazer o uso de delegação para evitar a duplicação de código.

Em sistemas baseados em JSP, por exemplo, também é possível obter os benefícios deste padrão. Em JSP existe um atributo da diretiva `page` que especifica de qual classe o *servlet*, gerado pelo processo de compilação automática da página, deve herdar. O trecho abaixo exemplifica este uso:

```
<%@page ... extends="SuperServlet" %>
...
```

Código de exemplo

As classes `SuperServlet` e `ServletComum` mostradas na seção Contexto são exemplos de implementação em Java do padrão *Super Component*.

Usos conhecidos

- **Sistema Wide** [64] - *Web Information of Development* usa este padrão para evitar a repetição de código de inicialização dos *servlets*. O *Super Component* do Wide, denominado de ServletWide, recupera, em seu método init, uma instância de um objeto do tipo Sistema e armazena-a em um de seus atributos. A classe Sistema encapsula toda a regra de negócio da aplicação e implementa os padrões de projeto *Facade* e *Singleton*. Os demais *servlets* do sistema possuem apenas método de serviço e usam o atributo do tipo Sistema para executar todas as regras de negócio sem precisar declarar ou iniciá-lo, pois a declaração e inicialização já foi herdada.

- **Portal encontre-e-compre** [28] - O Portal encontre-e-compre usa este padrão de forma similar ao WIDE.

- **FiS (Financial Services)** – Sistema desenvolvido pelo CESAR, cujo projeto contempla a migração dos Módulos de Contabilidade, Crédito, Lojistas e Serviços da HiperCard, para um novo ambiente tecnológico (J2EE). Estes módulos são integrados ao Sistema de Integrado de Crédito da HiperCard (SIC), ao R3/SAP e a outros sistemas legados. Também faz parte do projeto o desenvolvimento de um módulo de Controle de Acesso único e centralizado que poderá ser utilizado por qualquer aplicação da HiperCard disponível neste novo ambiente;

- **Central de Regulação SUS** - Sistema de *software* que possibilita tratar o atendimento a pacientes em uma dada região de saúde, de forma a garantir o melhor atendimento à população. Dois dos seus módulos: marcação de consultas e exames especializados e controle de internação hospitalar, desenvolvidos pelo CESAR [9], usam o padrão de projeto **Super Component** na implementação de seus componentes *Web* para evitar a duplicação de código de inicialização e finalização.

Capítulo 3

Framework para implementação de sistemas *Web*

Neste capítulo apresentamos um framework para implementação de sistemas baseados em servlets.

O principal objetivo do *framework* para implementação de sistemas *Web* é facilitar o desenvolvimento da camada de apresentação das aplicações *Web* [3], aumentando a produtividade, o reuso e o desacoplamento entre o código de processamento das requisições e o código de montagem de páginas, sem acrescentar complexidade de entendimento, desenvolvimento e distribuição ao sistema. O *framework* provê um mecanismo para estruturar e organizar serviços *Web* baseado em *servlets* de forma a evitar repetição de código. Mais especificamente ele funciona como apoio para a implementação do padrão de projeto **Web Handlers** que foi apresentado na Seção 2.2. Por isso os desenvolvedores de sistemas *Web* que desejarem implementar este padrão de projeto em seus sistemas podem usar o *framework* para auxiliar o trabalho.

Este *framework* é resultado do estudo sobre padrões, diretrizes e boas práticas para desenvolvimento de sistemas *Web*. Ele foi desenvolvido com o objetivo de melhorar a produtividade no desenvolvimento de sistemas do CESAR [9], permitindo um maior reuso e legibilidade dos componentes envolvidos na solução. Seus requisitos foram elicitados a partir da experiência prática pessoal com sistemas desenvolvidos no CESAR, antes e durante o desenvolvimento desta dissertação, já outros foram resultado das pesquisas de mestrado.

Hoje ele é usado em empresas como o CESAR [9] e Mobile [56] e está em contínuo desenvolvimento, afim de agregar novas funcionalidades e acompanhar as tendências e tecnologias de desenvolvimento.

Este capítulo está organizado da seguinte forma. A Seção 3.1 dá uma noção básica do funcionamento do *framework*, apresentando as principais funcionalidades e entidades do mesmo. Já a Seção 3.2 mostra o projeto do *framework*, detalhando como as entidades e funcionalidades foram implementadas e relatando o motivo de algumas das decisões do projeto. A Seção 3.3 apresenta um guia detalhado de uso do *framework*, nesta estão todas as informações necessárias para que os desenvolvedores criem as entidades requeridas para o funcionamento de seus sistema, além de informações de como customizar algumas funcionalidades padrões desta implementação. Por fim, a Seção 3.4 apresenta alguns sistemas *Web* comerciais que usam o *framework*.

3.1 Introdução

O *framework* é baseado na construção de entidades denominadas *handlers* (como já visto na Seção 2.2). Com esta estruturação é necessário criar um *handler* de processamento para cada operação do sistema e um de apresentação para cada página dinâmica. Cada requisição *Web* dispara uma execução do lado do servidor que dinamicamente monta um par de *handlers* (um de apresentação e um de processamento) para responder ao cliente. Esta composição dinâmica é baseada em parâmetros da requisição do cliente e no resultado da execução do código de negócio. A entidade responsável pela montagem deste par e delegação da requisição é o Controlador de *Handlers*, um dos principais elementos do nosso *framework*. Os *handlers* de processamento, além de conterem lógica de execução das operações do sistema, possuem código responsável pela validação dos dados vindos do cliente *Web* e código responsável pela geração de dados para a execução do seu par. Os *handlers* de apresentação também possuem validação de dados, além de código de montagem das páginas dinâmicas. Esta validação é necessária porque estes precisam validar os dados gerados pelo seu pares, já que eles são entidades independentes

e podem ser compostos de diferentes formas.

As composições possíveis de *handlers* de apresentação e processamento são definidas em um arquivo XML que é carregado durante a inicialização do sistema. Neste arquivo é definida uma lista de serviços, onde cada serviço possui um *handler* de processamento e um mapeamento de eventos em *handlers* de apresentação. Cada requisição do cliente *Web* dispara um serviço específico, o que significa executar o seu *handler* de processamento, e dependendo do evento gerado por esta execução, processar um de seus *handlers* de apresentação.

O *framework* possui um mecanismo de tratamento de erros que tem como objetivo tirar parte da responsabilidade pelo tratamento dos erros do programador. Os erros (*exceptions*) gerados pela execução do sistema não precisam ser tratados no escopo (*handler*) da chamada do método que a gerou, o programador pode se abstrair de determinados erros (não esperados) e tratar apenas os que interessá-lo. No arquivo XML mencionado no parágrafo anterior, o programador pode especificar que *handlers* de apresentação devem tratar determinados tipos de exceções em Java, desta forma toda vez que uma exceção do sistema for levantada e a mesma não for tratada por nenhum *handler* do sistema, o *framework* automaticamente associa esta exceção ao *handler* de apresentação que deve tratá-la, colocando-o para executar logo em seguida.

Esta implementação do *framework* foi construída baseado na versão 2.2 de *servlets* [13] e testada também na versão 2.3 [12].

3.2 Características do *framework*

Como já descrito na Seção 2.2, os *Web handlers* são componentes que são executados em resposta a requisições do cliente e existem dois tipos dos mesmos: Apresentação e Processamento. No *framework*, eles são definidos como classes Java e possuem características similares aos *servlets*, tais como:

- **Ciclo de vida bem definido e gerenciado pelo ambiente** - Os *handlers*, assim como os *servlets*, possuem um ciclo de vida bem definido e que é controlado pelo ambiente no qual eles estão executando;
- **Instanciação automática dos *handlers*** - O ambiente é responsável por localizar e carregar os *handlers* necessários para a execução da requisição;
- **Reload automático de *handlers*** - As modificações feitas em *handlers* já carregados em memória são atualizadas automaticamente pelo ambiente;
- **Passagem de parâmetros de configuração** - Com o uso de *handlers* também é possível recuperar parâmetros de configuração. O meio no qual são passados estes parâmetros é o mesmo dos *servlets* (geralmente arquivos do *Servlet Container*) e forma de recuperá-los também.
- **Uso de Filtros** - É possível usar o conceito de Filtros [39] para *handlers*, desde que o *Web Container* suporte a especificação 2.3 de *servlets* [12].

Enfim, tudo que é possível fazer com *servlets* também pode ser feito com os *Web handlers* já que estes são construídos sobre a arquitetura de *servlets*.

3.2.1 Ciclo de Vida

Os *handlers* possuem um ciclo de vida bem definido que envolve basicamente três estados: Inicialização, Serviço e Destruição. A Figura 3.1 dá uma idéia deste ciclo de vida. Como os *Web handlers* são baseados em *servlets*, ambos possuem um ciclo de vida muito similar.

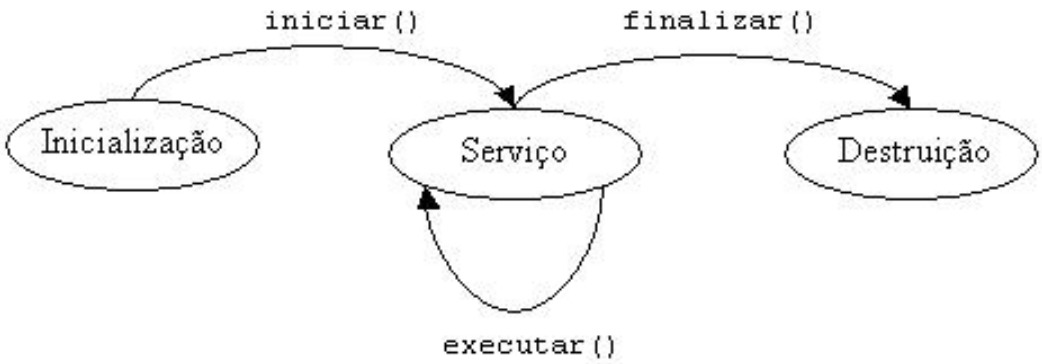

Figura 3.1: Ciclo de vida dos *Web Handlers*.

A Inicialização é a primeira fase do ciclo, todos os *handlers* passam uma única vez por este estado. É neste momento que o *handler* é carregado na memória e deve executar as operações relacionadas ao *setup* do serviço, ou seja, operações como alocação de recursos (por exemplo, conexão com banco de dados, abertura de arquivos), leitura de parâmetros de configuração, instanciação de objetos que executam as regras de negócio. A inicialização do *handler* é feita através de uma chamada ao seu método `iniciar`. É neste método que o programador deve colocar o código referente as operações de *setup*.

A fase de Serviço é o momento em que o *handler* responde as requisições dos clientes. O *handler* só pode entrar neste estado se tiver passado anteriormente pela Fase de Inicialização (como pode ser visto na Figura 3.1). Cada requisição dos clientes *Web* resulta na execução de um *thread* que deve fazer uma chamada ao método `executar` do *handler*. Para os *handlers* de apresentação este método delega a execução para outro método, o `apresentar`. Já para os *handlers* de processamento a execução é delegada para o método `processar`. Múltiplas requisições simultâneas resultam na execução de vários *threads* concorrentes.

A última fase do ciclo de vida é a Destruição, nesta o *handler* deve liberar todos os recursos alocados na Fase de Inicialização e, em alguns casos, executar operações de finalização (como por exemplo, registrar ocorrência no log, enviar e-mail de notificação, gravar estado em disco). A destruição do *handler* é feita através de uma chamada ao método `finalizar`, correspondendo a última chance do *handler* executar alguma operação antes de ser removido da memória, por isso é neste método que o programador deve colocar operações relativas à destruição do *handler*.

3.2.2 Implementação do *Framework*

O *framework* é composto por um conjunto de classes e interfaces escritas em Java que estão definidas sobre a API de *servlets* 2.2 e implementam o padrão de projeto *Web Handlers*, descrito na Seção 2.2. A Figura 3.2 mostra estes elementos e o relacionamento entre eles. Nas próximas seções eles são descritos com mais detalhes.

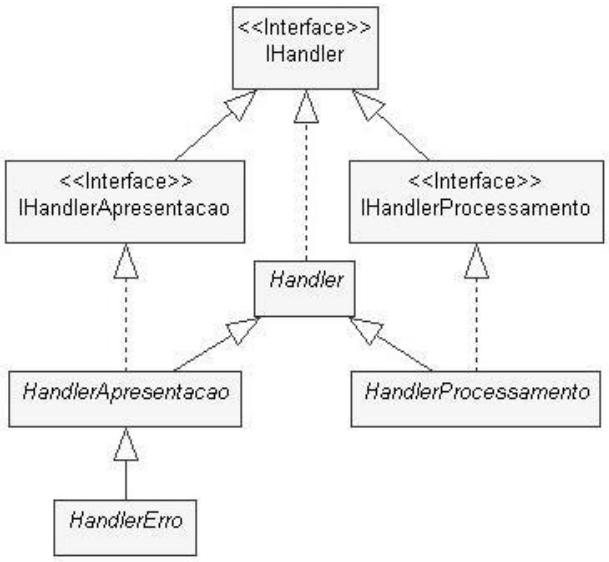

Figura 3.2: Diagrama UML dos componentes do *framework*.

Handler

Os *handlers* de apresentação e processamento possuem interfaces e parte do comportamento bastante similares. Por este motivo é interessante que exista uma classe que defina este comportamento compartilhado, servindo para dar apoio à construção de ambos os tipos de *handlers*. Este componente também é importante para que as demais classes do *framework* possam acessar os dois tipos de *handlers* de forma homogênea, ou seja, através de uma interface única.

O primeiro componente que dá suporte ao *framework* é a interface `IHandler`, ela especifica o comportamento comum aos dois tipos de *handlers*. Nela estão definidas três constantes e as assinaturas de quatro métodos.

```
1: public interface IHandler {
     ... \\ Declaração de constantes
2:   public void iniciar()throws HandlerException;
3:   public void executar(HttpServletRequest request,
       HttpServletResponse response)throws HandlerException;
4:   public void finalizar() throws HandlerException;
5:   public void validarDados(ServletRequest request,
       ServletResponse response)throws HandlerException;
6: }
```

Como pode ser visto, a interface `IHandler` define quatro métodos: `iniciar`, `executar`, `finalizar` e `validarDados`, que podem lançar a exceção `HandlerException`. Esta é uma exceção genérica do *framework*. Os métodos `iniciar`, `executar` e `finalizar` são métodos relacionados com o ciclo de vida do *handler* e já foram falados brevemente na Seção 3.2.1. O método `validarDados` é chamado em todas as requisições, sempre antes do `executar`, e é usado para que o programador possa verificar se todos os parâmetros esperados pela requisição estão presentes e são do tipo esperado. São passados dois argumentos para os métodos `executar` e `validarDados`, o primeiro do tipo `HttpServletRequest` e o outro `HttpServletResponse`. Estes são tipos da API de *servlets* e representam a entrada e saída do serviço, respectivamente.

O segundo componente que dá suporte ao *framework* é a classe abstrata `Handler`. Ela é uma implementação da interface `IHandler` e define a parte do comportamento que é comum aos dois tipos de *handlers*. Esta classe estende `HttpServlet`[1], para que os *handlers* possuam características similares aos *servlets*:

```
public abstract class Handler extends HttpServlet implements IHandler{
  protected boolean suportaGET = true;
  protected boolean suportaPOST = true;
  ...
}
```

As variáveis `suportaGET` e `suportaPOST` são usadas para especificar se o *handler* suporta requisições HTTP do tipo GET e POST. O valor *default* para elas é *true*, isto significa que os *handlers* suportam os dois tipos de operações, desde que o programador não mude os valores dos atributos. Estes valores podem ser alterados através da chamada ao método `suportaMetodo(int metodo)` da classe `Handler`, passando como parâmetro uma das constantes definidas na interface `IHandler`: POST, GET e GET_E_POST. As duas primeiras constantes representam as operações *Web* POST e GET e a última indica a união das duas operações. Por exemplo, quando o programador estiver escrevendo o código de um *handler* ele pode usar o trecho de código abaixo para garantir que o *handler* só vai responder requisições POST.

```
suportaMetodo(POST);
```

Os métodos `iniciar`, `finalizar` e `validarDados` são implementados com os corpos vazios. Isto foi feito para que o programador tenha a opção de implementar apenas os métodos que lhe for conveniente. Em situações em que o desenvolvedor não implementar alguns destes métodos não haverá problema algum, porque o *framework* vai executar os métodos vazios.

Alguns métodos que foram herdados da classe `HttpServlet` precisam ser redefinidos na classe `Handler` para que os *handlers* se comportem como *servlets*, são eles: `init`, `service` e `destroy`. Estes métodos estão relacionados com o ciclo de vida dos *servlets* e por isso o comportamento destes á basicamente fazer uma chamada ao método correspondente do ciclo de vida dos *handlers*, como pode ser visto no trecho abaixo:

[1]Classe da API de *servlets* que é a base para todos os *servlets*.

```
public final void init(ServletConfig config)
                  throws ServletException {

  super.init(config);
  iniciar();
}
public final void destroy() {
  super.destroy();
  finalizar();
}
public final void service(HttpServletRequest request,
                          HttpServletResponse response)
                  throws ServletException, IOException {
  executar(request,response);
}
```

Estes métodos foram definidos como `final` apenas para garantir que os desenvolvedores de *handlers* não possam mudar o comportamento destes tentando redefini-los. Nos métodos `init` e `destroy` é feita uma chamada ao método da superclasse; isto é feito para garantir a consistência dos *servlets* [24].

O método `executar` é o mais importante, já que ele especifica o comportamento para o tratamento das requisições dos clientes *Web*. Ele é um método interno do *framework*, por isso o programador não precisa saber nada a seu respeito.

O Controlador de *Handlers*, que é mostrado ainda nesta seção, implementa o padrão de projeto *Web Interceptor* (mostrado na Seção 2.1). Por isto os *handlers* não podem ser acessados diretamente pelo cliente *Web*, ao invés disto as requisições devem passar pelo Controlador de *Handlers* para só então serem recebidas pelos *handlers*. A primeira operação importante que o método `executar` faz é verificar se a requisição passou pelo interceptor, através de uma chamada ao método auxiliar `verificarSeguranca` passando o `request` como parâmetro (na linha 5 do trecho de código abaixo).

```
 1: public final void executar(HttpServletRequest request,
 2:      HttpServletResponse response) throws HandlerException{
 3:   String metodo;
 4:   metodo = request.getMethod();
 5:   verificarSeguranca(request);
 6:   if ((metodo.equals("GET") && suportaGET) ||
          (metodo.equals("POST") && suportaPOST)){
 7:     validarDados(request, response);
 8:     execucaoEspecifica(request, response);
 9:   }
10:   else {
11:     throw new MetodoNaoSuportadoException (metodo,this);
12:   }
13: }
```

Na linha 6 é feita uma verificação para garantir que o *handler* só executa requisições suportadas por ele. Se a verificação for válida, são executados os métodos `validarDados`

e `execucaoEspecifica`, caso contrário é lançada uma exceção indicando que o método HTTP requisitado não é suportado pelo *handler*. O método `execucaoEspecifica` é abstrato e é implementado nas classes `HandlerProcessamento` e `HandlerApresentacao`, subclasses de `Handler`.

Handler de Processamento

Os *handlers* de processamento são entidades que estão relacionadas com as operações do sistema propriamente ditas, ou seja, eles definem o comportamento apenas da parte de execução da requisição, deixando o tratamento da resposta para os *handlers* de apresentação.

Dois são os componentes que dão apoio à implementação desta entidade, definindo as assinaturas das operações e o comportamento comum aos *handlers* de processamento. O primeiro é a interface `IHandlerProcessamento`, que especifica o comportamento destes tipos de *handlers* e o outro é a classe abstrata `HandlerProcessamento` que define parte do comportamento. Estes componentes são apenas uma especialização dos componentes apresentados na seção anterior, como pode ser visto na Figura 3.3.

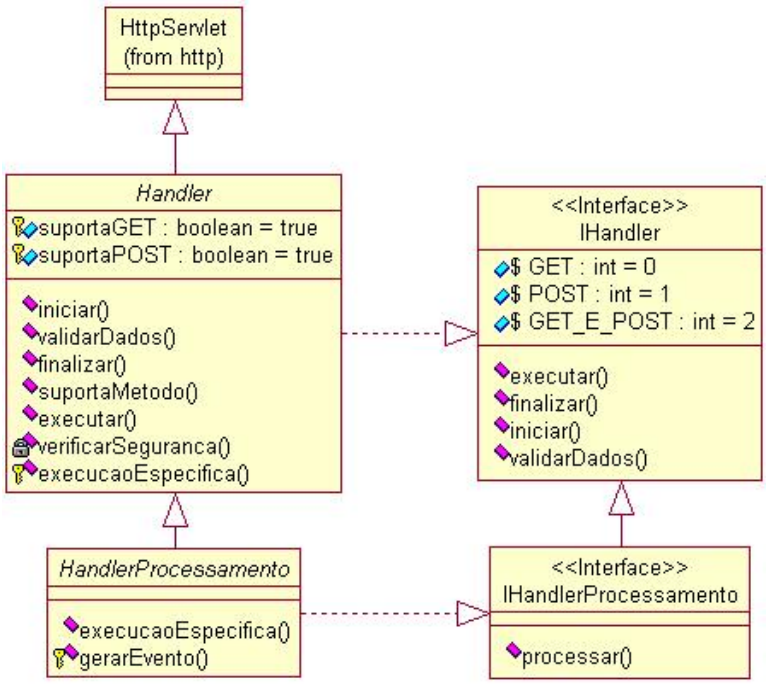

Figura 3.3: Diagrama das classes básicas do *framework* relacionadas com os *handlers* de processamento.

A interface `IHandlerProcessamento` é bastante simples, ela define apenas o método `processar` e estende a interface `IHandler`. Este método deve ser implementado pelos *handlers* de processamento definidos pelo desenvolvedor. Ele deve conter todas as operações relacionadas com a execução da requisição, desde a recuperação dos parâmetros do cliente até a passagem de dados para o *handler* de apresentação.

O último componente que dá suporte a implementação dos *Handlers* de processamento é a classe `HandlerProcessamento`, que é uma implementação da interface

`IHandlerProcessamento` e extensão da classe `Handler`. Esta classe implementa o método `execucaoEspecifica`, definido como abstrato na classe `Handler`. Seu comportamento é simplesmente fazer uma chamada ao método `processar` e qualquer erro que ocorra nesta execução é encapsulado em uma exceção do tipo `ProcessamentoException` e lançada pelo método:

```
public void execucaoEspecifica(HttpServletRequest request,
    HttpServletResponse response) throws ProcessamentoException{
  try{
    processar(request,response);
  }
  catch(Exception e){
    throw new ProcessamentoException(e);
  }
}
```

Outro método definido nesta classe é o `gerarEvento`, que deve ser usado pelos *handlers* de processamento para notificar o Controlador de *Handlers* que o seu processamento gerou um evento.

Mais detalhes do uso dos *Handlers* de apresentação são mostrados na Seção 3.3.2.

Handler de Apresentação

As entidades responsáveis pelo tratamento da resposta ao cliente são os *handlers* de apresentação. Estes não possuem execução de regra de negócio alguma, apenas código de formatação da resposta. A montagem das páginas dinâmicas é feita com base nos dados gerados pelo *handler* de processamento.

O *framework* define dois componentes básicos relacionados com *handlers* de apresentação: a interface `IHandlerApresentacao` e a classe abstrata `HandlerApresentacao`. O diagrama de classes destes elementos podem pode ser visto na Figura 3.4.

`IHandlerApresentacao` é a interface que especifica o comportamento básico dos *handlers* de apresentação, estendendo a interface `IHandler` e definindo apenas um novo método: `apresentar`. Este método deve ser implementado pelos *handlers* de apresentação definidos pelo desenvolvedores e deve conter operações relacionadas com a montagem e formatação da resposta ao cliente.

A classe `HandlerApresentacao` realiza a interface `IHandlerApresentacao` e estende da classe `Handler`. Assim como o `IHandlerProcessamento`, ela implementa o método `execucaoEspecifica`, mas chamando o método `apresentar` ao invés do `processar` e lançando, quando necessário, a exceção `ApresentacaoException`:

```
public void execucaoEspecifica(HttpServletRequest request,
    HttpServletResponse response) throws ApresentacaoException{
  try{
    apresentar(request,response);
  }
  catch(Exception e){
    throw new ApresentacaoException(e);
  }
}
```

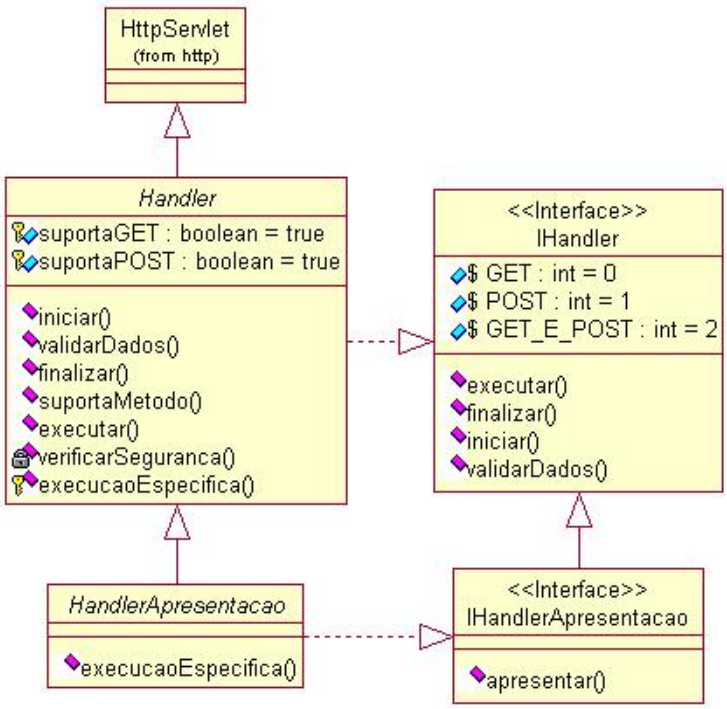

Figura 3.4: Diagrama das classes básicas do *framework* relacionadas com os *handlers* de apresentação.

Serviços

Geralmente, um *handler* isolado não pode responder às requisições dos clientes. Para que estes comecem a fazer sentido e sejam realmente úteis, é necessário que eles funcionem em pares, mais especificamente um de apresentação com um de processamento cooperando entre si. Neste *framework* toda requisição *Web* implica no processamento de um par de *handlers*, cada um de um tipo. Como já visto na Seção 2.2 estes *handlers* podem ser compostos de diferentes formas e cada um pode participar de quantos pares for necessário.

Para que o *framework* funcione como descrito no parágrafo anterior é necessário que o desenvolvedor especifique quais os pares de *handlers* possíveis para o seu sistema. O problema é que nem sempre é possível saber quais são estes pares antes da execução da requisição. Muitas vezes só é possível decidir que *handler* de apresentação deve ser executado, após a execução do *handler* de processamento. Isto acontece porque o sistema precisa apresentar diferentes respostas dependendo do resultado da execução das regras de negócio. Um exemplo bem simples desta situação é a execução de um login. Para implementar esta funcionalidade em um sistema *Web* é necessário criar um *handler* de processamento, que executa a operação de login no sistema. Mas a execução de um login pode implicar em duas situações: o login foi executado com sucesso ou login falhou. Para cada situação desta é necessário montar uma página de resposta diferente, o que implica em implementar *handlers* de apresentação diferentes para cada uma destas. Embora seja possível saber quais as opções de apresentação para uma execução, só podemos decidir qual delas executar após o *handler* de processamento processar.

Percebe-se que a especificação de pares de *handlers* não é suficiente para descrever o comportamento de um sistema baseado neste *framework*, por isso se faz necessário a criação de um novo conceito: serviço. Cada requisição de clientes *Web* implica na execução de um serviço. Este elemento nada mais é do que a composição de um *handler* de processamento com um mapeamento de eventos em *handlers* de apresentação. Executar um serviço significa, processar o *handler* de processamento associado, o que deve gerar um evento, e dependendo do evento gerado executar o *handler* de apresentação associado ao mesmo. A Figura 3.5 demonstra como seria o serviço de login exemplificado no parágrafo anterior.

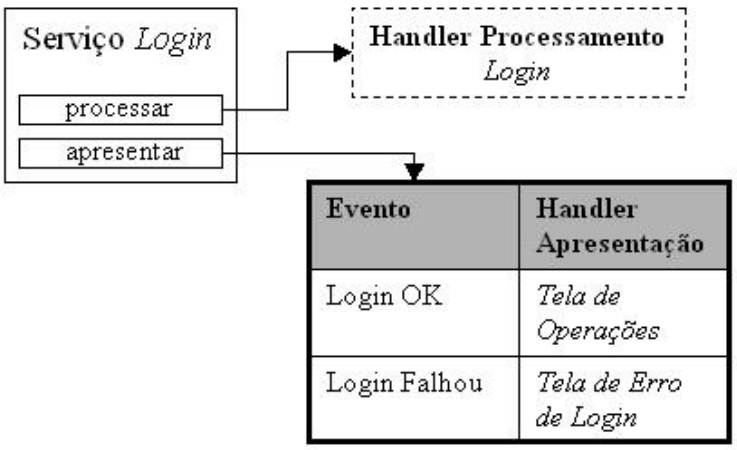

Figura 3.5: Especificação do serviço de login.

A classe `Servico`, mostrada na Figura 3.6, representa o componente serviço; ela possui um atributo `nome` que é o identificador do serviço, `processamento` que é o nome do *handler* de processamento associado ao serviço e o atributo `hashEventoApresentacao`, do tipo `HashMap`, que representa o mapeamento de evento em nome de *handler* de apresentação. Além destes atributos e métodos `getXXX` e `setXXX` associados, a classe possui os métodos para recuperar o nome do *handler* de apresentação baseado em um dado evento.

Um sistema *Web* é composto por um conjunto de serviços, que é representado pela interface `ConfiguracaoHandlers`, que tem este nome porque além dos serviços ela possui informações sobre o tratamento de erros. Através desta é possível recuperar os serviços pelo nome e obter informações necessárias para o tratamento de exceções que não são tratadas pelos *handlers* do sistema. Os métodos `getTratadorErro` são usados para recuperar o nome do *handler* de apresentação responsável por mostrar a tela de erro relacionada à exceção gerada. Este método possui três assinaturas: a primeira recebe o nome da exceção (mais especificamente, o nome da classe da exceção), a segunda o objeto que representa a exceção e a última nada recebe. Esta última versão do método é usada para tratar os demais tipos de erros, que não são especificados pelo programador.

O *framework* já oferece a realização desta interface, `ConfiguracaoHandlersDefault`, que armazena os serviços e as informações dos tratadores de erro em *Hashmaps* de Java. O desenvolvedor pode optar por implementar esta interface de outras formas. Maiores detalhes sobre esta funcionalidade podem ser obtidos na Seção 3.3.6.

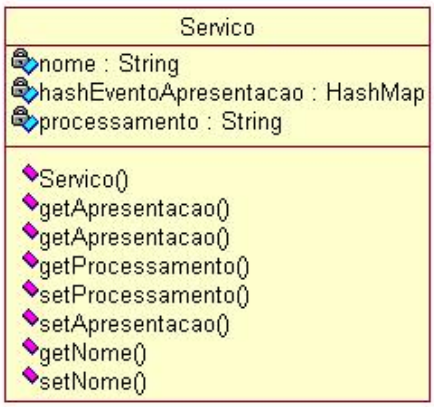

Figura 3.6: Diagrama UML da classe Servico.

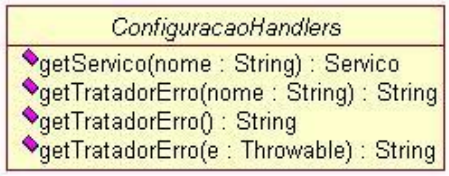

Figura 3.7: Diagrama UML da interface ConfiguracaoHandlers.

`MontadorServicos` é a entidade responsável por montar as configurações de *serviços* e de tratamento de erros. É através dela que o Controlador de *Handlers* passa as informações necessárias para montar os serviços e recuperá-los. Esta é uma classe abstrata que pode ser estendida pelo programador para que o mesmo customize a forma de carregar os serviços (esta funcionalidade pode ser vista com mais detalhes na Seção 3.3). O *framework* oferece um subclasse concreta, chamada `MontadorServicoXMLSimples`, que monta os serviços e informações de tratamento de erros à partir de um arquivo XML e as carrega na implementação `ConfiguracaoHandlersDefault`. A estrutura das classes abstrata e concreta do montador de serviço do *framework* é mostrada na Figura 3.8.

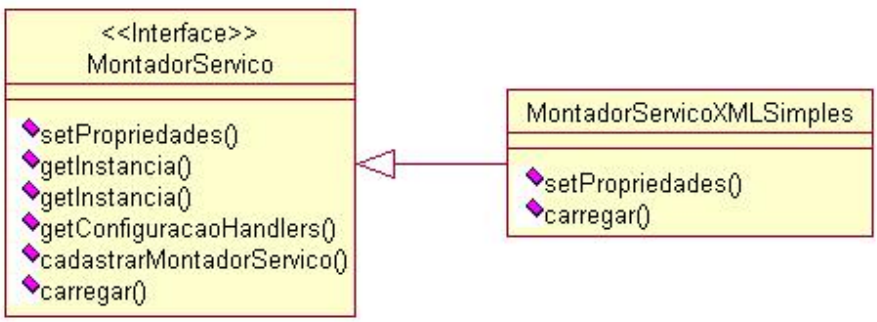

Figura 3.8: Diagrama de classes das entidades que implementam o montador de serviço.

A classe `MontadorServico` funciona como uma *Abstract Factory* [17]; toda instan-

ciação de objetos do tipo `MontadorServico` deve ser feita através de uma chamada ao seu método estático `getInstancia`, que recebe o nome da classe a ser instanciada como parâmetro. Esta classe também funciona como uma *cache* de montadores, ou seja, as classes carregadas pela primeira vez são armazenadas em uma estrutura do tipo `Hahmap` para depois serem consultadas. A implementação do método `getInstancia` pode ser vista no trecho de código a seguir.

```
 1: public abstract class MontadorServico {
 2:   ...
 3:   private static HashMap hashMontadoresServico = new HashMap();
 4:   public static MontadorServico getInstancia(String nome)
          throws ClassNotFoundException{
 5:     MontadorServico montador;
 6:     montador = (MontadorServico)hashMontadoresServico.get(nome);
 7:     if(montador == null){
 8:        montador = instanciarMontador(nome);
 9:        hashMontadoresServico.put(nome,montador);
10:     }
11:     return montador;
12:   }
13:  ...
14: }
```

A linha 6, verifica se já existe alguma instância do montador que se deseja carregar armazenado na *cache*. Caso não exista, é feita uma chamada ao método privado `instanciarMontador`, que através do mecanismo de reflexão de Java [41] localiza e instancia um objeto do tipo desejado (alguma subclasse de `MontadorServico`), armazenando-o na *cache* logo em seguida. Por fim, a instancia é retornada.

Como já comentado no parágrafo anterior, o método `instanciarMontador` usa um mecanismo muito poderoso para instanciar os montadores de serviços: reflexão. Com este mecanismo é possível instanciar objetos e executar métodos baseado em seus nomes (*strings*) em tempo de execução. O trecho completo deste método pode ser visto logo abaixo.

```
 1: private MontadorServico instanciarMontador(String nome){
 2:   Class classeMontador;
 3:   Constructor construtorDefault;

 4:   classeMontador = Class.forName(nome);
 5:   construtorDefault = classeMontador.getConstructor(null);
 6:   return (MontadorServico)construtorDefault.newInstance(null);
 7: }
```

A linha 4 carrega a classe do sistema de arquivos usando o *ClassLoader default* de Java. O resultado desta operação retorna um objeto do tipo `Class`, que é uma representação da classe em Java. A partir deste objeto é possível invocar métodos, construtores e consultar valores de atributos públicos, por exemplo. Na linha 5 consulta-se o construtor

default(sem parâmetros) desta classe. Por fim, o construtor é invocado sem se passar nenhum parâmetro, o que implica na instanciação e inicialização *default* de um objeto.

O método `setPropriedades` da classe `MontadorServico` é usado para que a classe que instancie o montador de serviços possa passar parâmetros que serão usados na execução do método `carregar`. Este último deve criar e carregar uma instância da classe `ConfiguracaoHandlers`. A implementação oferecida pelo *framework* cria uma instância da classe `ConfiguracaoHandlersDefault`, baseada no arquivo XML. Por fim, deve-se chamar o método `getConfiguracaoHandlers` para recuperar a configuração carregada. O diagrama de seqüência da Figura 3.9 dá uma idéia da ordem de execução destas operações.

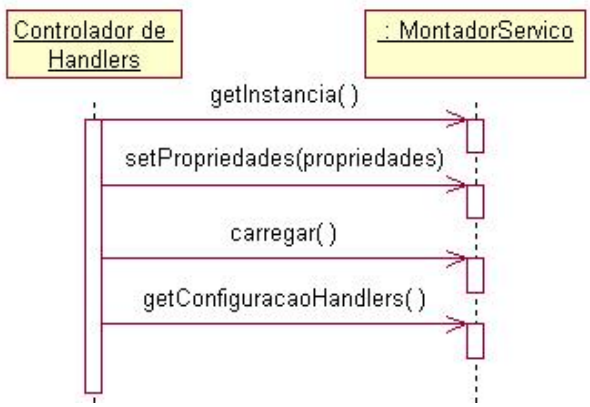

Figura 3.9: Diagrama de seqüência do processo de recuperação das configurações de serviços.

O `Controlador de Handlers`, mostrado na Figura 3.9, é responsável por instanciar o montador de serviços e gerenciar a execução dos *handlers*. Mais detalhes sobre o mesmo serão dados na próxima seção.

Controlador de Handlers

Um componente essencial para o funcionamento do *framework* é o Controlador de *Handlers*. Ele é responsável por receber todas as requisições *Web* e, só depois de interpretá-las, repassá-las para os *handlers* responsáveis pela sua execução. Este é o componente que implementa o padrão *Web Interceptor* (Veja a Seção 2.1) e controla toda a lógica de execução dos componentes do *framework*. Este elemento é implementado como um *servlet*, já que ele precisa receber as requisições *Web*.

Durante sua inicialização ele recupera uma instância do `MontadorServico`, mais especificamente de alguma das suas subclasses. Por *default* o controlador usa a implementação `MontadorServicoXMLSimples`, mas o desenvolvedor tem a flexibilidade de definir que implementação usar. Esta customização pode ser feita através da passagem de um parâmetro de configuração de *servlet* (um exemplo deste uso pode ser visto na Seção 3.3.6). Com o montador, o controlador recupera as informações sobre os serviços do sistema encapsulados em um objeto do tipo `ConfiguracaoHandlers`. Este último será usado durante todo o ciclo de vida do *servlet* para realizar decisões sobre que *handler* de processamento e apresentação executar para cada requisição *Web*. O diagrama

de seqüências da Figura 3.10 mostra o comportamento do controlador para executar uma requisição *Web*.

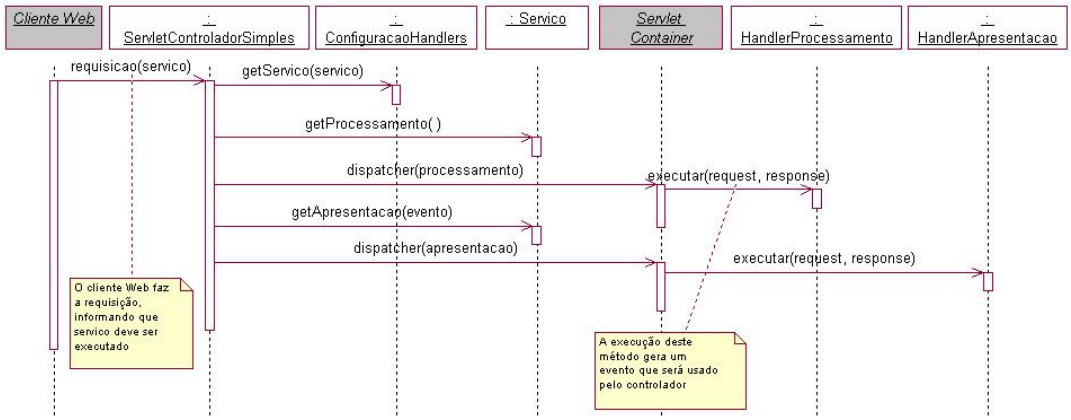

Figura 3.10: Diagrama de seqüência da execução de cada requisição *Web*.

O componente `Servlet Container` representa o *Servlet Container* de Java, que é o responsável pelo gerenciamento do ciclo de vida e execução dos *servlets*. Uma descrição mais detalhada da seqüência de operações da Figura 3.10 pode ser visto abaixo:

1. O `Cliente Web` deve fazer a requisição informando o `servico` que deseja executar;
2. O `ServletControladorSimples` consulta o `Servico` através de uma chamada ao método `getServico` da classe `ConfiguracaoHandlers`;
3. Com o `Servico`, o `ServletControladorSimples` recupera o nome do *handler* de processamento através da chamada ao método `getProcessamento`;
4. Este nome é usado para executar o `HandlerProcessamento` correto. Como os *handlers* são *servlets*, isto deve ser feito através de uma chamada via `dispatcher` para o `Servlet Container`, que por sua vez executa o `HandlerProcessamento` através de uma chamada ao seu método `executar`;
5. A execução do `HandlerProcessamento` gera um evento que deve ser usado pelo `ServletControladorSimples` para decidir que *handler* de apresentação deve montar a resposta para o cliente. Este evento é passado como parâmetro na chamada ao método `getApresentacao`, que retorna o nome do *handler* de apresentação do `Servico` corrente e que é responsável por tratar o evento passado;
6. Com o nome do *handler* de apresentação o `ServletControladorSimples` executa o `HandlerApresentacao` indiretamente através do `Servlet Container`.

Como já visto na Figura 3.10, o `ServletControladorSimples` não executa os *handlers* diretamente, ao invés disto ele requisita ao `Servlet Container` que os execute. Isto é feito para garantir a consistência do ambiente de *servlets*, já que estes são gerenciados pelo *Servlet Container*. O trecho de código abaixo exemplifica como é implementada a requisição ao *Container*, mostrada anteriormente como uma chamada ao método `dispatcher`.

```
dispatcher = request.getRequestDispatcher(handler);
if(dispatcher != null){
  dispatcher.include(request,response);
}
```

A chamada ao método `getRequestDispatcher` na variável `request`, que é do tipo `HttpServletRequest`, cria uma referência ao *handler* indiretamente pelo *Servlet Container* [12]. Com este objeto do tipo `RequestDispatcher`, representado pela variável `dispatcher`, é possível disparar o *handler* através de uma chamada ao método `include`.

O `ServletControladorSimples` está preparado para ser estendido e ele possui algumas operações que são definidas como vazias apenas para que suas subclasses possam definir algum comportamento para estas (detalhes sobre estas operações podem ser vistos na Seção 3.3.5). No entanto a maioria de seus métodos é definido como `final` para que as subclasses continuem preservando o comportamento básico do *framework*.

Tratamento de Exceções

O *framework* assume parte da responsabilidade pelo tratamento dos erros do sistema, deixando o programador responsável apenas por definir que *handlers* de apresentação devem ser executados quando determinadas exceções ocorrerem. Além disto ele é robusto o suficiente para que qualquer erro que ocorra durante a execução dos serviços seja mostrado de forma amigável ao usuário.

Todos os erros (*Exceptions*) gerados pela execução do sistema não precisam ser tratadas no escopo das chamadas aos métodos que as geraram. O programador pode se abstrair de determinados erros (não esperados) e tratar apenas os requisitados. A implementação do *framework* tem o compromisso de pegar todas exceções e mostrar as telas de erros associadas aos mesmos. Quando não existir nenhuma tela de erro associada, ele mostra uma tela padrão da implementação.

O método `mostrarTelaErroDefault` do Controlador é executado sempre que ocorre algum erro não tratado pelos *handlers* e requer como parâmetro todas as informações necessárias para o tratamento do erro: um parâmetro do tipo `Throwable`, que representa um erro em Java, e mais dois que são o `HttpServletRequest` e o `HttpServletResponse`.

```
private final void mostrarTelaErroDefault(Throwable erro,
  HttpServletRequest request,HttpServletResponse response){
  ...
}
```

A implementação deste método pode ser vista no trecho de código a seguir. Na linha 2, tenta-se recuperar o nome do tratador de erro (*handler* de apresentação) responsável por mostrar a tela associada ao erro através da chamada ao método `getTratadorErro` na `configuracaoHandlers` (variável que armazena toda a configuração de *handlers* e tratamento de exceções). Caso não exista tratador associado, tenta-se pegar o tratador *default* (chamada ao método `getTratadorErroDefault`). Na linha 7, o erro é adicionado como atributo do `request` para que o tratador possa recuperar esta informação. O nome do atributo adicionado é `$$erro$$`, apenas por decisão de projeto, mas qualquer outro nome seria permitido. As linhas 12 e 13 são exemplos da execução de situações onde não foi encontrado nenhum tratador para este tipo de erro.

```
 1:  ...
 2:  tratadorErro = configuracaoHandlers.getTratadorErro(erro);
 3:  if(tratadorErro == null){
 4:    tratadorErro = configuracaoHandlers.getTratadorErroDefault();
 5:  }
 6:  if(tratadorErro != null){
 7:    request.setAttribute("$$erro$$",erro);
 8:    dispatcher = request.getRequestDispatcher(tratadorErro);
 9:    if(dispatcher != null){
10:      dispatcher.include(request,response);
11:    }
12:    else{... // mostra a tela de erro padrão}
13:  }
14:  else{... // mostra a tela de erro padrão}
15:  ...
```

Uma última entidade relacionada com o tratamento de erros é o `HandlerErro`. Esta classe é uma extensão do `HandlerApresentacao` para apresentar especificamente tela de erros.

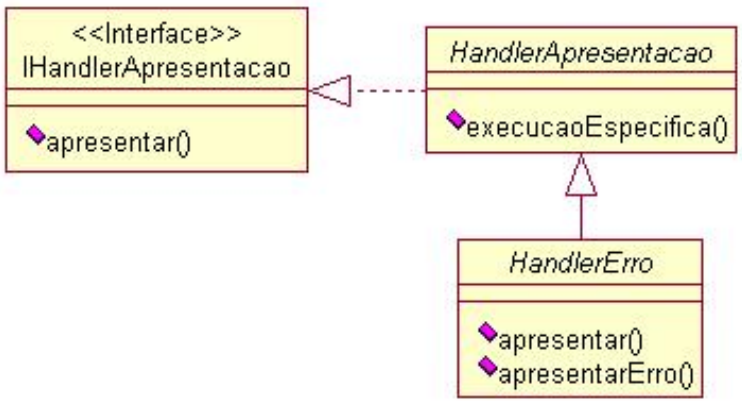

Figura 3.11: Diagrama de classes do *Handler* de erro.

Como pode ser visto na Figura 3.11 ele possui apenas um método a mais, que é `apresentarErro`, que além do `request` e `response` recebe como parâmetro o erro a ser tratado. Este método deve ser implementado pelos *handlers* de erro definidos pelo programador para que o mesmo se comporte como esperado. O método `apresentar` herdado de sua subclasse foi implementado e marcado como `final`, para evitar que o desenvolvedor tente implementá-lo e mude o comportamento do *framework*. A implementação desta nova classe pode ser vista no trecho de código a seguir.

```
...
public abstract class HandlerErro extends HandlerApresentacao {
  public final void apresentar(HttpServletRequest request,
    HttpServletResponse response) throws Exception{
```

```
    Exception excecao;
    excecao = (Throwable)request.getAttribute("$$erro$$");
    apresentarErro(request,response,excecao);
  }
  public abstract void apresentarErro(HttpServletRequest request,
    HttpServletResponse response,Exception excecao) throws Exception;
}
```

Como pode ser visto, o método `apresentar` foi implementado nesta subclasse. Ele basicamente recupera o erro que foi armazenado no `request` anteriormente e faz uma chamada ao método `apresentarErro`, que é abstrato e deve ser implementado em suas subclasses, ou seja, nos *handlers* de erro definidos pelo desenvolvedor.

3.3 Guia de utilização do *framework*

Nesta seção é apresentado um guia de implementação detalhado para utilização do *framework*. Seu principal objetivo é auxiliar arquitetos, projetistas e programadores a entenderem e usarem todos os aspectos existentes no *framework*.

3.3.1 Definindo os *handlers* de processamento

Os *handlers* de processamento estão diretamente relacionados com as operações executadas pelos usuários. Cada operação do sistema deve ser tratada por um *handler* diferente. Colocar no *handler* de processamento qualquer código que tenha a ver com dados de resposta ao cliente é errado e deve ser evitado. Em um sistema bancário, por exemplo, que possui operações de crédito, débito, login e saldo, é necessário criar quatro *handlers* de processamento, um para executar cada operação desta.

Colocar código relativo às regras de negócio do sistema diretamente na implementação dos *handlers* de processamento não é uma boa prática. Ao invés disto deve-se criar componentes que encapsulem estas regras e fazer com que o *handler* apenas invoque as operações destes. Os *handlers* são apenas entidades responsáveis por disponibilizar os serviços destes componentes na *Web*. Maiores detalhes sobre a implementação destes componentes podem ser obtidos em outros trabalhos [30, 65].

Apesar destes tipos de *handlers* não conterem regras de negócio diretamente em seus corpos, eles possuem diversas operações que devem ser executadas para adaptar a funcionalidade já definida nos componentes de negócio para a realidade da *Web*:

- Recuperar os parâmetros da requisição;
- Executar validações em cima desses parâmetros para garantir que os dados esperados estão presentes e que são do tipo correto;
- Fazer as conversões de tipo de alguns parâmetros da requisição (*strings*) para tipos primitivos em Java;
- Criar objetos de negócio com as informações da requisição convertidas;

- Consultar objetos de negócio do sistema, através de chamadas a métodos dos componentes que implementam as operações de negócio, passando as informações enviadas na requisição;
- Executar operações dos componentes de negócio passando os objetos de negócios criados/atualizados como parâmetro.

Não é obrigatório que os *handlers* de processamento executem todas as operações descritas acima, mas é muito comum ele executar a maioria destas.

Deve-se estender a classe `HandlerProcessamento` e implementar alguns de seus métodos para definir um *handler* de processamento. Estes métodos são

- `validarDados` – Validações em cima dos parâmetros da requisição;
- `iniciar` – Todo o código de inicialização do *handler*;
- `finalizar` – Todo o código de destruição do *handler*;
- `processar` – Código de execução das operações de negócio relativos a requisição.

Destes métodos, o único cuja implementação é mandatória é o `processar`, os demais são opcionais.

Validação de dados

A validação dos dados é feita em todas as requisições e é executada sempre antes do método `executar`. Existem diversos tipos de validações sobre os dados da requisição *Web* e várias abordagens para implementá-las. O foco desta seção é apenas como usar o *framework* para fazer validação de dados, ou melhor, onde colocar o código referente à validação. Maiores detalhes sobre as abordagens e tipos de validação podem ser vistos na Seção 4.2.

Em ambos os tipos de *handlers* existe um método específico para se colocar código de validação que é o `validarDados`. Este deve ser usado apenas para garantir que os dados necessários para dar continuidade a execução da requisição estão presentes e são do tipo esperado. O trecho de código a seguir exemplifica uma validação que é feita no *handler* de processamento que executa a operação de crédito e espera dois parâmetros do cliente *Web*: o número da conta e o valor a ser creditado.

```
 1: public void validarDados(ServletRequest request,
     ServletResponse response) throws HandlerException{
 2:   String txtNumeroConta, txtValor;
 3:   int numeroConta, valor;
 4:   txtNumeroConta = request.getParameter("conta");
 5:   txtValor  = request.getParameter("valor");
 6:   if(txtNumeroConta != null && txtValor != null){
 7:     try{
 8:       valor = Integer.parseInt(txtValor);
 9:     }catch(NumberFormatException nfe){
10:       throw new ProcessamentoException(...);
```

```
11:     }
12:     if(valor < 0 || numeroConta.length() != 6)
13:       throw new ProcessamentoException(...);
14:     }
15:   }
16:   else{throw new ProcessamentoException(...);}
17: }
```

Nas linhas 4 e 5 são recuperados os parâmetros da requisição com o nome `conta` e `valor`. Na linha 6 é feita uma verificação para saber se os dois estão presentes. Todo parâmetro de requisição *Web* é tratado como *string* em Java, por isso é necessário que seja feita uma verificação sobre este valor para garantir que ele pode ser convertido para um inteiro (linha 8). Por fim, é verificado se o valor a ser creditado é positivo e se o número da conta possui seis caracteres. Como pode ser visto nas linhas 10, 13 e 16, qualquer falha nas verificações faz com que seja lançada uma exceção do tipo `ProcessamentoException`.

Inicialização

Como já falado na Seção 3.2.1, todo *handler* passa uma única vez por este estado durante todo o seu tempo de vida. É neste momento que o *handler* deve executar as operações de inicialização do serviço ou alocar recursos que serão compartilhados por todas as requisições neste *handler*. Exemplo destes tipos de operações são conexão com banco de dados, abertura de arquivos, leitura de parâmetros de configuração, instanciação de objetos que executam as regras de negócio, etc. A inicialização do *handler* é feita através de uma chamada ao seu método `iniciar`, é neste método que o programador deve colocar o código referente as operações descritas anteriormente. O trecho de código abaixo exemplifica o uso deste método.

```
 1: public class HP_Credito extends HandlerProcessamento{
 2:   private Fachada sistema;
 3:   private Log logSistema;
 4:   ...
 5:   public void iniciar()
 6:     throws HandlerException {
 7:     try{
 8:       sistema = Fachada.getInstancia();
 9:       logSistema = GerenciadorLog.getLogOperacoes();
10:     }
11:     catch (Exception e) { ... }
12:   }
13: }
```

A inicialização do `HP_Credito` é bastante simples: ele precisa instanciar a Fachada (objeto que implementa o padrão de projeto Facade [17], ou seja, funciona como interface única para as regras de negócio do sistema) e o *log* de operações do sistema, representado pela variável `logSistema` do tipo `Log`.

É recomendável usar o padrão de projeto *Super Component* (Seção 2.4) para evitar a replicação de código de inicialização nos diversos *handlers* do sistema.

Finalização

A finalização do *handler* é feita através de uma chamada ao seu método `finalizar` e é sua última chance de executar alguma operação antes de ser removido da memória, por isso é neste método que o programador deve colocar operações relativas a destruição do *handler*. Em geral as operações de destruição devem ter o efeito inverso das definidas no método `iniciar`. Por exemplo, o `HP_Credito` tem o método `finalizar` que basicamente implementa chamadas a operações que liberam os recursos alocados em sua inicialização:

```
1: public void finalizar() throws HandlerException {
2:   try{
3:     sistema = Fachada.liberarInstancia();
4:     GerenciadorLog.liberarLog(logSistema);
5:   }
6:   catch (Exception e) { ... }
7: }
```

O padrão *Super Component* (visto na Seção 2.4) também deve ser usado para evitar duplicação de código de destruição nos *handlers*.

Processamento da requisição

Esta é a parte mais importante do *handler* de processamento; é nesta que o desenvolvedor deve colocar todo o código de execução das operações de negócio relativos à requisição do cliente. O processamento da requisição é feito através de uma chamada ao método `processar` do *handler*.

O trecho a seguir é um exemplo da implementação da operação de login no sistema. São passados dois parâmetros na requisição, o login e a senha do usuário. Na linha 8, é feita uma validação do usuário para conferir se o login e senha estão corretos através da chamada ao método `validarUsuario` do objeto `sistema`, que encapsula todos os serviços de negócio da aplicação. Se esta verificação for bem sucedida o *handler* de processamento consulta as informações completas do usuário, encapsuladas em um objeto do tipo `Usuario`, através da chamada ao método `recuperarUsuario` do `sistema`. Este objeto será usado pelo *handler* de apresentação que montará a tela inicial do sistema e por isso é armazenado no `request`. As linhas 11 e 14 geram um evento que será usado pelo *framework* para decidir que *handler* de apresentação deverá ser executado para montar a página de resposta ao cliente. As informações sobre o mapeamento de eventos em *handlers* de apresentação são definidas no arquivo XML de especificação dos serviços.

```
1: public void processar(HttpServletRequest request,
2:                       HttpServletResponse response)
3:    throws ProcessamentoException,SistemaException{
4:   String login,senha;
5:   Usuario usuario;
6:   login = request.getParameter("login");
7:   senha = request.getParameter("senha");
8:   if(sistema.validarUsuario(login,senha)){
9:     usuario = sistema.recuperarUsuario(login);
```

```
10:     request.setAttribute("usuario",usuario);
11:     gerarEvento("LoginOK")
12:   }
13:   else{
14:     gerarEvento("LoginFalhou")
15:   }
16: }
```

Percebe-se que o processamento só contém operações relativas à invocação das operações de negócio, e que todo código referente à montagem da apresentação deve ficar nos *handlers* de apresentação. Apesar destes tipos de *handlers* executarem regras de negócio, deve-se evitar ao máximo colocá-las diretamente no corpo do método `processar` (no exemplo anterior existem apenas chamadas ao objeto que encapsula as regras da aplicação).

3.3.2 Definindo os *handlers* de apresentação

Os *handlers* de apresentação estão diretamente relacionados com as telas do sistema. Cada modelo de tela do sistema deve ser tratado por um *handler* diferente. Colocar código relativo ao processamento das regras de negócio do sistema diretamente na implementação dos *handlers* de apresentação é errado e deve ser evitado.

Colocar o conteúdo da resposta diretamente dentro do código Java não é uma boa prática de implementação, além de trazer diversos problemas para o desenvolvimento e manutenção destes componentes. Maiores detalhes sobre este assunto pode ser obtido na Seção 4.1.

Os *handlers* de apresentação executam um conjunto bem específico de operações na maioria das situações de suas execuções, mas não é obrigatório que eles executem todas. Algumas destas são

- Recuperar os parâmetros da requisição, geralmente estes parâmetros são objetos de negócio da aplicação;
- Executar validações em cima desses parâmetros para garantir que os dados esperados estão presentes e que são do tipo correto;
- Recuperar as informações contidas nos objetos passados como parâmetro;
- Fazer ou delegar a composição dos dados recuperados dos parâmetros (dinâmicos) com a parte estática da página HTML a ser apresentada;
- Enviar o resultado da composição para o cliente *Web*.

Deve-se estender a classe `HandlerApresentacao` e redefinir alguns de seus métodos para definir um *handler* de apresentação. Estes métodos são

- `validarDados` – Validações em cima dos parâmetros da requisição;
- `iniciar` – Todo o código de inicialização do *handler*;

- `finalizar` – Todo o código de destruição do *handler*;
- `apresentar` – Código de execução da montagem da página de resposta para o cliente.

Destes métodos o único que é obrigatório implementar é o `apresentar`, os demais são opcionais. O padrão de projeto *Super Component* também deve ser usado nestas entidades para evitar a duplicação de código de inicialização e finalização destes.

Validação de dados

É importante colocar validações nos *handlers* de apresentação também, pois estes precisam validar os dados gerados pelos *handlers* de processamento, já que eles são entidades independentes e podem ser compostos de diferentes formas. Nestes casos a validação é um pouco diferente, pois geralmente os dados passados pelo seu par são objetos em Java e não tipos básicos. Um exemplo disto pode ser visto no trecho de código abaixo:

```
 1: public void validarDados(ServletRequest request,
      ServletResponse response) throws ApresentacaoException{
 2:   Object objeto;
 3:   objeto = request.getAttribute("banco");
 4:   if ((objeto == null)||!(objeto instanceof Banco)){
 5:     throw new ApresentacaoException(...);
 6:   }
 7:   objeto = request.getAttribute("mensagem");
 8:   if ((objeto == null ||!(objeto instanceof String)){
 9:     throw new ApresentacaoException(...);
10:   }
11: }
```

De fato, percebe-se que as validações são em cima de objetos em Java, ao invés de tipos primitivos. O *handler* de apresentação acima precisa receber dois parâmetros de seu par, um chamado `banco` e outro `mensagem`, dos tipos `Banco` e `String`, respectivamente.

Inicialização

Na maioria dos casos a inicialização dos *handlers* de apresentação é mais simples do que a dos *handlers* de processamento. Isto ocorre basicamente porque os primeiros não precisam alocar recursos caros, como conexão com banco de dados, objetos que encapsulam as regras de negócio, conexão com a rede, etc. Em sua inicialização os *handlers* de apresentação fazem operações como leitura de arquivos que possui o *template* da página, instanciação do mecanismo que faz a composição de informação dinâmica com a estática, etc. O trecho de código abaixo exemplifica a inicialização destes.

```
1:   ...
2: WebCompiler compiler;
3: String templatePagina;
4: ...
```

```
 5: public void iniciar()
 6:   throws HandlerException {
 7:   try{
 8:     templatePagina = Biblioteca.lerArquivo("template.html");
 9:     compiler = WebCompiler.getInstancia();
10:   }
11:   catch (Exception e) { ... }
12: }
13: ...
```

O `WebCompiler` implementa o padrão de projeto *Web Compiler* apresentado na Seção 2.3 e é usado durante todo ciclo de vida do *handler* para fazer a composição de informações dinâmicas e estáticas. A variável `templatePagina` guarda a parte estática da página de resposta ao cliente, ou seja a parte que não muda para as diversas requisições.

Finalização

A finalização do *handler* de apresentação também é a última fase do seu ciclo de vida e é neste momento que ele deve liberar os recursos alocados na inicialização. O trecho de código a seguir é um exemplo de finalização de um *handler* que possui a inicialização do exemplo anterior:

```
1: ...
2: public void finalizar()
3:   throws HandlerException {
4:   try{
5:     compiler = null;
6:   }
7:   catch (Exception e) {...}
8: }
9: ...
```

Montagem da resposta ao cliente

É neste momento que o *handler* de apresentação deve formatar a resposta aos clientes *Web* com base nas informações passadas para ele pelo seu *handler* de processamento par. Esta resposta pode ser de qualquer formato, como XML, WML (Wireless Markup Language), HTML, etc. O código de formatação da resposta deve ser colocado no método `apresentar` do *handler*.

```
1: public void apresentar (HttpServletRequest request,
2:                         HttpServletResponse response)
3:   throws ApresentacaoException, IOException {
4:   PrintWriter out = response.getWriter();
5:   String pagina;
6:   String nomeUsuario, ultimoAcesso;
7:   Usuario u = (Usuario) request.getAttribute("usuario");
8:   try {
```

```
 9:      nomeUsuario = usuario.getNome();
10:      ultimoAcesso = Formatador.formateData(u.getUltimoAcesso())
11:      pagina = compiler.executar(templatePagina,
           new String[]{"nomeUsuario","ultimoAcesso"},
           new String[]{nomeUsuario,ultimoAcesso});
12:      out.println(pagina);
13:    }catch (Exception e) {
14:      throw new ApresentacaoException ("Erro ...",e);
15:    }finally{ out.close(); }
16: }
```

Este método `apresentar` pertence ao *handler* que monta a tela inicial do sistema, após o sucesso da operação de *login*. Na linha 7 recupera-se as informações do usuário, encapsuladas no objeto do tipo `Usuario`, que foi armazenado no `request` pelo *handler* de processamento. Nas linhas 9 e 10, pega-se as informações dinâmicas necessárias para a montagem da página: o nome do usuário e a data do último acesso (convertida para *string* através da execução de `Formatador.formateData`). Com estas informações, executa-se o objeto que implementa o padrão de projeto *Web Compiler* (linha 11), através da chamada ao seu método `executar`. Por fim, a resposta ao cliente é enviada através da execução da linha 12.

Existem outras formas de compor as informações dinâmicas com as estáticas, maiores detalhes sobre estas abordagens podem ser obtidos na Seção 4.1.

3.3.3 Definindo os serviços do sistema

Um serviço é o elemento que representa a composição dos *handlers* para responder um determinado tipo de requisição. Já foi visto na Seção 3.2.2 que um serviço nada mais é do que a composição de um *handler* de processamento com um mapeamento de eventos em *handlers* de apresentação. Assim, executar um serviço significa processar o *handler* de processamento associado e executar o *handler* de apresentação associado ao evento gerado.

Os serviços de um sistema devem ser especificados em um arquivo XML para que possam ser executados a partir da implementação padrão do *framework*. Este arquivo deve estar localizado em qualquer diretório alcançado pelo CLASSPATH[2] e deve ter o nome "configuracao_servicos.xml".

A estrutura deste arquivo é bastante simples, seu elemento raiz é o `servidor`. Este elemento possui como subelementos *tags* de tratamento de erro e a definição dos serviços do sistema como pode ser visto no trecho abaixo.

```
<servidor>
  ... <!-- Tags de tratamento de erro -->
  <servicos>
     ... <!-- Tags de serviço -->
  </servicos>
</servidor>
```

[2]Conjunto de caminhos onde o *Classloader* de Java procura as classes e recursos a serem carregados.

O elemento `servicos` é composto por várias definições de serviço, onde cada uma destas possui informações necessárias para execução do serviço requisitado pelo cliente *Web*. O serviço descrito no trecho abaixo tem o nome `login`, é por este nome que o cliente deve invocá-lo. Para o serviço de login é definido que o *handler* de processamento será o `HP_Login`. Se o processamento gerar um evento `loginOK` será apresentado o `HA_TelaInicial`, mas se o evento for `loginFalhou` o `HA_LoginErrado` será executado. O atributo `default` do elemento `apresentacao` pode ser usado para definir o *handler* de apresentação responsável por tratar a saída caso não seja gerado nenhum evento, ou se o evento gerado não estiver presente na lista de opções.

```
<servico nome="login">
  <processamento nome="HP_Login"/>
  <apresentacao default="">
    <opcao evento="loginOK" handler="HA_TelaInicial"/>
    <opcao evento="loginFalhou" handler="HA_LoginErrado"/>
  </apresentacao>
</servico>
```

Neste caso, a falha do login não representa um erro de sistema pois é uma situação bastante esperada. Por este motivo não deve ser tratado pelo mecanismo de tratamento de erros do *framework* e sim como uma simples situação de evento.

O cliente *Web* deve acessar o serviço através da execução do *servlet* controlador do sistema e passar um parâmetro chamado `servico`, que contém o nome do serviço a ser executado. A chamada ao serviço de login via GET é feita através da submissão da seguinte URL:

```
http://www.bancoExemplo.com.br/servlet/Controlador?servico=login
```

Neste exemplo invoca-se o componente `Controlador`, que pode ser o nome do *servlet* que faz o papel de Controlador de *handlers* ou apenas um apelido[3] do que faz.

Uma característica interessante deste *framework* é que se pode usar JSP ao invés de *Handlers* de apresentação. Isto significa que em qualquer lugar do arquivo de configurações que tem referências para *handlers* de apresentação pode-se colocar JSPs que o *framework* continua funcionando da mesma maneira.

3.3.4 Tratando erros

Os erros (*Exceptions*) gerados que não tem a ver com as regras de negócio (por exemplo, erro de conexão com o BD, erros de Entrada/Saída, etc.) não precisam ser tratados no escopo do *handler* (mediante o uso do comando try-catch); o programador pode se abstrair destes e tratar apenas os que interessarem ao negócio.

No arquivo de configurações de serviços, o programador pode especificar que *handlers* de apresentação devem tratar determinados tipos de exceções em Java, desta forma toda vez que a execução do sistema lançar uma exceção e a mesma não for tratada por nenhum *handler* do sistema, o *framework* automaticamente pega esta exceção e executa o tratamento apropriado. Este tratamento é procurar o *handler* de apresentação que

[3]Estes apelidos podem ser definidos nos arquivos de configuração do *Servlet Container*.

deve tratá-la, colocando-o para executar logo em seguida. O trecho de código a seguir mostra um exemplo do trecho de configuração do tratamento erros.

```
<servidor>
  <tratamento_erro default="HandlerErroDefault">
    <throwable nome="SQLException" handler="/jsp/TelaErroBD.jsp"/>
    <throwable nome="IOException" handler="HandlerErroIO"/>
  </tratamento_erro>
   ... <!-- Configuração dos serviços -->
</servidor>
```

A especificação de tratamento de erro acima define que os erros do tipo `SQLException` que não forem tratados pelos *handlers* ou outras classes do sistema devem ser tratados e apresentados para usuário pelo JSP `TelaErroBD.jsp`. As exceções do tipo `java.io.IOException` são tratadas pelo *handler* de apresentação `HandlerErroIO`. Qualquer outro tipo de exceção deve ser tratada pelo `HandlerErroDefault`.

3.3.5 Estendendo o Controlador de *handlers*

O *framework* disponibiliza uma implementação do Controlador de *handlers* que já atende aos requisitos básicos de muitos sistemas. No entanto, ele está preparado para que o desenvolvedor possa estendê-lo e adicionar comportamento ao mesmo. Existem diversos métodos que podem ser redefinidos pelo desenvolvedor:

- `processarGET` – É executado antes dos *handlers* e apenas em requisições do tipo GET;
- `processarPOST` – Também é executado antes dos *handlers*, mas apenas em requisições POST;
- `preprocessar` – É executado antes do `processarGET` e `processarPOST` e em qualquer situação;
- `verificaAcesso` – É executado antes do `preprocessar` e tem como objetivo fazer verificações de segurança sobre as requisições dos clientes *Web*;
- `iniciar` – É executado durante a inicialização do Controlador e onde o desenvolvedor deve alocar recursos necessários para a execução dos outros métodos;
- `finalizar` – É executado antes da destruição do Controlador. É onde o programador deve liberar os recursos alocados na inicialização e outras operações referentes à finalização do serviço;

Os três primeiros métodos podem ser redefinidos em situações onde o desenvolvedor deseja usar o padrão de projeto *Web Interceptor*, ou seja, em situações onde há duplicação de código no início da execução das requisições nos diversos *handlers* do sistema.

3.3.6 Customizando a definição de serviços

O comportamento padrão do *framework* define que o desenvolvedor deve especificar as informações do serviço em XML, e as classes do *framework* interpretam estas montando uma estrutura em memória que representa as configurações de serviços do sistema. Apesar desta funcionalidade atender aos requisitos de muitos sistemas, este *framework* foi definido de forma que possa ser facilmente estendido para que o desenvolvedor possa customizar sua implementação da melhor maneira que lhe convir.

Um exemplo simples de customização que pode ser feita é, ao invés de usar um arquivo XML para armazenar as informações do serviço, usar um banco de dados ou até mesmo outro formato de arquivo. Mais ainda, é possível que seja um requisito da aplicação que as configurações do serviço estejam constantemente mudando e que o sistema deve estar sempre se ajustando a essas mudanças. Para implementar um requisito como este é necessário entender duas entidades do *framework*: `MontadorServico` e `ConfiguracaoServicos` (já definidas na Seção 3.2).

O `MontadorServicos` é a classe responsável por configurar e montar as definições de serviços e tratamento de erros. Além disto, ela funciona como uma *Factory* [17] de Montadores de Serviço. O primeiro passo para a customização é estender a classe `MontadorServicos` e redefinir o seu método `carregar`, que é o responsável por instanciar o `ConfiguracaoHandlers`:

```
public class MontadorServicosBD extends MontadorServico{
  ...
  public abstract void carregar() throws MontadorServicoException{
    Connection conexao;
    ... // Faz a conexão JDBC com o BD onde
    ...  //estão as informações do serviços
    servicos = new ConfiguracaoServicosBDSemCash(conexao);
  }
```

Como pode ser visto no trecho acima a implementação do método é bastante simples. Ele primeiro abre a conexão JDBC com o banco de dados onde estão armazenadas as informações dos serviços e depois instancia um objeto do tipo `ConfiguracaoHandlersBD`, armazenando-o no atributo `servicos` que foi herdado. Esta classe representa a implementação do `ConfiguracaoHandlers` que recupera os dados dos serviços de um banco de dados, sem armazená-los em memória.

A classe `ConfiguracaoHandlersBD` implementa a interface `ConfiguracaoHandlers`, que define os métodos que devem ser implementados. O seu construtor recebe como parâmetro uma `Connection`, que é armazenada em um atributo da classe e é usado para acessar o banco de dados e obter as informações dos serviços. Um dos métodos que deve ser implementado é o `getServico` que recupera um objeto do tipo `Servico` a partir de um nome. Este método faz o acesso ao banco de dados e instancia um objeto do tipo `Servico`, baseado nos dados consultados, que vai ser retornado.

```
public class ConfiguracaoHandlersBD implements ConfiguracaoHandlers{
  private Connection conexao;
  ...
  public ConfiguracaoHandlersBD(Connection conexao){
```

```
        this.conexao = conexao;
    }
    public Servico getServico(String nome){...}
    ...
}
```

Por fim, é necessário adicionar um parâmetro de configuração `MontadorServico` ao controlador; ele informa que implementação de montador utilizar. Todos os *Servlets Container* suportam está funcionalidade, no entanto a forma de implementá-la é diferente em cada um destes.

3.4 Usos conhecidos

A implementação atual do *framework* é usada em duas empresas de desenvolvimento de sistemas: CESAR e Mobile. Quatro sistemas baseados em *servlets* estão usando o *framework* com sucesso:

- **Web2Billing** [57] – É uma solução completa de EBPP (*Eletronic Bill Presentation and Payment*), desenvolvida pela Mobile, e que permite a criação, geração, gerenciamento, apresentação, consulta e pagamento de faturas *online*;
- **FiS (Financial Services)** – O projeto contempla a migração dos Módulos de Contabilidade, Crédito, Lojistas e Serviços da HiperCard, para um novo ambiente tecnológico (J2EE). Estes módulos são integrados ao Sistema de Integrado de Crédito da HiperCard (SIC), ao R3/SAP e a outros sistemas legados. Também faz parte do projeto o desenvolvimento de um módulo de Controle de Acesso único e centralizado que poderá ser utilizado por qualquer aplicação da HiperCard disponível neste novo ambiente;
- **Fep (Call Center no FEP)** – Desenvolvimento de uma aplicação para a Central de Atendimento HiperCard que autorizará compras no autorizador FEP (Front End Processor) e no sistema legado SIC (Sistema de Integrado de Crédito da HiperCard) via browser;
- **Gin (Sistema de Gestão Interna)** – Sistema de apoio a gestão interna do CESAR com cadastros e relatórios gerais, além de englobar os sistemas financeiro e avaliação de colaboradores.

Capítulo 4

Diretrizes para desenvolvimento da camada de apresentação de sistemas *Web*

Neste capítulo apresentamos um conjunto de diretrizes para o desenvolvimento de sistemas Web baseados em servlets.

Durante o desenvolvimento e projeto de sistemas *Web*, várias são as decisões que os projetistas, arquitetos e engenheiros precisam tomar para implementar características inerentes a estes tipos de sistemas. Na maioria das vezes existem diferentes abordagens ou soluções para implementar cada uma destas características. A decisão de escolha de uma determinada solução em um projeto de *software* para *Web* em Java é uma tarefa complicada, tanto pelo fato de existirem várias soluções [26], quanto por ser uma tecnologia nova e em constante evolução, que poucas pessoas conhecem profundamente.

Por isso, este capítulo tem como objetivo fornecer diretrizes para as pessoas envolvidas no desenvolvimento de sistemas *Web* em Java, possibilitando que elas tomem decisões que melhor se adequem aos requisitos do seu tipo de aplicação. Para isso são apresentadas as diferentes abordagens para implementar cada uma das características comuns aos sistemas desta natureza. Em paralelo, são feitas comparações entre as abordagens na tentativa de avaliar quais delas devem ser aplicadas em determinadas situações. Estas comparações são resumidas em tabelas, ao final de cada uma das seções equivalentes às características (Seções 4.1, 4.2, 4.3 e 4.4). Para cada um dos critérios de avaliação são atribuídas notas na forma de símbolos de estrela ($\star$), que indicam quão positivo é a abordagem.

As seguintes características de sistemas *Web* foram analisadas neste trabalho.

- Formatação da apresentação, está relacionada a forma de montar a página de resposta ao cliente *Web*;
- Validação de dados, como validar os dados (realizar críticas) que são informados pelo usuário;
- Persistência do estado do cliente, como armazenar informações relativas ao estado do cliente entre requisições, já que o protocolo HTTP não tem conceito de estado [15];
- Arquitetura dos componentes *Web*, está relacionada a forma como estruturar os vários componentes *Web* da aplicação para responder às diferentes requisições.

4.1 Formatação da apresentação

Todo componente *Web* que precisa processar requisições e enviar respostas dinâmicas como resultado deste processamento possui duas partes bem distintas, mas que geralmente são tratadas como uma única. Estas são processamento da requisição e formatação da apresentação. O processamento da requisição é a parte responsável por recuperar os parâmetros da requisição, configuração, sessão do usuário ou outras fontes, e depois invocar métodos de negócio. Já a formatação da apresentação é a parte que recebe as informações resultantes da invocação dos métodos de negócio e, usando alguma técnica específica, compõe essas informações dinâmicas com a parte estática da resposta (que descreve apenas a aparência da página) para gerar a resposta para usuário.

Como existem diversas técnicas para gerar a resposta ao usuário, nesta seção são apresentadas as principais técnicas, além de tecnologias que dão suporte a implementação de cada uma destas. Por fim, é feita uma análise comparativa entre elas com o objetivo de definir que técnicas são mais apropriadas para determinadas situações. As

técnicas estudadas nesta seção são *Servlet* puro, Processamento de *templates*, Tratamento de elementos HTML como objetos Java, JSP puro, JSP com *View Helpers* e Uso de tecnologias baseadas em XML.

4.1.1 *Servlet* puro

Esta técnica se resume a codificar trechos da página de resposta ao usuário (geralmente textos HTML) diretamente no código do componente *Web*. No caso de componentes *Web* implementados como *Servlets*, estes trechos são representados por *strings* em Java e enviados para o cliente através de chamadas ao método `println` de um objeto do tipo `PrintWriter`, que no contexto de *servlets* representa o canal de resposta ao cliente *Web*. O trecho de código abaixo exemplifica o uso desta técnica.

```
  ...
1: StringBuffer texto;
2: String mensagem;
3: mensagem = consultarMensagem();
4: texto = new StringBuffer("<HTML><HEAD><TITLE>");
5: texto.append("Exemplo de Servlets com HTML");
6: texto.append("</TITLE></HEAD><BODY>");
7: texto.append(mensagem);
8: texto.append("</BODY></HTML>");
9: out.println(texto);
```

Percebe-se que ao longo do processamento do programa é feito uma série de concatenações de *strings*, através de chamadas ao método `append` de um objeto do tipo `StringBuffer`[1]. Na linha 3 é feita uma chamada ao método `consultarMensagem` que retorna uma mensagem dinâmica que deve ser inserida na página de resposta, logo após a abertura da *tag* `<body>`. Por fim, é enviada a resposta, representada pela variável `texto`, ao cliente *Web*.

A aplicação desta técnica é comum no desenvolvimento de sistemas *Web*, principalmente por ser simples, intuitiva e fácil de entender. Além do mais a maioria dos tutorias, livros e artigos que introduzem o assunto de *Servlets* apresentam esta técnica como forma de gerar páginas dinâmicas. Apesar destes pontos positivos esta abordagem apresenta uma série de desvantagens, principalmente em relação a manutenção e desenvolvimento, apresentadas abaixo.

- Impossibilidade de ver o *layout* da página sem executar o componente *Web*. Muitas vezes durante o desenvolvimento de sistemas para *Web* é necessário visualizar a página a ser gerada, mesmo antes do final do desenvolvimento dos componentes. Esta técnica exige a compilação e execução do componente para isto.

- Conflito entre as responsabilidades dos *designers* e engenheiros de *software*. Geralmente as páginas ficam prontas antes da parte Java do sistema começar a ser implementada, e elas sofrem muitas modificações durante o desenvolvimento. Isto

[1]Classe da API Java que representa um *string* em memória e permite que sejam feitas manipulações em seu conteúdo afim de alterar o seu valor.

torna esta abordagem muito limitante, pois qualquer modificação no formato da página tem que ser feita dentro do código Java e por engenheiros de software, quando mudanças no *layout* da página deveriam ser feitas por *designers*.

- Qualquer modificação no *layout* da página obriga o desenvolvedor a recompilar a aplicação. Como o código HTML está misturado com o código dos componentes *Web*, qualquer modificação no *layout* da página requer que o código fonte dos componentes seja modificado diretamente, e para que as modificações efetivadas é necessário que o sistema seja recompilado. Esta necessidade de recompilar o código fonte dos componentes para qualquer modificação, mesmo as mais simples, diminui muito a produtividade dos desenvolvedores.

- Dificulta a legibilidade do código dos componentes *Web*. Como existe código HTML diretamente dentro do código fonte dos componentes, temos que tratar com duas linguagens dentro de um único arquivo. Os construtores da linguagem de programação se confundem com os de HTML e isto faz com que o código fonte se torne ilegível para sistemas muito grandes. Nestes casos, mesmo que a funcionalidade do sistema seja simples, se a página HTML gerada necessitar de muita informação dinâmica, o código será grande e complicado. A complexidade do componente *Web* não deveria depender do *layout* ou tamanho da página que vai ser gerada.

- Confusão entre os caracteres especiais de Java e caracteres de HTML. Existem alguns caracteres da linguagem HTML que são tratados em Java como caracteres especiais, por isso sempre que estes são encontrados no código do *servlet* o compilador interpreta-os de outra forma acusando erro de sintaxe dentro do *servlet*. A maioria destes caracteres são muito importantes para a estrutura do HTML, e geralmente são impossíveis de serem evitados. Alguns exemplos destes tipos de caracteres são: " (aspas) e \ (barra). A única solução para este problema é colocar o caractere \ (barra) antes de cada caractere especial que causar erro, isto faz com que o compilador entenda o próximo caractere normalmente, como desejado pelo desenvolvedor. Colocar caractere \ em alguns trechos de código HTML, além de ser uma tarefa cansativa, modifica a estrutura sintática original da página fazendo com que fique mais difícil entender ou modificar o código HTML diretamente através do *servlet*.

4.1.2 Processamento de *templates*

Templating é uma técnica usada para aumentar o poder das páginas HTML estáticas. Com *templating* adiciona-se trechos de código a estas. Estes trechos são interpretados durante a execução da requisição, gerando resultados dinâmicos que são inseridos na página. Estas páginas que são extensões de HTML são chamadas de *templates*. Existem diversas implementações de *templating*, desde as mais simples, que suportam apenas variáveis, até mais complexas, que possuem suporte a comandos, funções, declarações. Entre estas implementações podemos destacar FreeMarker [61], WebMacro [29], Velocity [52] e a API do CESAR [9]. Um exemplo de um *template* em FreeMarker pode ser visto no trecho abaixo.

```
<HTML><HEAD>
<TITLE> Exemplo de Servlets com HTML </TITLE>
</HEAD><BODY>
$mensagem
</BODY></HTML>
```

O *template* é bastante similar a uma página HTML pura, no entanto ela define uma variável chamada `mensagem`, que será substituída por informação dinâmica durante o processamento da requisição.

Este tipo de tecnologia é utilizada em conjunto com *servlets*, da seguinte forma: o *servlet* se encarrega de obter as informações dinâmicas e, por fim, executa o *template* passando estas informações (a execução do *template* é feita através de mecanismos específicos da tecnologia utilizada). A execução do *template* se resume a preservar a parte estática da página e inserir os dados passados para ele em seus devidos lugares. As principais características desta abordagem são:

- Permite a visualização do *layout* da página independente da execução do componente *Web*. A maioria dos *templates*, apesar de não serem HTML, podem ser parcialmente visualizados no *browser*.
- Permite o desenvolvimento do componente *Web* paralelamente à construção do *layout* da página, possibilitando a coexistência de papéis diferentes, como engenheiro de *software* e *designers*.
- Não há necessidade de recompilação do código em situações de mudança no *layout* da página, pois geralmente os *templates* são definidos em fontes não compiláveis.
- Facilita a legibilidade do componente *Web*, já que o código referente a formatação da página está definido externamente (no *template*).
- O uso de funcionalidades mais avançadas, tais como comandos, funções. pode aumentar a complexidade do *template* e conseqüentemente diminuir a legibilidade do mesmo. Além do mais, este tipo de uso pode trazer conflitos entre as atividades do *designer* e o do engenheiro de *software*, já que o *template* poderá possuir muita lógica de processamento, além da formatação da página.

4.1.3 Tratamento de elementos HTML como objetos Java

Nesta abordagem os elementos da página HTML são tratados como objetos Java dentro do corpo do *servlet*, o que significa que o componente *Web* não manipula informação HTML diretamente, mas trabalha com objetos Java que representam a página e os elementos HTML que o componente precisa manipular. Algumas implementações deste tipo de modelo são XMLC [53] e Tags [46]. Estas APIs são muito úteis para lidar com elementos de HTML dentro do código Java, oferecendo vários recursos de criação, consulta e manipulação de elementos de HTML, além de tratarem os elementos de HTML como se fossem estruturas em árvore.

O XMLC é um compilador que converte documentos HTML em classes Java, o que significa que para cada arquivo HTML é gerado uma classe que é a representação

Java do HTML. Desta forma os componentes *Web* podem manipular as *tags* de uma página HTML e inserir informação dinâmica as mesmas apenas acessando classes Java. Estas classes geradas possuem um estrutura baseada em DOM [14], o que facilita sua manipulação e entendimento.

A API de Tags não cria classes que representam a página HTML, o que ela faz é interpretar dinamicamente um arquivo HTML e montar uma estrutura em árvore, genérica, que representa a página. Desta forma o componente *Web* pode navegar pela estrutura em árvore lendo, alterando e inserindo informações à estrutura original. Esta estrutura gerada também é baseada em DOM e possui alguns facilitadores que permitem a localização e manipulação dos nós da árvore de modo bastante otimizado. Um trecho de código que exemplifica o uso da API Tags pode ser visto a seguir:

```
   ...
1: TagTiller leitor =  new TagTiller(reader);
2: Tag thePage = leitor.getTilledTags();
3: Text text = Inspector.locateByText(thePage,"$mensagem");
4: text.replaceSelf("Usando API de Tags");
5: out.println(thePage.toHTML());
   ...
```

Na primeira linha é criado um objeto do tipo `TagTiller`, passando como parâmetro a variável `reader`, que representa um *stream* para um arquivo HTML. Este objeto converte um texto HTML em uma estrutura em árvore, que é representada pelo tipo `Tag`. A partir deste objeto, que é a raiz da árvore, pode-se navegar pela estrutura e localizar os descendentes de duas formas: descendo um nível na árvore a cada iteração ou realizando buscas baseadas nas informações das *tags*. A linha 3 é um exemplo de uma busca por elementos texto, ou seja, elementos que não são *tags* ou atributos de *tags*. Neste caso tenta-se localizar o ponto do documento que possui a palavra `$mensagem`. Após o texto ser localizado, ele é substituído por uma frase (na linha 4). Apesar desta frase ser estática no texto, ela poderia ter sido gerada dinamicamente como resultado de alguma operação do sistema. Por fim, o resultado final da árvore é convertido para texto e enviado como resposta para o cliente *Web*.

De fato, esta abordagem é capaz de separar o código HTML do Java de maneira simples e poderosa. No entanto, o uso deste tipo de abordagem pode trazer problemas em algumas situações comuns ao desenvolvimento de sistemas *Web*:

- Apesar do componente *Web* não tratar com elementos HTML diretamente, ele precisa manipular objetos Java que são estruturalmente equivalentes às páginas. Em alguns casos, para manipular e inserir informações dinâmicas nestas estruturas os componentes *Web* precisam navegar pelos objetos e conseqüentemente conhecer a estrutura do documento. Isto faz com que o código do *servlet* fique bem dependente da estrutura do documento HTML. Como conseqüência negativa, em páginas cuja estrutura é complexa o componente *Web* também será.

- Mudanças estruturais no *layout* da página geralmente requerem mudança no código do componente *Web*. Como os objetos que o componente *Web* precisa manipular são a representação estrutural da página HTML, algumas mudanças nesta estrutura requerem que o código do componente seja ajustado para navegar na nova

estrutura. Com o uso de XMLC o problema é ainda maior, devido a necessidade de regerar as classes que representam as páginas.

4.1.4 JSP puro

Páginas JSP possuem o mesmo objetivo dos *servlets*: gerar conteúdo dinâmico para clientes *Web*. A grande diferença entre as duas tecnologias está no modelo de programação, ou seja, na forma como desenvolver programas. As páginas JSP podem ser vistas como uma extensão das páginas HTML comuns, já que elas podem conter trechos de código Java ou *tags* especiais de JSP inseridas entre código HTML. Esta tecnologia é muito similar a técnica de *templating* (visto na Seção 4.1.2), a única diferença é que o uso desta substitui a necessidade de *servlets*, enquanto que os *templates* trabalham em conjunto com os *servlets*.

O trecho de código abaixo exemplifica um JSP simples que monta uma página contendo a data e hora da execução da requisição. Como pode ser visto, o trecho de código inserido entre os delimitadores de expressões (representados por `<%=` e o `%>`) nada mais é do que um comando Java que retorna a data corrente, que é automaticamente convertida para *string* e inserida no local onde o comando aparece.

```
...
<html>
  <body>
    <h1>
      <%= Calendar.getInstance().getDate() %>
    </h1>
  </body>
</html>
```

JSP é uma tecnologia simples e que pode ser utilizada para desenvolver desde aplicações simples até sistemas de grande porte. Ela é totalmente baseada em *servlets*, o que significa que ambas as tecnologias possuem o mesmo poder de expressividade. Apesar desta tecnologia ser de mais alto nível que *servlets*, com o objetivo de auxiliar o desenvolvedor de sistemas *Web*, ela apresenta algumas falhas e problemas comuns à abordagem de *servlets* puro (apresentado na Seção 4.1.1). As principais características do uso de JSP puro podem ser vistas logo abaixo:

- Os papéis do *designer* e engenheiro de *software* entram em conflito. Tanto as modificações no formato da página, quanto as alterações na lógica de processamento da requisição são feitas na mesma fonte (arquivo JSP).
- A legibilidade da página JSP fica comprometida para sistemas de médio e grande porte. Três linguagens devem ser tratadas na página JSP: Java, *tags* JSP e HTML. Os elementos de cada uma das linguagens tendem a se confundir e o código da página tende a se tornar ilegível para sistemas muito grandes.
- *Servlet* puro tem o problema de inserir código HTML dentro do código Java, o que aumenta a complexidade do componente *Web* e traz vários problemas já relatados na Seção 4.1.1. JSP foi proposto como solução para este tipo de problema, e de

fato ele resolve o problema, no entanto o seu uso acrescenta um problema similar que é inserir código Java dentro do código HTML. Percebe-se que o modelo de programação dos *servlets* e JSP são bem diferentes, no entanto o problema de misturar Java com HTML é o mesmo.

- Não há necessidade de recompilação quando a página é modificada. Na verdade a recompilação da página JSP é feita automaticamente sem intervenção do desenvolvedor. O ambiente onde as páginas JSP são executados se encarrega de checar se houve alguma alteração na página e automaticamente inicia o processo de compilação, se necessário.

- O *layout* da página pode ser visto sem necessidade do componente *Web* estar finalizado. Existem diversas ferramentas que permitem a visualização do formato de páginas JSP.

4.1.5 JSP com *View Helpers*

Já foi constatado na Seção 4.1.4, que o uso de JSP puro não é recomendável e apresenta problemas similares à abordagem de *Servlet* puro. Por este motivo se faz necessário o uso de um modelo mais refinado que tenta preservar as vantagens do uso de JSP e ao mesmo tempo resolver os problemas apresentados pela mesma. O uso de *View Helpers* tem como objetivo evitar a inclusão de código Java diretamente nas páginas JSP. Para isto são definidos componentes, denominados *View Helpers*, que devem possuir o código Java necessário para o processamento da requisição. As páginas JSP apenas usam estes componentes.

Existem duas formas de implementar este tipo de componente, através de JavaBeans ou com uso de *Tags* JSP customizáveis. A primeira destas é mais simples. Basicamente cria-se um ou vários componentes Java que possuem a lógica de processamento da requisição e na página JSP são inseridas apenas chamadas para a execução dos serviços destes componentes. Um exemplo desta implementação pode ser vista no trecho de página JSP abaixo:

```
...
<html>
  <body>
    <h1>
      <% BeanXXX.executarParteYYYDaRequisicao(request); %>
    </h1>
  </body>
</html>
```

Mesmo usando os *View Helpers*, percebe-se que não há como evitar os pequenos trechos de código Java referente a invocação dos JavaBeans na página JSP.

A outra forma, através de *Tags* customizáveis, é mais elegante, no entanto mais complexa. Neste caso são definidas *Tags* especiais que podem ser usadas na página JSP da mesma forma como as *Tags* pré-definidas de JSP são usadas. No trecho abaixo pode-se ver a mesma página JSP do exemplo anterior, mas implementada com *Tags* customizáveis.

```
...
<html>
  <body>
    <h1>
      <tags_estendidas:execucaoXXXDaParteYYY/>
    </h1>
  </body>
</html>
```

Esta implementação é bem mais elegante e torna a página JSP mais uniforme, sem código Java algum. No entanto a definição de *Tags* customizáveis é uma atividade não trivial, pois requer a definição de componentes Java responsáveis por executar a lógica da *Tag* e a definição de algumas configurações em recursos específicos do ambiente onde a *Tag* vai ser executada [24].

4.1.6 Uso de tecnologias baseadas em XML

O uso de XML [4] vem se tornando bastante popular e o número de tecnologias de suporte a esta tem crescido bastante nos últimos tempos. XML pode ser usada com diferentes propósitos, inclusive para auxiliar o processo de montagens de páginas dinâmicas [4]. Existem mecanismos simples e poderosos para transformação de documentos XML em páginas HTML. Uma tecnologia que dá suporte a este tipo de transformação é o XSL (*Extensible Stylesheet Language*) [8]. O uso desta abordagem se resume a manipular documentos XML, que representam os dados da aplicação, e por fim formatá-los para que possam ser visualizados pelo *Web browser*. Esta formatação é feita através do uso da linguagem de transformação XSL. O trecho abaixo mostra um exemplo bem simples de um documento XML que representa os dados de um usuário da aplicação. Neste exemplo um usuário possui três informações: o nome, login e a data do último acesso.

```
<usuario>
  <nome>Gibeon Aquino</nome>
  <login> gsaj </login>
  <ultimoacesso> 10/03/2002 </ultimoacesso>
</usuario>
```

Para que estas informações possam ser vistas pelo usuário *Web* no *browser* é necessário escrever código XSL que guie a transformação do documento XML em uma versão HTML. O trecho abaixo exemplifica um documento XSL que define as transformações:

```
1: <xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
2:    <xsl:template match="/">
3:      <HTML><BODY>
4:       <xsl:apply-templates select="/usuario" />
5:      </BODY></HTML>
6:    </xsl:template>
7:    <xsl:template match="/usuario">
8:     <H1>  Login = <xsl:value-of select="./login" /> </H1>
```

```
 9:        Ultimo Acesso de <b> <xsl:value-of select="./nome" /> </b>
10:        foi no dia <b> <xsl:value-of select="./ultimoacesso" /> </b>
11:    </xsl:template>
12: </xsl:stylesheet>
```

Como pode ser visto no trecho acima, um documento XSL possui algumas *tags* especiais que localiza os elementos contidos no documento XML e consultam seus valores. A *tag* `xsl:template` indica a definição de um *template* e possui um atributo denominado `match`, que indica a que elemento do documento XML o *template* deve ser aplicado. No exemplo anterior, o primeiro *template* (iniciado na linha 2) é aplicado a um elemento abstrato que referencia o documento XML como um todo. O resultado desta aplicação é a abertura de uma página HTML (`<HTML><BODY>`), a aplicação de outro *template* ao elemento `usuario` e, por fim, o fechamento das *tags* de HTML (`</BODY></HTML>`). O segundo *template*, que é aplicado para cada elemento `usuario`, seleciona os valores contidos nos subelementos `login`, `nome` e `ultimoacesso` mediante o uso da *tag* de XSL `xsl:value-of`, inserindo-os no meio de um trecho de código HTML. O resultado final do processamento deste XSL pode ser visto a seguir:

```
<HTML><BODY>
  <H1>  Login = gsaj </H1>
  Ultimo Acesso de <b> Gibeon Aquino </b>
  foi no dia <b> 10/03/2002 </b>
</BODY></HTML>
```

De fato, XSL é muito simples e o seu objetivo é apenas de formatação da página. Por este motivo, ela evita que o desenvolvedor misture código de processamento da requisição com a lógica de formatação da apresentação. Esta abordagem, apesar de interessante, apresenta algumas limitações:

- XSL é uma linguagem bem limitada. Por isso, fazer alguns tipos de processamento e tratamento dos dados (documento XML), é uma tarefa árdua e custosa ou, em muitos caso, impossível.
- Necessidade de conhecimento de uma terceira linguagem. Apesar da linguagem ser simples, é necessário o conhecimento de alguns elementos básicos da sua sintaxe para que o desenvolvedor possa usá-la de forma correta e alcançar seus objetivos finais. O modelo de desenvolvimento com XSL é bastante diferente do modelo procedural da maioria das linguagens de programação; ela é baseada no casamento de elementos XML e na aplicação de *templates* aos mesmos.
- Os construtores da linguagem XSL geralmente se misturam com os de HTML, o que causa uma certa confusão. Para transformações de XML para HTML é necessária a inclusão da parte estática do documento HTML diretamente no documento XSL. Isto faz com que os construtores das duas linguagens fiquem misturados, o que pode tornar o documento XSL ilegível em casos onde a transformação ou a página final é relativamente complexa.

- Causa um *overhead* desnecessário em aplicações cujos dados não são baseados em XML. Em várias aplicações, os dados ou entidades do sistemas são representados por objetos Java, o que torna o sistema elegante e intuitivo. Estes tipos de aplicações precisam ficar convertendo os seus objetos Java em representações XML para que ela possa ser processada por um XSL e gerar uma resposta HTML que será visualizada pelo cliente *Web*. Esta conversão implica em *overhead* de processamento, o que pode custar caro para alguns tipos de aplicações, e também aumenta a complexidade do programa, já que torna-se necessário definir rotinas de conversão Java em XML para cada entidade do sistema que necessitar ser apresentada para o cliente *Web*.

4.1.7 Análise comparativa entre as abordagens

Seis foram as abordagens para formatação da apresentação discutidas nas seções anteriores: *Servlet* puro, Utilização de *templatings*, HTML como elemento Java, JSP puro, JSP com *View helpers* e Tecnologias baseadas em XML. O objetivo desta seção é comparar estas abordagens com base em características importantes para sistemas *Web*. A Tabela 4.1 dá uma visão resumida do comportamento das abordagens para cada uma das características listadas na tabela.

Características	**Abordagens**					
	***Servlets* puro**	**Utilização de *templatings***	**HTML como elemento Java**	**JSP puro**	**JSP com *view helpers***	**Tecnologias baseadas em XML**
Visualização independente da execução		⋆ ⋆		⋆ ⋆	⋆ ⋆	⋆
Conflitos entre papéis	⋆	⋆ ⋆ ⋆	⋆	⋆	⋆ ⋆ ⋆	⋆ ⋆
Código de processamento indedependente do *layout*	⋆	⋆ ⋆ ⋆	⋆	⋆	⋆ ⋆	⋆
Legibilidade	⋆	⋆ ⋆	⋆	⋆	⋆ ⋆	⋆
Conflitos entre caracteres Java e HTML	⋆	⋆ ⋆	⋆ ⋆	⋆ ⋆	⋆ ⋆	⋆ ⋆
Mistura mais de uma linguagem na mesma fonte	⋆	⋆ ⋆	⋆ ⋆ ⋆	⋆	⋆ ⋆	⋆
Performance	⋆ ⋆ ⋆	⋆	⋆	⋆ ⋆ ⋆	⋆ ⋆	⋆

Tabela 4.1: Comparações entre as abordagens de formatação da apresentação.

Muitas vezes durante o desenvolvimento dos componentes *Web* se faz necessário a visualização do *layout* da página a ser gerada sem que o componente *Web* (elemento que contém a lógica de processamento da requisição) seja executado. As abordagens de *Servlet* puro, HTML como elemento Java e XSL realmente só permitem a visualização do *layout* da página com a execução do componente *Web*.

Algumas abordagens de formatação da apresentação tratam o código de processamento da requisição e a definição de *layout* da página como um único componente. Estes dois tipos de código são desenvolvidos e mantidos por papéis diferentes:

4.2 Validação de dados da requisição

No desenvolvimento de sistemas, especialmente aqueles que recebem entrada de usuários, como sistemas *Web* ou aplicações *stand alone* Java com GUI (*Graphical User Interface*), é gasto muito tempo com a escrita de código de validação das entradas (lógica para tratamento de erros dos usuários). Geralmente estas validações precisam estar presentes em várias partes do sistema e muitas vezes este código é duplicado, o que dificulta o desenvolvimento e a manutenção do sistema.

Em sistemas *Web*, este problema se torna ainda maior, já que a GUI é definida em HTML e é executado na máquina do cliente enquanto que o *backend* é escrito em Java (por exemplo) e reside no servidor. Nestes casos é necessário escrever código de validação para ser executado na GUI, e por esta estar residente em um *browser*, as regras de validações devem ser definidas em JavaScript[2]. Estas validações precisam ser feitas no cliente *Web* para evitar que o usuário só receba as mensagens de erro de validação após executar a requisição (ir e voltar do servidor). Por outro lado, são vários os trabalhos na área de segurança que indicam que a aplicação não deve confiar inteiramente no cliente *Web* e, por isso, estas regras devem ser replicadas também do lado do servidor ([63],[62],). Percebe-se que a replicação de código de validação nestes casos é inevitável e, o que é pior, cada um sendo escrito em linguagens diferentes (JavaScript e Java).

Várias são as técnicas para escrita de código de validação. Neste trabalho dividimos estas em três categorias principais: Simplificada, Estruturada e Estruturada customizável. Cada categoria corresponde a uma seção, onde é proposto um modelo de implementação baseado nas técnicas existentes, que tenta unir as vantagens de cada técnica e resolver os problemas oferecidos por elas. Por fim, é feita uma comparação qualitativa destas com o objetivo de guiar os desenvolvedores e projetistas a escolherem qual delas usar durante o desenvolvimento de sistemas.

Apesar de analisarmos cada uma das abordagens focando em sistemas para *Web*, elas podem ser generalizadas e usadas em outros tipos de sistemas, preservando grande parte de suas semânticas.

4.2.1 Abordagem Simplificada

Esta é forma mais comum de implementar validação de dados, principalmente por ser simples e intuitiva. Ela envolve basicamente a escrita de código Java de validação

[2]JavaScript é uma linguagem de *script* interpretada. Ela é usada no desenvolvimento de *Web sites* e pode ser inserida em páginas HTML para ser executada pelo *browser*. Sua sintaxe possui algumas coisas em comuns com a linguagem Java, mas a similaridade entre as duas é apenas esta.

diretamente no corpo dos *servlets*, e de JavaScript diretamente nas páginas HTML.

No trecho de código de um *servlet*, mostrado logo abaixo, que recebe a hora e minuto como parâmetro da requisição, percebe-se como são feitas as validações sobre os dados da requisição nesta abordagem:

```
public void doGet(ServletRequest request,ServletResponse response)
 throws IOException, ServletException{
  int hora,minuto;
  try{
    hora = Integer.parseInt(request.getParameter("hora"));
    minuto = Integer.parseInt(request.getParameter("minuto"));
  }catch(NumberFormatException nfe){...}
  if(hora < 0 || hora > 23){...}
  if(minuto < 0 || minuto > 59){...}
}
```

De fato, verifica-se que o código de validação é bastante simples e intuitivo. O *servlet* recebe dois parâmetros da requisição, a hora e o minuto, tenta convertê-los para inteiro, caso não seja possível notifica o usuário do erro (através da execução do código referente ao tratamento da exceção `NumberFormatException`). São validados, também, os limites permitidos para cada um dos parâmetros nas linhas subseqüentes, e da mesma forma é executado um trecho de código referente à notificação do erro ao usuário caso as validações falhem.

Como já falado, estas validações também precisam estar presentes no cliente *Web*, e por isso têm que ser definidas em JavaScript e embutidas na página HTML na qual os dados são informados pelo usuário. Um trecho da página HTML na qual o usuário informa o valor da hora e minuto, submetendo-os para o *servlet* do exemplo anterior pode ser visto abaixo:

```
<form name="formulario" action="ServletXX" method="GET">
  <input type="text" name="hora">
  <input type="text" name="minuto">
  <input type="button" onClick="javascript:submeterDados();">
</form>
```

Nela o usuário deve informar o valor da hora e minuto. Ao clicar no botão definido no formulário, é executada a função JavaScript `submeterDados`. Esta função faz a validação dos dados informados pelo usuário e depois submete o formulário. A implementação em JavaScript desta função de validação é bastante simples. Ela executa basicamente as mesmas validações que são feitas no trecho de código em Java do *Servlet* exemplificado anteriormente:

```
1: function submeterDados(){
2:   var hora, minuto;
3:   var form = document.formulario;
4:   hora = form.hora.value;
5:   minuto = form.minuto.value;
```

```
6:    if(hora < 0 || hora > 23){...}
7:    if(minuto < 0 || minuto > 59){...}
8:    form.submit();
9: }
```

Na linha 3, é feita uma referência ao formulário definido na página HTML com o nome `formulario`. Nas linhas 4 e 5, pega-se os valores informados pelo usuário (hora e minuto, respectivamente). Depois são feitas as validações para verificar se os valores respeitam os limites estabelecidos, executando o código de tratamento do erro caso contrário. Por fim, submete-se o formulário ao *Servlet* através da execução de seu método `submit`.

Apesar desta abordagem ser bastante simples e aplicável em qualquer situação, a codificação destas regras se torna bastante repetitiva, pois em muitos casos as regras são as mesmas e a única coisa que muda são os seus valores. Por exemplo, para hora e minuto a regra é a mesma, ou seja, verificar se o dado pertence a um determinado intervalo, mas os valores para os intervalos é que são diferentes (a hora deve ser entre 0 e 23, enquanto o minuto entre 0 e 59).

Em sistemas de grande porte, a tendência é que a quantidade de código duplicado torne-se muito grande. Isto acontece principalmente porque a quantidade de dados a serem validados é bem maior, e muitos deles são usados em várias partes do sistema. Outro problema típico de aplicações *Web* é a necessidade de escrita de código de validação em JavaScript, o que implica na replicação de código nas diversas páginas HTML do sistema e, o que é pior, a necessidade de escrita das mesmas regras em diferentes linguagens (Java e JavaScript).

4.2.2 Abordagem Estruturada

Esta abordagem se baseia no fato de que as diversas regras de validação existentes podem ser agrupadas em categorias, e estas podem ser facilmente definidas e parametrizadas de forma que possam ser reusadas e configuradas para serem usadas nas diferentes partes do sistema. De fato, algumas destas são identificadas de forma bastante intuitiva por serem usadas na maioria das situações onde é necessária a realização de validação:

- **Validação de tipo** – Verifica se o dado é do tipo esperado, tal como inteiro, ponto flutuante, texto, data.
- **Validação de obrigatoriedade** – Verifica se o dado está presente, ou seja, se ele não é nulo.
- **Validação de tamanho** – Verifica se o dado tem o tamanho especificado. É geralmente usado para textos, mas pode ser estendido para números também.
- **Validação de intervalo** – Verifica se o valor respeita os limites estabelecidos. Exemplos de restrições de limites são maior que e menor que (inclusivo ou exclusivo). Esta restrição pode ser aplicada para caracteres, data, textos.
- **Validação de pertence** – Verifica se uma entrada do usuário pertence a uma lista de valores predefinida.

- **Validações de padrões** – Verifica se um dado valor respeita um determinado formato. Exemplos destes são "dd-mm-aaaa", "mm/aaaa".

- **Validações de segurança** – Verifica se o conteúdo do dado pode fazer com que a aplicação se comporte de forma inesperada. Por exemplo, uma técnica utilizada por *hackers* é a injeção de comandos, onde comandos que podem trazer problemas para aplicação são inseridos entre os dados e executados pelo sistema normalmente. Exemplos destes são comandos SQL que violam ou apagam os dados da aplicação, ou em alguns casos simplesmente dão acessos a informações sigilosas.

A idéia desta abordagem é o desenvolvimento de uma API em Java para realizar vários tipos de validações necessárias em sistemas que recebem dados de usuários. Esta API deve suportar os tipos de validações listados anteriormente, e o mais importante é que ela seja flexível o suficiente para que novos tipos de validações possam ser criados pelo programador de forma simples e sem mudança no mecanismo original.

A classe cujos objetos representam regras de validação é a `ValidationRule`, uma classe abstrata que possui um atributo chamado `name`, métodos de acesso ao mesmo e duas versões do método `validate`. O atributo `name` representa o nome da regra, que deve ser único para cada regra definida. O método `validate` é abstrato e deve ser definido pelas implementações concretas das regras de validação; este é o método onde deve estar definido todo código de validação. A primeira versão recebe apenas um parâmetro, que é o campo ao qual a regra será aplicada, e a outra recebe um mapeamento de nomes e valores de campos (`HashMap` de Java), que representa um conjunto de campos ao qual a regra deve ser aplicada.

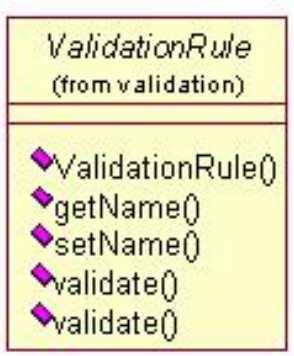

Figura 4.1: Definição da classe `ValidationRule` em UML.

A API já deve oferecer a implementação dos tipos mais comuns de validação; uma delas é a de tipo, cujo esboço pode ser visto no trecho de código abaixo. Como pode ser visto, ela deve herdar da classe `SingleFieldValidationRule` e implementar o método `validate`, esta última classe estende a `ValidationRule` e representa regras de validações que são aplicadas a um único campo. Nela tenta-se transformar o campo `fieldValue` (representado como *string*) para o tipo esperado. Caso não seja possível (ocorra uma exceção do tipo `NumberFormatException`), é criada e lançada uma exceção de validação representada pela classe `ValidationException`.

```
 1: public class TypeValidation extends SingleFieldValidationRule{
 2:   public void validate(String fieldValue)
        throws ValidationException{
 3:     try{
 4:       if(type.equals("int")){Integer.parseInt(fieldValue);}
 5:       else if(type.equals("double")){
                                Double.parseDouble(fieldValue);}
 6:       ...
 7:     }catch(NumberFormatException nfe){
 8:        throw new ValidationException(fieldValue,type);
 9:     }
10:   ...
```

Não é necessário para o desenvolvedor saber detalhes da implementação acima, ele tem apenas que se preocupar com o nome dos parâmetros da regra, neste caso apenas `type`, para que ele possa configurar a regra da forma desejada. O trecho abaixo é um exemplo da instanciação de uma regra chamada `integer-validation` que verifica se o campo é do tipo inteiro:

```
TypeValidation type = new TypeValidation("integer-validation");
type.setType("int");
```

Existem alguns tipos de regras mais complexas que envolvem vários campos, onde a validação de um deles é dependente do valor dos outros. Um exemplo deste é a validação de datas (quando o dia, mês e ano são recebidos cada um em um campo), onde alguns meses têm 31 dias e outros não. Neste caso, se o valor do campo dia for 31 o campo mês tem que ser restringido. Para estes tipos de regras que envolvem vários campos existe uma classe básica específica, a `MultipleFieldValidationRule`, e seu método `validate` recebe uma `HashMap` (mapeamento de nome de campos em seu valores). Esta classe também herda de `ValidationRule`, como pode ser visto na Figura 4.2.

Algumas regras podem ser construídas pela composição das já existentes. Por exemplo, a regra que faz a validação de campos inteiro positivo é a composição da regra que verifica o tipo do campo (`int`) com a que valida o limite (≥ 0). A classe que dá suporte à composição de regras de validação é a `CompositeValidationRule`; ela possui dois atributos do tipo `ValidationRule` e outro que indica qual o operador a ser aplicado. Três são os operadores suportados: **or** – ou lógico, **and** – e lógico e o **xor** – ou exclusivo lógico. O trecho de código abaixo exemplifica como criar a regra dos inteiros positivos usando regras pré-definidas. Como pode ser visto quatro são os parâmetros do seu construtor: o nome da regra, primeiro operando, segundo operando e o operador, nesta ordem.

```
SingleFieldValidationRule typeValidation, limitValidation;
... // Cria as regras de tipo e limite
... // Configura as regras
CompositeValidationRule comp =
   new CompositeValidationRule("integer-positive-validation",
                               typeValidation,
                               limitValidation,
                               CompositeValidationRule.AND);
```

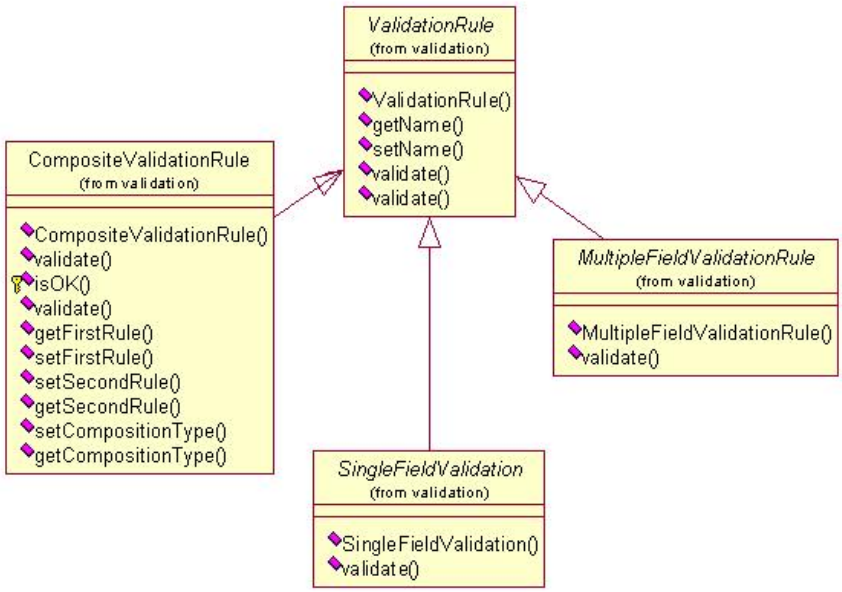

Figura 4.2: Hierarquia das classes básicas que representam as regras de validação.

Aplicação das regras de validação

Instanciar e configurar cada regra desta onde se deseja usá-la é uma tarefa bastante custosa e pode acarretar em duplicação de código, por isso este *framework* oferece uma classe que é a responsável por manter todas as regras de validação e serve como interface para aplicação das regras. No momento de realizar a validação, ao invés do desenvolvedor instanciar e configurar cada regra de validação, ele basicamente solicita a esta classe que aplique determinada regra em um ou mais campos:

```
Validator validador = Validator.getInstance();
validador.applyRule("integer-positive-validation",campo1);
```

O diagrama UML da classe `Validator` pode ser visto na Figura 4.3, onde verifica-se que o único método abstrato é o `load`. Este deve ser usado para que o desenvolvedor instancie e configure todas as regras de validação, cadastrando-as no repositório padrão da classe. As duas versões do método `applyRule` devem ser usados para aplicar a validação em um campo ou em vários deles, respectivamente. O método `setValidationRuleRepository` deve ser chamado em situações onde se deseje mudar a implementação do repositório que armazena as regras de validação, que deve ser subclasse de `ValidationRuleRepository`. A implementação padrão deste repositório é do tipo `ValidationRuleRepositoryDefault`, que armazena os objetos em memória.

O método `getInstance` deve ser usado pela aplicação para recuperar uma instancia do `Validator`, mas existem duas versões deste método: uma que recebe o nome da implementação de `Validator` a instanciar e outra que nada recebe e assume que o nome da classe a ser instanciada está armazenada em um arquivo de propriedades da API. No trecho de código a seguir, pode ser vista uma implementação concreta da classe `Validator`.

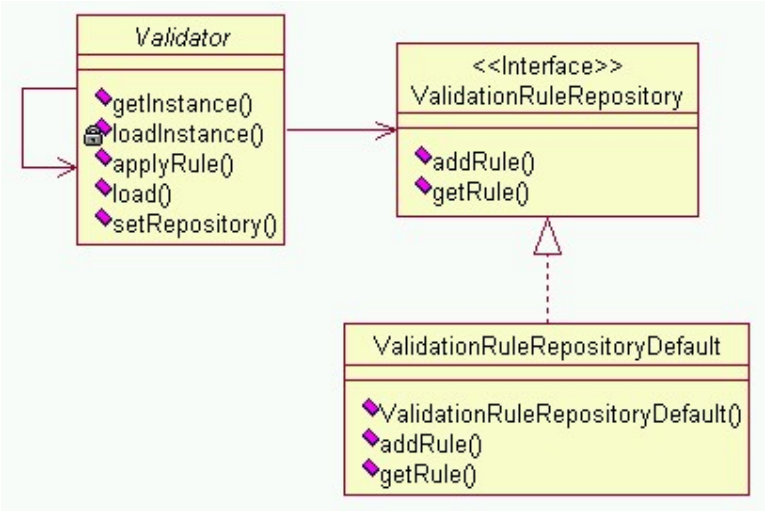

Figura 4.3: Diagrama UML da classe abstrata Validator.

```
class ValidatorExample extends Validator{
  public void load(){
    SingleFieldValidationRule typeValidation, limitValidation;
    typeValidation = new TypeValidation("integer-validation");
    limitValidation = new LimitValidation("positive-validation");
    typeValidation.setType("int");
    limitValidation.setBottom(0);
    limitValidation.setInclusive(true);
    repository.addRule(typeValidation);
    repository.addRule(limitValidation);
  }
}
```

Como pode ser visto, no método `load` coloca-se todo o código referente a instanciação, configuração e armazenamento das regras de validação.

Representação textual da regra de validação

Uma funcionalidade interessante deste *framework* é a capacidade de gerar um texto customizável que representa uma determinada regra já definida na implementação do `Validator`. Esta representação textual pode ser configurada de forma que o texto gerado possa ter o formato desejado. Para definir o formato deste texto basta criar um arquivo com o mesmo nome da classe que representa a regra de validação, mas com a extensão `.vrt` (*Validation Rule Text*). Este arquivo funciona como um *template*, onde o programador pode colocar qualquer texto referente a regra em questão. O trecho abaixo representa o *template* para gerar o código JavaScript da regra de validação de tipo (mostrada anteriormente), ele deve possuir o nome `TypeValidation.vrt`:

```
function $rule-name$(var field){
  var type = $type$;
```

```
    if(type == "int"){... // Verifica se o campo é inteiro}
    else if(type == "double")){...// Verifica se é double}
    ...
}
```

Nos *templates* pode ser colocado qualquer tipo de texto, neste caso foi JavaScript, mas nada impediria escrever outros tipos de texto como Delphi, C. As palavras entre o caracteres $, representam referências para os atributos das regras de validação (informados pelo método `setXXX`), neste arquivo a palavra `$type$` será substituído pelo valor do atributo `type` da regra de validação `TypeValidation` durante a conversão para texto. Por exemplo, para a regra `integer-validation` esta referência `type` será substituída por `int`.

Por fim, o desenvolvedor só precisa invocar o método `getTextualFormat` da classe `Validator`, passando o nome da regra que ele deseja ter o formato textual, como no trecho abaixo:

```
String textualForm;
Validator validator = Validator.getInstance();
textualForm = validator.getTextualFormat("integer-validation");
```

Esta abordagem resolve parcialmente o problema da duplicação de código de validação e facilita a vida dos desenvolvedores, provendo um conjunto de classes que podem ser configuradas e usadas em diferentes situações. No entanto ela ainda possui um problema: sem usar algum tipo de fonte não compilável para definir os parâmetros das regras de validação, ou seja, os limites, padrões, tipos, etc., mudanças nas informações das regras irão sempre resultar em recompilação do código.

4.2.3 Abordagem Estruturada Customizável

Esta abordagem é uma extensão da apresentada na Seção 4.2.2. Ela usa todas as classes e conceitos definidos anteriormente, além de embutir funcionalidades para resolver o problema de recompilação do código toda vez que as propriedades das regras de validação mudarem.

As categorias de regras de validação devem ser definidas em classes Java (assim como na abordagem anterior), no entanto as configurações delas (nome, valores dos parâmetros) devem ser definidas em uma segunda fonte, não compilável e, o que é mais importante, de forma descritiva.

Como XML é bastante poderosa, flexível, portátil e está se tornando um padrão para descrição de informação, escolhemos usá-la como linguagem para descrição das regras de validações. Outra alternativa seria usar um arquivo de propriedades de Java, no entanto o poder de expressividade destes é bastante limitado e representar alguns conceitos nestes seria uma tarefa árdua ou deselegante. Com isso, a partir de uma descrição de regras de validação em XML, será possível configurar as classes que representam as regras e aplicá-las onde necessário.

Um exemplo de instanciação e configuração da regra de validação de tipo inteiro, mostrada na Seção 4.2.2 pode ser vista no trecho de XML abaixo:

```
<rule class="TypeValidation" name="integer-validation">
  <parameter>
    <name> type </name>
    <value> int </value>
  </parameter>
</rule>
```

De fato, este trecho define uma regra de validação chamada `integer-validation` do tipo `TypeValidation` (classe definida na Seção 4.2.2), onde o valor do parâmetro `type` é `int`. Com o suporte do *framework*, este trecho de XML substitui a necessidade de instanciar uma regra do tipo `TypeValidation`, informar os parâmetros através de chamadas aos métodos `setXXX` e cadastrá-la em alguma implementação concreta da classe `Validator`.

Também é possível expressar regras que são composição de outras já existentes, de forma simples, intuitiva e sem precisar criar classes adicionais. Por exemplo, o trecho de XML abaixo descreve a regra `integer-positive-validation` que é a composição das regras já definidas `integer-validation` e `positive-validation`. Como o tipo da composição é `and`, as duas regras têm que ser validadas:

```
<composite-rule name="integer-positive-validation">
  <composition type="and">
    <rule> integer-validation </rule>
    <rule> positive-validation </rule>
  </composite>
</composite-rule>
```

Nesta abordagem não é necessário definir uma subclasse de `Validator` e implementar o método `load`, contendo a instanciação, configuração das regras de validação e cadastro delas no repositório de regras (já visto na Seção 4.2.2). Todas estas operações são executadas automaticamente por um componente que interpreta as definições de regras de validação em XML. Estas regras devem ser definidas em um único arquivo XML (denominado `validation-rules.xml`). Elas devem ter a sintaxe similar aos dois trechos de XML mostrados anteriormente, e o arquivo completo deve ter o seguinte formato:

```
<validation-rules-definition>
  <!-- Definição das regras de validação -->
</validation-rules-definition>
```

Como pode ser visto ele possui o elemento raiz `validation-rules-definition`, que no seu corpo pode ter uma ou mais definições de regras de validação. O componente responsável por interpretar este arquivo, instanciar as regras e aplicá-las aos campos desejados é a classe `Validator`. Para utilizá-la basta recuperar uma instância da mesma, através da chamada ao método `getInstance`, que retorna um objeto *Singleton* do tipo `Validator` e que contém todas as regras definidas no arquivo `validation-rules.xml`. Por exemplo, assumindo que a definição `integer-positive-validation`, mostrada anteriormente nesta seção, está presente no arquivo XML de definição das regras, o trecho de código abaixo aplica a regra de inteiro positivo (`integer-positive-validation`) ao campo `campo1`:

```
...
Validator validador = Validator.getInstance();
validador.applyRule("integer-positive-validation", campo1);
...
```

Como já comentado na Seção 4.2.2, a API de validação deve oferecer a implementação das regras de validação mais comuns, como: validação de tipo, obrigatoriedade, tamanho, intervalo, padrões. Além do mais, através do arquivo XML, é possível oferecer também as configurações mais comuns destas regras, tais como validação de tipo inteiro, ponto flutuante, *string*, validação de inteiros positivos, alfanuméricos.

4.2.4 Análise comparativa entre as abordagens de validação

Nesta seção foram apresentadas três abordagens para validação de dado de entrada: Simplificada, Estruturada e Estruturada Customizável. Para cada uma delas foi dada uma idéia do funcionamento, características e exemplos de uso. Agora realizamos uma análise comparativa entre elas e explanamos de forma mais objetiva e resumida as características de cada uma das abordagens. A Tabela 4.2 mostra o resultado da análise das três abordagens de validação com base em quatro características importantes no desenvolvimento de sistemas.

Características	**Abordagens**		
	Simplificada	**Estruturada**	**Estruturada Customizável**
Simplicidade de uso	⋆ ⋆	⋆	⋆ ⋆ ⋆
Duplicação de código	⋆	⋆ ⋆	⋆ ⋆ ⋆
Mudanças na configuração exigem recompilação	⋆	⋆	⋆ ⋆ ⋆
Qtde de código Java por validação	⋆	⋆ ⋆	⋆ ⋆ ⋆
Performance	⋆ ⋆ ⋆	⋆ ⋆	⋆

Tabela 4.2: Comparações entre as abordagens de validação de dados de entrada.

A abordagem Simplificada é bastante simples de se usar já que exige apenas conhecimento da linguagem Java e de suas APIs básicas. Por este motivo ela é bastante utilizada pelos desenvolvedores. Já a abordagem Estruturada exige um maior conhecimento por parte do desenvolvedor, que precisa conhecer as classe que representam cada tipo de validação e, principalmente, o funcionamento das classes que controlam o registro e aplicação das regras de validação. A última abordagem, Estruturada Customizável, é uma extensão da anterior, usando as classes definidas pela outra, mas criando algumas funcionalidades que tem como objetivo dar uma maior abstração do seu mecanismo de funcionamento interno. São exatamente estas funcionalidades que tornam o uso desta abordagem mais simples. Na maioria dos casos o desenvolvedor vai configurar e aplicar as validações aos campos de forma declarativa, apenas em XML, e sem necessidade de conhecimento da linguagem Java.

A duplicação de código é uma característica indesejada em qualquer tipo de sistema. Por este motivo é importante que as soluções propostas promovam o maior reuso possível.

As abordagens Estruturada e Estruturada customizável promovem o reuso do código de validação mediante a definição de componentes que encapsulam as regras de validação e conseqüentemente não apresentam problemas relativos a duplicação de código. Já na abordagem Simplificada, não há preocupação com o reuso de código e, por este motivo, ela facilita a replicação de código.

Mudanças nos requisitos da aplicação são bastante comuns durante o desenvolvimento e mesmo após o seu término. Estas mudanças podem fazer com que as regras de validações sofram pequenas modificações. Nas abordagens Simplificada e Estruturada as definições das regras de validação são feitas em Java e por isto exigem recompilação sempre que houver mudanças nas mesmas. Já na abordagem Estruturada Customizável as definições das regras são feitas em uma fonte não compilável (arquivo XML) e por este motivo podem ser alteradas sem necessidade de recompilação.

Geralmente o código referente às validações de entrada de dados são responsáveis por uma parte significante do total de código da aplicação. Esta situação ocorre porque a maioria das aplicações utilizam a abordagem Simplificada, que não reusa código e conseqüentemente exige a escrita de muito código para cada validação necessária. A Estruturada promove o reuso das regras de validação e desta forma exige a escrita de uma quantidade menor de código, mas ainda assim é necessário a escrita de código referente a criação e configuração de cada uma das regras usadas no sistema. A última abordagem, Estruturada Customizável, exige menos código do que as outras duas. Ela continua promovendo o reuso das regras de validação, e ainda permite que a criação e configuração das regras do sistema seja feita de forma automática baseada nas informações declaradas no arquivo XML.

Para finalizar, a última característica analisada é a Performance. A primeira abordagem não acrescenta nenhum problema de performance, já que não exige nenhum processamento além da execução do código da própria regra de validação. A Estruturada acrescenta um pequeno *overhead* devido a necessidade de criação de objetos que representam as regras de validação, além do registro e consulta dos mesmos na estrutura de armazenamento das regras (*HashMap* de Java). A abordagem Estruturada Customizável, além do *overhead* relativo à criação, registro e consultas dos objetos que representam às regras de validação, possui uma perda de performance (apenas no *startup* do sistema) relativa a interpretação do arquivo XML com as definições das regras de validação da aplicação.

4.3 Arquitetura dos componentes *Web*

Nesta seção iremos investigar um dos aspectos mais importantes para o desenvolvimento de sistemas *Web*. Este é o que primeiro deve ser avaliado durante o projeto do sistema pois influencia os demais aspectos. Este trata basicamente da distribuição dos componentes *Web* para atender aos diferentes tipos de requisição. Em outras palavras, ele tem como objetivo definir como serão feitas as associações entre requisições e componentes *Web* responsáveis pela execução das mesmas.

Vários são os modelos de associação de requisições a *Servlets*, por isso aqui tentamos identificar os mais comuns: monolítico, orientado a páginas, orientado a operações e baseado em eventos.

4.3.1 Modelo monolítico

O modelo monolítico é o mais simples de todos. Nele só existe um único componente *Web* que responde por todas as requisições feitas ao sistema. Esta abordagem é muito simples e usada em sistemas amadores e de pequeno porte. A Figura 4.4 ilustra este tipo de abordagem.

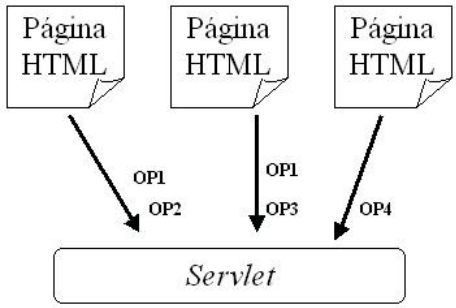

Figura 4.4: Modelo monolítico de estruturação de sistemas *Web*.

Apesar desta abordagem ser bastante simples, ela só deve ser aplicada em sistemas pequenos (com três ou quatro tipos de requisições diferentes). O uso dela em sistemas de maior escala não é recomendável por trazer diversos problemas:

- **Código extenso** – Como todas as requisições são executadas por um único *Servlet*, seu código fica muito extenso, o que implica em dificuldade de desenvolvimento e manutenção do mesmo.
- **Dificulta o trabalho em equipe** – Com o código definido em uma única classe, o trabalho em grupo torna-se difícil. Por isso, além de perdas em legibilidade, vai haver perdas significativas em produtividade.
- **Perda em performance** – Como um único componente *Web* trata todas as requisições, seu código se transforma em um amontoado de `if`s, cada um para um tipo de requisição. Dependendo da quantidade e da localização daquele que executa a operação requerida, o *servlet* pode ter um tempo de resposta ruim, mas em geral esta perda não é significativa.

4.3.2 Modelo orientado a operação

Nesta abordagem são definidos vários *servlets*, cada um para executar uma determinada operação do sistema. Cada tipo de requisição é executada por um único *servlet* e vice-versa. O que este faz é basicamente invocar as operações de negócios relativas a requisição, e por fim, montar a página de resposta para o cliente *Web*. Por exemplo, em um sistema bancário que possui operações de crédito, débito e saldo, são criados três *servlets*, um para cada operação (como pode ser visto na Figura 4.5).

Esta técnica resolve os problemas identificados com a utilização da abordagem monolítica, já que a lógica de tratamento das requisições fica modularizada (dividida entre os vários *servlets*), o que facilita a manutenção e desenvolvimento do sistema. Mas

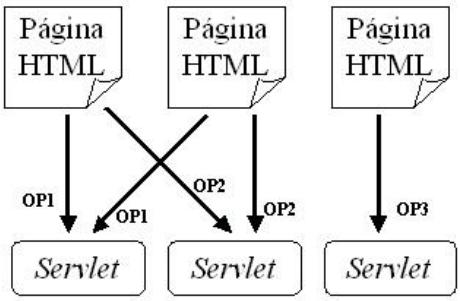

Figura 4.5: Modelo de estruturação orientado a operações.

ainda assim ela apresenta sérias limitações em algumas situações que são bem comuns em sistemas *Web*. A primeira destas é a duplicação de código para implementar sistemas onde diferentes operações precisam gerar como resposta a mesma página para o usuário. Nestes casos, a lógica de montagem da página é replicada em cada um dos *servlets* criados para executar estas operações. Um exemplo de sistema onde esta situação pode ser observada é ilustrada na Figura 4.6. Nela as operações de `credito`, `debito` e `login` retornam como resultado a página `Menu de Operações`, o que implica na criação de três *servlets* (um para cada operação), cada um contendo o trecho de montagem e formatação da página `Menu de Operações` replicadas em seus corpos.

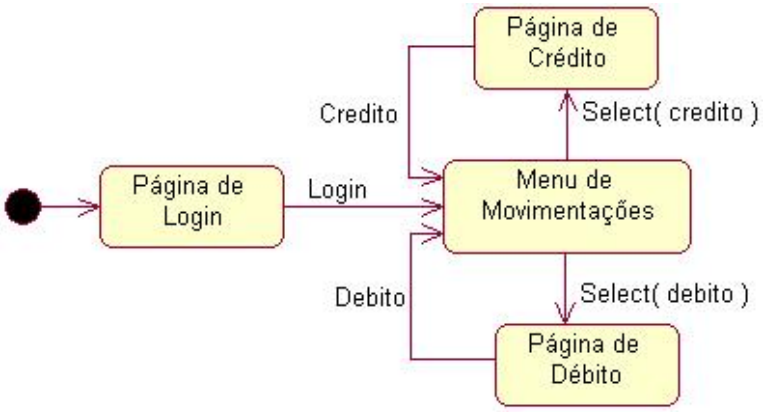

Figura 4.6: Exemplo de um sistema bancário.

Outra situação onde está técnica apresenta limitações é quando a execução de uma determinada operação pode resultar em respostas diferentes para o usuário. Esta situação também é bastante comum, um exemplo dela é a execução do `login` da Figura 4.6. Apesar de não estar ilustrado, a página de resposta resultado da execução desta operação pode ser duas: O `Menu de Operações`, quando o login for executado com sucesso, ou uma página de erro, caso o login do usuário falhe. Em situações como esta o uso desta abordagem pode fazer com que o código do *servlet* fique muito extenso, pois para implementá-la será necessário colocar uma seqüência de `ifs` (um para cada página de resposta possível) em seu corpo, como pode ser visto no trecho abaixo:

```
... //executa as operações de negócio relativas a operação
if(cond1){... // Monta a página de resposta 1}
else if(cond2){... // Monta a página de resposta 2}
...
else {...}
```

4.3.3 Modelo orientado a página

A estruturação orientada a página é muito parecida com a abordagem apresentada na Seção 4.3.2 (Modelo orientado a operações). A diferença básica está no fato de que existe um *servlet* para cada página do sistema, ao invés de um por operação (como pode ser visto na Figura 4.7). Nela o *servlet* que monta uma determinada página é responsável por receber todas as requisições originadas dela, executar a operação solicitada, e por fim, delegar a montagem da página de resposta para outro *servlet* (o responsável pela página).

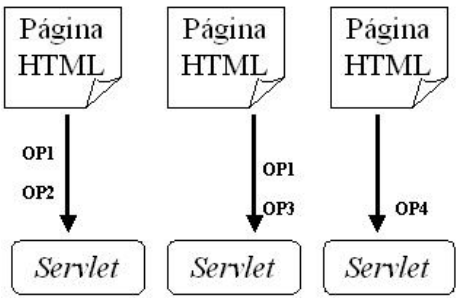

Figura 4.7: Modelo de estruturação orientado a páginas.

Por exemplo, para implementar o sistema bancário exemplificado na Figura 4.6, seria necessária a criação dos *servlets* `ServletPaginaLogin`, `ServletMenuOperacoes`, `ServletPaginaCredito` e `ServletPaginaDebito` (um para cada página do sistema). Em cada um destes deve estar o código relativo a montagem de suas respectivas páginas. O trecho de código abaixo dá uma idéia do código do `ServletCredito`.

```
...
operacao = request.getParameter("operacao");
if(operacao.equals("credito"){
   ... //executa as operações de negócio relativas ao crédito
   response.sendRedirect("ServletMenuOperacoes")
}
else{... // Monta a página de Credito }
```

Através do valor do parâmetro `operacao`, o *servlet* verifica se o crédito deve ser executado, caso contrário ele assume que a requisição é apenas para montar a página de crédito. Para requisições relativas a operação de crédito, ele executa as regras de negócio devidas e delega a formatação da página para o *servlet* responsável pela montagem do menu de operações (neste caso, o `ServletMenuOperacoes`).

Esta abordagem é bastante interessante e pode ser usada em muitas situações. Uma delas é para diminuir o número de *servlets* do sistema, já que o mesmo *servlet* pode executar várias operações. Por outro lado, esta característica apresenta algumas desvantagens:

- Não é possível evitar duplicação de código em situações onde a mesma operação pode ser executada a partir de diferentes páginas (situação que não é incomum em sistemas *Web*). Nestes casos sempre vai haver replicação do código da operação nos *servlets* responsáveis pela montagem destas páginas.

- Em situações onde várias operações podem ser executadas a partir da mesma página *Web*, o uso desta abordagem pode fazer com que o código do *servlet* fique muito extenso, pois será necessário colocar uma seqüência de `if`s (um para cada operação a ser executada) em seu corpo.

4.3.4 Modelo baseado em eventos

As estruturações orientadas a operações e páginas possuem algumas limitações para tratar situações que são bastantes comuns em sistemas *Web*. Por isso se faz necessário um modelo que preserve as vantagens de cada uma delas, mas seja capaz de resolver as limitações impostas por elas.

A idéia é separar a lógica de execução das operações e de montagem da página de resposta em entidades distintas. Este modelo é basicamente a utilização do padrão de projeto *Web handlers* (visto na Seção 2.2) ou sua implementação concreta: o *framework* apresentado na Seção 3.

Com o uso do modelo baseado em eventos, os problemas de duplicação e complexidade do código são resolvidos e, o que é melhor, ele pode ser usado em todas as situações relatadas anteriormente. Além do reuso de código, esta solução possui diversas vantagens: Grande flexibilidade na composição das partes de apresentação e processamento, mudanças nos mecanismos de montagem da apresentação não causam efeito nas entidades de processamento, e facilita a implementação de biblioteca de componentes.

Maiores detalhes sobre o funcionamento, vantagens e limitações podem ser vistos nas Seções 2.2 e 3.

4.3.5 Análise comparativa entre as abordagens

As quatro abordagens mostradas nesta seção são bastante usadas em sistemas *Web*. Cada uma delas possui vantagens e desvantagens, que dependendo do sistema ao qual elas são aplicadas podem ter maior ou menor relevância. Por este motivo é difícil apontar a melhor entre elas sem haver falhas na argumentação. Esta seção tem como objetivo explanar de forma mais objetiva e resumida as qualidades e problemas de cada uma das abordagens baseado em alguns critérios mais relevantes para aplicações desta natureza. Tendo conhecimento desta informações o desenvolvedor pode escolher a abordagem mais interessante para o sistema apenas analisando que características são mais relevantes no seu contexto.

A Tabela 4.3 mostra o resultado da análise das quatro abordagens, com base em cinco características importantes no desenvolvimento de sistemas. A primeira delas é a

Características	Abordagens			
	Monolítico	**Orientado à páginas**	**Orientado à operações**	**Baseado em eventos**
Legibilidade	⋆	⋆ ⋆	⋆ ⋆	⋆ ⋆ ⋆
Produtividade	⋆	⋆ ⋆	⋆ ⋆	⋆ ⋆ ⋆
Evita duplicação de código	⋆ ⋆	⋆	⋆	⋆ ⋆ ⋆
Performance	⋆	⋆ ⋆ ⋆	⋆ ⋆ ⋆	⋆ ⋆
Número de componentes *Web*	Um	#operações	#páginas	1 + #páginas + #operações

Tabela 4.3: Comparações entre as abordagens de arquitetura para sistemas *Web*.

legibilidade, que diz respeito a facilidade de compreender o código do componente *Web*, sem considerar aspectos tais como estilo de programação, complexidade das APIs utilizadas ou nível de dificuldade das regras de negócio aplicadas. A abordagem Monolítica tende a dificultar a legibilidade, enquanto que a baseada em eventos tende a facilitar. Os modelos orientado à páginas e operações tendem a facilitar a legibilidade, mas em algumas situações, já apresentadas nas Seções 4.3.2 e 4.3.3, podem tornar o código mais complexo.

A produtividade é outra característica importante no desenvolvimento de qualquer tipo de sistema. A única abordagem que tem problemas de produtividade é a Monolítica porque dificulta o trabalho em equipe.

A capacidade de facilitar o reuso também é muito importante, já que evita a duplicação de código e, conseqüentemente, facilita a manutenção e o desenvolvimento do sistema. A estruturação Monolítica evita a duplicação de código já que toda a lógica do sistema está em um único componente. As estruturações orientada à operações e páginas podem causar duplicação de código em algumas situações (ver detalhes nas Seções 4.3.2 e 4.3.3). A abordagem baseada em eventos facilita o reuso porque modulariza mais os componentes *Web* do sistema.

Outra característica analisada foi a performance. As abordagens orientada à páginas e à operações não causam muita perda de performance porque geralmente o seus processamentos são bem simplificados. A estruturação Monolítica deixa o código do componente *Web* muito extenso e com muitas escolhas (seqüências de `if`), por isso a performance pode ser afetada. Já o modelo baseado em eventos causa uma pequena perda de performance por causa da necessidade de processamento a mais para tratamento dos eventos.

Por fim, o número de componentes *Web* (mais especificamente, o número de classes) do sistema é outra característica a ser analisada, pois pode acrescentar uma dificuldade operacional que impacta na produtividade. O modelo Monolítico implementa todo o processamento das requisições em um único componente *Web*. O número de componentes necessários para implementar os modelos orientado à páginas e operações é igual ao número de operações e páginas do sistema, respectivamente. Já o modelo baseado em eventos tem o número de componentes igual a soma das operações e páginas de todo o sistema, acrescido de um (componente que faz o papel do FrontController).

4.4 Persistência do estado do cliente

Os *servlets* são programas em Java que residem em um servidor *Web* e executam operações em resposta as requisições HTTP. O protocolo HTTP é um protocolo cliente-servidor baseado em pedido-resposta [15]. Por questões de simplicidade, ele é *stateless*, ou seja, é um protocolo sem estado conversacional. Isto significa que não é possível o servidor saber (de forma natural, sem precisar de programação adicional) se um cliente já fez uma requisição anterior, qual o estado atual do cliente e outras informações em relação a história de solicitações de um determinado cliente.

A principal questão abordada nesta seção é como manter informações entre requisições do mesmo cliente em um ambiente que é *stateless*. Apesar da persistência de estado do cliente não ser uma coisa natural do protocolo HTTP, existem diversos artifícios que tornam o armazenamento do estado possível. Por isso é necessário realizar algumas comparações entre estas abordagens, tentando levantar os pontos negativos e positivos de cada uma, para se ter diretrizes sobre que alternativa escolher para o desenvolvimento do sistema *Web*.

Existem basicamente três formas de armazenar o estado de um cliente em aplicações *Web*. Algumas delas são muito simples pois requerem pouca programação e pouco entendimento do mecanismo de armazenamento, no entanto não são muito poderosas para armazenar qualquer tipo de estado. Outras são mais complexas em termos de programação, mas não apresentam limitação quanto ao tipo de estado que são capazes de armazenar. As abordagens principais são Uso de *hiddens*, Uso de *cookies* e Uso de sessões.

O mesmo exemplo, parte de um sistema bancário, foi implementado usando as três abordagens para que seja possível realizar uma comparação mais concreta destas. O diagrama de estados de UML [6] da Figura 4.8 especifica o comportamento deste exemplo. A primeira tela do sistema é responsável pelo login, onde o usuário deve informar o número da conta e senha. Após executar o login, ele vai para o menu de operações. Nesta tela ele pode selecionar uma das operações disponíveis (neste exemplo, apenas a de crédito) e ir direto para a tela da operação. Na página de crédito o usuário deve informar o valor a ser creditado. A resposta da execução do crédito é uma página de sucesso.

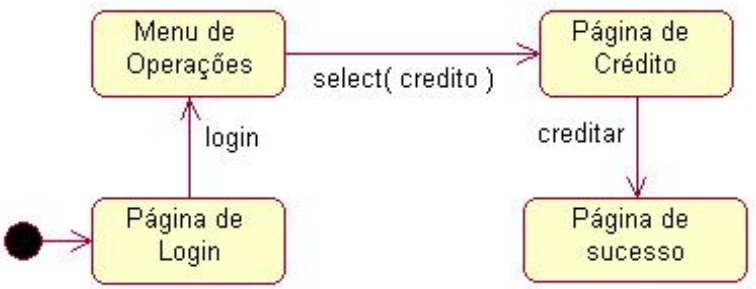

Figura 4.8: Exemplo do sistema bancário simplificado.

Por questões de simplicidade este sistema foi implementado usando a estruturação orientada a operações (apresentado na Seção 4.3.2). Para isto é necessário criar três *servlets*, cada um responsável por executar uma das operações. Os `ServletLogin`,

`ServletSelect` e `ServletCreditar` executam as operações `login`, `select` e `creditar`, respectivamente.

4.4.1 Usando hiddens

Hidden é um tipo de componente contido em formulários *Web* e, sendo elemento da linguagem HTML, não tem relação direta alguma com Java. Eles são usados para armazenar valores em uma página dinâmica enviada a um cliente *Web*, e possuem as mesmas características dos outros componentes de formulários HTML, tais como botões, entradas de texto, listas. Sua grande peculiaridade é que ele é um componente que não possui propriedades visuais, ou seja, não são visíveis ao cliente quando a página é mostrada pelo *browser*. No trecho abaixo pode-se conferir como é feita a definição de *hiddens* em HTML. Neste exemplo é definido um *hidden* chamado `conta`, cujo valor é `123-4`:

```
<form name="formTeste" action="ServletTeste">
  <input type="hidden" name="conta" value="123-4">
  <input type="text" name="valor">
  ...
</form>
```

Em requisições HTTP feitas através de formulários HTML, os valores contidos nos *hiddens* são enviados ao servidor como parâmetro da requisição, assim como os dos outros componentes de formulário. Os programas que vão manipular a requisição têm acesso a esses parâmetros e podem fazer o processamento baseado nos valores desses campos. O trecho abaixo exemplifica como um *servlet* pode acessar os parâmetros da requisição (*hiddens* e outros componentes):

```
String conta = request.getParameter("conta");
String valor = request.getParameter("valor");
```

O mecanismo de persistir o estado do cliente através do uso de *hiddens* é bastante simples. Os valores que fazem parte de seu estado, e que precisam persistir entre requisições, são definidos como *hiddens* na página dinâmica montada pelo *servlet* e enviada para o cliente. O *browser*, ao receber a página, apresenta-a sem mostrar qualquer vestígio dos *hiddens* (o usuário só pode ver estes valores se solicitar a visualização do código fonte ao *browser*). Quando o formulário desta página for submetido ao servidor, os *hiddens* são enviados de volta como parâmetros da requisição e o *servlet* que receber a requisição pode recuperar estes valores, fazer o processamento necessário para a resposta, inserir outros, ou os mesmos, *hiddens* na página de resposta e enviar a nova página de volta ao cliente. A Figura 4.9 dá uma idéia da utilização de *hiddens* para guardar o estado do cliente.

Implementação do exemplo

- `ServletLogin` – Recebe como parâmetro da requisição o número da conta e senha do usuário, faz a verificação destas informações e retorna o menu de operações como resposta. Nesta página de retorno é inserido um *hidden* chamado `conta` contendo o número da conta do usuário, o que deve identificar o cliente *Web*.

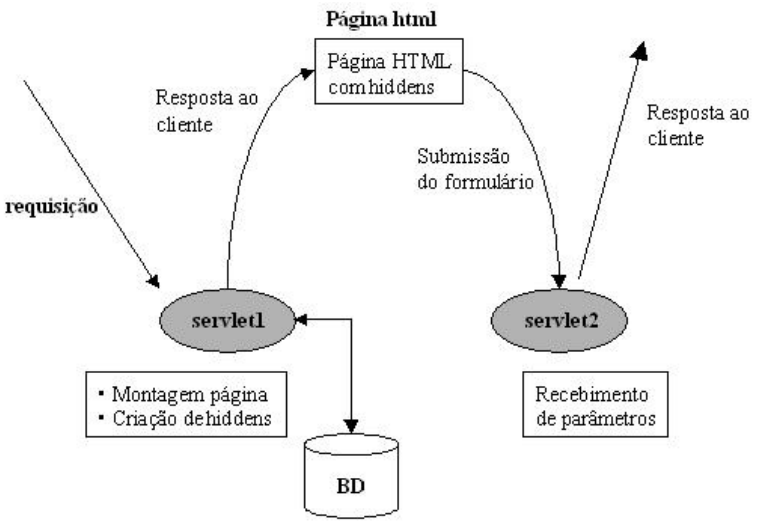

Figura 4.9: O mecanismo de persistir o estado do cliente através de *hiddens*.

- `ServletSelect` – Ele recebe o número da conta do usuário (armazenada anteriormente como *hidden*) como parâmetro e simplesmente monta a página de crédito, sem executar operação alguma, repassando a conta do usuário em um *hidden* na resposta.
- `ServletCreditar` – Recebe o número da conta e o valor, faz o crédito na conta e retorna uma página de sucesso.

Vantagens e desvantagens

As principais vantagens desta abordagem são

- **Simplicidade** – A implementação de persistência utilizando *hiddens* é muito simples e requer pouca programação pois a API de *servlets* já oferece uma abstração muito grande do mecanismo de recuperação de parâmetros. O desenvolvedor só precisa saber qual o nome dos parâmetros a recuperar e os métodos apropriados da API já realizam toda a tarefa para ele.
- **Não consome memória no servidor** – Todo o estado está armazenado na página do cliente, não existe consumo de memória no servidor para armazená-lo. Esta característica se torna ainda mais relevante em sistemas cujo número de acessos é muito grande, pois estamos trabalhando com sistemas cliente-servidor onde normalmente existe apenas um servidor para responder às solicitações e armazenar o estado de milhares ou milhões de usuários.
- **Suportado por qualquer *browser*** que suporte HTML padrão – Os *hiddens* são componentes da linguagem HTML, por isso qualquer *browser* em conformidade com os padrões da linguagem é capaz de suportá-los.

É possível identificar, também, algumas desvantagens deste mecanismo:

- **Só é possível o armazenamento de valores *strings*** – Os *hiddens* são elementos de HTML que são enviados e armazenados como *strings*, o que faz com que seja impossível armazenar outros tipos de objetos Java neles. Isto é um fator limitante para muitas aplicações, pois normalmente o estado que se deseja manter são objetos que não podem ser representados como simples *strings*. Um exemplo deste é o `java.util.Collection`[3]. Muitas aplicações fazem consultas a um banco de dados que retornam muitos registros, no entanto elas só querem mostrar uma parte destes registros cada vez que o cliente fizer uma nova solicitação. Uma forma de implementar isto seria fazer a consulta, mostrar parte dos dados e armazenar o `Collection` de forma que, na próxima requisição do mesmo cliente o *servlet* possa saber onde está armazenada a consulta dele e desta forma não ser necessário refazer a consulta no banco de dados. Em sistemas onde os objetos que necessitam manter-se persistentes entre requisições são objetos de negócio da aplicação, o que pode ser feito é armazenar apenas os identificadores (chaves) destes objetos nos *hiddens*. Desta forma, toda requisição que precisar de outras informações que não façam parte do identificador dos objetos tem que consultar o objeto no banco de dados novamente. Dependendo dos tipos de objetos e de suas dependências, estas consultas podem afetar fortemente a performance do sistema. Em sistemas onde o acoplamento entre os objetos é muito grande, ou eles possuem muitos atributos, esta abordagem não é uma boa escolha.

- **Dificuldade em armazenar estado composto** – Em casos onde o estado a ser armazenado é muito grande (no sentido de número de atributos que precisam ser armazenados), o que pode acontecer é que os *servlets* vão ter que manipular muitos parâmetros para compor este estado.

- **Os *hiddens* podem ser facilmente vistos pelo cliente** – Em alguns casos não é desejável que os valores que representam o estado dos clientes sejam vistos e conhecidos pelo próprio cliente. Isto permite que as informações manipuladas pela aplicação sejam conhecidas, o que facilita ataques de *crackers*.

- **As informações do estado do cliente trafegam pela rede** – Com o uso desta abordagem os dados que representam o estado do cliente trafegam pela rede do cliente ao servidor e vice-versa, o que é bastante perigoso para a segurança do cliente e do sistema. Em sistemas cuja segurança é essencial esta abordagem só deve ser utilizada em conjunto com uma infra-estrutura que garanta um canal de comunicação seguro entre cliente e servidor, como por exemplo https.

- **Persistência fraca das informações** – Um problema que acontece constantemente com a utilização desta solução é: mesmo quando algum dos parâmetros da requisição que representam o estado do cliente não é necessário para alguns *servlets*, eles precisam recuperá-los e montar a nova página de resposta inserindo todos estes valores de estado, pois o próximo *servlet* que receber uma requisição deste mesmo cliente pode precisar deles. Um exemplo deste fato é o que acontece com o `ServletSelect`. Ele não precisa executar operação alguma com o número

[3]Classe da API de Java que representa uma coleção de objetos e dispões de métodos para acesso aos elementos armazenados na mesma.

da conta do cliente, mesmo assim ele precisou recuperá-lo e montar uma nova página com esta informação.

4.4.2 Usando cookies

Os *cookies* também podem ser usados para persistir informações entre diferentes requisições do mesmo cliente. A maioria dos *browsers* suportam o uso de *cookies*, apenas *browsers* muito antigos não implementam este mecanismo.

Os *cookies* são informações que podem ser adicionadas à resposta ao cliente, assim como os *hiddens*; a única diferença é que eles não ficam na página HTML e não podem ser vistos de forma tão simples. Na verdade os *cookies* são enviados como informações de cabeçalho do protocolo HTTP e são armazenados em algum recurso de forma transparente pelo *browser* do cliente. O tipo de recurso que armazena os *cookies* depende da implementação do *browser*, não existe nenhuma especificação que obrigue-os a armazenar os *cookies* em recursos específicos. Os *cookies* trafegam pela rede, sempre que é feita uma requisição e sempre que é dada uma resposta. Eles são reenviados de forma automática, sem intervenção da aplicação, até que o seu tempo de vida expire.

Um *cookie* nada mais é do que um elemento que contém um nome e um valor e pode ser modificado sempre que for desejado pelo desenvolvedor da aplicação. Além de ter esta característica, os *cookies* contêm algumas informações adicionais que são úteis para o desenvolvedor da aplicação e para o *browser* que vai gerenciá-lo. Estes atributos são tempo de vida, se ele deve ser armazenado de forma segura, descrição do *cookie*, domínio ao qual ele pertence. Na API de *servlets* pode-se manipular os *cookies* através da classe `Cookie` [33]. Esta classe oferece diversos métodos para recuperar e modificar os atributos descritos acima.

Cookies são atribuídos pelo servidor, usando informações adicionadas ao cabeçalho de resposta HTTP. Na API de *servlets*, os *cookies* são salvos um por vez no cabeçalho de resposta, usando o método `addCookie` da classe `HttpServletResponse`. Os *cookies* também são passados de volta para o servidor através de campos adicionados ao cabeçalho da requisição. Nesta API, a recuperação de *cookies* pode ser feita usando o método `getCookies` da classe `HttpServletRequest`.

O trecho de código abaixo é um exemplo de adição de um *cookie*, chamado `conta`, com o valor `123-4` à resposta enviada ao cliente *Web*:

```
Cookie cookie;
cookie = new Cookie("conta","123-4");
response.addCookie(cookie);
```

A recuperação dos *cookies* enviados pelo *browser* ao servidor é feita através do método `getCookies` do `request`. Uma limitação desta API é que não existe uma forma de recuperar apenas um *cookie*, baseado no nome. Ao invés disto, o programa tem que recuperar o *array* com todos os *cookies* do cliente e percorrê-lo para encontrar o elemento com o nome desejado. No exemplo abaixo, é garantido que o cliente só possui um *cookie*, por isso pega-se o primeiro elemento do *array*:

```
Cookies[] cookies;
Cookie cookieConta;
```

```
String conta;
cookies = request.getCookies();
cookieConta = cookies[0];
conta = cookieConta.getValue();
```

Implementação do exemplo

- `ServletLogin` – Recebe como parâmetro da requisição o número da conta e senha do usuário, faz a verificação destas informações, adiciona um *cookie* com o número da conta à resposta e, por fim, retorna a página com o menu de operações.
- `ServletSelect` – Monta a página de crédito, sem executar operação alguma.
- `ServletCreditar` – Recebe o valor a ser creditado como parâmetro da requisição, e o número da conta como um *cookie*, faz o crédito na conta e retorna uma página de sucesso.

Vantagens e desvantagens

As principais vantagens do uso de *cookies* são

- **Uma vez armazenado o *cookie* pode ser mantido até que a aplicação deseje liberá-lo** – Uma das desvantagens do uso de *hiddens* é que a aplicação precisa ficar manipulando (com o objetivo de repassar) as informações do estado do cliente, mesmo sem precisar delas naquele momento. Com o uso de *cookies* não há a necessidade de ficar repassando o estado, pois uma vez armazenados na máquina do cliente eles só são removidos em duas situações: se a aplicação pedir explicitamente para removê-los ou se o tempo de vida deles expirar.
- **Persistência de estado por um longo período** – Uma característica interessante dos *cookies* é que eles podem persistir durante muito tempo, até mesmo quando o usuário sai da aplicação que os inseriu ou quando o navegador é fechado. Para controlar o tempo de vida do *cookie* na máquina do cliente, existe um atributo que pode ser modificado a qualquer momento. A classe `Cookie` permite a modificação desse atributo através da chamada ao método `setMaxAge`, que informa o tempo máximo de vida do *cookie*. Após o tempo de vida ter terminado, o *browser* pode removê-lo automaticamente a qualquer momento.
- **Não consome memória no servidor** – Os *cookies* são armazenados no cliente, por isso não existe consumo de memória no servidor para armazená-los. Apesar de não consumir memória no servidor existe um limite de *cookies* que podem ser armazenados na máquina do cliente (Esta limitação será melhor descrita nas desvantagens).

As desvantagens desta abordagem são

- **Só é possível o armazenamento de valores *strings*** – Os *cookies* são enviados e armazenados como *strings*. Assim como os *hiddens*, estes possuem os mesmos problemas relatados anteriormente.

- **Os valores dos *cookies* podem ser vistos pelo cliente** – Dependendo da configuração do *browser* do cliente, ele é alertado que a resposta da requisição está prestes a armazenar um *cookie* em sua máquina e neste alerta aparece o nome e o valor do *cookie*. Outra forma de visualizar informação de *cookies* é encontrando o recurso onde o *browser* armazena os *cookies* do cliente. Na maioria dos *browsers* estas informações são mantidas em arquivos texto e podem ser lidas por qualquer usuário.

- **A API não oferece métodos refinados para manipulação de *cookies*** – Como visto nos exemplos anteriores, o método que retorna os *cookies* do cliente para o *servlet* é o `getCookies` da classe `HttpServletRequest`. Este método tem como tipo de retorno um *array* de *cookies*. A API de *servlets* deveria oferecer métodos mais refinados que retornassem apenas um *cookie* específica, ou seja, oferecer algum método como o `getCookie(String name)` que retornaria apenas o *cookie* com um determinado nome. Com a API atual, o *servlet* que deseja consultar os valores dos *cookies* tem que percorrer o *array* procurando qual o *cookie* que tem determinado nome. O pior é que esta procura se repete em todo *servlet* que precisa trabalhar com *cookies*.

- **O usuário pode desabilitar o uso de *cookies*** – Vários *browsers* oferecem para o usuário a opção de negar que *cookies* sejam armazenados em suas máquinas. Por isso é comum a ocorrência de situações onde o *browser* do cliente não esta habilitado para o uso de *cookie* e a aplicação precisa armazenar *cookies* para executar todas as operações de forma correta.

- **Limitações de espaço para cookies** – Existem algumas limitações em relação a quantidade de *cookies* que podem ser armazenados na máquina do cliente e ao tamanho de cada um. A primeira limitação é que não é possível, para o mesmo servidor, armazenar mais de vinte *cookies* na máquina do cliente. A outra limitação é que cada *cookie* só pode ter no máximo um tamanho de quatro *kilobytes*. Apesar desta característica se apresentar como desvantagem, é muito difícil as aplicações necessitarem deste limite.

4.4.3 Usando sessões

O mecanismo de persistência do estado do cliente através do uso de sessões é muito poderoso. Cada sessão está diretamente ligada a um usuário, onde devem estar armazenadas todas as informações do seu estado. A sessão do usuário persiste por um período de tempo específico, através de várias requisições, e os valores armazenadas nela só podem ser manipulados pelo seu proprietário.

Em Java este gerenciamento de sessões é totalmente transparente para o desenvolvedor. Existem várias classes e métodos disponíveis na API de *servlets* que ajudam o desenvolvedor a manipulá-las. São poucas as operações que o programador precisa saber para desenvolver aplicações que usem sessões.

O objeto sessão nada mais é do que uma *hashtable* que contém o mapeamento entre identificadores e objetos. Estes mapeamentos podem ser adicionados, alterados, consultados ou removidos da sessão do usuário. A classe em Java que representa este conceito é a `HttpSession`, e os métodos para manipular as informações da sessão são

`setAttribute`, `getAttribute` e `removeAttribute` [4]. O trecho de código abaixo demonstra o uso de sessão para armazenar um objeto do tipo `Conta` da sessão do usuário:

```
Conta c = new Conta("123-4");
sessao.setAttribute("conta",c);
```

Da mesma forma, o trecho abaixo pode ser usado para recuperar o valor previamente armazenado na sessão do usuário:

```
Conta c = (Conta) sessao.getAttribute("conta");
```

Além dos mapeamentos, a sessão possui atributos como identificador, momento de criação, último acesso, período máximo que pode ficar inativo. Todos estes atributos podem ser recuperados por chamadas aos métodos `getXXX`, por exemplo.

Para recuperar a sessão do usuário, basta fazer uma chamada ao método `getSession` da classe `HttpServletRequest`. Este método retorna a sessão do usuário associado a requisição corrente. Um exemplo de recuperação de sessão de usuários pode ser visto no trecho abaixo, que implementa o método `service` de um *servlet* qualquer:

```
public void service(HttpServletRequest request,
                    HttpServletResponse response){
  HttpSession sessao;
  sessao = request.getSession();
  ...
}
```

O método `getSession`, sem parâmetros, cria uma nova sessão caso o usuário ainda não possua uma. Mas existe outra versão deste mesmo método que recebe um booleano como parâmetro, que indica se o mesmo deve ou não criar a sessão quando ela não existir.

As sessões são mantidas no servidor *Web*, no entanto é necessário que cada cliente *Web* guarde o identificador de sua sessão para que a mesma possa ser posteriormente recuperada. Na API de *servlets* esta informação pode ser armazenada no cliente de duas formas: através de *cookies* ou por reescrita de URL.

O mecanismo padrão é através do uso de *cookies*. A API automaticamente cria um *cookie* contendo o identificador da sessão e o envia para o cliente *Web* sempre que é criada uma nova sessão. Para recuperar a sessão do usuário, ela se baseia neste *cookie*, que é retornado em toda requisição deste usuário. Toda esta manipulação é feita internamente pelas classes da API, não havendo necessidade alguma de intervenção do programador.

O problema do mecanismo padrão, baseado em *cookies*, é que ele não funciona caso o cliente *Web* não suporte *cookies* ou esteja habilitado para não aceitá-los. Uma forma de contornar esta limitação de *cookies* é através do uso de reescrita de URLs. Aí é necessário adicionar uma informação extra, que identifique a sessão, a cada URL que referencia *servlets* no sistema. Por exemplo, na URL `http://www.algo.br/servlet/TesteSessao;jsessionid=123` é adicionada a informação de sessão `jsessionid` = `123`, a qual indica que o identificador da sessão do usuário que requisitar esta URL é `123`. A adição

[4] Os métodos `setValue`, `getValue` e `removeValue` estão depreciados desde da versão 2.2 da API de *servlets* [12].

desta informação pode ser feita automaticamente através de uma chamada ao método `encodeURL` da classe `HttpServletResponse`, passando-se a URL original. O trecho de código abaixo exemplifica este uso, onde a URL representada por `urlOriginal` é reescrita adicionando-se a informações da sessão e retornada para o cliente *Web* como um *link* HTML:

```
String urlReescrita = response.encodeURL(urlOriginal);
out.println("<A HREF=\"" + urlReescrita + "\">...</A>");
```

O mecanismo de reescrita é muito mais poderoso, pois pode ser usado em qualquer situação. No entanto, o uso deste requer que os *servlets* reescrevam constantemente as URLs do sistema.

Implementação do exemplo

- `ServletLogin` – Recebe como parâmetro da requisição o número da conta e senha do usuário, faz a verificação destas informações, cria uma nova sessão e adiciona o número da conta a esta, por fim, retorna a página com o menu de operações;
- `ServletSelect` – Monta a página de crédito, sem executar operação alguma.
- `ServletCreditar` – Recebe o valor a ser creditado como parâmetro da requisição e o número da conta recupera da sessão, faz o crédito na conta e retorna uma página de sucesso.

Vantagens e desvantagens

As vantagens de se usar sessões para persistir o estado do cliente são

- **As informações do estado do usuário não podem ser vistas pelo cliente *Web*** – O uso de sessões evita o problema que acontece com o uso *hiddens* e *cookies*, ou seja, do cliente *Web* poder visualizar os nomes e valores dos parâmetros que guardam o estado do usuário. Todas as informações referente ao usuário ficam armazenadas no servidor, a única que vai para o cliente é uma identificação da sessão para que o *Container* seja capaz de identificar cada cliente.
- **As informações do estado do usuário não trafegam pela rede** – Todas as informações armazenadas na sessão do usuário ficam residentes apenas no servidor e não há a necessidade delas serem enviadas a cada requisição e resposta, como é o caso dos *hiddens* e *cookies*. O tráfego de dados sigilosos de um lado para outro é perigoso para a segurança do cliente e do sistema, por isso esta abordagem aumenta a segurança do sistema e conseqüentemente sua qualidade.
- **Persistência de estado por um longo período** – Assim como os *cookies*, a sessão pode persistir durante muito tempo, não há a necessidade de rearmazenar o estado a cada requisição pois uma vez armazenado, os dados do usuário persistem enquanto durar sua sessão. O desenvolvedor não pode configurar o tempo de vida da sessão do usuário, no entanto ele pode definir o valor de *timeout* para estas, em outras palavras, o máximo de tempo as quais podem ficar inativas entre requisições.

Existem duas formas de configurar este tempo: fazendo uma chamada ao método `setMaxInactiveInterval(int)` ou através do arquivo de configurações[5] do *Servlet Container*.

- **Possibilidade de armazenar qualquer tipo de objeto como estado** – Isto torna esta abordagem muito poderosa pois qualquer tipo de estado pode ser armazenado na sessão, desde os mais simples (como *strings*) até os mais complexos (como estruturas inteiras de objetos). Isto, além de facilitar o desenvolvimento de sistemas, pode melhorar muito a performance destes, pois ao invés de armazenar identificadores de objetos e precisar construí-los em toda requisição, o sistema pode armazenar os objetos completos e recuperá-los diretamente.

As desvantagens do uso de sessões são

- **Consome memória no servidor** – Os objetos mantidos pelas sessões ficam armazenados no servidor, por isso há consumo de memória nele. Em sistemas com milhares de usuários simultâneos, ou quando o estado de cada cliente é muito grande, pode ocorrer o consumo excessivo de memória para armazenar as sessões de todos os usuários.

- **Não persistem após o fechamento do *browser*** – Apesar de ser possível manter a sessão sem ter a preocupação em ficar repassando-a, o estado não pode ser mantido depois que o usuário fecha o *browser*, como pode ser feito com o uso de *cookies*.

É recomendável usar a primeira abordagem para armazenar o identificador da sessão (uso de *cookies*), pois é bem mais simples de ser implementado pelo desenvolvedor. No entanto, se faz necessário ter como contingência a implementação de reescrita de URLs para que em situações onde os *coookies* não são suportados estas possam ser usadas.

4.4.4 Análise comparativa entre as abordagens

Três foram as abordagens para persistência do estado do cliente apresentadas: Uso de *hiddens*, de *cookies* e sessão. Foi dada uma idéia do funcionamento de cada uma delas, das vantagens e desvantagens do seu uso. Nesta seção é feita uma análise comparativa entre estas abordagens baseada em nove características que são importantes para sistemas *Web*. O resultado da análise pode ser visto na Tabela 4.4.

A primeira destas é a simplicidade de uso. O uso de *hiddens* é o mais simples de todos pois tudo é baseado em HTML e manipulação de *strings*. O uso de *cookies* e sessão requerem um pouco mais de conhecimento pois exige um conhecimento de alguns métodos e classes da API de *servlets*.

O armazenamento do estado do cliente *Web* exige um certo consumo de memória, que na maioria dos casos é bem pequeno. Quando este estado é armazenado do lado do cliente *Web*, não há muita preocupação com este consumo, já que cada cliente vai armazenar apenas o seu estado. No entanto, quando o estado é armazenado do lado servidor, é necessário ter uma atenção maior já que este terá que armazenar o estado de

[5]Cada *Servlet Container* possui sua própria forma de configurar este valor.

Características	Abordagens		
	Hiddens	***Cookies***	**Sessão**
Simplicidade	⋆ ⋆ ⋆	⋆ ⋆	⋆ ⋆
Consumo de memória no servidor	⋆ ⋆	⋆ ⋆	⋆
Garantia de persistência	⋆ ⋆ ⋆	⋆	⋆ ⋆
Tipos de objetos como estado	*strings*	*strings*	Qualquer
Persistência de estado composto	⋆	⋆	⋆ ⋆ ⋆
Tráfego de informações do estado pela rede	⋆	⋆	⋆ ⋆ ⋆
Período de persistência do estado	⋆	⋆ ⋆ ⋆	⋆ ⋆
Limitação do tamanho do estado a ser armazenado	Não existe	20 *cookies* de 4K cada	Memória do servidor
Privacidade do estado	⋆	⋆	⋆ ⋆ ⋆

Tabela 4.4: Comparações entre as abordagens de persistência do estado do cliente *Web*.

todos os clientes que realizam acessos ao sistema. Em sistemas que podem ter milhares de usuário fazendo acesso é possível que o consumo de memória para persistir o estado seja muito grande. As abordagens de *hiddens* e *cookies* não requerem esta atenção, já que armazenam o estado no cliente *Web*. O uso de sessão é que requer uma certa cautela pois todo o estado, de todos os clientes, fica armazenado no servidor.

O uso de *cookies* apresenta um problema em relação a garantia de armazenamento do estado, pois os clientes *Web* podem desabilitar o armazenamento de *cookies* e desta forma fazer com que a persistência do estado não funcione. O mecanismo de sessão dos *servlets* faz uso de *cookies* por *default*, por isso, nestes casos, se cliente desabilitar o uso de *cookies* o mecanismo não deve funcionar de forma adequada. No entanto o desenvolvedor pode usar reescrita de URLs para manipular a sessão e garantir que a persistência do estado funcione. O uso de *hiddens* não apresenta problemas deste tipo pois é puramente baseado em HTML, que é aceito por qualquer navegador *Web*.

Com o uso de *hiddens* e *cookies*, só é possível o armazenamento de informações do tipo *strings*. Outra limitação destas duas abordagens é que elas dificultam o armazenamento de estado composto, ou seja, estados que são compostos por várias informações. O uso de sessão não apresenta nenhuma destas limitações. Com ele é possível armazenar qualquer tipo de objeto Java.

Na Internet é preciso ter cuidado com as informações que trafegam pela rede. Por questões de segurança é desejado que o mínimo de informação possível trafegue pela *Web*. O uso de *hiddens* e *cookies* exige que as informações do estado do cliente trafeguem pela rede a cada requisição. Já o uso de sessões evita este tipo de problema, pois o estado do cliente fica armazenado no servidor. Este risco pode ser minimizado com o uso de protocolos de comunicação seguros, como HTTPS.

Como as informações dos *hiddens* ficam armazenadas diretamente na página HTML é necessário que a cada requisição o componente *Web* recoloque as informações do estado na página HTML de resposta. Isto significa que o período de vida do estado nesta abordagem é muito curto e que a aplicação precisa ficar sempre rearmazenando-a, o que requer código a mais no componente *Web*. As sessões persistem enquanto o usuário estiver acessando o sistema, sem necessidade de renovação periódica (também é possível

configurar o tempo de vida e o tempo de *timeout* da sessão). Os *cookies* são mais poderosos em relação a este aspecto, eles podem persistir durante um longo período, mesmo que o usuário saia do sistema ou até mesmo quando a máquina do cliente é desligada.

Outra característica que vale a pena analisar é a limitação do tamanho do estado que pode ser armazenado. Com o uso de *hiddens* não existe nenhuma limitação, já que as informações vão estar na página HTML e, teoricamente, ela pode ter qualquer tamanho. A abordagem de sessões está limitada à quantidade de memória no servidor alocada para executar o *container* de *servlets*. Já o uso de *cookies* possui uma limitação bem definida que é de vinte *cookies* de quatro *Kilobytes* cada.

Outro requisito de segurança é que não seja possível saber quais informações do estado do cliente o sistema persiste. Isto se dá ao fato de quanto mais aspectos da aplicação forem conhecidos, mais fácil será atacá-la. As informações contidas nos *hiddens* podem ser vistas facilmente mediante o código fonte da página visualizada no *browser*. Os *cookies* são armazenados em meios específicos da implementação do navegador e dependendo deste meio é possível visualizar os valores dos *cookies*. Já as informações armazenadas na sessão ficam apenas no servidor e não podem ser vistas pelos clientes.

Capítulo 5

Conclusões

Neste capítulo apresentamos as conclusões finais deste trabalho de dissertação, discutindo as principais contribuições, apresentando os trabalhos relacionados e sugerindo trabalhos futuros que permitam melhorar os resultados obtidos neste.

Este capítulo foi dividido em três seções. A primeira: Contribuições e Conclusões (Seção 5.1) detalha as contribuições e conclusões gerais do trabalho. A subseqüente: Trabalhos Relacionados (Seção 5.2), apresenta alguns trabalhos relacionados a este. Por fim, a Seção 5.3 (Trabalhos Futuros) detalha as possíveis extensões deste trabalho.

5.1 Contribuições e Conclusões

De fato, o desenvolvimento de sistemas *Web* é mais complexo do que se pensa, como já relatado no Capítulo 1. Inclusive o desenvolvimento da camada de apresentação destes tipos de aplicação geralmente é feito sem se seguir os bons princípios de qualidade de *software*, o que acarreta em dificuldades de desenvolvimento e manutenção. Por este motivo se faz necessário a criação de ferramentas que dêem um melhor suporte ao desenvolvimento desta camada, tais como guias e diretrizes de desenvolvimento, componentes de *software*, *frameworks* e padrões de projetos, etc.

As principais contribuições deste trabalho foram a documentação de alguns padrões de projeto, um *framework* para implementação e a criação de algumas diretrizes para desenvolvimento de sistemas *Web*. Detalhes destas contribuições são dados nas Seções 5.1.1, 5.1.2 e 5.1.3, respectivamente.

5.1.1 Padrões de Projeto para sistemas *Web*

No Capítulo 2 apresentamos quatro padrões de projetos para desenvolvimento da camada de apresentação de sistemas *Web*. Estes padrões de projeto foram documentados segundo formatos bem conhecidos na literatura de padrões [17, 7]. Eles foram identificados no desenvolvimento de vários sistemas e, de fato, cada um destes descreve uma solução geral para um problema recorrente no contexto de sistemas *Web*.

O primeiro deles, *Web Interceptor* (Seção 2.1), tem o objetivo de evitar a duplicação de código no início das operações de execução das requisições em sistemas *Web* estruturados de forma modular. Para isto é definido um componente intermediário (*Web Interceptor*) na comunicação entre o cliente (*browser*) e os componentes *Web*. Sendo assim todas as requisições passam pelo intermediário, onde a parte comum aos demais componentes é executada, e depois o mesmo delega a requisição para o componente responsável por ela.

O segundo é o *Web Handlers* (Seção 2.2), que é baseado na construção de entidades denominadas *handlers*, que podem ser de dois tipos: *Handlers* de Apresentação e *Handlers* de Processamento. Os primeiros contêm apenas código relativo à montagem das páginas dinâmicas e os outros possuem código (ou chamadas) relativo à execução da lógica de negócio. Com esta estruturação é possível evitar a duplicação de código e complexidade na estruturação de sistemas *Web* com relacionamento M:N entre a apresentação e o processamento.

O padrão de projeto *Web Compiler* (Seção 2.3) cujo problema que procura resolver é como desenvolver uma aplicação *Web* de forma a evitar que o *layout* das páginas HTML estejam misturadas com a lógica de execução das operações do sistema. A solução recorrente para este problema é separar o *layout* da apresentação (geralmente HTML) e o código de processamento da requisição (Java), deixando-os em entidades diferentes

e fazendo a ligação destes através de um componente específico, chamado aqui de *Web Compiler*.

Por último, o *Super Component* (Seção 2.4) descreve a solução para o problema inerente ao desenvolvimento de sistemas *Web* em Java: como evitar a duplicação de código de inicialização e destruição nos diversos componentes *Web* de um sistema? A solução se baseia no mecanismo de herança de linguagens Orientadas a Objetos (OO) [32, 5], criando um componente que contém apenas os métodos de inicialização e destruição (*Super Component*), com o código comum a todos os componentes *Web* do sistema. Os demais componentes *Web* devem estendê-lo e apenas especificar a operação para execução da requisição.

5.1.2 *Framework* para implementação de sistemas *Web*

No Capítulo 3, descrevemos o funcionamento do *framework* de implementação de sistemas *Web* que foi construído durante o desenvolvimento desta tese. Ele tem como objetivo melhorar a produtividade no desenvolvimento de sistemas *Web* permitindo um maior reuso e legibilidade dos componentes envolvidos na solução através do desacoplamento entre o código de processamento das requisições e o código de montagem de páginas. Mais especificamente ele pode ser visto como um *framework* de apoio a implementação do padrão de projeto *Web Handlers* que foi apresentado na Seção 2.2.

Vários são os benefícios do uso do *framework* para o desenvolvimento da camada de apresentação de sistemas *Web*, entre eles: grande flexibilidade na composição das partes de apresentação e processamento, maior reuso de código, facilita a implementação de sistemas que requerem diferentes formatos de saída para a mesma operação e facilita a implementação de componentes (*handlers* de apresentação e processamento) que podem ser reusados em outros sistemas.

O uso do *framework* também traz alguns problemas ao desenvolvimento de sistemas *Web*, tais como aumento do número de classes necessárias para implementar um sistema e acrescenta uma complexidade à implementação dos componentes *Web* devido a separação entre as partes de processamento e apresentação.

Apesar destas desvantagens, os benefícios alcançados com o uso do *framework* são suficientes para justificar sua adoção em sistemas de médio e grande porte. De fato, a experiência prática do uso deste *framework* no CESAR e Mobile tem mostrado que o mesmo é bastante eficaz para desenvolvimento de aplicações *Web*.

Por fim, o *framework* realmente consegue alcançar, de forma muito simples, o seu objetivo: facilitar o desenvolvimento de aplicações *Web*, aumentando a produtividade, o reuso e o desacoplamento entre o código de processamento das requisições e o código de montagem de páginas, sem acrescentar complexidade de entendimento, desenvolvimento e distribuição ao sistema.

5.1.3 Diretrizes e Técnicas para desenvolvimento de sistemas *Web*

No Capítulo 4 apresentamos várias abordagens para implementar características inerentes a sistemas *Web*, onde também são feitas comparações entre estas abordagens para avaliar quais delas devem ser aplicadas em determinadas situações. Este capítulo tem

como objetivo guiar as pessoas envolvidas no desenvolvimento de sistemas *Web* em Java a tomarem decisões que melhor se adequem aos requisitos do seu tipo de aplicação, já que a decisão de escolha de uma determinada solução em um projeto de *software* para *Web* em Java é uma tarefa bastante complicada.

As características inerentes a sistemas *Web* analisadas neste capítulo foram Formatação da apresentação (Seção 4.1), Validação de dados (Seção 4.2), Persistência do estado do cliente (Seção 4.4) e Arquitetura dos componentes *Web* (Seção 4.3). Ao final de cada uma destas seções é feita uma comparação qualitativa entre as diversas técnicas usadas para implementá-la.

A formatação da apresentação está relacionada a forma de montar a página de resposta ao cliente *Web*. O código de formatação da página em um componente *Web* é a parte que recebe as informações resultado da invocação dos métodos de negócio e, usando alguma técnica específica, compõe essas informações dinâmicas com a parte estática da resposta (que descreve apenas a aparência da página) para gerar a resposta para usuário. Várias foram as técnicas de formatação da apresentação apresentadas nesta seção, entre elas *Servlet* puro, Processamento de *templates*, Tratamento de elementos HTML como objetos Java, JSP puro, JSP com *View Helpers* e Uso de tecnologias baseadas em XML.

Já a validação de dados está relacionada a como realizar críticas sobre os dados que são informados pelos usuário. No desenvolvimento de sistemas é gasto muito tempo com a escrita de código de validação e muitas vezes este código fica duplicado em várias partes do sistema. Em sistemas *Web* este problema se torna ainda maior, já que estas regras devem ser replicadas tanto no servidor quanto no cliente. Três foram as técnicas para escrita de código de validação analisadas neste trabalho: Simplificada, Estruturada e Estruturada customizável.

Arquitetura dos componentes *Web* trata basicamente da distribuição dos componentes *Web* para atender aos diferentes tipos de requisição. Esta característica é uma das mais importantes para o desenvolvimento de sistemas *Web* e, geralmente, é a primeira a ser avaliada durante o projeto do sistema. Vários são os modelos de associação de requisições a *servlets*, neste trabalho tentamos identificar os mais comuns: monolítico, orientado a páginas, orientado a operações e baseado em eventos.

Por último, a persistência do estado do cliente trata o seguinte problema: Como armazenar informações relativas ao estado do cliente entre requisições, já que no protocolo HTTP não existe conceito de estado? Existem basicamente três formas de armazenar o estado de um cliente em aplicações *Web*. Algumas delas são muito simples pois requerem pouca programação e pouco entendimento do mecanismo de armazenamento, no entanto não são muito poderosas para armazenar qualquer tipo de estado. Outras são mais complexas em termos de programação, mas não apresentam limitação quanto ao tipo de estado que são capazes de armazenar. As abordagens principais para implementação da persistencia do estado de clientes *Web* são uso de *hiddens*, uso de *cookies* e uso de sessões.

5.2 Trabalhos Relacionados

Nesta seção apresentamos alguns trabalhos relacionados que também dão suporte ao desenvolvimento de sistemas *Web*. Alguns deles definem ferramentas que funcionam de forma semelhante às apresentadas, enquanto outros provêem um suporte complementar

a este trabalho. Também são apresentados os trabalhos que serviram como inspiração para alguns pontos desta dissertação. Os seguintes trabalhos, que têm alguma relação com este, são detalhados nas próximas seções.

- *Framework Struts* [51];
- J2EE *Patterns* [1];
- Técnicas de validação em Java [31, 47];
- Geração e Execução de Testes de Aceitação de Sistemas *Web* [2].

5.2.1 *Framework Struts*

O *Struts* é um *framework*, código aberto, criado por Craig R. McClanahan e incorporado a *Apache Software Foundation* (ASF) em 2000. O *framework Struts* é um dos subprojetos do projeto Jakarta [21] e tem como objetivo auxiliar a construção de aplicações *Web* com Java *Servlets* e *JavaServer Pages* (JSP).

O *Struts* dá suporte ao desenvolvimento de arquiteturas beseadas no padrão de arquitetura *Model-View-Controller* (MVC) [7], também conhecido como Modelo 2 no contexto de desenvolvimento na plataforma J2EE.

O *Struts* dá suporte a construção das três entidades do padrão MVC: *Model*, *View* e *Controller*. O *Model* é construído como JavaBeans [37] e é a entidade que representa os dados da aplicação. O *View* é implementado como páginas JSP comuns. O *Struts* define *tags* especiais, que podem ser usadas pelo desenvolvedor para auxiliar a construção da *View*. Para a construção do *Controller* já é oferecido a implementação de um componente (*Servlet*) que é responsável por receber as requisições, executar a ação associada a requisição, recuperar o modelo e executar a *View*. A ação é definida pelo desenvolvedor como uma classe (*Action class*) que encapsula a execução da requisição.

De fato, os *Struts* é muito similar ao *framework* apresentado neste trabalho. Ambos definem um componente *Controller* (implementa o padrão de projeto *Web Interceptor*) que funciona como ponto de entrada para os outros componentes do sistema. O *handler* de processamento possui características similares às *Actions classes*. O componente *View* no *Struts* tem que ser um JSP, enquanto que no *Web handlers* ele pode ser implementado como um JSP ou um *handler* de apresentação. A configuração do sistema nos dois *frameworks* é feita mediante o uso de um arquivo XML.

5.2.2 J2EE *Patterns*

J2EE *Patterns* é um catálogo de padrões para plataforma J2EE que foi inicialmente publicado no Java *Developer Connection* [35] em Março de 2000. Ele surgiu como uma idéia do *Sun Java Center* (SCJ), grupo de arquitetos da Sun que realiza trabalhos de consultoria na plataforma Java e J2EE em todo o mundo. Este grupo reconheceu a necessidade de documentar as soluções adotadas, já que na literatura não existia nada especifíco para este tipo de ambiente.

O catálogo de padrões J2EE engloba a documentação de padrões de projeto e arquitetura para *servlets*, JSPs e EJBs; também descreve algumas considerações de projetos, más práticas e técnicas de *refactorings* neste tipo de ambiente. Os padrões de projeto documentados neste trabalhos foram os seguintes:

Camada de Apresentação

- *Intercepting Filter* – Facilita o pré e pós-processamento de uma requisição.
- *Front Controller* – Provê um controle centralizado para gerenciamento do processamento das requisições.
- *View Helper* – Encapsula a lógica que não está relacionada a formatação da apresentação em componentes *Helpers*.
- *Composite View* – Cria uma agregação de apresentação a partir de componentes atômicos.
- *Service to Worker* – Combina o *Dispatcher View* em coordenação com o *Front Controller* e o *View Helper*.
- *Dispatcher View* – Funciona de forma similar ao *Service to Worker*, cedendo mais atividades ao processamento do *View*.

Camada de Negócio

- *Business Delegate* – Desacopla a apresentação dos serviços, e provê uma interface *Facade* e *Proxy* [17] para os serviços.
- *Value Object* – Usado para a passagem de informações entre camadas.
- *Session Facade* – Esconde a complexidade dos objetos de negócio e centraliza as solicitações aos serviços.
- *Composite Entity* – Define uma recomendação para construção de *entity beans*.
- *Value Object Assembler* – Constrói *Value Objects* compostos a partir de múltiplas fontes de dados.
- *Value List Handler* – Gerencia a execução de *queries*, *caching* e processamento de resultados.
- *Service Locator* – Esconde a complexidade da localização e criação dos serviços de negócio.

Integração com Camada de Dados

- *Data Acess Object* – Abstrai a fonte de dados e provê acesso transparente aos dados.
- *Service Activator* – Facilita o processamento assíncrono para componentes EJB.

5.2.3 *Técnicas de validação em Java*

Alguns trabalhos, que apresentam sugestões de técnicas de validação, serviram como base para categorização das técnicas de validação desta dissertação: Simplificada, Estruturada e Customizável (apresentadas nas Seções 4.2.1, 4.2.2 e 4.2.3, respectivamente).

Inclusive o *framework* de validação apresentado nas Seções 4.2.2 e 4.2.3 foi construído com o objetivo de melhorar as técnicas apresentadas nestes trabalhos relacionados, já que nenhum deles atendiam de forma aceitável todos os requisitos de validação desejados para sistemas *Web*.

5.2.4 Geração e Execução de Testes de Aceitação de Sistemas *Web*

A dissertação de mestrado apresentada em [2] tem como objetivo definir uma linguagem para descrição de casos de testes de aceitação com alto nível de abstração e reuso para sistemas *Web*. Além disto, neste trabalho, foram produzidos geradores automáticos de códigos que permitem gerar código de componentes *Web* baseado em informações dos casos de testes.

Os componentes *Web* parcialmente gerados por esta ferramenta seguem o padrão *Web handlers*. Inclusive o gerador de código também gera as configurações de serviços do sistema.

5.3 Trabalhos Futuros

O desenvolvimento de sistemas *Web* abrange um escopo muito grande, por isso não é possível em um único trabalho analisar todos os seus aspectos e facetas. Esta dissertação teve um escopo mais prático, voltado às fases de projeto e implementação de sistema *Web*. Por isso vários outros trabalhos, focados nas atividades de análise, testes, implantação, gerência de sistemas *Web*, podem ser desenvolvidos para complementar este.

Outros trabalhos focados nas atividades de projeto e implementação também podem ser desenvolvidos, com o objetivo de melhorar e estender esta dissertação. Nas Seções 5.3.1, 5.3.2 e 5.3.3 são apresentadas melhorias que podem ser feitas em cada uma das questões tratadas nesta tese.

5.3.1 Diretrizes para desenvolvimento de sistemas *Web*

Quatro características de sistemas *Web* foram investigadas no Capítulo 4: Formatação da apresentação (Seção 4.1), Validação de dados da requisição (Seção 4.2), Arquitetura dos componentes *Web* (Seção 4.3) e Persistência do estado do cliente (Seção 4.4). Existem outras características inerentes a sistemas *Web* que podem ser analisadas com objetivo de produzir diretrizes que apóiem o seu desenvolvimento:

- **Personalização** – Este é um requisito muito comum em aplicações *Web* e significa que as páginas apresentadas aos clientes *Web* podem ter características diferentes dependendo do tipo de usuário. Esta personalização pode ser de dois tipos: por perfil (estática) ou por usuário (dinâmica). No primeiro caso, existe um conjunto

de estilos de páginas cadastrados no sistema e cada usuário está associado a algum estilo deste. Alguns sistemas permitem que os usuários escolham qual destes estilos ele deseja para suas páginas do sistema, já outros não permitem esta flexibilidade e a associação é feita pelo administrador do sistema, baseada no perfil do usuário (por exemplo, gerente, funcionário, usuário comum). O segundo tipo de personalização é mais flexível e permite que o usuário possa configurar diversos aspectos da aparência de suas páginas no sistema, tais como tamanho e estilo de fonte, cor de pano de fundo, posição de certas informações.

- **Segurança** – Sistemas *Web*, geralmente estão publicados na Internet e podem ser acessados por usuários em todo mundo. Esta característica exige um controle maior da segurança da aplicação, tanto a segurança dos clientes quanto a segurança do sistema propriamente dito. A segurança de aplicações *Web* engloba funcionalidades tais como autenticação de usuários, autorização de acesso, sigilo de informações armazenadas e trafegadas pela rede.

- **Internacionalização** – Uma exigência que tem se tornado comum para aplicações é o suporte multilingue. Em sistemas *Web* este requisito ainda é mais exigido, já que muitas destas estão publicadas na Internet e conseqüentemente podem ser acessadas por pessoas em todo mundo.

A solução para validação de entrada de usuários proposta na Seção 4.2.3 é bastante interessante e pode ser usada em várias situações apresentando mais benefícios do que as demais. No entanto, a funcionalidade de configuração das regras através de fontes não compiláveis (arquivo XML) acrescenta um problema que geralmente só é identificado em tempo de execução (e nas demais podia ser identificado durante a compilação). Como os nomes e valores dos atributos são informados em uma fonte não compilável, não é possível realizar uma checagem de tipo nos valores dos atributos, nem uma checagem semântica dos nomes dos campos, durante o processo de compilação das classes. A solução para este problema seria a construção de uma verificador semântico para a definição de regras de validação. Desta forma, após o processo de compilação normal das classes Java, o desenvolvedor poderia usar o verificador para garantir que os nomes e tipos dos atributos informados estão corretos.

5.3.2 Padrões de Projetos

Os quatro padrões de projetos para sistemas *Web* catalogados neste trabalho (Seção 2) foram identificados durante o desenvolvimento de alguns sistemas *Web* desenvolvidos no CESAR. Após a análise e documentação destes padrões outras sistemas foram desenvolvidos, inclusive de porte bem maior do que os anteriores. Por isso, se faz necessário um trabalho de análise nestes novos sistemas, com o objetivo de identificar novos padrões de projeto *Web*.

5.3.3 *Framework* para sistemas *Web*

Com o uso do *framework* nos três projetos descritos na Seção 3.4, foi possível identificar uma série de novos requisitos que podem enriquecer ainda mais esta implementação.

Novas funcionalidades devem ser implementadas para atender as necessidades identificadas:

- Melhorar o tratamento de erros – Na versão atual os erros só podem ser tratados por *handlers* de apresentação, no entanto isto se mostrou insuficiente para alguns dos sistemas. Muitas vezes se faz necessário executar algum processamento com o erro ocorrido (por exemplo, gravar no *log*, trocar a exceção por outra, etc.) antes de mostrar a tela para usuário. Por isso, uma das melhorias possíveis é a associação de erros a serviços;

- Acrescentar informações que indiquem se o serviço suporta GET e POST – O desenvolvedor poderá acrescentar informações ao serviço sobre o suporte de requisições GET e POST diretamente no arquivo XML;

- Agrupar operações relacionadas – Uma variação da implementação para resolver o problema do aumento do número de classes do sistema é permitir que o desenvolvedor possa agrupar operações relacionadas ou semelhantes em um único *handler*. Com isso seria necessário implementar um mecanismo de execução de *handlers* mais elaborado, onde além de informações a respeito dos *handlers* a serem executados numa determinada requisição, deveria haver parâmetros indicando que operação executar em cada um deles;

- Permitir que um serviço possa ser definido como uma cadeia qualquer de *handlers* – Alguns processamentos são bastante complexos e muitas vezes parte deles é usada em outros processamentos do sistema. Para melhorar o reuso de código, é necessário permitir que se defina pequenos *handlers* de processamento e que estes possam ser compostos para serem responsáveis pelo processamento de um serviço. De forma mais genérica e flexível, é interessante que o *framework* suporte a definição de um serviço na forma de uma cadeia (ou composição) de *handlers* de qualquer tipo. Para isso, também é necessário a definição de uma linguagem (ou ferramenta) que facilite a definição das composições entre *handlers*;

- Permitir a customização de algumas propriedades do *framework* – Algumas propriedades que hoje estão fixas na implementação do *framework* poderão ser parametrizadas. O nome e o caminho do arquivo XML de configurações dos serviços poderá ser informado pelo desenvolvedor. Outra propriedade que poderá ser customizada é o nome do parâmetro *Web* que informa o serviço a ser executado pelo controlador;

- Verificador de tipos para a composição de *handlers* – Um dos principais objetivos do *framework* é facilitar o reuso dos componentes *Web*, permitindo que os *handlers* possam ser compostos de diferentes formas para atenderem requisições distintas. Na prática, a composição entre *handlers* só é possível se as informações de entrada exigidas por um deles for igual as de saída geradas pelo outro. Por isso, se faz necessário o desenvolvimento de uma ferramenta que verifique se as composições entre os *handlers* são compatíveis e não vão gerar erros em tempo de execução.

O *framework* apresentado nesta tese é bem focado na estruturação dos componentes *Web* e não abrange outras características inerentes a sistemas *Web*. Por isso, é possível

a definição de outros *frameworks* ou implementação de novas características neste, que auxiliem o desenvolvimento de sistemas *Web*, e facilite a implementação de questões tais como segurança, personalização, internacionalização e validação de dados do usuário.

Apêndice A

Esquema do Arquivo de Configuração do *Framework Web Handlers*

```
<?xml version="1.0" encoding="UTF-8" ?>

<!ELEMENT servidor ( tratamento_erro, servicos ) >

<!ELEMENT tratamento_erro ( throwable+ ) >
<!ATTLIST tratamento_erro default NMTOKEN #REQUIRED >

<!ELEMENT throwable EMPTY >
<!ATTLIST throwable nome NMTOKEN #REQUIRED >
<!ATTLIST throwable handler NMTOKEN #REQUIRED >

<!ELEMENT servicos ( servico+ ) >

<!ELEMENT servico ( processamento, apresentacao ) >
<!ATTLIST servico nome NMTOKEN #REQUIRED >

<!ELEMENT processamento EMPTY >
<!ATTLIST processamento nome NMTOKEN #REQUIRED >

<!ELEMENT apresentacao ( opcao* ) >
<!ATTLIST apresentacao default NMTOKEN #REQUIRED >

<!ELEMENT opcao EMPTY >
<!ATTLIST opcao handler NMTOKEN #REQUIRED >
<!ATTLIST opcao evento NMTOKEN #REQUIRED >
```

Apêndice B

Esquema do Arquivo de Configuração do *Framework* de Validação

```
<?xml version="1.0" encoding="UTF-8" ?>
<!ELEMENT name ( #PCDATA ) >

<!ELEMENT composition ( rule+ ) >
<!ATTLIST composition type NMTOKEN #REQUIRED >

<!ELEMENT description ( #PCDATA ) >

<!ELEMENT validation-rules-definition ( rule+, composite-rule ) >

<!ELEMENT value ( #PCDATA ) >

<!ELEMENT rule ( #PCDATA | description | parameter )* >
<!ATTLIST rule name NMTOKEN #IMPLIED >
<!ATTLIST rule class NMTOKEN #IMPLIED >

<!ELEMENT composite-rule ( description, composition ) >
<!ATTLIST composite-rule name NMTOKEN #REQUIRED >

<!ELEMENT parameter ( name, value+ ) >
```

Bibliografia

[1] Deepak Alur, John Crupi, and Dan Malks. *Core J2EE Patterns – Best Practices and Design Strategies*. Prentice Hall, March 2001.

[2] Eduardo Aranha. Geração e Execução de Testes de Aceitação de Sistemas *web*. Master's thesis, Centro de Informática – Universidade Federal de Pernambuco, Abril 2001.

[3] Joseph J. Bambara, Paul R. Allen, et al. *J2EE Unleashed*. Sams, first edition edition, 2002.

[4] Mark Birbeck et al. *Professional XML*. Wrox Press, second edition, May 2002.

[5] Grady Booch. *Object–Oriented Analysis and Design with Applications*. Benjamin/Cummings, second edition, 1994.

[6] Grady Booch, Ivar Jacobson, and James Rumbaugh. *Unified Modeling Language – User's Guide*. Addison–Wesley, 1999.

[7] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *A System of Patterns: Pattern–Oriented Software Architecture*. John Wiley & Sons, 1996.

[8] K. Cagle et al. *Professional XSL*. Wrox Press, first edition, May 2001.

[9] CESAR Centro de Estudos e Sistemas Avançados do Recife. Web site do CESAR. http://www.cesar.org.br.

[10] Bill Kennedy Chuck Musciano. *HTML & XHTML: The Definitive Guide*. O'Reilly & Associates, Inc., fourth edition, August 2000.

[11] Robin Cover. *WAP Wireless Markup Language Specification (WML)*. OASIS – Organization for the Advancement of Structured Information Standards, public review edition, August 2001.

[12] Danny Coward. Java Servlet Specification version 2.3. Disponível em http://java.sun.com/products/servlets, 17th Setember 1999.

[13] James Duncan Davidson and Danny Coward. Java Servlet Specification version 2.2. Disponível em http://java.sun.com/products/servlets, 17th December 2001.

[14] W3C Architecture Domain. Dom – Document Object Model. Disponível em http://www.w3.org/DOM.

[15] R. Fielding et al. *Hipertext Transfer Protocol – HTTP/1.1.* Network Working Group, rfc 2616, proposed standard edition, June 1999.

[16] David Flanagan. *JavaScript: The Definitive Guide.* O'Reilly & Associates, Inc., second edition, 1997.

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object–Oriented Software.* Addison–Wesley, 1994.

[18] Simson Garfinkel et al. *Web Security, Privacy & Commerce.* O'Reilly & Associates, Inc., second edition, 2001.

[19] Gin. Sistema de Gestão Interna. http://www.cesar.org.br.

[20] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification.* Addison–Wesley, second edition, 2000.

[21] Apache Group. The Jakarta Project Site. Disponível em http://jakarta.apache.org.

[22] OMG Object Management Group). OMS's CORBA website. Disponível em http://www.corba.org.

[23] Shishir Gundavaram. *CGI Programming on the World Wide Web.* O'Reilly & Associates, Inc., first edition, 1996.

[24] Marty Hall. *Core Servlets and JavaServer Pages.* Prentice Hall, 2000.

[25] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process.* Addison–Wesley, 1999.

[26] Budi Kurniawan. *Java for the Web with Servlets, JSP, and EJB: A Developer's Guide to J2EE Solutions.* New Riders, first edition, April 2002.

[27] Lattes. Sistema de Fomento Lattes. Disponível em http://www.cnpq.br/ servicos-restritos/.

[28] Listel. Portal Encontre e Compre. Disponível em http://www.listel.com.br.

[29] Web Macro. Web Macro home page. Disponível em http://www.webmacro.org.

[30] Tiago Massoni, Vander Alves, Sérgio Soares, and Paulo Borba. PDC: Persistent Data Collections pattern. In *First Latin American Conference on Pattern Languages Programming, Sugarloaf PLoP*, Rio de Janeiro, Brazil, 3th–5th October 2001. Submmited.

[31] Brett McLaughlin. Validation with Java and XML Schema, Part 1, 2, 3 and 4. Disponível em http://www.javaword.com, September 2000.

[32] Bertrand Meyer. *Object–Oriented Software Construction.* Prentice–Hall, second edition, 1997.

[33] Sun Microsystems. Java 2 platform, enterprise edition, v 1.3 api specification. Disponível em http://java.sun.com/j2ee/sdk_1.3/techdocs/api/.

[34] Sun Microsystems. Java 2 platform, standard edition, v1.2.2 api specification. Disponível em http://java.sun.com/products/jdk/1.2/docs/api.

[35] Sun Microsystems. Java Developer Connection web site. Disponível em http://developer.java.sun.com/developer/.

[36] Sun Microsystems. Java Remote Method Invocation (RMI). Disponível em http://java.sun.com/ products/jdk/1.2/docs/guide/rmi.

[37] Sun Microsystems. The javabeans 1.01 specification. Available at http://java.sun.com/products/javabeans/docs/beans.101.pdf.

[38] Sun Microsystems. Products & apis java. Disponível em http://java.sun.com/products/.

[39] Sun Microsystems. The Essentials of Filters. Version 1.1. Sun Microsystems. Disponível em http://java.sun.com/products/servlet/Filters.pdf.

[40] Sun Microsystems. The Java Community Process. Disponível em http://www.jcp.org/.

[41] Sun Microsystems. Reflection Api Documentation. Disponível em http://java.sun.com/j2se/1.3/docs/guide/reflection/spec/java-reflectionTOC.doc.html, 1998.

[42] Sun Microsystems. Enterprise java beans specification. Version 2.0, Final Release, 22th August 2001.

[43] Sun Microsystems. Javaserver pages specification version 1.2. Disponível em http://java.sun.com/products/jsp, 27th August 2001.

[44] NEWStorm. Newstorm *web site*. Disponível em http://www.newstorm.com.br.

[45] NEWStorm. Notitia *web site*. Disponível em http://www.notitia.com.br/.

[46] Hothouse Objects. Tags - HTML Toolkit for Java. Disponível em http://www.hothouseobjects.com.

[47] Victor Okunev. Validation with pure Java. Disponível em http://www.javaword.com, December 2000.

[48] PHP. Hypertext Preprocessor. Disponível em http://www.php.net/.

[49] Rani Pinchuk and Yonat Sharon. The skin pattern. *7th. Pattern Languages of Programs Conference*, August 2000.

[50] Roger Pressman. *Software Engineering–A Practioner's Approach*. McGraw-Hill, third edition edition, 1994.

[51] Apache Jakarta Project. Struts. Disponível em http://jakarta. apache.org/struts.

[52] Apache Jakarta Project. Velocity Template Engine. Disponível em http://jakarta.apache.org/velocity.

[53] Enhydra XMLC Project. Open Source Java/XML Presentation Compiler. Disponível em http://xmlc.enhydra.org.

[54] Prospectar. Sistema de Prospecção Tecnológica. Disponível em http://prospectar.cesar.org.br/admin.

[55] E. Rescorla. *HTTP Over TLS – HTTPS/1.1.* Network Working Group, rfc 2818 edition, May 2000.

[56] Mobile S.A. Mobile *web site*. Disponível em http://www.mobile.com.br/.

[57] Mobile S.A. Web-2-Billing – Web 2 Billing Pagamentos online. Disponível em http://www.web2billing.com.br.

[58] Bill Shannon. Java 2 Platform Enterprise Edition Specification, v1.3. Disponível em http://java. sun.com/products/j2ee, 21th August 2001.

[59] Bill Shannon. Java 2 Platform Enterprise Edition Specification, v1.3. Disponível em http://java.sun.com/j2ee, July 2001.

[60] Sérgio Soares. Desenvolvimento Progressivo de Programas Concorrentes Orientados a Objetos. Master's thesis, Centro de Informática – Universidade Federal de Pernambuco, Fevereiro 2001.

[61] SourceForge.net. Freemarker – an open-source html template engine for java servlets. Disponível em http://freemarker.sourceforge.net/.

[62] Inc. SPI Dynamics. Sql injection – are your web applications vulnerable? A White Paper from SPI Dynamics. Disponível em http://www.spidynamics.com/.

[63] SQLSecurity.com. SQL Security web site. Disponível em http://www.sqlsecurity.com.

[64] UNDP. Wide – Web of Infornation for Development. http://www.wide.org.br.

[65] Euricelia Viana and Paulo Borba. Integrando Java com Bancos de Dados Relacionais. In *III Simpósio Brasileiro de Linguagens de Programação*, pages 77–91, Porto Alegre, Brasil, 5–7 de Maio 1999.

[66] Larry Wall et al. *Programming Perl.* O'Reilly & Associates, Inc., third edition, 2000.

[67] A. Keyton Weissinger. *ASP in a Nutshell.* O'Reilly & Associates, Inc., second edition, 2000.