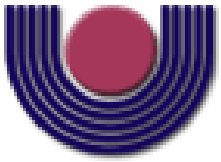

UNIOESTE – Universidade Estadual do Oeste do Paraná

CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS

Colegiado de Ciência da Computação

Curso de Bacharelado em Ciência da Computação

Adaptação dinâmica de componentes de interface Web em diferentes domínios de aplicação

André da Silva Queiróz

CASCAVEL

2011

ANDRÉ DA SILVA QUEIRÓZ

ADAPTAÇÃO DINÂMICA DE COMPONENTES DE INTERFACE WEB EM DIFERENTES DOMÍNIOS DE APLICAÇÃO

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação, do Centro de Ciências Exatas e Tecnológicas da Universidade Estadual do Oeste do Paraná - Campus de Cascavel

Orientador: Prof. Anibal Mantovani Diniz

CASCAVEL

2011

ANDRÉ DA SILVA QUEIRÓZ

ADAPTAÇÃO DINÂMICA DE COMPONENTES DE INTERFACE WEB EM DIFERENTES DOMÍNIOS DE APLICAÇÃO

Monografia apresentada como requisito parcial para obtenção do Título de *Bacharel em Ciência da Computação*, pela Universidade Estadual do Oeste do Paraná, Campus de Cascavel, aprovada pela Comissão formada pelos professores:

Prof. Anibal Mantovani Diniz (Orientador)
Colegiado de Ciência da Computação,
UNIOESTE

Prof. André Luiz Brun
Colegiado de Ciência da Computação,
UNIOESTE

Prof. Marcio Seiji Oyamada
Colegiado de Ciência da Computação,
UNIOESTE

Cascavel, 03 de Novembro de 2011.

DEDICATÓRIA

Este trabalho é dedicado à minha família e minha esposa, aos meus amigos, aos profissionais da computação, e a quem ler este texto.

AGRADECIMENTOS

Primeiramente gostaria de agradecer a Deus, pois, sem sua ajuda durante esta caminhada, nada disso seria possível.

Gostaria de agradecer à minha família que sempre apoio não somente nos meus estudos, mas na minha vida também. Agradeço então, à minha mãe Sueli de Fátima da Silva, ao meu pai Alceu de Oliveira Queiróz, e aos meus “brothers”, meus irmãos Adelar da Silva Queiróz e Joelison da Silva Queiróz, que sempre me apoiaram neste trabalho e em tudo de faço.

Agradeço a minha amada esposa Carolina Almeida Romani, que sempre esteve ao meu lado, me apoiando de todas as formas e ainda me ouvindo comentar de termos, que qualquer um fora da computação, chamaria de Grego. O resultado desse trabalho não seria possível sem a sua dedicação, amor e companheirismo.

Agradeço também ao meu orientador, é também grande amigo professor Anibal Mantovani Diniz, por sua disposição, dedicação, incentivo para a realização deste trabalho. Foi uma honra ser orientado por um dos profissionais mais éticos que eu conheço.

Agradeço aos professores do curso de Ciência da Computação por compartilharem seus conhecimentos para consolidar o profissional que sou hoje.

Agradeço aos meus amigos da faculdade, principalmente à Aline Vaplak, Anderson R. Slivisnki, Allan Bello, Allysson Carapeços, Carlos H. França, Cleiton Balansin, Diego R. Hachmann, Jean P. Varela, Lucas Batistussi, Lucas Inácio, Odair M. Souza, Tharle J. Camargo, Osmar dos Santos e Fábio K., Gustavo R. Krüger, que viveram comigo estes cinco anos de curso, de diversão, estudos, dedicação e finalmente a vitória. Agradeço também aos amigos que não continuaram o curso, mas que também foram muito importantes.

Agradeço também aos meus amigos Cassiano C. Casagrande, Claudir G. Junior, Rafael Voltolini e Thiago Rodrigues da OrbitSistemas pelo apoio e por todo o conhecimento passado, que foi muito útil para a realização deste trabalho.

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

AJAX	*Asynchronous JavaScript com XML*
AOCS	*Sistemas de Controle de Atitude e Órbita*
CBSE	*Engenharia de Software Baseada em Componentes*
CGI	*Common Gateway Interface*
CI	*Casos de Implementação*
COT	*Commercial off-the-shelf*
CRUD	*Create, Retrieve, Update e Delete*
CSS	*Cascading Style Sheets*
DBC	*Desenvolvimento Baseado em Componentes*
DTO	*Objeto de Transferência de Dados*
EJB	*Enterprise Java Beans*
HTTP	*HyperText Transfer Protocol*
JCP	*Java Community Process*
JNDI	*Java Naming and Directory Interface*
JSF	*JavaServer Faces*
JSP	*Java Server Pages*
JVM	*Java Virtual Machine*
MVC	*Model View Controller*
OSGI	*Open Services Gateway Initiative*
SGBD	*Sistema Gerenciador de Banco de Dados*
WIS	*Web-based Information Systems*

Sumário

Resumo

O desenvolvimento de sistemas para o ambiente Web, assim como os convencionais, esbarra em muitas dificuldades, onde a constante variação dos requisitos dos sistemas é uma das maiores. Os *frameworks* Web facilitam o desenvolvimento padronizado de aplicações Web, tornando-o mais produtivo. Dentre estes *frameworks*, destaca-se neste trabalho o JavaServer Faces (JSF), que é voltado à criação de páginas Web, e segue o princípio do Desenvolvimento Baseado em Componentes (DBC). O JSF fornece uma gama de componentes que podem ser reutilizados nas aplicações clientes. No entanto, o modelo de programação do JSF, que utiliza a composição de componentes nas páginas, gera um alto acoplamento entre a visão e a lógica da aplicação. Portanto, neste trabalho foi desenvolvido um estudo de caso que uniu os conceitos de *framelets* e a adaptação dinâmica de componentes JSF, para geração de relatórios reutilizáveis em diferentes domínios de aplicação. A utilização de *framelets* facilitou a construção da estrutura reutilizável, ao reduzir o acoplamento dos submódulos e com a distribuição de funcionalidade em vários *framelets*, diminuiu-se a complexidade de manutenção e evolução do módulo. O principal retorno deste trabalho está ligado à metodologia de reutilização elaborada no projeto da implementação, que poderá ser extendida para outros segmentos e tecnologias com os mesmos problemas do JSF.

Palavras-chave: Reutilização, JSF, Web, Desenvolvimento Baseado em Componentes (DBC), *Framelets*.

Capítulo 1

Introdução

É sabido, que a Internet trouxe muitos benefícios para a comunicação das pessoas, porém, as primeiras formas de transmissão de informação pela Internet eram puramente estáticas, ou seja, acesso a documentos específicos através de endereços pré-definidos.

No entanto, esta organização não passava de acesso a arquivos ou livros, de forma remota. Progressivamente, foram estudadas melhores maneiras de manipular essas informações. É nesse contexto, que surgiram as páginas dinâmicas.

Isso levou ao surgimento das aplicações Web, também chamadas de sistemas Web, que permitem uma maior interatividade com o usuário do navegador, uma vez que o conteúdo pode se adaptar dinamicamente às suas ações. Estas aplicações, convencionalmente trabalham sobre a arquitetura Web, isto é, são implantadas em servidores de aplicação, que recebem as requisições para a aplicação, e em última instância, retornam a resposta ao usuário.

A tecnologia Asynchronous Javascript com XML (AJAX) permite aos componentes de uma página Web, submeter de forma independente suas mudanças de estado ao servidor, e também, que o retorno dessas ações se reflita apenas nos segmentos desejados. Com isso, o usuário é beneficiado, pois, a diminuição do trafego de dados permite uma interação mais rápida e natural do sistema Web.

Sistemas Web, da mesma forma que os convencionais, objetivam atender os requisitos dos clientes, mas, diferentemente destes últimos, possuem maior complexidade. Sendo que, a demanda crescente por mudanças nesses sistemas, a diversidades de tecnologias e padrões utilizados, e as diversas necessidades presentes no ambiente Web, se destacam como as principais problemáticas nesse paradigma de programação.

Para apoiar o desenvolvimento de sistemas Web, tem-se o surgimento de várias linguagens e processos de software. Porém, de pouco adianta ter à disposição esses elementos se o desenvolvimento ocorre de forma desorganizada. Desse modo, o *framework* é uma estrutura

capaz de padronizar o desenvolvimento, uma vez que atende a um domínio de aplicação específico, mas seguindo padrões de projeto, e consolidando sua infra-estrutura com maior confiabilidade.

Assim, os desenvolvedores tem uma gama de tecnologias à disposição, primeiramente o Common Gateway Interface (CGI), e posteriormente outras, como a linguagem PHP, a plataformas Java e .NET, entre tantas oferecidas pelo mercado.

Além disso, outra grande ferramenta para acelerar a produtividade de desenvolvimento de software é a programação baseada em *frameworks*, que centralizam numa única estrutura vários artefatos voltados para um domínio específico de aplicação. Entre os vários disponíveis para o desenvolvimento de sistemas WEB utilizando a plataforma Java, encontram-se o Spring (SPRING, 2011), o Struts (STRUTS, 2011), o Google Web Toolkit (GWT) (GWT, 2011) e o JavaServer Faces (JSF) principalmente.

Nestes *frameworks*, ocorre o chamado Desenvolvimento Baseado em Componentes (DBC), onde são construídas as páginas Web através da composição dos componentes gráficos providos pelos *frameworks*.

O *framework* tratado neste trabalho é o JSF, pois, segundo Teixeira (2008), dentre estes *frameworks* citados acima, o que possui mais pontos positivos nos critérios avaliados, é o JSF. Para que se possam utilizar os componentes da tecnologia JSF, os programadores inserem em meio ao código HTML as *tags* dos componentes, *tags* estas que possuem atributos que se ligam diretamente com a lógica da aplicação.

Esse modelo de programação com componentes é praticado em outros *frameworks* e outras plataformas, além do JSF. No entanto, quando se desenvolve dessa forma uma estrutura complexa para uma dada aplicação e, deseja-se reutilizá-la noutra aplicação, a forma como é feita a integração entre a lógica da aplicação e a interface do usuário pode dificultar ou mesmo inviabilizar a reutilização. Uma vez que, como afirma Lobo Filho (2010), o JSF possui alto acoplamento entre a apresentação, ou seja, seus componentes de interface, e a lógica da aplicação.

Este trabalho propôs um estudo de caso que envolveu a construção de um módulo de relatório reutilizável, denominado MRR, que através de adaptações dinâmicas de componentes do *framework* JSF buscou-se a reutilização em diferentes domínios de aplicação. A arquitetura deste projeto foi baseada em pequenas estruturas, denominadas *framelets*, que semelhantes aos *frameworks* fornecem serviços sobre um segmento específico.

A definição de uma metodologia de reutilização envolvendo o projeto e elaboração de artefatos que utilizem *framelets* e possam ser extendidos para outros contextos e tecnologias, foi considerada a principal expectativa deste trabalho.

1.1 Objetivos

1.1.1 Objetivo Geral

Construir uma estrutura para a geração dinâmica de interfaces Web e adaptação de regras de negócio focadas na reutilização de componentes prontos em diferentes domínios de aplicação.

1.1.2 Objetivos Específicos

a) revisão bibliográfica sobre a internet e sistemas WEB;
b) revisão bibliográfica sobre tecnologias e arquitetura WEB;
c) revisão bibliográfica sobre padrões de projeto, frameworks e engenharia de software baseada em componentes (CBSE);
d) revisão bibliográfica sobre reutilização de software;
e) propor solução para reutilização de regras de geração de interfaces com o uso de *framelets*;
f) implementar estudo de caso com a utilização de *framelets* aplicados sobre o framework JSF.
g) analisar os resultados sobre o critério de reutilização através de *framelets*.

1.2 Estrutura do trabalho

Neste primeiro capítulo foi apresentado a problema alvo, a justificativa, e os objetivos deste trabalho. Esta monografia esta estruturada da seguinte maneira:

a) no capítulo 1 são apresentados a introdução, objetivos do trabalho e a estrutura do trabalho;

b) no capítulo 2 são apresentados conceitos relacionados aos sistemas Web, desde o modelo cliente/servidor e características das aplicações Web à containers Web, no 7.1Apêndice A são encontradas mais informação das tecnologias Java para Web;

c) no capítulo 3 são citados vários conceitos relacionados ao desenvolvimento de sistemas, como: padrões de projetos, engenharia de software baseada em componentes, *frameworks* e *framelets*. Além disso, são apresentados os fundamentos da reutilização de software, elencando suas características principais, vantagens e desvantagens;

d) no capítulo 4 é feito um estudo detalhado do *framework* JSF, contextualizando seus elementos principais, o funcionamento de seu ciclo de vida e conceituando AJAX e como esta tecnologia se relaciona ao JSF atualmente;

e) no capítulo 5 são descritas as características do estudo de caso, bem como a metodologia de reutilização aplicação no seu desenvolvimento. Neste capítulo também se encontram os requisitos funcionais do módulo, a descrição dos diagramas de casos de uso e de componentes e a descrição de cada *framelet* sendo relacionados às funções de suas classes;

f) no capítulo 6 são encontradas análises e avaliações do estudo de caso sobre aspectos como: facilidade de uso, compatibilidade, manutenabilidade, vantagens e limitações, discussões sobre o uso de *framelets*, dentre outros;

g) no capítulo 7 estão presentes as discussões em relação ao trabalho e seus principais resultados comparando o uso de *framelets* em outros contextos e evidenciando os principais retornos desse trabalho.

Capítulo 2

Sistemas WEB

Assim que surgiu, a Internet era uma rede de comunicação muito simples, composta basicamente por conteúdo em páginas estáticas, tais como textos, imagens e outros elementos. A princípio o usuário não tinha nenhuma interação dinâmica com o servidor, ou seja, o conteúdo não se alterava com relação às preferências do usuário ou conforme as suas escolhas naquela página. Verificou-se que a mudança da forma deste serviço poderia trazer mais interatividade e robustez às aplicações. Com isso, houve alterações no modelo cliente/servidor em uso até então, para que as aplicações que inicialmente executavam somente do lado cliente, migrassem para o lado servidor (TEIXEIRA, 2008).

Nesse contexto, surgiram muitos recursos para prover maior interatividade, entre elas estão: a tecnologia CGI, a linguagem PHP e as plataformas Java e .NET. Tais recursos possibilitaram que as requisições do cliente, ao invés de gerar recursos estáticos, gerassem recursos dinâmicos adaptadas a diferentes situações (LOBO FILHO, 2010).

Assim, com a crescente utilização da linguagem Java para o desenvolvimento de sistemas corporativos, a JCP[1] viu a necessidade de que a plataforma Java permitisse o desenvolvimento de sistemas Web, assim como as necessidades de serviços de alto nível citadas na JSR 32[2], que são: gestão de ciclo de vida, sessões de armazenamento e recuperação de dados, segurança, mapeamento, contexto e configuração de dados.

Nesta época, e seguindo este princípio, foram desenvolvidas tecnologias para a plataforma Java, objetivando o desenvolvimento de aplicações Java para Web. Foram criadas então três tecnologias Java: Applet, Servlet e o Java Server Pages (JSP). Applets são pequenas aplicações clientes, que executam na máquina virtual Java instalada no navegador do usuário. Provavelmente o cliente necessitará de um Plug-in Java, e pode ser necessário um certificado

[1] JCP (Java Community Process – a JCP é um mecanismo padrão, livre para entrada e participação de qualquer membro, que define especificações técnicas para a tecnologia Java (JCP).
[2] JSR 32 – a JSR (Java Specification Requests) 32 define uma API de uso geral que é destinada ao processamento de baixo nível de aplicações sob a Internet em clientes, bem como servidores (JSR 32).

digital para que o *applet* execute com sucesso (ORACLE, 2001). O Servlet é basicamente uma classe Java, que recebe um objeto de requisição, executa certo processamento e envia uma resposta adequada.

Já o Java Server Pages (JSP) é uma extensão da tecnologia Servlet e tem como objetivo facilitar o desenvolvimento de páginas, separando as camadas de negócio e a de apresentação (SANTOS, 2007). Por último, foi também incorporada a tecnologia Java Server Faces (JSF) como uma extensão dos Servlets e JSP. Esta tecnologia traz maior produtividade ao desenvolvimento, pois ao implementar o padrão de projeto Model View Controller (MVC), a camada de visão e camada de controle são separadas.

Com isso, a utilização de padrões de projeto, como o MVC no framework JSF, simplificou muitas tarefas relacionadas a problemas constantes no desenvolvimento de interfaces gráficas para aplicações Web (CASTILHO, 2007). No entanto, segundo Grott (2003), um dos desafios dos frameworks para desenvolvimento Web, é ser uma solução genérica e, ao mesmo tempo permitir flexibilidade e reusabilidade do código-fonte.

2.1 Modelo cliente/servidor na Web

Na arquitetura cliente/servidor, a comunicação se baseia em requisições. Um ou mais, clientes requisitam recursos e/ou serviços a um processo servidor. Este último, por sua vez, executa certo processamento e devolve ao cliente o recurso solicitado**.**

Riccioni (2000) e Larman (2000) afirmam que, o modelo cliente/servidor divide os segmentos das aplicações em: acesso a banco de dados no lado servidor; apresentação no lado cliente e as regras de negócio no servidor e/ou cliente (GROTT, 2003).

Existem muitas variações do modelo cliente/servidor. Grott (2003), cita como as principais:

a) *fat client* (“cliente gordo”) e *thin server* (“servidor magro”);
b) *thin client* (“cliente magro”) e *fat server* (“servidor gordo”);
c) *fat client* e *fat server*.

Para este trabalho será considerado o modelo “cliente magro e servidor gordo”, pois o *framework* JSF trabalha dessa forma, concentrando a lógica de negócio no servidor. Nessa abordagem, o usuário não tem acesso a processamentos críticos/restritos do sistema, e o servidor sendo uma máquina mais robusta, contribui para a *performance* da aplicação.

A arquitetura cliente/servidor foi escolhida para operar na Web, onde o lado cliente é um aplicativo específico, browser ou navegador, que requisita informações ao servidor por meio do protocolo HyperText Transfer Protocol (HTTP), e apresenta a resposta ao usuário. Já o lado servidor, responde com serviços ou dados, às requisições de diversos clientes ou servidores feitas a ele (WINCKLER e PIMENTA, 2002).

Segundo Winckler e Pimenta (2002), o atual sucesso da Web é devido a dois fatores, principalmente: a "arquitetura simples, mas eficiente e uma interface igualmente simples, originalmente baseada no paradigma de hipertextos". Os autores, afirmam também que o modelo cliente/servidor pode também ser chamado de *request/response (pedido/resposta)*, visto que a comunicação necessária para requisição de dados ou outras operações no servidor, envolve um pedido (browser) e uma resposta esperada para este pedido (servidor Web).

A Web foi projetada de forma a atuar em diferentes ambientes e plataformas, ou seja, nem o lado cliente e nem o lado servidor se preocupam com esses elementos. Cada ponta está apenas interessada em executar a sua tarefa, seja ela de requisição ou de resposta (KUEHNE, 2007).

Nesta seção, será elucidado como o modelo cliente/servidor funciona, e como ele se insere no contexto das aplicações Web. Além disso, pretende-se demonstrar a forma de navegação na Web, em contraste ao modelo de aplicações *desktop*.

2.1.1 Funcionamento do Modelo Cliente/Servidor

Como citado anteriormente, a forma de comunicação de aplicações que atuam sobre o modelo cliente/servidor é diferente de aplicações *desktop*, onde o acesso a métodos, dados e serviços, geralmente são requisitados a um processo distante do seu contexto.

O modelo cliente/servidor é muito abrangente, podendo ser aplicado não somente às aplicações Web. Logo, desse ponto em diante, será citado o modelo, como sendo aplicado no contexto de sistemas Web.

Na Web o cliente pode estar utilizando diferentes plataformas para interação com servidores. Como a comunicação acontece através de protocolos iguais, não é necessário que ambos os lados tenham hardware semelhantes, ou seja, as mudanças de plataforma ou hardware são imperceptíveis à execução.

Além disso, como citado anteriormente, o tipo de interação com o servidor, pode gerar dois tipos de dados, estáticos ou dinâmicos.

Para tornar os sistemas baseados na Web dinâmicos, em 1995 surge a tecnologia *Common Gateway Interface* (CGI), que permitiu uma maior interatividade com o servidor, com a execução de aplicações neste, a partir do navegador Web (YEAGER e MCGRATH, 1996 apud (TEIXEIRA, 2004)).

A portabilidade ainda é garantida, visto que a comunicação entre o navegador e o servidor Web continua sendo pelo formato HTML. O CGI permite que o servidor tenha o recurso para acessar aplicativos, e que fique aguardando requisições de acesso aos aplicativos. O servidor interpreta requisições HTML com os pedidos de execução de aplicação, estas aplicações podem ir de simples serviços como email ou FTP, a Sistemas Gerenciadores de Banco de Dados (SGBDs) ou sistemas complexos (TEIXEIRA, 2004).

Com a tecnologia CGI, as páginas continuam estáticas no servidor, sob o ponto de vista do usuário. Entretanto, o pedido feito através do navegador Web ao servidor, executará uma aplicação para gerar a página dinamicamente. Logo, o CGI nada mais é, do que é um protocolo de comunicação que diz ao servidor qual aplicativo chamar para a geração de páginas dinâmicas.

Na Figura 2.1 é ilustrado o funcionamento do modelo cliente/servidor em páginas estáticas, considerando apenas solicitações de páginas HTML.

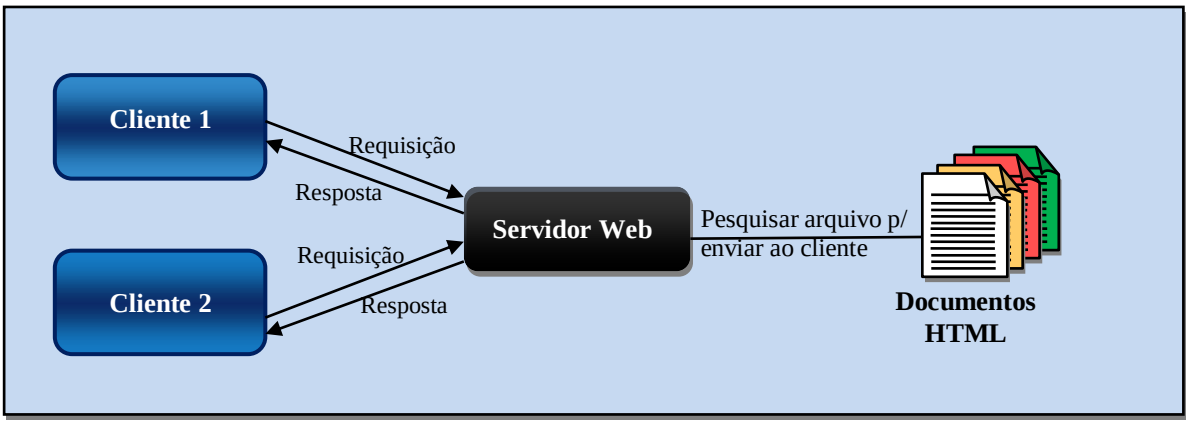

Figura 2.1: Interações cliente servidor no modelo de páginas estáticas

Embora a CGI servisse como primeira solução para a geração de páginas dinâmicas, não demorou a surgirem novas demandas e que consequentemente, problemas fossem identificados. Segundo Orfali et al. (1999), CGI, é um protocolo *stateless*, assim, por padrão, não permite manter estado no servidor. Além disso, os autores afirmam que, existe o sobrecarregamento do servidor decorrente das muitas solicitações para criação de processos novos.

A Figura 2.2 mostra outros recursos que auxiliam no processo de geração de páginas dinâmicas. Pode-se observar que os dados obtidos ainda podem ser estáticos, mas há a possibilidade de retornar como resultado, a mescla de páginas dinâmicas e estáticas ou então, *download* de arquivos gerados segundo parâmetros da requisição, por exemplo.

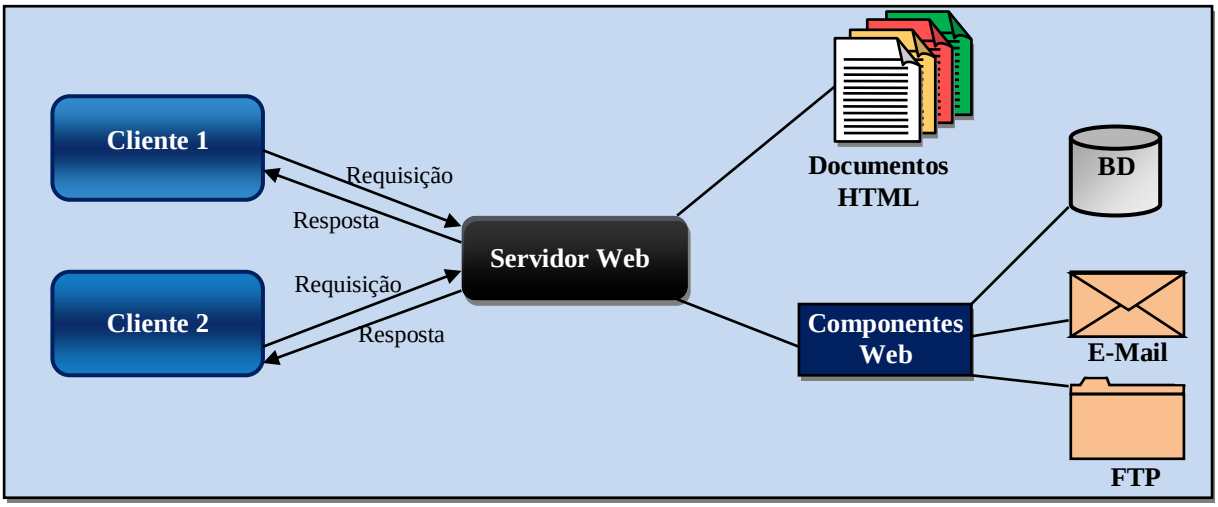

Figura 2.2: Modelo Cliente/Servidor de Aplicações Web Dinâmicas (adaptado de Mário Teixeira (2004))

As deficiências encontradas na tecnologia CGI dão uma visão de como seria o funcionamento das novas tecnologias para páginas dinâmicas, isso inclui: manutenção do estado no servidor, serviços compartilhados no servidor e maior interatividade com o cliente.

Embora existam distinções entre páginas estáticas e dinâmicas, vale salientar que, no desenvolvimento de aplicações desta natureza existe uma mescla de informações dinâmicas incorporadas em informações estáticas.

2.1.2 Servidores Web versus Servidores de Aplicação

Servidores Web podem ser definidos como programas de computador com a finalidade de receber requisições HTTP e devolver respostas HTTP com o conteúdo esperado pelos clientes, no caso, os navegadores ou outros programas que utilizam a especificação. Ainda que servidores Web tenham diferenças técnicas, em todos o aspecto comum é o uso do protocolo HTTP nas requisições e respostas (KUEHNE, 2007).

Atualmente o termo servidor Web é utilizado de uma maneira genérica, porém nesta seção serão indicadas algumas diferenças entre servidores Web, como o servidor Apache HTTP, e os servidores de aplicação, como o IIS (Microsoft Internet Information Services), Tomcat, Glassfish e JBoss.

Na grande maioria das vezes, as definições de servidor Web e servidor de aplicação são usadas sem distinção. Contudo, servidores de aplicação são muito mais robustos, e em sua maioria contém um servidor Web internamente.

No contexto de servidor de aplicação tem-se o termo *middleware*. Sistemas categorizados como *Middleware*, fornecem infra-estrutura para que aplicações corporativas utilizem seus serviços de uma forma abstrata. Logo, estes sistemas são mediadores da aplicação com outras aplicações ou o próprio sistema operacional. Além disso, *middlewares* permitem que o desenvolvimento seja focado nas regras de negócio, e não na comunicação e integração das aplicações.

Portanto, servidores de aplicação são *middlewares*, pois dão suporte à execução de aplicações, e respondem por elementos gerais da maioria das aplicações, como a segurança e a disponibilidade. Na sequencia estão as principais características encontradas nos servidores de aplicação (GARTNER, 2011):

a) Abstrai das aplicações a tarefa de balanceamento de carga;
b) Disponibilidade para atuar em modo usuário ou em background (daemon);
c) Aperfeiçoa o uso de recursos através do compartilhamento com as aplicações;
d) Podem agregar componentes e arbitrar o acesso aos mesmos;
e) Integra os componentes com os recursos da plataforma de execução;
f) Recursos comuns, como conexão com banco de dados, gerenciamento de transações, segurança e tolerância a falhas, são disponibilizados como serviços para as aplicações.

Ainda segundo Gartner (2011), de acordo com um levantamento feito em 2005 para verificar quais os servidores de aplicação disponíveis, entre os cincos produtos mais utilizados, estavam quatro que seguiam a especificação J2EE. Assim, grande parte da comunidade de TI, está convencida de que servidores de aplicação são servidores Java EE.

Entretanto, há uma necessidade eminente de alargar novamente a definição de servidores de aplicação, com o advento da computação em nuvem e o do Open Services Gateway Initiative (OSGI)[3]. Trazendo a necessidade de redefinir estes servidores como: qualquer sistema que forneça serviços para os desenvolvedores, e que possuem as características citadas acima (OTTINGER, 2008).

[3] A plataforma OSGI prove um Framework núcleo e serviços de plataforma. O framework suporte em tempo de execução para executar e gerenciar o ciclo de vida de várias aplicações em um ambiente seguro e modularizado. O Framework tem quatro camadas: segurança, modularização, ciclo de vida e serviço (CHAPPELL e KAND, 2009).

2.1.2.1 Containers

Conforme citado anteriormente, em aplicações tradicionais tudo que ocorre no sistema deve ser previsto e tratado pelos desenvolvedores. Porém, em sistemas distribuídos esta tarefa se torna muito mais complexa e crítica, e para evitar a ocorrência ou propagação de falhas é necessário delegar certas tarefas a um sistema específico, o container (*contêiner*).

> *Os contêineres são a interface entre um componente e a funcionalidade específica da plataforma de nível baixo (sic) que dá suporte ao componente. Antes que um componente Web, enterprise bean ou cliente da aplicação possa ser executado, ele deve ser montado em uma aplicação J2EE e implantado no seu contêiner.* (FERLIN, 2004, pg. 21).

A plataforma Java EE oferece dois modelos de contêineres:

a) container *Web* – dá suporte para a execução de aplicações nas tecnologias Servlet, JSP, JSF, dentre outras.
b) container EJB – voltado para aplicações corporativas. Dá suporte à execução de aplicações que usam a tecnologia EJB (*Enterprise Java Beans*).

Não é foco deste trabalho, citar containers EJB, neste sentido, o termo *contêiner* ou container, fará referência a contêiners Web no decorrer do texto.

2.2 Aplicações Web

Juntamente com o surgimento de novas linguagens e tecnologias Web, surge o conceito de aplicações Web. Conforme Lobo Filho (2010) são aplicações que usam a Web como suporte para sua execução, seja através de uma rede, Internet ou Intranet, ou por meio de execução local. Além disso, qualquer software que utilize um navegador Web como base para sua apresentação, é considerado uma aplicação Web.

Faz-se necessário nesta seção, diferenciar aplicação Web e aplicação na Web. Uma aplicação na Web é qualquer aplicação que utilize a Web como suporte para sua execução, como exemplo, um sistema de gerenciamento de arquivos, que utilize a arquitetura Web para suas operações. Por outro lado, uma aplicação Web, essencialmente implementa o paradigma hipermídia (ligações associativas entre documentos) (JACYNTHO, 2008).

Para Conallem (2000), as aplicações Web são divididas em dois grupos, são eles: Sites Web e sistemas Web. Sites Web foram criados por Tim Bernes-Lee (QUADROS, 2002). São sistemas distribuídos de hipermídia, com o objetivo de disponibilizar acesso a documentos e outros dados distribuídos em computadores pela Internet (WINCKLER e PIMENTA, 2002).

Alguns exemplos segundo Ginige e Murugesan (2001) são: "*websites* informativos, interativos, aplicações transacionais ou baseadas em fluxo que executam na Web, ambientes colaborativos de trabalhos, comunidades online e portais".

Nesse tipo de aplicação, as informações são interligadas através de um componente essencial para a estrutura navegacional da Web, o *hiperlink* (SOUZA, 2007).

Um *hiperlink*, nada mais é que uma referência para uma página Web ou então uma ação, que é enviada ao servidor para processamento. A primeira definição, é a base dos Sites Web, pela sua estrutura estaticamente definida, já a segunda, é como os sistemas Web trabalham, ou seja, as regras de navegação são e estão definidas na aplicação hospedada no servidor.

2.2.1 Sistemas Web

Sistemas Web são sistemas que permitem aos usuários a execução de lógica de negócio utilizando um navegador Web. Assim, esse tipo de sistema é capaz de gerar através das interações do usuário, páginas dinâmicas para operações em banco de dados (WINCKLER e PIMENTA, 2002).

Souza (2007) cita os chamados Sistemas de Informação Baseados na Web (*Web-based Information Systems* – WISs), que são equivalentes aos sistemas tradicionais, todavia disponíveis na Web, ou Intranet. É esta categoria de aplicação Web que este trabalho focará desse ponto em diante.

Sobre alguns aspectos do desenvolvimento de sistemas, sistemas Web são muito semelhantes aos sistemas tradicionais. Contudo, muitas das tarefas ditas simples para um sistema tradicional, geram muita complexidade quando a mesma é levada ao contexto Web.

Destaca-se em sistemas Web, uma grande heterogeneidade da equipe de trabalho, pois demanda grande diversidade de conhecimentos e grande instabilidade dos requisitos do sistema (JACYNTHO, 2008). Ainda, esses sistemas são muito complexos, uma vez que, existe a composição de muitos recursos, com variedade de: tecnologias e padrões de projeto; e pelas necessidades que são inerentes ao ambiente Web, como: segurança, interfaces claras e objetivas, rapidez e compatibilidade.

Nesse contexto de desenvolvimento, é necessário um grande planejamento para garantir que ao final de um projeto, que se tenha expansibilidade, compatibilidade, qualidade e manutenabilidade.

2.2.2 Tecnologias para interfaces Web

A princípio, as páginas Web eram essencialmente páginas com texto e imagens interligadas através de *hiperlinks*. No entanto, os recursos suportados pela linguagem HTML, não eram suficientes para aplicações com diferentes demandas e complexidades.

Ao mesmo tempo que o ambiente Web trazia muitos benefícios para o desenvolvimento e para a utilização por parte do usuário, haviam barreiras para a evolução das aplicações advindo da simplicidade do HTML 1.0 (WINCKLER e PIMENTA, 2002).

Surge então as tecnologias com maior impacto na personalização e especificação de interfaces Web, o CGI, CSS (Cascading Style Sheets) e JavaScript (WINCKLER e PIMENTA, 2002). Essas tecnologias quebraram muitos paradigmas da programação Web, como trazer recursos para páginas Web que permitiram que estas se comportassem como uma aplicação tradicional.

Além da interatividade e robustez proporcionada pela linguagem JavaScript, as interfaces que eram limitadas à estilos básicos e "enxutos", agora podem contar com a formatação e personalização dos componentes visuais, através da linguagem de estilos CSS.

Nesta seção, serão apresentadas as tecnologias CSS e JavaScript, visto que, as mesmas serão utilizadas amplamente no estudo de caso, e de modo geral, grande parte das aplicações Web de hoje as utiliza.

2.2.2.1 CSS (Cascading Style Sheets)

O CSS (Cascading Style Sheets) é um recurso para adicionar estilos (p. ex, fontes, cores, espaçamento) em documentos Web (CSS). Segundo Winckler e Pimenta (2002), o CSS provê meios de personalizar a apresentação de páginas Web através de um conjunto de comandos de formatação e especificação de propriedades tipográficas, dentre outras. O CSS é vinculado a documentos HTML para alterar a apresentação, sem adicionar comandos específicos. Sendo que, o mesmo autor destaca isso como a principal vantagem do CSS, pois garante a independência de plataforma.

A primeira versão do CSS, CSS 1.0, é de 1996, e trouxe avanços ao HTML 1.0 que utilizava comandos explícitos em suas *tags* para configurar o estilo das páginas. O CSS em sua versão 2.1, já é padrão para mudança de estilos em páginas HTML e suas extensões. Através do atributo "*style*" dos comandos da linguagem HTML é possível definir quais atributos se deseja aplicar em determinado(s) componente(s). Além disso, existe o atributo

"*class*", que permite reutilização de estilos, que podem estar ou não no mesmo documento onde é chamado.

2.2.2.2 JavaScript

A linguagem JavaScript foi desenvolvida pela empresa Netscape em 1995, que objetivava tornar os sistemas Web mais interativos (Goodman (2001) (apud Marafon (2006)). Hoje a mesma está padronizada e difundida nas implementações dos navegadores atuais, sendo a linguagem base para processamentos no lado cliente em aplicações Web (*client side scripting*) (LOBO FILHO, 2010).

Na sequência Marafon (2006) define algumas características da linguagem JavaScript:

a) variáveis com tipagem dinâmica;
b) linguagem interpretada;
c) funções JavaScript são executadas no Browser cliente;
d) suporte a gerenciamento de eventos.

A principal desvantagem está na manutenção dos sistemas, pois parte da lógica da aplicação fica na apresentação do sistema, espaço inadequado para a lógica de negócio (RIBEIRO et al., 2006).

O JavaScript que está contido numa página HTML, é interpretado pelo browser para que sejam realizadas as mudanças devidas, admitindo aplicar dinamismo em componentes HTML (WINCKLER e PIMENTA, 2002).

Esta seção buscou conceituar e caracterizar sistemas (aplicações) Web e como tecnologias como a linguagem JavaScript e o CSS, dão suporte ao desenvolvimentos destes. Seguindo esta idéia, a seção 7.1A.1 do 7.1Apêndice A , tratará dos Servlets. Os Servlets são outra tecnologia disponível para o desenvolvimento de sistemas Web, mas, que se destacou por unir a este paradigma a linguagem Java para o desenvolvimento.

Capítulo 3

Padrões de Projetos, Engenharia de Software Baseada em Componentes (CBSE), Reutilização, Frameworks e Framelets

Neste capítulo serão tratados assuntos relacionados com a problemática do trabalho, no contexto de engenharia de software. Primeiramente a seção de padrões de projeto, visa mostrar as características, classificações, contextualização no ambiente JSF e as desvantagens de seu uso.

A seção de Engenharia de Software Baseada em Componentes (CBSE), tratará da programação com foco em componentes, ao invés de classes. Tendo a base sobre os componentes, a seção de Reutilização, irá tratar da reutilização de artefatos de software e principalmente componentes no desenvolvimento de software.

Na seção *frameworks*, serão citadas as características, vantagens e desvantagens dos *frameworks*, elencando assim os elementos que consolidaram seu sucesso no desenvolvimento de sistemas.

Por fim, a seção *framelets*, visa elucidar as características dessa estrutura, suas diferenças em relação aos *frameworks* e sua composição, para que fique claro como esta poderá ser utilizada para a solução proposta da problemática deste trabalho.

3.1 Padrões de Projeto

Os padrões surgem com uma linguagem comum entre os desenvolvedores, visando um melhor entendimento da comunicação de entidades e indivíduos. Devido à existência de padrões com as mais diversas finalidades, os mesmos estão em crescente uso no desenvolvimento de sistemas. Logo, se a eficiência de uma solução foi comprovada num dado projeto, pode-se reutilizar esse esforço em outros projetos com a mesma garantia (GROTT, 2003).

Padrões relacionam problemas e soluções. Com os padrões pode-se documentar um problema recorrente e sua solução num contexto característico, para expor esses dados. O objetivo do padrão é promover reutilização no decorrer do tempo (BOOCH e SCIENTIST, 2003).

Segundo Gamma et al. (1995) padrões de projeto podem ser vistos como soluções que deram certo, ou seja, foram comprovadas na prática, e posteriormente difundidas como uma boa prática. Tais soluções têm apoio de especialistas, de modo a trazer o máximo de benefícios com sua adoção.

Portanto, essa abordagem segundo Sommerville (2007) pode ser considerada, reutilização de projetos mais abstratos, onde não há inclusão de detalhes da implementação. De acordo com o mesmo autor, os padrões de projeto esbarram em elementos como: características do objeto, herança e polimorfismo, para garantir generalidade. Todavia, sua aplicação deve ser ocorrer independentemente do contexto aplicado.

Na sequência são apresentadas algumas características presentes em padrões (BOOCH e SCIENTIST, 2003):

a) são observados através da experiência;
b) normalmente são documentados de maneira estruturada;
c) evita desperdício com "reinvenção da roda";
d) presentes em diferentes níveis de abstração;
e) estão em contínua evolução;
f) são artefatos reutilizáveis;
g) relacionam as melhores práticas aos projetos;
h) podem se unir a outros padrões para solucionar um problema maior.

Um padrão de projeto é composto por quatro elementos essenciais de acordo com Gamma et al. (1995):

a) um **nome** que seja suficiente para descrever de maneira sucinta o contexto e aplicação do padrão. Conforme afirma Grott (2003), a nomeação de padrões fornece uma linguagem comum entre desenvolvedores, facilitando o projeto com alto nível de abstração;
b) uma descrição do **problema** que expõem em que situação o padrão se aplica e qual finalidade no contexto. Pode incluir descrição de problemas específicos. Além

disso, há certas restrições que devem ser obedecidas para que a aplicação do padrão seja correta (FERLIN, 2004);

c) uma **solução** que descreva como o problema pode ser resolvido, com base nos relacionamentos e responsabilidades dos elementos no projeto. Portanto, não se tem uma solução concreta, mas sim, um modelo abstrato geralmente ilustrado graficamente, que serve de base para a solução efetiva;

d) as **consequências** apresentam os resultados e conciliações de se aplicar o padrão. Isso apoio a tomada de decisão dos projetistas, mensurando se o padrão pode ou não ser aplicado numa situação especifica.

3.1.1 Classificação de Padrões de Projeto

Padrões de software geralmente são classificados, em padrões de: análise, arquitetura, construção e codificação. A diferença entre estas, está no nível de abstração.

Padrões são classificados em cinco categorias através de sua aplicação, são elas:

a) interface;
b) responsabilidade;
c) construção;
d) operação;
e) extensão.

Neste trabalho é feita referência a padrões de construção, ou seja, padrões que atuam em nível de classes e objetos.

Os padrões de construção, segundo Ahmed e Umrysh (2002) podem ser divididos nas seguintes categorias:

a) criação – padrões voltados à configuração e inicialização de objetos. Por exemplo, o padrão *singleton* (única instância de um objeto);
b) estrutural – padrões responsáveis pela estruturação de interfaces e o relacionamento entre suas classes concretas;
c) comportamental – padrões que determinam um comportamento específico em um conjunto de objetos.

3.1.2 Padrões Relacionados à Tecnologia JSF

A implementação JSF contém vários padrões. Nesta seção são descritos sucintamente como cada padrão aparece no desenvolvimento (ou mesmo na implementação) JSF;

3.1.2.1 Padrões de Interface

Padrões de Interface estão relacionados ao conceito de interface, que por sua vez, representa a abstração de um conjunto de métodos e atributos. É um artefato que concede acesso de objetos de outras classes aos métodos de sua classe concreta (METSKER, 2004).

Na sequência estão os padrões de interface que o JSF utiliza.

3.1.2.1.1 Composite

Composite como sua própria tradução sugere, é uma composição de objetos no qual objetos podem conter outros objetos. Desse modo, existem os compostos, que representam um agrupamento de objetos individuais, e há também estes últimos, que são de caráter primitivos.

Com o Composite é possível modelar sistemas, de forma que objetos possam contém grupos de objetos primitivos, e também outros grupos. Além disso, objetos compostos e primitivos devem implementar uma mesma classe abstrata, para definir um comportamento uniforme para ambos (METSKER, 2004).

O padrão Composite em JSF está presente na estrutura de uma *View*. Cada *View* possui um árvore de componentes onde cada nó implementa a mesma interface, assim facilita a navegação por cada um destes elementos. O objeto UIViewRoot que representa a *View*, também implementa a interface UIComponent, comum aos componentes JSF, embora o usuário não interaja com este (MANN, 2005).

Em relação à implementação utilizando a tecnologia JSF, pode-se ver o uso de objetos compostos com a aplicação do tag *h:form*, para criação de formulários. Através dela é possível agregar componentes JSF (que implementam UIComponent), código HTML e outros componentes como este, de forma transparente.

3.1.2.1.2 Façade

Este padrão é muito útil quando há a necessidade de ter uma interface em comum para as classes, ou seja, o Façade simplifica o acesso aos métodos complexos de um subsistema. Com isso, tal padrão fornece à camada de apresentação a lógica de negócios, sem amarrações, por exemplo, com os nomes e formatos de métodos (GAMMA et al., 1995).

Segue abaixo, alguns elementos do JSF onde é verificada a aplicação do padrão Façade:

a) acesso a listas em componentes;

b) acesso a maps;
c) recuperação e atualização de propriedades através de métodos Getters e Setters;
d) EL com padronização para acesso a métodos e propriedades;
e) chamadas de métodos remotos.

3.1.2.2 Padrões de Responsabilidade

Os objetos normalmente possuem métodos e informações que permitem aos mesmos trabalharem sozinhos. Contudo, pode ser necessário delegar responsabilidades a outro objeto, buscando centralização, intensificação ou limitação da responsabilidade de objetos comuns (METSKER, 2004).

Nas seções abaixo, são descritos os padrões de responsabilidade relacionados com a tecnologia JSF.

3.1.2.2.1 Singleton

Este padrão é aplicado para garantir que apenas uma instância de determinada classe seja fornecida, existindo um único ponto de acesso (SHALLOWAY e TROTT, 2004).

Em JSF são encontradas muitas aplicações desse padrão. Abaixo são descritas algumas das situações em que o padrão Singleton é encontrado:

a) quando uma aplicação JSF é inicializada o servidor de aplicação cria uma instância do objeto *FacesContext* para o usuário. A própria aplicação pode fazer acesso a este objeto nas classes, mas a instância acessada é única na aplicação;

b) desde a tecnologia JSP é possível acessar vários objetos na sessão do usuário, para obter informações da requisição. Os mesmos também são instâncias únicas para um usuário em específico;

c) como citado anteriormente quando definido um *managedBean*, pode-se escolher seu escopo, caso o escopo seja *session* tem-se a criação de uma instância única para o usuário, mas se o escopo for *application*, assim que criada a instância deste objeto ela será compartilhada por todos os usuários que acessarem a aplicação.

3.1.2.2.2 Proxy

Quando o acesso a determinadas partes de um objeto deve ser controlado, ao invés desse objeto ser acessado diretamente, entre o ponto de requisição e o mesmo, existe um objeto com

função de procurador (Proxy). Este objeto é responsável por arbitrar o que pode ser acessado para cada caso (BOOCH e SCIENTIST, 2003).

Na tecnologia Java Naming and Directory Interface (JNDI) existe um objeto Proxy responsável por identificar através da requisição do cliente, uma instância de objeto associada. No JSF algo semelhante ocorre com o uso de *managedBeans* e na estrutura de navegação das páginas.

Assim, quando é feita uma chamada a um *managedBean* numa página JSF, tem-se um identificador deste objeto. Este identificador é tratado pelo *FacesServlet* que age como um Proxy localizando a instância do objeto requisitado.

Além disso, no modelo de navegação do JSF tem-se o uso de Proxy para redirecionamento de páginas. Por exemplo, quando um método retorna uma *String* que está registrada nas regras de navegação, o controlador de navegação processa esse objeto e define qual página exibir.

3.1.2.2.3 Observer

Quando a mudança de estado de um objeto é de interesse de vários outros objetos, fazer com que cada um destes sejam notificados, é uma tarefa complexa. Contudo, o padrão Observer define uma dependência de um-para-muitos, fazendo com que o objeto ao mudar de estado, seus dependentes sejam notificados e atualizados (METSKER, 2004).

Este padrão é muito utilizado no JSF, visto que esta tecnologia segue um modelo orientado a eventos. Os mais comuns usos do padrão Observer no JSF está na chamada de métodos no managedBean e na renderização de componentes de tela.

O primeiro caso acontece quando é feito um vínculo entre um evento de um componente e um método do *managedBean*. No momento que o usuário disparar aquele evento as operações registradas para ele serão executadas, ou seja, acontece a execução do método.

Outra aplicação desse padrão, está na renderização dos componentes JSF. Pois, quando é disparado um evento, pode ser feita uma chamada para recarregar os componentes, quando isso acontecer os filhos de cada componente devem ser recarregados também. O JSF realiza esta operação por ter a mesma interface para todos os componentes, de modo que o ouvidor de renderização ao recarregar um componente já tem conhecimento do método a ser chamado.

3.1.2.3 Padrões de Operação

Os padrões contidos nesta categoria fornecem serviços às classes, ou seja, distribuem operações ao longo de diversas classes (METSKER, 2004).

Um exemplo de padrão de operação, é o Template Method. O objetivo desse padrão é fornecer às classes estruturas pré-definidas, com funcionalidade específica, mas com métodos abstratos para que certos passos possam ser redefinidos por estas classes (GAMMA et al., 1995).

Em JSF este padrão surge quando se utilizam modelos prontos de páginas. Onde estes modelos possuem partes invariantes, assim como permite às subpáginas definirem os comportamentos que podem variar.

3.1.2.4 Padrões de Extensão

Segundo Metsker (2004), "os padrões orientados à extensão tratam de contextos nos quais precisamos acrescentar comportamento específico para uma coleção de objetos ou acrescentar novos comportamentos a um objeto sem alterar a sua classe".

O padrão Iterator é um dos padrões de extensão existentes. Assim como o Decorator, este traz facilidades extras no desenvolvimento, visto que, com o mesmo é possível acessar os objetos de diferentes estruturas de dados sem que seus comportamentos sejam conhecidos, sem que as estruturas internas deles sejam envolvidas. Com isso, modificações feitas na estrutura de dados não afetam o código fonte onde foram utilizadas (H. DEITEL e P. DEITEL, 2005).

O padrão Iterator utiliza uma interface padrão entre as estruturas de dados, chamada "iterator", é através dos métodos desta interface que os dados são reconhecidos de maneira transparente em diferentes estruturas de dados.

No JSF este padrão é muito utilizado. Em componentes de exibição de dados, por exemplo, *dataTable* e *dataList*, os dados são acessados através de estruturas de dados do Java, como: List, ArrayList, Vector, Map, e sua extensões; mas, não é necessário que o desenvolvedor preocupe-se em varrer estas estruturas, pois o próprio componente a "itera" para obter os dados. Além disso, existem componentes pertencentes às bibliotecas de extensão do JSF que iteram listas para criação dinâmica de outros componentes.

3.1.3 Desvantagens do uso de padrões

Utilizar padrões no desenvolvimento de sistema como citado, traz benefícios. No entanto, possuem suas desvantagens e limitações. Como há abstração sobre os elementos da aplicação, não há uma forma de reutilizar o código fonte do padrão aplicado. Isso não reduz a complexidade de aplicá-lo novamente. Além disso, tem-se uma forte ligação entre um padrão e seu contexto de aplicação, caso não exista um padrão para um novo contexto, é necessário um grande comprometimento na busca de um que atenda a nova necessidade (GROTT, 2003).

Por um lado, usar padrões é uma forma eficaz de reuso, por outro, tem-se um alto custo para implementação nos processos do projeto. De modo que, apenas profissionais experientes os utilizam com eficiência, pois, é necessário identificar em que circunstâncias genéricas um padrão pode ser aplicado. A complexidade é própria de padrões, forçando os projetistas a ter um conhecimento e compreensão de muitos padrões, diferentemente de componentes executáveis, onde conhecer as interfaces já é suficiente (SOMMERVILLE, 2007).

3.1.4 Model View Controller (MVC)

O padrão arquitetural Model View Controller (MVC) se tornou popular através do uso na linguagem Smalltalk e hoje é frequentemente usado na construção de interfaces de usuário. O mesmo tem por objetivo a separação das aplicações em três camadas distintas: Modelo, Visão e Controlador. O Modelo representa os dados da aplicação principal e as regras de negócio. A Visão faz a exibição dos dados ao usuário e também serve como meio de interação da aplicação. E finalmente, o Controlador trata as interações ou dados de entradas do usuário (DUDNEY et al. 2004).

A interface de usuário (Visão) é a parte com maior variabilidade na maioria dos sistemas, e isso pode ocorrer em relação a requisitos, ou até mesmo na plataforma da aplicação. Logo, em aplicações altamente acopladas, uma pequena mudança na Visão, pode propagar alterações em todo o sistema (REENSKAUG, 1979).

Nesse contexto, o padrão MVC promove uma forma flexível e reutilizável para resolver esses problemas, uma vez que dissocia o Modelo, a Visão e o Controlador em componentes de uma aplicação, mantendo a comunicação entre eles.

O padrão MVC permite que as regras de negócio sejam acessadas e visualizadas através de diferentes interfaces de usuário, pois, neste padrão as partes lógicas da aplicação não conhecem e nem precisam conhecer as visualizações sendo exibidas (SANTOS, 2008).

Abaixo segue a Figura 3.1, a mesma ilustra o diagrama padrão da arquitetura MVC. Onde, fica a cargo do Controlador direcionar as interações vindas da Camada de Apresentação, para o Modelo. O Modelo por sua vez, executa as regras de negócio da aplicação e pode realizar a persistência dos dados. Na sequência, pode ser necessário que as operações executadas no Modelo retornem alguma notificação para a interface, novamente, o Controlador é encarregado de exibir as mensagens ou mudanças dos dados.

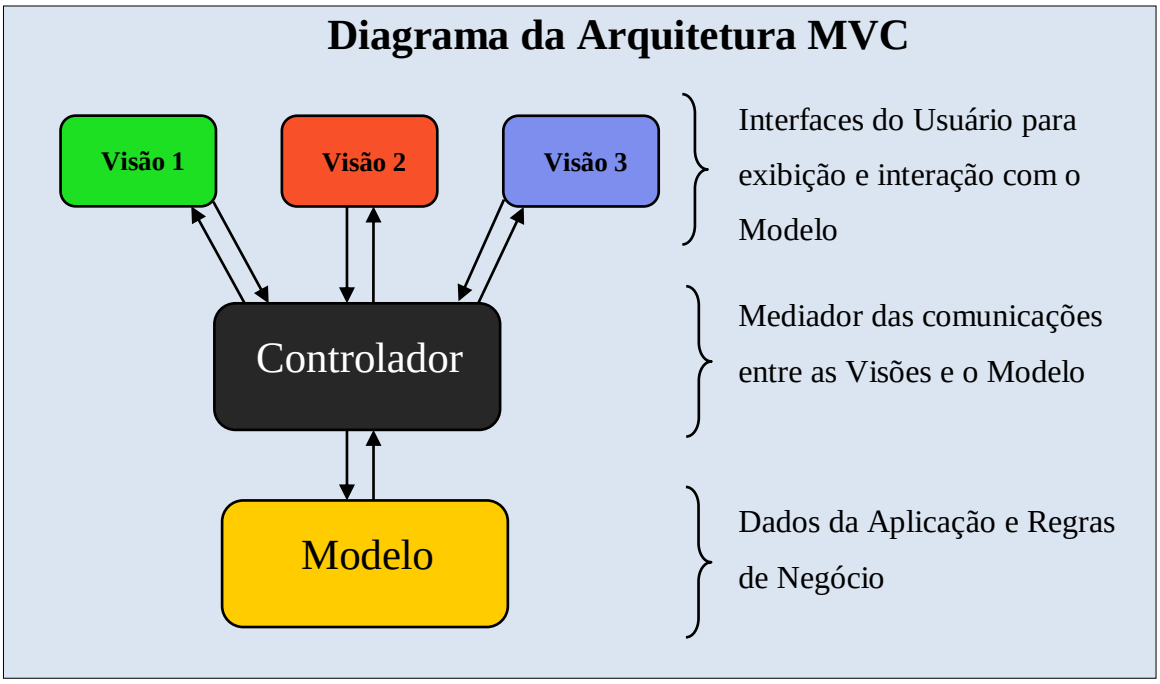

Figura 3.1: Diagrama que demonstra a Arquitetura MVC

Pode-se destacar várias vantagens da utilização da arquitetura MVC, assim como existem certas consequências de sua aplicação. A seguir algumas delas são destacadas de acordo com Dudney et al. (2004):

a) este padrão diminui o acoplamento entre as interfaces dos usuários e o Modelo, permitindo assim, diferentes visualizações para um dado modelo;
b) as mudanças ocorridas no Modelo são transmitidas automaticamente a Visão, através do Controlador;
c) este padrão incentiva a reutilização de componentes de interface e os controladores podem ser trocados, mantendo o mesmo Modelo;
d) para aplicação pequenas, onde existe um baixo acoplamento das camadas, aplicar MVC pode resultar em maior complexidade. Porém, mesmo nesse

contexto este padrão se aplica, pois, geralmente as aplicações começam pequenas, mas se tornam complexas no decorrer de sua evolução;

e) fora o conjunto de aplicações que não requerem uso de Controladores, por exemplo, em acesso apenas para leituras, a Visão e Controlador estão intimamente ligados, e isso pode limitar o nível de reuso;
f) o número de atualizações recebidas do Modelo pelas Visões e Controladores, podem ser excessivas, cabendo a necessidade de fazer registro de algumas de algumas delas. Portanto, as Visões e Controladores devem ser registrados restringindo somente as partições do Modelo que lhes interessam.

Até este ponto foi elucidado o padrão MVC de maneira genérica, mas como o foco deste trabalho são os sistemas Web, especificamente Java Web, na próxima seção será mostrado como esse padrão arquitetural se relaciona e se adapta ao contexto Web.

3.1.4.1 Modelo-2 - MVC para a Web

As aplicações Web executam sobre o protocolo HTTP sem armazenamento de estado. Desse modo, as aplicações Web implementam o padrão MVC de maneira diferente, mantendo grande partes dos benefícios deste. O termo Modelo 2 se refere à adaptação do padrão MVC original para uso em aplicações Web.

Geralmente os usuários interagem num sistema Web, acessando com uso de um navegador Web páginas HTML. Nestas páginas, uma interação, como ao pressionar um botão, pode gerar uma requisição ao servidor para realização de alguma operação.

Um Servlet Controlador pode estar atendendo as requisições, onde, ao receber uma requisição o mesmo interage com o Modelo, e na sequência determina a qual Visão enviar a resposta, podendo ainda, a Visão consultar o modelo para realizar esta operação.

No Modelo-2 pode existir apenas um Controlador para a aplicação Web. Com isso, facilita operações como: gerenciamento de segurança, gerenciamento de usuários e gerenciamento do fluxo de controle da aplicação. Este tipo de Controlador é conhecido como *Front Controller* (BOOCH, 2003).

No padrão *Front Controller*, o servidor Web delega a um controlador frontal todas as requisições recebidas, e este passa a gerenciar o processo, criando objetos de classe com base em configurações pré-definidas, para a execução do serviço solicitado. O objeto criado nesse processo, ao terminar o serviço retorna o resultado ao Controlador Frontal, que por sua vez,

irá determinar se constrói uma página, redireciona a visualização, dentre outras operações (SOUZA, 2007).

Segue abaixo a Figura 3.2, que representa a execução de aplicações que implementam o Modelo-2 do MVC, voltado para a Web. A primeira ação acontece através de uma requisição do Navegador Web, então, o Servidor Web recebe a mesma e delega ao Controlador Frontal a responsabilidade de seu processamento. O primeiro passo que o Controlador Frontal executa, é carregar as configurações da aplicação, como mostra na figura, para definir que objetos instanciar. Na sequência, executar o passo dois que identifica qual ação (serviço) deste objeto executar, o resultado obtido é então repassado a uma tecnologia que pode renderizar uma página de resultados.

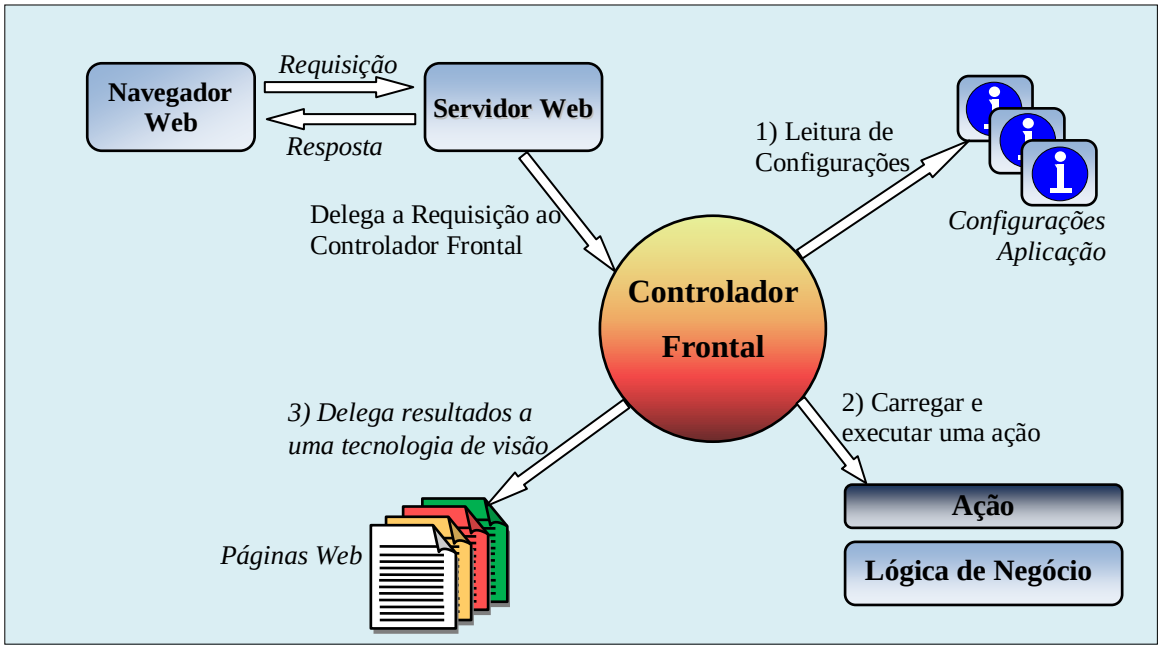

Figura 3.2: Diagrama funcional do Modelo-2 do MVC (SOUZA, 2007).

No Modelo-2 um controlador pode necessitar para a execução da aplicação, de componentes auxiliares ou então configuração em XML, e então atuar como intermediadores entre o Modelo e o Controlador. Este cenário pode ser encontrado em grandes aplicações, em que a Visão não acessa diretamente o Modelo. Neste caso, o Controlador ou seus objetos auxiliares obtém um conjunto de dados em forma de Objeto de Transferência de Dados (DTO) (BOOCH, 2003), podendo fornecer a Visão o acesso direto aos dados.

A separação de camadas advinda do Modelo-2 do MVC, proporciona outra vantagem além das já citadas para o modelo MVC padrão. De acordo com Mann (2005), este padrão permite um maior controle de erros, pois, um erro na Visão não compromete o código da aplicação ou

o modelo. Do mesmo modo, um erro no modelo não afeta o código da aplicação ou a visão. Portanto, esta separação facilita a construção de testes unitários para cada camada, além de permitir o trabalho independente das equipes sobre cada camada.

3.2 Engenharia de Software Baseada em Componentes (CBSE)

No final da década de 1990 surge a CBSE (Component-Based Software Engineering) como uma abordagem focada na reutilização de componentes no desenvolvimento de sistemas. A principal motivação para sua criação foram as frustrações encontradas nas tentativas de aplicar reuso na orientação a objetos. As classes de objetos por serem entidades muito detalhadas e específicas, frequentemente necessitavam ser ligadas a uma aplicação no momento da compilação. Com isso, o desenvolvedor precisa conhecer detalhes das classes, o que por consequência, pode levar à necessidade de acesso ao código fonte de componente. Embora as previsões iniciais apontassem para o sucesso dessa abordagem, na prática nenhum mercado para objetos individuais se desenvolveu (SOMMERVILLE, 2007).

O Desenvolvimento Baseado em Componentes é uma abordagem da CBSE, e pode ser usada para a solução destes problemas, segundo o que aponta a seção 3.3.1.

A CBSE é conceituada por Heineman e Councill (2001) (apud Oliveira e Paula (2009)) como: “A CBSE é atualmente a forma mais rápida de produzir software, com menos esforço e de alta qualidade. Ela está tornando-se um elemento indispensável no desenvolvimento de software no mundo”.

Para Pressman (2010), a CBSE tenta conseguir seguinte princípio: “Um conjunto de componentes de software normalizados, pré-construídos, é disponibilizado para se enquadrar em um estilo arquitetural específico para algum domínio de aplicação. Então, a aplicação é montada usando esses componentes, em vez de partes discretas de uma linguagem de programação convencional”.

Ainda segundo Pressman, a CBSE compreende duas abordagens simultâneas da engenharia: engenharia de domínio e desenvolvimento baseado em componentes (DBC). A engenharia de domínio objetiva encontrar componentes candidatos a reuso dentro de um

domínio de aplicação. Já o desenvolvimento baseado em componentes, cria, adapta ou pesquisa os componentes que podem ser utilizados para os requisitos que o cliente deseja.

3.2.1 Desenvolvimento Baseado em Componentes

O desenvolvimento baseado em componentes é uma atividade da CBSE que acontece simultaneamente com a engenharia de domínio. Anterior ao DBC, grande parte dos produtos de software desenvolvidos, eram blocos monolíticos, ou seja, uma estrutura com alto acoplamento e rigidez. Com esta abordagem, os sistemas podem ser definidos através de componentes com função bem definida, diminuindo a complexidade de inter-relacionamentos de módulos e clareando os relacionamentos entre os componentes do sistema (LUCRÉDIO, 2009).

O DBC é uma abordagem de desenvolvimento baseada na composição de módulos já existentes. Contudo, existem especialistas em orientação a objetos que não compreendem como agrupar objetos em componentes. Existem ainda, profissionais que consideram seu trabalho superior aos outros, visto que não confiam em algo como os sistemas de prateleiras (Commercial-off-the-shelf – COTs), o que os leva à reconstrução de código em vez de reutilizar (OLIVEIRA e PAULA, 2009).

Segundo Spagnoli e Becker (2003), o DBC contribui para a manutenção dos sistemas, pois, permitem tanto sua atualização por meio da integração de novos componentes, quanto à evolução dos componentes já existentes. Além disso, conforme diz Brown (1998) (apud Spagnoli e Becker (2003)), o grande interesse pela DBC atualmente, ocorre devido às tecnologia permitirem o uso dos componentes e composição destes para o desenvolvimento de aplicações, mas também tem relação com as mudanças na forma como as aplicações são desenvolvidas, utilizadas e mantidas.

No DBC têm-se duas abordagens de desenvolvimento, o desenvolvimento **de** componentes e o desenvolvimento **com** componentes. O desenvolvimento de componentes abrange o projeto, criação, e documentação do componente. E o desenvolvimento com componente compreende a utilização de componentes existentes através da composição para o desenvolvimento de sistemas (LUCRÉDIO, 2009). A segunda abordagem será descrita logo na sequência, sendo que a primeira abordagem será elucidada em meio à próxima seção.

Existem cinco atividades intrínsecas ao desenvolvimento com componentes, propostas por Brown (1997) (apud Spagnoli e Becker (2003)). Segundo Spagnoli e Becker (2003), essas

mesmas são possíveis etapas no DBC, ou seja, não existe ordem de execução e também a obrigatoriedade de todas as etapas. Abaixo são explanadas as cinco atividades conforme o autor:

a) seleção – é busca e seleção de componentes aptos para o desenvolvimento do sistema. Investiga a qualidade e as propriedades do componente, assim como seu ambiente de execução. Esta etapa retorna uma listagem de componentes candidatos, que serão analisados na próxima fase. Além disso, sua principal dificuldade é a definição do que pode ser considerado um componente pela aplicação;

b) qualificação – etapa que visa avaliar se o componente candidato se enquadra nos requisitos necessários no sistema. Normalmente as documentações e especificações são analisadas detalhadamente. Também são efetuadas interações com equipes de desenvolvedores dos componentes, e teste dos componentes;

c) adaptação – a adaptação é uma operação muito importante para a reutilização de componentes e seu objetivo é corrigir conflitos que impessão o projetista de aplicar o componente numa aplicação;

d) composição – esta etapa é responsável por unir os componentes em uma infra-estrutura que possibilite a união dos componentes seguindo especificações em comum nos componentes;

e) atualização – esta é geralmente a última etapa do DBC, e abrange a atualização parcial ou total dos componentes, mantendo funcionalidade e interfaces iguais.

3.2.2 Engenharia de Domínio

Segundo Clements (2005), "Engenharia de domínio consiste em encontrar pontos comuns entre sistemas para identificar componentes que podem ser aplicados a muitos sistemas e identificar famílias de programas, que são posicionadas para tirar plena vantagem desses componentes". A engenharia de domínio visa a identificação, construção, catalogação e disseminação de artefatos, que tenham a possibilidade de ser utilizada num domínio de aplicação específico. De acordo com Pressman (2010), a engenharia de domínio pode ser dividida em três processos: análise, construção e disseminação. Abaixo são descritos cada um destes processos (LUCRÉDIO, 2009):

a) análise – compreende o desenvolvimento de documentação de identificação do domínio, pontos comuns, variáveis de domínio, identificação de subdomínio;

b) construção – envolve o projeto de uma arquitetura própria do domínio e que atenda às especificidades encontradas na fase de análise, e o suporte ao seu gerenciamento com base nos diferentes subdomínios identificados;

c) disseminação – o último processo é a implementação do projeto, resultando em artefatos como: componentes, ferramentas de modelagem, geradores de código, e outros.

Para Pressman (2010), a mais comum aplicação da engenharia de domínio é no contexto de engenharia de software orientada a objetos. E nesse contexto são tem-se os seguintes passos:

1) definição do domínio a ser pesquisado;
2) classificar os elementos retirados do domínio;
3) experimentar uma amostra significativa das aplicações do domínio;
4) cada aplicação da amostra deve ser avaliada para a definição dos critérios de análise;
5) desenvolvimento de um modelo de análise para os objetos.

3.2.3 Componentes

Muito se fala sobre componentes, mas o que realmente caracteriza um componente? De acordo Hurwitz (2009) (apud Oliveira e Paula (2009)), "Um componente é um pacote de software que contém uma coleção de serviços relacionados e atributos que abrangem a funcionalidade completa de algum problema de negócio".

Outra definição que esclarece melhor este termo é a de Szyperski (2002): "um componente de software é uma unidade de composição com interfaces bem especificadas em contrato e dependências explícitas do contexto. Um componente de software pode ser independentemente implantado e pode ser composto por terceiros".

Entretanto, a definição de componente não está restrita apenas a uma tecnologia ou aplicação específica, ou seja, qualquer artefato de software que possuir uma interface com pontos de acesso aos seus serviços, pode ser considerada um componente (WCOP 97 (apud Silva, 2000)).

Na Tabela 3.1 são elencadas algumas características de um componente baseando em CBSE (Sommerville, 2007):

Tabela 3.1: Características de Componentes. Adaptado de (Sommerville, 2007, pg. 294)

Plano	Descrição
Padronizado	O componente usando na CBSE deve seguir algum modelo padronizado de componente. Modelo este que pode determinar interfaces, documentação, composição, metadados e implantação.
Independente	Deve ser possível efetuar a composição e implantação sem a necessidade de outros componentes exclusivos.
Passível de composição	O componente deve ser passível de composição, ou seja, este artefato de software deve fornecer interfaces para as efetivas interações externas.
Implantável	Um componente para ser implantável, deve necessariamente operar de forma independente sobre uma plataforma de componentes que siga o mesmo modelo do componente.
Documentado	A documentação do componente auxilia os usuários na tomada de decisão de usar ou não o componente.

3.3 Reutilização

3.3.1 Introdução

Existem muitas aplicações que são desenvolvidas sem planejamento para evolução e reaproveitamento de código fonte, isso gera um projeto de baixa qualidade e soluções com foco muito específico. Nesse contexto, o termo reutilização se refere a reutilizar em novos projetos, módulos de projetos consolidados, ou seja, testados e difundidos com sucesso (GROTT, 2003).

As primeiras formas de reutilização eram direcionadas sobre o código fonte, principalmente pelas linguagens orientadas a objeto. Segundo Gamma (1995) (apud GROTT (2003)), a desenvolvimento de sistemas no paradigma orientado a objetos é muito complexo, porém, desenvolver projetos reutilizáveis é mais complexo ainda. Até certo ponto, a orientação a objetos atende à demandas de reutilização, porém, os projetistas se sentem

frustrados por não conseguirem atingir um amplo nível de reuso segundo a proposta inicial (OLIVEIRA e PAULA, 2009).

O Desenvolvimento Baseado em Componentes (DBC) busca resolver algumas carências encontradas na orientação a objetos em relação à reutilização de software. O componente por ser um elemento de mais alto nível que as classes, resulta em um baixo acoplamento com outros componentes e numa maior coesão interna. Além do mais, os acoplamentos necessários são executados através do uso de interfaces e conectores, de modo que possam ser facilmente substituídos por outras estruturas que conheçam estas ligações (MURTA, 2006).

Com isso, o desenvolvimento tem muitos ganhos, pois o uso de componentes reutilizáveis supre a necessidade de criação de novos, e evita o consequente tempo despendido nisso. Somado a isso, tem-se o aumento da qualidade do projeto no todo, quando se utiliza componentes reutilizáveis de qualidade (REIS, 2002).

Reutilização de software não abrange somente o reaproveitamento de código fonte. Não que seja uma prática errada, mas é uma das técnicas menos produtivas de reutilização (GROTT, 2003). Além dessa forma de reutilização, é possível reutilizar vários outros artefatos, são eles (D'SOUZA e WILLS, 1998):

a) código compilado e objetos executáveis;
b) código fonte (classes e métodos);
c) teste de automatizados;
d) diagramas e modelos: colaboração, frameworks, padrões;
e) interface com usuário, "*look and feel*";
f) planos, estratégias e regras de arquitetura

Entretanto, ainda hoje a reutilização é entendida de forma errada, pois muitos projetistas acham que é possível reutilizar artefatos de aplicações específicas, sem adaptações, o que gera artefatos reutilizáveis com elementos de um domínio de aplicação específico. Geralmente a reutilização não é adotada de forma sistemática, isso pode levar os desenvolvedores a criarem componentes para cada projeto, ao invés de reutilizar. Além disso, tanto armazenar adequadamente componentes, quando investir na qualidade dos mesmos, são fatores que aumentam o grau de reutilização de uma aplicação (VILLELA, 2000).

3.3.2 Requisitos para o reuso de software

Apesar da existência de muitas abordagens voltadas à reutilização de software, abaixo são descritos alguns conceitos comuns entre elas, conforme citado por Krueger (1992) (apud Lucrédio (2009)):

a) abstração – a abstração é essencial para que o reutilizador possa compreender as funcionalidades de um artefato de forma "macro", ou seja, esconder detalhes técnicos de um artefato facilita a viabilidade de sua reutilização;

b) seleção – Segundo Arango (1988), bibliotecas de artefatos reutilizáveis facilitam a identificação, seleção e busca de forma eficiente e facilitada. Dessa forma, o armazenamento adequado destes artefatos, facilita posteriormente, sua busca e recuperação, elementos essenciais para a reutilização (VILLELA, 2000).

c) adaptação – o processo de adaptação de artefatos para um contexto diferente pode ser uma tarefa muito difícil. Para contornar esse esforço, frequentemente as abordagens de reutilização buscam a criação de artefatos genéricos, que são adaptados através de parâmetros, configurações, ou então de pequenas modificações.

d) integração – a integração se torna uma problema, quando se deseja integrar artefatos de software projetados para diferentes arquiteturas.

3.3.3 Tipos de Reutilização

A reutilização, como já citado, pode ocorrer de várias formas. De acordo com Grott (2003) são descritas algumas delas na sequência:

a) reutilização de código fonte – é a forma mais utilizada de reutilização. Onde, trechos de código com determinadas funcionalidades, são adicionados em outras partes de um sistema. Ainda segundo o autor, embora haja certo benefício por reutilizar um trabalho já feito, essa é a forma menos eficiente de reutilização;
b) reutilização de herança – quando uma classe herda funcionalidades de outra, existe uma reutilização do código gerado e validado na classe superclasse;
c) reutilização de modelos – é a utilização dos modelos que fornece um modelo inicial para os elementos do projeto como: criação de interfaces, padronização de código fonte, elaboração de casos de uso e outros. Apesar de trazer muita organização nos

projetos, esta estratégia de reutilização necessita de constantes atualizações dos modelos;

d) reutilização de componentes – por possuírem função bem definida, os componentes são utilizados nas aplicações para fornecerem serviços a um baixo custo de acoplamento. Sua estrutura interna não precisa ser exibida, o que se busca, é que o componente cumpra sua função independente de como foi projeto internamente;
e) reutilização de frameworks – este tipo de reutilização se baseia no aproveitamento de estruturas geralmente voltadas a uma área específica. O *framework* concentra certo conhecimento fornecido como base inicial para as aplicações, ou seja, representando as funcionalidades comuns a um domínio de aplicação;
f) reutilização de artefatos – refere: casos de uso, documentação, diagramas; ou seja, elementos reutilizados de um projeto, para dar origem a outro projeto (FURLAN, 1998 apud GROTT, 2003). Subtrai o tempo de elaboração de artefatos de software necessários para o início de um projeto descendente;
g) reutilização de padrões – como já citado, utilizar padrões promove o reuso, pois padrões representam o conceito de técnicas que deram certo em determinado projeto, e sua generalização trará as mesmas vantagens/desvantagens relatadas em sua documentação.

A forma de reutilização mais relevante para este trabalho é a reutilização de componentes, pois o estudo de caso será constituído de uma estrutura reutilizável, com função específica, ou seja, um agregado de componentes para fornecer serviços.

3.3.4 Vantagens

Sommerville (2007) destaca como principal vantagem na reutilização, a redução de custo para a especificação, projeto, implementação e validação de componentes de software. Além disso, o autor destaca outras vantagens como mostra a Tabela 3.2.

Tabela 3.2: Benefícios do reuso de software. Adaptado de (Sommerville, 2007, pg. 276).

Benefício	Explicação
Aumento da confiabilidade	Geralmente um software reusado é mais confiável, pois já foi experimentado e testado, e eventuais erros que foram encontrados estão corrigidos.

Redução nos riscos de processos de desenvolvimento	O custo de desenvolvimento de um software é um problema constante. Como o custo de um software existente já é conhecido, reutilizar módulos desse software facilita o gerenciamento de custos no projeto.
Especialistas com trabalho focado	Os especialistas podem concentrar os trabalhos, no desenvolvimento de software reutilizável que reúnam seus conhecimentos específicos.
Seguem padrões	Pode-se utilizar um conjunto de componentes reusáveis para definir um padrão de interface com o usuário. Além disso, seguir padrões de estruturação de projeto, de código fonte, documento, dentre outros, acelera o desenvolvimento, a compreensão e manutenção dos sistemas.
Aceleração do desenvolvimento	O reuso de software pode acelerar a construção de um sistema, pois o tempo de desenvolvimento e validação pode ser reduzido.

3.3.5 Desvantagens

A reutilização, assim como traz benefícios, também apresenta alguns problemas, como expressos abaixo (LUCRÉDIO, 2009):

a) quando um artefato é modificado, tem-se novamente o trabalho de testá-lo para garantir que não será introduzido erro nos locais onde é usado;
b) como citado anteriormente, a abstração e a seleção são fatores essenciais na reutilização de software. Porém, dependem muito do talento e intelecto humano, e por serem fatores com certa subjetividade, não é algo próprio de todo indivíduo;
c) para o uso de um artefato desconhecimento é necessário um estudo, caso isso demande muito tempo, geralmente faz o desenvolvedor desistir da reutilização e partir para a criação de um novo artefato.

Além disso, Sommerville (2007) cita como problema principal da reutilização o tempo para o entendimento de se um componente é adequado à reutilização em determinada situação. Outros problemas citados pelo mesmo autor são descritos na Tabela 3.3.

Tabela 3.3: Problemas advindos da reutilização Fonte: Adaptado de (Sommerville, 2007, pg. 277)

Problema	Explicação
Aumento no custo de manutenção	No caso onde o código fonte do componente reutilizável não está disponível, o custo com manutenção é aumentado, pois os elementos reutilizados podem progressivamente ficarem incompatíveis com o sistema.
Baixa disponibilidade de ferramentas de apoio	Muitas vezes as ferramentas CASE podem não suportar o desenvolvimento baseado em reuso. Assim, dificultando ou mesmo impossibilitando a integração da ferramenta com um conjunto de componentes.
Síndrome do não-inventado-aqui	Alguns projetistas têm preferência pela criação de seus próprios componentes, por acreditarem que podem ter maior grau de aprimoramento e maior controle.
Criação e manutenção de uma biblioteca de componentes	Assegurar que uma biblioteca de componentes reutilizáveis seja utilizada pelos desenvolvedores, não é uma tarefa simples. As técnicas disponíveis atualmente para catalogação, classificação e recuperação de componentes de software são imaturas.
Busca, interpretação e adaptação de componentes reutilizáveis	Os componentes devem ser facilmente encontrados em uma biblioteca de componentes, interpretação (compreendidos) e, podem ser adaptados para se integrar a um novo ambiente.

Vale ressaltar uma importante observação feita por Pressman (2010), sobre a redução de tempo e esforço advindo do reuso de software:

> *Apesar disso ser verdade para aqueles que reusam software em projetos futuros, o reuso pode ser caro para aqueles que precisam projetar e construir componentes reusáveis. Estudos indicam que projetar e construir componentes reusáveis pode*

> *custar de 25% a 200% a mais do que o software-alvo. Em alguns casos, o diferencial de custo pode não ser justificável.*

Nesse contexto, uma das melhores estratégias pode ser a reutilização de artefatos, quando possível, feitos por outros desenvolvedores/empresas, pois segundo afirmou Pressman (2010) no trecho acima, os custos extras estão relacionados ao desenvolvimento "de componentes reutilizáveis", e não no desenvolvimento "com componentes reutilizáveis".

3.4 Frameworks

Frameworks podem ser considerados estruturas que abstraem um domínio de aplicação, e deve ser especializado em aplicações desse contexto. O desenvolvimento utilizando *frameworks*, objetiva generalidade da aplicação em relação aos elementos do domínio tratado. Além disso, torna a aplicação mais flexível, ou seja, facilita a manutenção e extensibilidade (SILVA, 2000).

Coad (1992) define *framework* como uma estrutura composta de relacionamentos entre objetos e classes, de maneira genérica, para compor uma aplicação concreta. Além disso, Grott (2003, p. 59) afirma que, "Frameworks possibilitam reutilizar não só componentes isolados, como também, toda a arquitetura de um domínio específico".

Os frameworks surgem no cenário onde, tarefas semelhantes são realizadas diversas vezes em diferentes aplicações, contextos e por diferentes equipes de desenvolvedores. Logo, verificou-se a possibilidade de generalizar os métodos/processos envolvidos, a fim de estender o uso sem acoplamento de regras de negócio com o domínio da aplicação.

Portanto, o reuso adquirido através dos *frameworks*, segundo Nash (2003) acelera o processo de desenvolvimento, pois reduz os esforços e tempo dedicado à construção de componentes de software, reutilizando código e projeto simultaneamente.

Pode-se dizer que um *framework* determina a arquitetura de uma aplicação. Assim, todos os artefatos do projeto: objetos, classes, interfaces, relacionamentos e sequencia de execução; ficam subordinados às regras do *framework*. Desse modo, o desenvolvedor na maioria dos casos, não precisará despender tempo para a modelagem de arquitetura em sua aplicação (FERLIN, 2004).

Fayad e Schmidt (1997) (apud Sommerville (2007)) dividem os *frameworks* em três classes, são elas:

a) *frameworks* de infra-estrutura de sistema são direcionados ao desenvolvimento de infra-estrutura para sistemas, no contexto de comunicação, interface com o usuário e compiladores;
b) *frameworks* de integração com *middleware* são compostos por um conjunto de classes e objetos associados, que dão suporte à comunicação de componentes e à troca de dados. *Frameworks* para desenvolvimento com objetos distribuídos estão inclusos nesta classe;
c) *frameworks* corporativos são voltados à aplicações específicas, e geralmente são focados no usuário final. São estruturas complexas de se desenvolver, mas que possibilitam a criação de muitas aplicações.

De acordo com Larman (2002) (apud Ferlin (2004)), um *framework* é dotado de certas características:

a) representa um conjunto coeso de classes que se unem para prover serviços à uma aplicação;
b) possue classes concretas e abstratas (principalmente), e pode ter também, métodos abstratos ou concretos;
c) pode obrigar o usuário a estender as classes do *framework* para uso, reformulação ou agregação dos serviços do mesmo;
d) define através de classes concretas e abstratas as interações entre os elementos de sua estrutura;
e) segue o **Princípio de Hollywood** – "*Não nos chame, nós iremos chamá-lo*" - responsabiliza-se pela sequencia de execução da aplicação, ou seja, fora as chamadas de métodos dentro das regras de negócio, as demais são efetuadas pelo *framework*.

Para garantir o reuso na construção de *frameworks*, podem ser utilizados vários padrões de projeto, visto que, estes facilitam a documentação da arquitetura, representam uma linguagem comum entre os desenvolvedores e permitem a separação das camadas nas aplicações. Ferlin (2004) complementa dizendo que, "Os padrões de projeto permitem que a arquitetura de um framework possa se adequar a vários aplicativos diferentes, sem a necessidade de uma remodelagem".

3.4.1 Classificação dos Frameworks

3.4.1.1 Horizontal e Vertical

Frameworks horizontais não são específicos para um domínio de aplicação, servindo como infra-estrutura nas aplicações, na comunicação, interfaces gráficas e gerenciamento de dados (GROTT, 2003).

Por outro lado, os *frameworks* verticais são elaborados com base na experiência numa área específica. Portanto, são *frameworks* especializados em determinado domínio de aplicação, fornecendo suporte (tarefas comuns) para as aplicações desse domínio (GROTT, 2003).

Entretanto, neste trabalho serão tratados massivamente *frameworks* horizontais, no caso JSF e Richfaces, visto que estes fornecem elementos de infra-estrutura para a construção de interfaces com usuários, abstração de comunicação, facilidades no tratamento de dados, e outros.

3.4.2 Inversão de Controle

Como citado anteriormente, ao usar um *framework* na aplicação, ocorre uma "Inversão de Controle", conceito definido como, mudança no fluxo de controle de aplicações. A inversão de controle é um padrão de projeto segundo Hammant (2008). O mesmo autor sugere que componentes da aplicação não devem se preocupar com o seu próprio controle.

Assim, as aplicações não seguem o modelo tradicional, onde as chamadas de métodos eram especificadas pelo programador, com a inversão de controle, o *framework* se responsabiliza pela sequência de execução da aplicação. Com isso, o *framework* pode agir como um esqueleto extensível, de modo que os métodos das aplicações especializam os métodos genéricos deste (JOHNSON e FOOTE, 1988).

3.4.2.1 Injeção de Dependência (DI)

Segundo Fowler (2004), quando se deseja que instâncias de classes tenham dependência de outras classes para determinada operação, é importante que apenas interfaces estejam envolvidas no relacionamento, e não uma implementação específica da operação. Além de Fowler (2006), Gamma et al. (1995) também segue o mesmo princípio.

Baseados nessas teorias surgiram os Frameworks de DI (Injeção de Dependência). Estes são *frameworks* que se responsabilizam por instância objetos e todas as dependências deles, ou seja, através das configurações de dependências entre as classes o *Framework* saberá quais

classes instância (SOUZA, 2007). Logo, na DI o trabalho de instanciar as classes, não mais pertence a uma classe principal. O *Framework* se responsabiliza por gerenciar o processo de instanciação de classes e pode tomar para si também o fluxo de controle da aplicação.

Na tecnologia JSF também é utilizada a injeção de dependência, por exemplo, no uso de *managedBean*. Estes, assim que configurados, para usa-los não é necessário nada além de referenciá-los na página Web, já que sua instância é criada pelo *container* Web na primeira referência encontrada. Além disso, é tarefa do *container* salvar e obter dados dos atributos do *managedBean* e chamar métodos deste, sem envolver implementação dessas operações por parte do desenvolvedor.

3.5 Framelets

O desenvolvimento com uso de *framework*, como citado anteriormente, trazem muitos benefícios, por exemplo, a reutilização da experiência de projetistas e reuso de padrões de projeto embutidos nele.

Contudo, o *framework* geralmente é uma estrutura caixa-preta, ou seja, os desenvolvedores nem sempre podem ter a disposição detalhes de projeto, assim como os códigos fontes do *framework*. Com isso, segundo Grott (2003), apesar de um *framework* ter uma documentação clara, apresenta mesmo assim um grau de dificuldade que pode levar os desenvolvedores a não utilizá-lo.

Nesse sentido, vários autores elencaram alguns problemas associados a *frameworks* complexos, como expressos na sequência (CAMARGO, 2006):

a) projetar um *framework* é um tarefa difícil, principalmente devido a: complexidade e tamanho do *framework*, dificuldades no processo de projeto, processo iterativo;
b) a reutilização é difícil de ser aplicada, uma vez que o *framework* contém uma arquitetura básica para aplicações de determinado domínio. Onde, o projetista deve compreendê-la adequadamente para o efetivo reuso;
c) uma grande dificuldade dos *framework* está relacionada à composição destes. Uma vez que um *framework* toma para si o fluxo de controle da aplicação, é muito difícil combinar vários outros que façam o mesmo, sem que a integridade seja quebrada;
d) os *frameworks* obrigam que as aplicações clientes levem consigo classes deste, que não são utilizadas por elas.

Esses problemas resultam do conceito tradicional de que um *framework* deve ser a base, ou seja, um esqueleto complexo para se desenvolver uma aplicação inteira. Logo, o *framework* torna-se uma grande coleção de classes fortemente acopladas, o que quebra os princípios da modularização e os torna difícil de combinar com outros *frameworks* similares. Interfaces de herança e dependências de lógicas ocultas, não podem ser gerenciadas por programadores. A solução mais praticada nessa situação é utilizar *framework* caixa-preta que são especializados pela composição ao invés da herança. No entanto, isso facilita o uso, mas limita sua adaptabilidade e mantém problemas relacionados a design e combinação de estruturas (PREE, 1999).

Segundo Pree e Koskimies (2000), o princípio não é considerar o desenvolvimento com *frameworks* um problema, mas considerar a granularidade de onde são aplicados. Para enfrentar tais problemas, deve-se avaliar o código fonte de vários sistemas, para descobrir que pequenos módulos são implementados várias vezes de uma maneira similar. Nesse sentido, surge o conceito de *framelets*, que é a aplicação de conceitos de *frameworks* para o desenvolvimento de componentes pequenos e reutilizáveis.

A idéia por traz dos *framelets* é ter várias entidades pequenas altamente adaptáveis, voltadas para determinado modelo, e que possam ser facilmente compostas nas aplicações (PASETTI e PREE, 2000).

Grott (2003) elencou baseado nos trabalho de Pree e Koskimies (2000), algumas características do *framelet* em contraste com os *frameworks*, são elas:

a) não causa a inversão do controle principal de uma aplicação;
b) tem pequena quantidade de classes, geralmente doze classes no máximo;
c) possui interface simplificada e intuitiva;
d) pode ser auto-definido em um domínio de aplicação;
e) funciona isoladamente de outros *framelets*.

Taligent (2003) (apud Grott, 2003) cita outras características do *framelet*:

a) é composto de implementações concretas de uso direto;
b) possui poucas classes herdadas e métodos que possam ser sobrescritos;
c) concretiza e implementa funcionalidades semelhantes numa mesma abstração;
d) fragmenta grandes abstrações em outras menores, onde cada uma tem uma funcionalidade pequena e bem definida;
e) permite implementar cada abstração em um objeto;

f) tem foco na composição, reduzindo a quantidade de classes e a complexidade no *framework* cliente.

A utilização *framelet* diferente do que acontece no uso de *frameworks* não toma o fluxo de controle principal da aplicação, isso acontece somente onde o *framelet* está efetivamente sendo usado, ou seja, eles também seguem o princípio de *Hollywood* descrito na seção 3.4 (CAMARGO, 2006).

3.5.1 Framelets versus Componentes

As definições de *framelet* propostas por Pree e Koskimies (2000) podem dar a entender que um *framelet* é sempre um componente. No entanto, existem componentes com escopo muito maior do que um *framelet* propõe, e os componentes podem ser voltados para mais de um modelo. Neste trabalho, será utilizada a definição de componente proposta por Reenskaug (1995), onde um componente pode assumir vários modelos dependendo de sua especificação, podendo agregar outros artefatos de granularidade mais fina, já o *framelet* é voltado para apenas um modelo.

Portanto, como o *framelet* é voltado a apenas um modelo, logo, não é "correto" desenvolver um *framelet* que implemente a funcionalidade de vários modelos simultaneamente. Dessa forma, para desenvolver um modelo mais amplo, utilizam-se componentes, onde estes delegam tarefas aos *framelets* (PASETTI e PREE, 2000)

Nesse contexto, *framelets* se fundamentam como artefatos que utilizando da abordagem tradicional de "dividir para conquistar", simplificando o processo de desenvolvimento de sistemas. Os mesmos atuam sobre um conjunto de pontos de variação, ou seja, casos de uso um domínio, de modo a assumir subconjuntos menos de destes. Onde, um *framelet* possui um subconjunto de objetos totalmente integrados entre si, mas com baixo acoplamento com os demais subconjuntos (*framelets* ou objetos simples).

3.5.2 Desenvolvendo Framelets

O *framelet* é uma estrutura que fornece determinado serviço, sendo que, este é usado por meio de interfaces, que também determina o que de abstrato pode ser implementado na aplicação. Além disso, sua arquitetura deve ser bem definida e que possibilite interação com outros *framelets* e aplicações clientes (GROTT, 2003; PREE, 1999).

Para efetivar a construção dos *framelets* é essencial identificar e fragmentar um domínio de aplicação em itens, para isso pode-se seguir as etapas abaixo (GROTT, 2003):

a) mapear em objetos as abstrações principais encontradas de uma aplicação;

b) identificação de requisitos repetidos em vários sistemas;

c) usar uma pequena arquitetura para problemas pequenos;

d) identificar em meio a vários comportamentos, um elemento em comum entre eles.

Segundo Coad (1997) (apud Grott, 2003), para desenvolver um *framelet* pode-se seguir um padrão de análise.

Ainda, um *framelet* é um artefato que define basicamente duas interfaces: uma para chamada aos seus serviços e outra para identificar suas partes abstratas que serão especializadas nas aplicações. A primeira é apropriada para reestruturar código legado, ou seja, quando é removido um módulo que representa serviços logicamente relacionados num sistema, um *framelet* pode substituir este módulo fornecendo o mesmo serviço através de uma interface compatível. Já a última corresponde à interface de especialização: os métodos chamados devem ser implementados de diferentes maneiras para cada diferente aplicação (PREE e KOSKIMIES, 1999).

3.5.3 Aplicação dos Framelets

Framelets definem construções arquiteturais que são disponibilizados para uso em outros *framelets* ou diretamente como uma instância em aplicações. *Framelets* podem exportar três tipos de construtores:

a) componentes: unidades configuráveis e pré-definidas que podem ser usadas sem alterações;
b) interfaces: conjuntos de assinaturas de métodos;
c) padrão de projeto: soluções arquitetônicas para um projeto de um problema específico de um domínio de aplicação.

Observa-se que estas construções existem em diferentes níveis de abstração. Componentes são objetos concretos para uso direto em aplicações. Interfaces são classes abstratas que necessitam do fornecimento de uma implementação especifica na para o uso, ou podem servir como um protocolo padronizado. Por fim, padrão de projeto são soluções arquitetônicas abstratas, voltadas a um dado problema e que devem ser desenvolvidos para sua aplicação, como já citado neste trabalho.

Como mostrado na Figura 3.3, interfaces e componentes podem ser exportados de dois modos, “horizontalmente” para outros *framelets* ou componentes, ou “verticalmente” instanciados em aplicações. Já os padrões de projeto, não existem efetivamente em nível de aplicação, logo, são exportados apenas na “horizontal”, como metodologias passíveis de reutilização em outros *framelets* ou componentes (PREE, 1999).

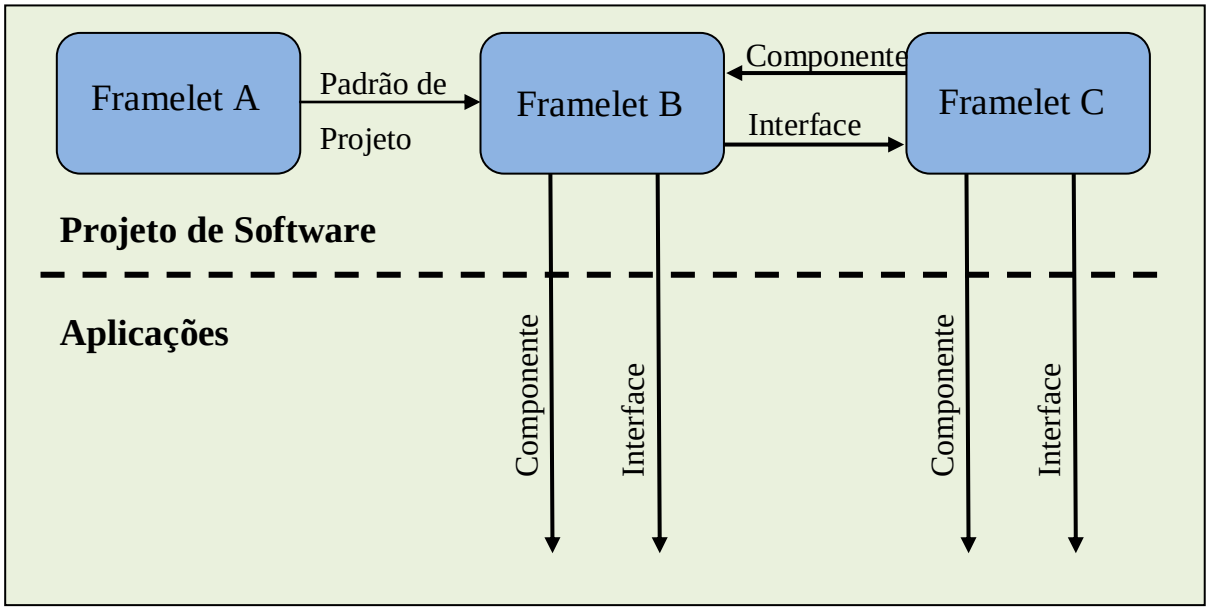

Figura 3.3: Desenvolvimento de aplicações com uso de *framelets* (adaptado de Grott (2003)).

Portanto, uma aplicação final consiste de classes abstratas em conjunto com componentes padrões fornecidos, que podem ser sobrescritos e classes abstratas serem especializadas. De modo que, as classes abstratas e objetos concretos que compõem a aplicação são subconjuntos das interfaces e componentes fornecidos pelos *framelets*.

3.5.4 *Framelets* aplicados no desenvolvimento de um *Framework* para Sistemas de Controle de Atitude e Órbita (Attitude and Orbit Control System – AOCS)

O objetivo do projeto AOCS foi o projeto e prototipação de um *framework* baseado em componentes para um Sistema de Controle de Atitude e Órbita (AOCS) de satélites. Este projeto teve início em agosto de 1999, com a parceria da Agência Espacial Européia (ESA) e o departamento de Ciência da Computação da Universidade de Konstanz, localizada na cidade Konstanz, Alemanha. O mesmo foi encerrado em setembro de 2000, no Laboratório de Controle Automação da Politécnica de Zurique, após o gerente de projeto do *framework* AOCS deixar a Universidade de Konstanz.

Um software AOCS atualmente é reescrito do zero para cada novo sistema. Mas sua estrutura geral e os requisitos de um AOCS sofrem pequenas mudanças de missão para missão, permitindo assim um nível substancial de reutilização. Por exemplo, todos os sistemas AOCS executam operações como telecomandos, geração de telemetria, detecção de falhas, entre outras. Com base nisso, o projeto de um *framework* AOCS, visa à identificação de funcionalidades comuns a esses sistemas, para construir uma solução reutilizável (PASETTI, 2002).

Um AOCS possui tipicamente as seguintes funcionalidades (CECHTICKY e PASETTI, 2002):

a) telemetria – formatação e envio de dados de telemetria;
b) telecomando – gestão de telecomandos;
c) detecção de falha – gerenciamento de verificações para detecção de falhas no software autônomo AOCS;
d) recuperação de falha – medidas corretivas para neutralizar falhas detectadas;
e) controlador – gerenciamento de algoritmos de controle para controle de loops operados pelo AOCS;
f) manobra – gestão de manobra com rotacional, roda de descarga, etc;
g) reconfiguração – gerenciar reconfiguração de unidade;
h) unidade – gerenciar sensores externos e atuadores.

De acordo com Pasetti (2011), os objetivos específicos propostos para o projeto de *framework* AOCS, são:

a) reutilização de arquitetura: a grande maioria dos sistemas AOCS são re-projetos em diferentes missões. Caso o mesmo contratante seja responsável pelas missões, existe a reutilização de fragmentos de códigos, porém, a arquitetura não pode ser completamente reutilizada. O foco é desenvolver um software que possua uma arquitetura genérica do segmento de AOCS, viabilizando a reutilização da arquitetura em missões;
b) reutilização de componentes – a reutilização de código, como citado na seção 3.3.3, dificulta refatorações e testes quando ocorre evolução das rotinas utilizadas. Portanto, o objetivo é desenvolver um sistema composto por componentes reutilizáveis como unidades de uso direto;
c) extensibilidade de arquitetura: em sistemas AOCS, para estender funcionalidades é necessário modificar sub-rotinas de algoritmos.

d) projeto orientado a objetos: os sistemas AOCS padrões seguem um paradigma modular. Contudo, grande parte dos mesmos são baseados meramente em objetos, e não nos princípios formais da orientação a objetos.

Para cada funcionalidade de uma AOCS descrita acima, foram definidas interfaces abstratas para separar o seu gerenciamento de sua implementação, foi criado um componente núcleo para gerenciar a mesma e desenvolvido componentes com implementações padrões para operações repetitivas. Portanto, o *framework* AOCS é constituído de *framelets* independentes que cooperam através do intercâmbio de dados (PASETTI e PREE, 2000).

Com base no projeto de *framework* AOCS, pode-se considerar algumas heurísticas para a identificação de *framelets*, são elas (PASETTI, 2002):

a) mapear um conjunto (grupo) de requisitos da aplicação para um *framelet* – identificar requisitos que se repetem em muitas aplicações. Isso pode ser feito através da inspeção dos conteúdos presentes nos documentos de requisitos do usuário;
b) desenvolver um *framelet* voltado a um ou mais *hot-spot's* – um hot-spot é um ponto adaptável que precisa ser especializado em uma aplicação. A identificação de pontos adaptáveis é a fase inicial de projeto de uma estrutura reutilizável. Um grande ponto adaptável pode servir de base para a concepção de um *framelet*;
c) construção de *framelets* em torno de padrões de projeto – padrões de projeto geralmente são compostos por um aglomerado de classes cooperantes para um dado problema de projeto sem considerar questões de controle da execução.
d) para aplicações embarcadas, mapear tarefas para *framelets* – tarefas geralmente são funcionalidades auto-suficientes e tem limites bem definidos entre si. Este tipo de mapeamento assegura que os *framelets* tenham funcionalidades dissociadas, facilitando sua criação e a vinculação em uma estrutura de integração;
e) mapear casos de uso abstratos para um *framelet* – com muita frequência casos de uso são utilizados para identificar classes abstratas em um domínio, mas isso pode ser estendido para identificar *framelets*, visto que um *framelet* possui certo grau de variabilidade de comportamento.

Um dos principais retornos do projeto do *framework* AOCS é o conceito de casos de implementação. Um caso de implementação (CI) é um projeto abstrato que é definido no início do projeto de um *framelet*, e são modificados e consultados paralelamente ao

desenvolvimento do *framelet*. Pode-se dizer que os CIs descrevem como os *framelets* associados são utilizados, do mesmo modo, que um caso de uso descreve como utilizar uma aplicação. Na fase final do desenvolvimento dos *framelets*, cada CI pode ser usado como uma "receita de bolo" tanto para sua reutilização, quanto para o desenvolvimento de outra solução que os tome como base (PASETTI e PREE, 2000).

A definição anterior, deixa claro que os casos de implementação devem abranger a totalidade das funcionalidades de um documento de requisitos, sendo na fase inicial de projeto de estruturas utilizando *framelets*, cada caso de implementação é deve ser definido como um *framelet* (DONOHOE, 2000).

De acordo com Pasetti (2002), os casos de implementação não seguem uma metodologia formal em suas descrições. No projeto AOCS, estes foram descritos de maneira sistemática, mas informal, com as seguintes informações:

a) objetivo do caso de implementação;
b) descrição do caso de implementação;
c) *Framelets* envolvidos no caso de implementação;
d) construções de *framelets* envolvidos no caso de implementação;
e) hot-spot de *framelet* envolvidos no caso de implementação;
f) Pseudo-código mostrando como é a criação do caso de implementação.

Segue abaixo (Tabela 3.4) um exemplo de um dos casos de implementação do projeto AOCS. O pseudo-código não foi mostrado, pois, exigiria outras fundamentações que não cabem numa exemplificação dos casos de implementação (PASETTI e PREE, 2000):

Tabela 3.4: Exemplo de Caso de Implementação do Projeto AOCS

Objetivo	Construir um telecomando para executar uma atitude de manobra de giro
Descrição	Atitudes de giro são normalmente iniciadas por um comando de solo (telecomando). Este caso de implementação mostra como construir um telecomando para executar uma atitude de manobra de giro
Framelets	FrameletTelecommand FrameletManoeuvreManagement
Construções do Framelet	Interface Telecommand – forma de

	exportação do “FrameletTelecommand”; Interface AocsManoeuvre e componente ManoeuvreManager – forma de exportação do “FrameletManoeuvreManagement”
Hot-spots de Framelet	Definir de tipos e características de telecomandos
Pseudo-código	******

Tomando como base as heurísticas identificadas no projeto AOCS, este trabalho segue estas abordagens para a construção de uma estrutura reutilizável em diferentes domínios. Contudo, não serão utilizados casos de implementação para descrever os *framelets* deste trabalho, pois, os mesmos serão explicitados no capítulo que descreve a implementação, sua citação neste trabalho, se deve a importância que esta abordagem teve para a documentação e reutilização no projeto do *framework* AOCS.

Capítulo 4

JSF (Java Server Faces)

O JSF é um framework lançado em setembro de 2002, especificado pela JCP (Java Community Process) em conjunto com a Sun Microsystem (GEARY e HORSTMANN, 2005 (apud (MARAFON, 2006)).

Este é o principal framework para desenvolvimento de aplicações Java Web da especificação Java Enterprise Edition (Java EE), portanto é multiplataforma. JSF é um framework orientado à requisições e, segue a arquitetura MVC, fundamentado na programação baseada em componentes (LOBO FILHO, 2010).

Segundo Ramos (2008), a tecnologia JSF tem como objetivo evitar que o desenvolvedor crie páginas JSP semelhantes à criação de páginas HTML, mas sim, como se fossem interfaces de um aplicativo tradicional. Para isso, esta tecnologia fornece um conjunto de componentes dispostos em bibliotecas, assim como na tecnologia JSP.

Logo, o framework processa os componentes de tela, e gera uma página ao navegador, e se responsabiliza por responder às requisições posteriores. Além disso, Marafon (2006) afirma que o JSF foi projetado para seguir um modelo orientado a eventos, para se aproximar do modelo de desenvolvimento de aplicações *desktops*.

Teixeira (2008) cita algumas características do JSF:

a) desenvolvimento de interfaces Web utilizando componentes pré-definidos;
b) acesso a componentes através de *tags* JSP;
c) viabiliza a reutilização dos componentes das páginas;
d) faz ligação entre eventos do cliente com tratadores.

De acordo com Borges (2007), existem quatro classificações para as classes JSF:

a) aplicação: se divide em duas seções, a classe *Application* que representa a aplicação e os *backing beans*, que contém a lógica de negócio da aplicação;

b) contexto: classes que objetivam acessar os dados de uma dada requisição;

c) manipulação de eventos: podem ser *listeners* ou apenas métodos no *backing bean*, responsáveis pelo tratamento de eventos disparados pelo usuário;

d) componentes gráficos: são os componentes da apresentação, onde o JSF pode manipulá-los através de código Java.

A Figura 4.1 demonstra o nível de abstração, começando pelo processamento de requisições no servidor *Web*, métodos e classes para utilizar *servlets*, páginas composta por HTML mais código Java no JSP, e finalmente na utilização de JSF tem-se acesso ao maior nível de abstração.

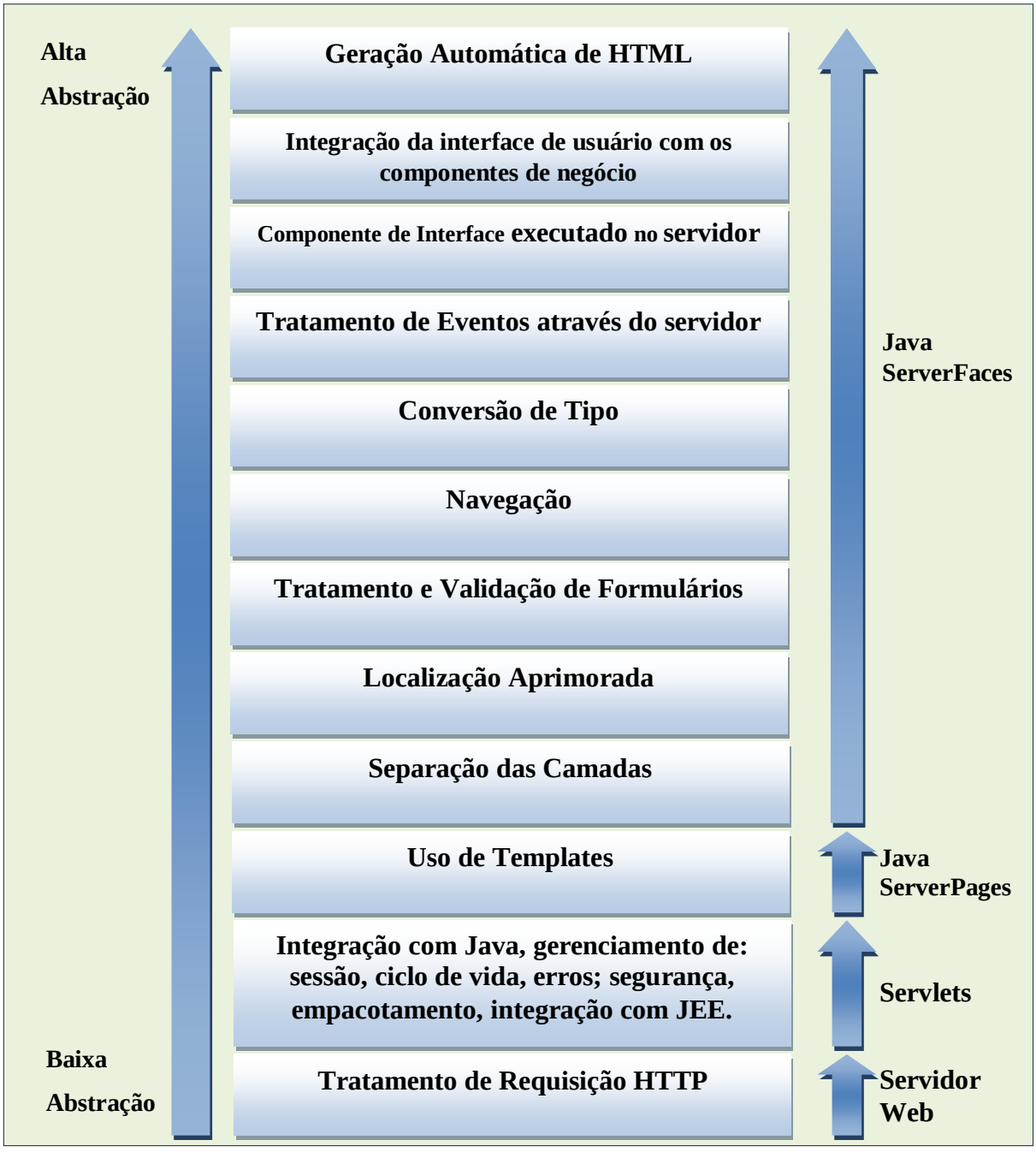

Figura 4.1: Disposição dos recursos e tecnologias do JSF por nível de abstração (BORGES, 2007)

4.1 Contextualizando a Tecnologia JSF

Nesta seção, são elencados alguns elementos com base no trabalho de Marafon (2006), estes são essenciais para a compressão das discussões acerca da tecnologia JSF, que serão realizadas neste trabalho.

4.1.1 FacesServlet

As requisição são processadas pelo *FacesServlet*, que é um *servlet* controlador da tecnologia JSF, responsável por receber e responder às mesmas (MANN, 2005).

Toda aplicação Web desenvolvida em JSF possui um *FacesServlet*, classe que está presente no pacote javax.faces.webapp. Todas as requisições que seguem um padrão de URL registrado no arquivo WEB.xml para o *FacesServlet*, são tratadas por este (KURNIAWAN, 2004).

A classe *FacesServlet* segue o mesmo ciclo de vida dos Servlets, pois implementa os métodos *init*, *service* e *destroy* da interface javax.servlet.Servlet.

4.1.2 Ouvidores e Eventos

É através dos eventos que os usuários interagem como os elementos da apresentação de um sistema. Logo, um evento vai desde um simples clique num componente, até a execução de uma lógica de negócio complexa em decorrência deste clique. A implementação JSF utiliza um modelo JavaBean para a manipulação de eventos, muito semelhante ao tratamento de eventos no Swing. Assim, os componentes da aplicação JSF pode disparar eventos, e tanto os desenvolvedores, quando os próprios componentes podem registrar ouvidores para lidar com os eventos (MANN, 2005).

A tarefa de tratar eventos sempre foi algo muito complexo para o desenvolvedor, pois fatores como decodificar requisições/resposta e vincular funções JavaScript aos eventos, eram realizados sobre cada componente pelo próprio desenvolvedor.

Portanto, para se trabalhar com eventos no JSF, não é necessária preocupação com as requisições e como tratá-las, visto que, a própria implementação suporta o desenvolvimento orientado a eventos, assim como está disponível na programação tradicional. O JSF fornece por padrão quatro tipos de eventos: eventos de mudança de valor, eventos de ação, eventos de modelos de dados e eventos de fase.

Eventos de mudança de valor são ativados quando ocorre mudança no valor do componente por parte do usuário. Já para componentes de comando como um botão ou link, existe os eventos de ação, que podem fazer chamadas a lógicas de negócio no *BackingBean*. Eventos de modelo de dados ocorrem quando é selecionada uma linha de um componente de múltiplas linhas de dados. Finalmente, eventos de fase são eventos disparados antes e depois de uma fase do ciclo de vida JSF, e podem ser padronizados ou reimplementados e registrados pelo programador (MANN, 2005).

4.1.3 View ID

Cada *View* é composta por uma árvore contendo os dados dos componentes instanciados por uma requisição e, por um identificador exclusivo. Numa página, as *View IDs* podem assumir três estados:

a) *New View – View* recém criada e sem dados sobre os componentes;
b) *Initial View* – quando é realizada a inicialização da página são carregados os dados dos componentes;
c) *PostBack* – restauração da página para uma *View* já existente (resultado aparece na mesma página solicitada).

4.1.4 FacesContext

É o objeto que engloba todos os componentes da tela que estão instanciados no servidor, isso inclui as *View Ids*, e outros dados sobre as aplicações em execução (GEARY e HORSTMANN, 2010).

Este objeto é criado para cada requisição JSF, e possui três objetos que lhes são passados pelo container Web, são eles: javax.servlet.ServletContext, javax.servlet.ServletRequest e javax.servlet.ServletResponse. O objeto *FacesContext*, permite acesso as informações de estado da requisição atual de qualquer fase do ciclo de vida JSF (KURNIAWAN, 2004).

4.1.5 Backing Beans

É um objeto Java que se comunica com as páginas JSF, ligando os campos da página HTML com os atributos do objeto e fazendo o processamento sobre eles (MARAFON, 2006). Esses elementos também chamados de *managedBeans*, são muito importantes para a tecnologia JSF, pois é através deles que são armazenadas e recuperados dados, assim como, acontece a interação do usuário com as regras de negócio da aplicação.

Em JSF, os *ManagedBeans* armazenam o estado das páginas, manipulam os dados e controlam o fluxo de execução da aplicação (ORACLE, 2011).

Os *ManagedBeans* podem ser declarados de duas formas. A primeira é no arquivo *faces-config.xml* da aplicação, e a segunda acontece por meio de *annotations* na própria declaração da classe.

Quando uma expressão com o nome de um *managedBean* for encontrada na página, a implementação JSF instancia o objeto da classe que este referencia. Este objeto permanece instanciado por um tempo determinado, tempo este chamado escopo e definido na configuração do *managedBean* (GEARY e HORSTMANN, 2010).

Na sequência tem-se os escopos e suas *annotations* associadas, que de acordo com Oracle (2001) um *managedBean* pode assumir:

a) *application* (@ApplicationScoped) – os objetos nesse escopo são compartilhados por todos os usuários;
b) *session* (@SessionScoped) – neste cada usuário tem acesso a sua instância do objeto, e essa instância permanece até que o navegador seja fechado;
c) *view* (@ViewScoped) – neste escopo os dados persistem nas interações de uma única página;
d) *request* (@RequestScoped) – os objetos desse tipo persistem seus dados apenas no decorrer de uma única requisição HTTP;
e) *none* (@NoneScoped) – indica um escopo não definido;
f) *custom* (@CustomScoped) – um escopo definido pelo usuário.

A Figura 4.1 demonstra como é realizada a declaração de uma classe como *managedBean* utilizando a *annotation "*javax.faces.bean.ManagedBean", e a definição do seu escopo como sessão, utilizando a *annotation* "javax.faces.bean.SessionScoped". O atributo *name* da *annotation* @ManagedBean pode ser omitido, nesta situação, é dado ao *managedBean* o nome derivado do nome da classe, com a inicial em letra minúscula (GEARY e HORSTMANN, 2010).

```
1.  @ManagedBean (name="usuario")
2.  @SessionScoped
3.  public class UsuarioController implements Serializable {
4.      ...
5.  }
```

Figura 4.1: Trecho de código de uma classe managedBean com escopo do tipo sessão

Ainda segundo Oracle (2011), os *managedBeans* são objetos considerados “preguiçosos”, ou seja, são instanciados apenas se forem referenciados em uma página em execução. Contudo, no JSF 2.0 há a possibilidade de forçar a instanciação do objeto através da expressão “eager=true”, passada como parâmetro na *annotation* que define o *managedBean*.

4.1.6 Expressões de Linguagem (EL)

No JSF, as expressões de linguagem geralmente associam componentes de tela com *backingBeans* ou objetos de modelo nas aplicações. As mesmas são avaliadas em tempo de execução, geralmente quando uma página está em exibição, e não no instante da compilação. Além disso, as EL permitem que se faça referência aos elementos do *managedBean* sem a inclusão de código Java na visão como acontecia na tecnologia JSF (MANN, 2005).

As EL tem as seguintes funções na tecnologia JSF (ORACLE, 2001):

a) avaliação imediata ou adiada de expressões;
b) disponibilidade para obter e atualizar dados;
    i) leitura dinâmica dos dados dos elementos do *managedBean*, manipulação de estrutura de dados e objetos implícitos;
    ii) grava dinamicamente os dados de um formulário nos componentes do *managedBean*;
c) capacidade para invocar métodos das aplicações JSF;
    i) chamadas de métodos estáticos e públicos de forma facultativa;
d) realizar operações aritméticas dinamicamente.

O JSF derivou o recurso de EL do JSP 2.0, mas, existem diferenças fundamentais (MANN, 2005):

a) JSF utiliza o símbolo (#) para marcar o início da declaração de uma expressão, oposto ao cifrão ($) usado em JSP;
b) as expressões do JSF suportam recuperação e atualização do valor de propriedades;
c) o JSF EL pode referenciar métodos dos objetos;
d) no JSF as expressões podem ser avaliadas usando código Java comum, sem interferência do JSF ou JSP.

Na Tabela 4.1 tem-se um conjunto de operadores disponíveis para uso nas EL do JSF. Sendo que, esses recursos facilitam o atendimento a diferentes necessidades de interação entre a visão e as regras de negócio ou modelo de dados.

Tabela 4.1: Operadores lógicos e matemáticos disponível para a criação de expressões de linguagem em JSF (MANN, 2005; GEARY e HORSTMANN, 2010)

Elementos	Opcional	Função
.		Acessar propriedade ou método de um managedBean, ou entrada em HashMap
[]		Acessar uma matriz ou lista de elementos, ou entrada em HashMap
()		Criar uma sub-expressão e controla a ordem de avaliação
? :		Expressão Condicional: ifCondicional ? ValorTrue : ValorFalse
+		Adição
-		Subtração e números negativos
*		Multiplicação
/	div	Divisão
%	mod	Módulo (resto de divisão)
==	eq	Igual (para objetos, usa-se o método equals())
!=	ne	Diferente
<	lt	Menor que
>	gt	Maior que
<=	le	Menor ou igual que
>=	ge	Maior ou igual que
&&	and	AND lógico
||	or	OR lógico
!	not	NOT lógico
empty		Testa valor vazio (null, String vazia, ou vetor vazio, Map ou Coleção sem valores)

4.1.7 web.xml (Deployment Descritor)

As aplicações Web da plataforma JEE geralmente necessitam configurar alguns elementos que estão contidos no descritor de implantação de aplicação Web. Estas configurações ficam no arquivo "web.xml", e as aplicações JSF devem especificar certas configurações, que incluem as seguintes:

a) o servlet responsável por processar as requisições JSF;

b) o mapeamento de identificadores para o servlet de processamento;

c) o caminho para o arquivo de configuração de recursos (*faces-config.xml*), caso não estiver no local padrão.

Além desses elementos, existem outras configurações opcionais que podem ser especificar no arquivo descritor de implantação, como:

a) especificar onde o estado do componente será persistido;
b) restringir acesso a certos domínios de páginas;
c) habilitar validação XML;
d) adição de filtros;
e) adição de listeners.

Por meio da Figura 4.2 pode-se verificar uma estrutura básica do arquivo web.xml nas aplicações. Onde, são declarados inicialmente os elementos de declaração de XML, e em seguida a *tag* "web-app", que abrange todas as configurações da aplicação. Como configuração específica da aplicação, tem-se a declaração de que a classe "ServletBanco" é um servlet com nome "ServletExemplo". Posteriormente, este último nome é mapeado para endereço na aplicação, que objetiva chamar a classe servlet registrada.

```
1.  <?xml version="1.0" encoding="ISO-8859-1" ?>
2.  <!DOCTYPE web-app
3.      PUBLIC "-//Sun Microsystems. Inc//DTD Web Application 2.3//EN"
4.      "http://java.sun.com/dtd/web-app_2_3.dtd" >
5.  <web-app>
6.      <servlet>
7.          <servlet-name>ServletExemplo</servlet-name>
8.          <servlet-class>ServletBanco</servlet-class>
9.      </servlet>
10.
11.     <servlet-mapping>
12.         <servlet-name>ServletExemplo</servlet-name>
13.         <url-pattern>/ServletURL</url-pattern>
14.     </servlet-mapping>
15. </web-app>
```

Figura 4.2: Exemplo de *Deployment Descritor* com configuração de Servlets.

4.1.8 Navegação e faces-config.xml

Aplicações Web não são centralizadas em um único arquivo, ou seja, existem muitas páginas por onde o usuário pode acessar. A ação de se locomover entre diferentes páginas é definida como navegação.

A tecnologia JSF tem um sistema de navegação muito robusto, onde, o manipulador de navegação é responsável pelo carregamento de uma página, baseado no resultado lógico, também chamado de *outcome*, do método de ação. Assim, existem regras de navegação para

as páginas, de forma que o manipulador de navegação interpreta essas regras para definir que página carregar (MANN, 2005).

Tais regras são definidas no arquivo de configuração de navegação (geralmente com o nome *faces-config.xml*). Mas as definições destas regras podem ser feitas por meio de ambientes de desenvolvimento, ficando a cargo dos mesmos criarem a organização dos dados em formato XML no arquivo *faces-config.xml*.

O JSF 2.0 não obriga a existência do arquivo de configuração devido aos *annotation*, mas podem existir um ou mais nos casos de: definição manual de regras de navegação, registro de conversores e validadores, e outras configurações de integração de tecnologia. Os mesmos possuem um formato portátil de configuração por utilizar XML (ORACLE, 2011). É dessa maneira que as ferramentas de apoio ao desenvolvedor, facilitam a criação de regras de navegação e registro de nomes para objetos, conversores e validadores.

A navegação pode ser de dois tipos: estática ou dinâmica. Na navegação estática o endereço da página é especificado diretamente no atributo de ação da página solicitante. Contudo, a navegação referida nesta seção é a navegação dinâmica.

A navegação dinâmica, como o próprio nome sugere, acontece com base nas entradas do usuário, por exemplo, numa página de autenticação, em caso de sucesso o usuário pode utilizar o sistema, caso contrário pode ser redirecionado à visualização para uma página de erro ou exibido nesta uma mensagem de erro.

A Figura 4.3 ilustra um exemplo básico de fluxo de páginas, onde se tem a inicialização do arquivo *login.jsp*, se houver sucesso na autenticação o método no *managedBean* retorna a *String* "sucesso", redirecionando imediatamente para a página *principal.jsp*. Mas se a autenticação falhar, o retorno é "erro", redirecionando para a página *erro.jsp*. Estando autenticado pode-se fechar a seção através do retorno "fecharSessao", voltando novamente para a página de autenticação. Além disso, estando na página de erro pode-se ter um componente de ação que retorne "voltar", e assim redirecionar para uma nova tentativa de autenticação na página inicial.

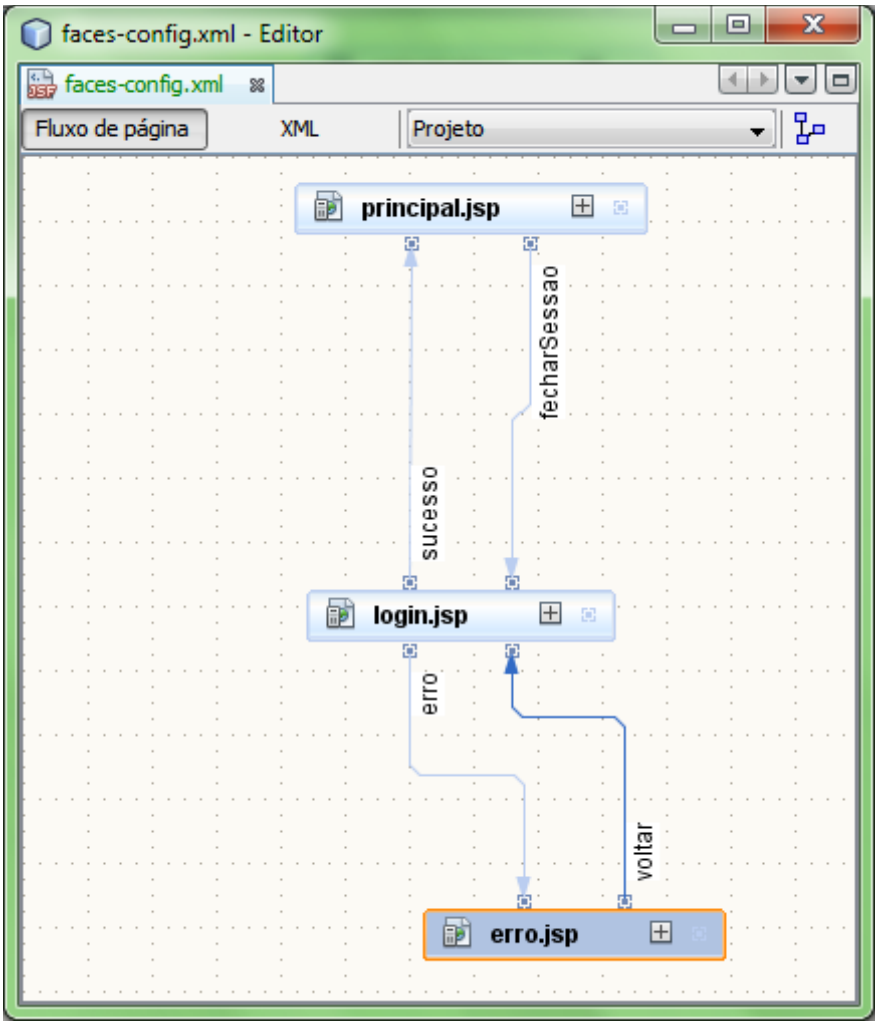

Figura 4.3: Figura que demonstra como o módulo de configuração visual IDE Netbeans exibe as regras de navegação das páginas

Já a Figura 4.4 exibe a organização das regras de navegação no arquivo *faces-config.xml*. Verificam-se na mesma alguns elementos que definem as regras, são eles:

a) *navigation-rule* – define uma nova regra de navegação, que pode conter zero ou mais casos de navegação;

b) from-view-id – diz a que páginas se aplica a regra. Caso não seja definida uma página, a regra se aplica a todas as páginas;

c) navigation-case – um caso de navegação, ou seja, situação onde será feito um redirecionamento;

d) from-outcome – define qual a String considerada em determinado caso de navegação;

e) to-view-id – diz para que página vai ser redirecionado quando o retorno do método coincidir com o *from-outcome* do mesmo caso de navegação.

Agora que os elementos de configuração de navegação foram elucidados pode-se explicar melhor a Figura 4.4.

A primeira regra define casos de navegação para a página "*login.jsp*", onde o primeiro caso define que ao receber uma *String "*erro" (linha 11), a visualização atual é redirecionada para a página *erro.jsp* (linha 12). Já o segundo caso, mostra que se receber uma *String* "sucesso" (linha 15), o redirecionamento é feito para a página *principal.jsp*.

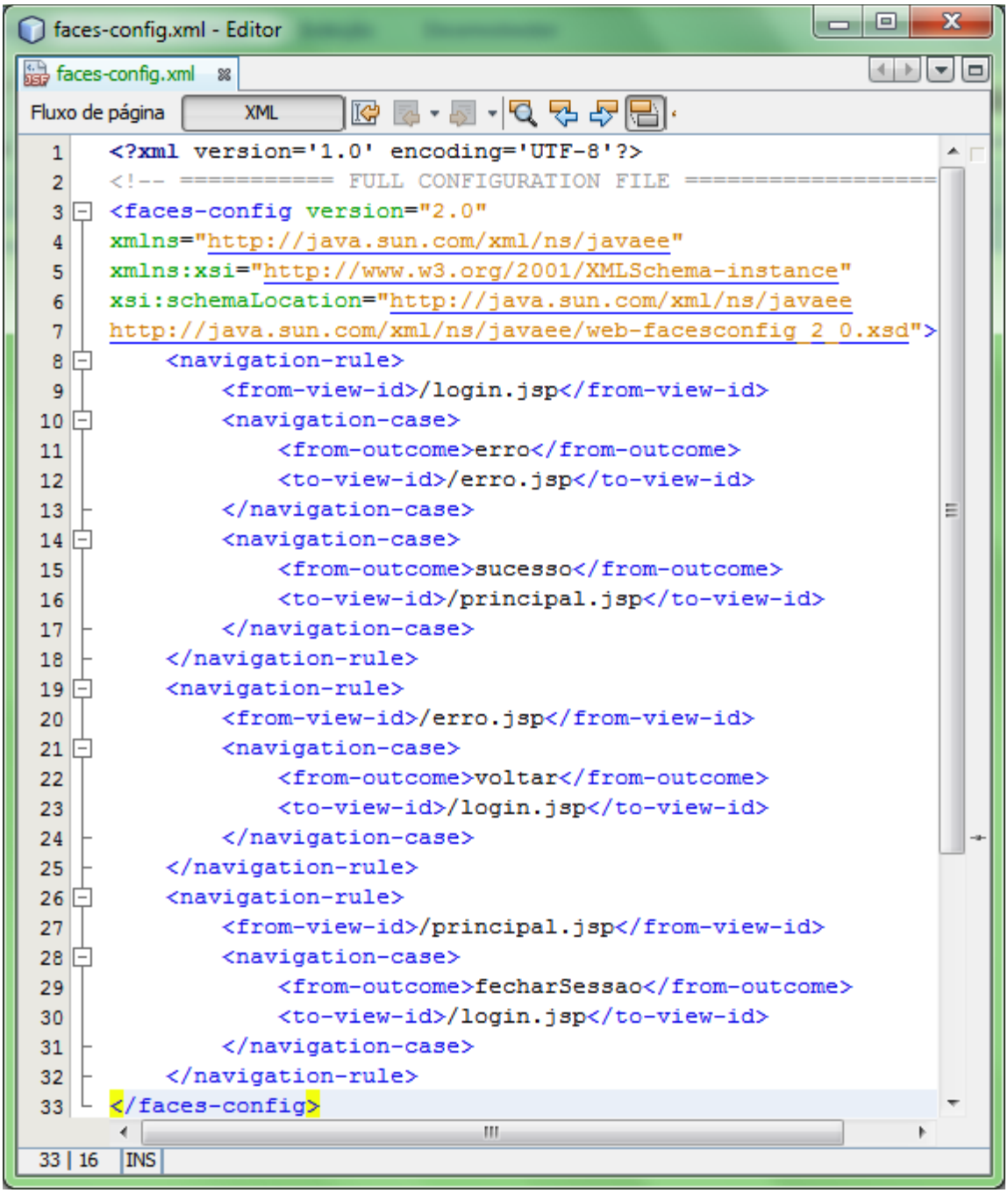

Figura 4.4: Esta figura mostra os dados em XML gerados pela IDE Netbeans

A tecnologia JSF 2.0 suporta ainda, regra de navegação implícita, isto é, regra não definida no arquivo de configuração de recursos na aplicação. Tal regra ocorre quando é passada diretamente uma página na *String* de saída de um método, concatenada com o atributo "*?faces-redirect=true*" (redirecionar visualização).

Entretanto, o arquivo *faces-config.xml* não é voltado apenas para a configuração de navegação na aplicação, nele até a versão 1.2 do JSF era efetuado o mapeamento das classes

da aplicação para identificadores, assim como definição de escopo das instância dessas classes, ou seja, esta era a definição dos *managedBeans*. Na implementação JSF 2.0 usar o arquivo *faces-config.xml* para este fim é opcional, visto que, com os *annotation* a definição de nomes e escopo na aplicação, são realizadas facilmente na própria implementação da classe.

4.1.9 Componentes UI

Componentes UI ou componentes de interface com o usuário, são os componentes do JSF responsáveis pela interação com o usuário da apresentação. Estes componentes têm a vantagem da reutilização de outros subcomponentes no seu desenvolvimento. Com isso, o desenvolvimento é simplificado e agilizado, permitindo ainda a abstração do uso do componente por parte de outros programadores e a divisão do trabalho dos programadores por área de conhecimento (CONCEIÇÃO, 2008).

4.1.10 Renderizadores

No JSF encontram-se dois padrões de renderização. No primeiro, cada componente é responsável pela própria renderização. Já no segundo, o processo de renderização é conduzido por classes chamadas renderizadores, e estão contidas em *render kits*, que geralmente seguem apenas uma forma de saída.

Nesse contexto, o JSF possui um *Render Kit* HTML, o que não impede o desenvolvimento de um novo, por exemplo, geração de arquivos PDF. Quando o renderizador atua no sentido do lado servidor para o lado cliente, o mesmo faz a transcrição do componente JSF para uma saída especificada. Contudo, quando se tem o sentido inverso, o renderizador decodifica a requisição, obtém os parâmetros e define os valores de um componente com base nesses parâmetros (ORACLE, 2011).

Na Figura 4.5 é observado o primeiro código JSF deste trabalho, o mesmo é composto por um componente de saída de texto (*h:outputText*) e dois componentes de entrada de texto (*h:inputText*). Já a Figura 4.6, exemplifica como os renderizadores fazem a transcrição dos componentes JSF da Figura 4.5 para código HTML.

```
1.  <p>
2.      <h:outputText id="saidaTexto" value="Testando saída!" />
3.  </p>
4.  <h:form id="meuFormulario">
5.      <p>
6.          <h:inputText/>
7.      </p>
8.      <p>
9.          <h:inputText id="entradaTexto"/>
10.     </p>
11. ...
12. </h:form>
```

Figura 4.5: Exemplo de código usando componentes JSF

```
1.  <p>
2.      <span id="saidaTexto">Testando saída?</span>
3.  </p>
4.  <form id="meuFormulario" method="post"
5.      action="/jia-standard-components/client_ids.jsf"
6.      enctype="application/x-www-form-urlencoded">
7.      <p>
8.          <input type="text" name=" meuFormulario:_id1" />
9.      </p>
10.     <p>
11.         <input id="meuFormulario:entradaTexto" type="text"
12.             name=" meuFormulario:entradaTexto " />
13.     </p>
14. ...
15. </form>
```

Figura 4.6: Código gerado pelo renderizador HTML do JSF

4.1.11 Validadores

Uma das necessidades do desenvolvimento de interfaces é verificar a validade dos dados inseridos pelo usuário. Para isso, deve ser usado algum artifício de validação dos valores. No caso pode-se validar os dados usando JavaScript no lado cliente e/ou tratamento dos valores através de lógica no lado servidor.

O uso de JavaScript traz como um benefício, evitar o processamento das validações por parte do servidor. Entretanto, usar esse tipo de validação não garante a segurança das validações, pois existem técnicas para alterar as funções de validação e alterar campos ocultos, permitindo com isso, quebrar a integridade dos dados (CONCEIÇÃO, 2008).

Contudo, tanto na validação no lado cliente quanto no lado servidor, pode ser gerada uma lógica muito complexa, dificultando com isso, a compreensão e manutenção do código.

Nesse sentido, o JSF auxilia na validação dos dados de três maneiras: validação no nível do componente UI, utilizando métodos de validação no *managedBean* ou então, usando validadores, ou seja, classes de validação especializadas. Geralmente na validação de componentes UI encontra-se validações simples, como a verificação de campos obrigatórios e

comprimento dos valores de entrada. O uso de métodos de validação no *managedBean* se fundamenta em validar vários campos segundo uma lógica específica (MANN, 2005).

Validadores externos geralmente são usados de maneira genérica. Com isso, classes de validação podem ser usadas sobre diferentes componentes de forma adaptativa. Usar validação no lado servidor é uma boa estratégia, visto que nem todos os clientes têm suporte a *Scripting*.

Na Figura 4.7, verifica-se o uso da validação no componente UI <*h:input*>, para isso, foi usada a *tag* padrão do JSF para invocar validadores, <*f:validate*>. Além disso, foi passado o mínimo e máximo tamanho do valor do campo, dessa forma, se o comprimento do valor ultrapassar o especificado, uma mensagem de erro é lançada à fila de mensagens, para ser exibida ao cliente.

```
1.  <h:inputText>
2.      <f:validateLength minimum="6" maximum="12" />
3.  </h:inputText>
```

Figura 4.7: Exemplo de validação de comprimento de valor no componente InputText do JSF

Já para a validação de valores através de métodos no *managedBean*, os valores são definidos nos atributos da classe, e na sequência os métodos podem analisar elementos como: comprimento, tipo de dados, números fora do intervalo, e outros.

Finalmente, quando se cria uma classe de validação no JSF, esta deve ser registrada através de um identificador (*validator-id*) e a sua hierarquia de pacotes juntamente com seu nome (*validator-class*), em um arquivo XML de configuração segundo a Figura 4.8.

```
1.  <validator>
2.      <validator-id>tcc.ValidarDataHoje</validator-id>
3.      <validator-class>org.tcc.validadores.ValidarDataHoje
4.      </validator-class>
5.  </validator>
```

Figura 4.8: Configuração de um validador no JSF

A Figura 4.9 ilustra o uso do validador registrado na Figura 4.8 anterior.

```
1.  <h:inputText>
2.      <f:validator validatorId="tcc.ValidarDataHoje" />
3.  </h:inputText>
```

Figura 4.9: Usando um validador não padrão no JSF

4.1.12 Conversores

Todos os valores de uma requisição são *Strings*, isso trazia na tecnologia JSP muito trabalho para a conversão, visto que, cada parâmetro não *String*, em JSP devia ser convertido adequadamente pelo programador. Além disso, também cabia ao programador a tarefa

inversa, a de converter o atributo não *String* para *String*, e assim enviar aos componentes de tela algo compreensivo pelo cliente.

Nesse contexto, através da Figura 4.10, pode-se verificar que no trecho de código *Scriptlet*, o recebimento do valor de um identificador como *String*, na sequência tem-se a conversão do valor para inteiro e o tratamento de erros. Posteriormente o componente que irá mostrar o código identificador deve ser definido, para isso foi utilizada uma função *JavaScript*, chamanda *inicio()*, para capturar o valor e defini-lo no componente, fazendo o processo inverso, ou seja, conversão para *String*.

```
1.  <%
2.      Integer codigoIdentificador = null;
3.      String codTemp = request.getParameter("in_identificador");
4.      if(codTemp != null) {
5.          try{
6.              codigoIdentificador = Integer.parseInt(codTemp);
7.          catch(Exception e) {
8.              codigoIdentificador = -1;
9.          }
10.     }
11. %>
12. <body onLoad="inicio();" >
13.     <form >
14.         Identificador:
15.         <label type="text" id="identificador" ></label>
16.     </form>
17. </body>
18. <script>
19.     function inicio() {
20.         var id = <%=id %>;
21.         document.getElementById("identificador").value = "" +  id;
22.     }
23. </script>
```

Figura 4.10: Exemplo de validação no código JSP

Os conversores do JSF surgiram nessa problemática, para retirar do programador a responsabilidade sobre estas operações tão maçantes do JSP. Assim como nos validadores, existem conversores fornecidos por padrão no JSF, para os tipos mais comuns como: *java.util.Date*, *Integer*, *String*, *Double* e *Float*.

Entretanto, é permitido ao desenvolvedor construir seus próprios conversores, para uma necessidade específica.

Em JSF existem três formas de vincular um conversor a um componente. A primeira forma é adicionar o atributo *converter* na descrição dos componentes, neste caso deve-se passar o caminho da classe de conversão desejada. A segunda forma ocorre por meio da declaração de

uma *tag* para cada conversor de propósito específico. As duas primeiras formas são apresentadas na Figura 4.11 e Figura 4.12, respectivamente.

```
1.  <h:outputText value="#{TCC.horas}" converter="javax.faces.DateTime" />
```

Figura 4.11: Uso do atributo converter para converter o atributo horas (String) para uma saída formata em

```
1.  <h:outputText value="#{TCC.horas}">
2.      <f:convertDateTime />
3.  </h:outputText>
```

Figura 4.12: Conversão do atributo horas em String para uma saída formata em horas usando tags do conversor

Já a terceira forma utiliza a *tag converter*, Figura 4.13, onde o conversor é especificado com base no seu identificador (ID) registrado, identificador este, que pode ser padrão do JSF ou então especificado pelo desenvolvedor.

Segue abaixo uma lista de IDs dos conversores padrões existentes no JSF:

a) *javax.faces.DateTime* – usado pela *tag f:convertDateTime*;

b) *javax.faces.Number* – usado pela *tag f:convertNumber*;

c) *javax.faces.Boolean*, *javax.faces.Byte*, *javax.faces.Character*, *javax.faces.Double*, *javax.faces.Float*, *javax.faces.Integer*, *javax.faces.Long*, *javax.faces.Short*, *javax.faces.BigDecimal*, *javax.faces.BigInteger* – automaticamente usados para manipularem atributos no *BackingBean.*

```
1.  <h:outputText value="#{TCC.horas} >
2.      <f:converter converterId="javax.faces.DateTime"/>
3.  </h:outputText>
```

Figura 4.13: Conversão utilizando a tag converter

Assim como mostrado na seção de Validadores, o desenvolvedor pode criar conversores para uso específico, mas para isso o conversor deve ser registrado em arquivo de configuração. O classe é registrada através de um identificador (*converter-id*) e o caminho da classe (*converter-class*), no arquivo *faces-config.xml* como mostra a Figura 4.14.

```
1.  <converter>
2.      <converter-id>tcc.ConverteToHoras</converter-id>
3.      <converter-class>
4.          org.tcc.conversores.ConverteToHoras
5.      </converter-class>
6.  </converter>
```

Figura 4.14: Configuração de um validador no JSF

Na Figura 4.15, faz-se uso do validador registrado na Figura 4.14 anterior.

```
1.  <h:inputText>
2.      <f:converter converterId="tcc.ConverteToHoras"/>
3.  </h:inputText>
```

Figura 4.15: Trecho de código demonstrando uso de conversores personalizados

4.1.13 Mensagens

Um dos grandes problemas no desenvolvimento de interfaces do usuário está ligado à exibição de mensagens de erro. Os erros podem ser divididos em dois tipos: erros de aplicação (lógica de negócios, banco de dados ou erros de conexão, por exemplo) e erros de entrada de usuário (como texto inválido ou campos vazios) (MANN, 2005).

Erros de aplicação envolvem operações críticas para o sistema, ou seja, o sistema depende do sucesso destas para sua execução. Assim, podem ser geradas páginas totalmente diferentes da atual, informando o erro ou auxiliando nas medidas a serem tomadas.

Já para erros de entrada de usuário, na maioria das vezes a execução permanece na página atual, mas os campos com erros são destacados através de mensagens.

Vale destacar, que o tratamento de erro pode estar envolvido em muitas páginas, gerando a necessidade de seguir um padrão nas mensagens de erro.

No JSF, no decorrer do ciclo de vida, um objeto Java pode criar uma mensagem e adicioná-la na fila de mensagens do contexto JSF. A responsabilidade por exibir estas mensagens ao usuário fica a cargo da fase Renderização da Resposta. Geralmente estas mensagens estão associadas aos componentes JSF na tela (GEARY e HORSTMANN, 2010).

Geralmente o tipo de mensagem mais comum, é a de erro, porém, existem quatro tipos de mensagens no JSF:

a) atenção;
b) erro;
c) fatal;
d) informação.

Para exibir mensagens o JSF utiliza duas *tags*, *h:messages* e *h:message*. A primeira *tag*, *h:message*, mostra todas as mensagens armazenadas no *FacesContext* durante o ciclo de vida JSF. Estas mensagens podem não ser vinculadas a um componente diretamente, nesse caso o atributo *globalOnly* da *tag* deve conter o valor *true* (por padrão *false*).

Para exibição de mensagens para um dado componente, existe a *tag h:message*, onde é obrigatória a presença do atributo *for* que especifica através do id, o componente referenciado. Neste comando as mensagens não são cumulativas para cada componente, permanecendo somente a mensagem mais recente deste.

Através da Figura 4.16 é possível verificar o uso de mensagens específicas para cada componente *h:inputText* através do atributo *for* da *tag h:message*. Além disso, utiliza-se a *tag*

h:messsage para exemplificar o seu funcionamento. Tem-se dois campos de entrada com ids "entradaDados1" e "entradaDados2" respectivamente, e a restrição é definida por meio do atributo *required* (campo obrigatório) com valor igual à *true*.

```
1.      <h:form>
2.          <h:outputText value="Valor 1:" ></h:outputText>
3.          <h:inputText id="entradaDados1" required="true" ></h:inputText>
4.          <h:message for="entradaDados1"
5.                  style="color: red; font-weight: bold; font-style: italic;" >
6.          </h:message>
7.          <br></br>
8.          <h:outputText value="Valor 2:" ></h:outputText>
9.          <h:inputText id="entradaDados2" required="true" ></h:inputText>
10.         <h:message for="entradaDados2"
11.                 style="color: red; font-weight: bold; font-style: italic;" >
12.         </h:message>
13.         <br></br>
14.         <h:commandButton action="foi" type="submit" value="Executar Ação">
15.         </h:commandButton>
16.     </h:form>
17.     <h:messages ></h:messages>
```

Figura 4.16: Trecho de código para demonstrar o uso de mensagens globais e mensagens por componente

Através da Figura 4.17, pode-se verificar como são exibidos no navegador erros do tipo campo obrigatório. Como mostra a figura, ao lado de cada campo, há os componentes de mensagens específicos por componente, e mais abaixo, existe o componente de mensagem global, que como citado, agrupou todas as mensagens de erros da página.

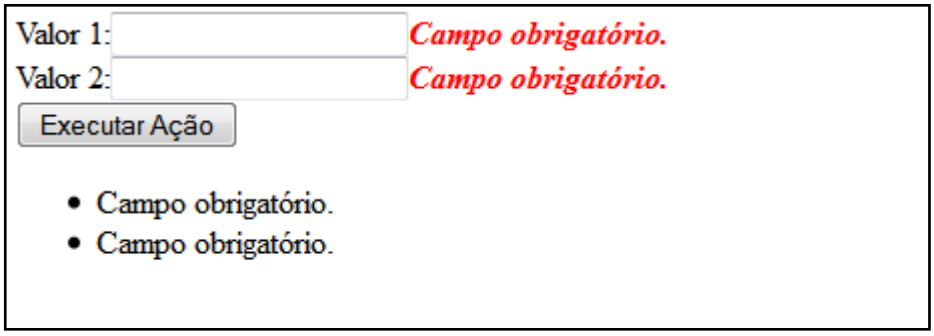

Figura 4.17: Figura com exibição de mensagens através dos comandos do JSF

4.2 Ciclo de Vida da Tecnologia JSF

O funcionamento do JSF é baseado num ciclo de vida composto por fases, que são executadas de maneira sequencial, indo desde a requisição até a resposta. Apesar do desenvolvimento no *Framework* JSF não necessitar do conhecimento de seu funcionamento interno, conhecer as fases do ciclo de vida, fornece base para a solução de erros de todo tipo e complexidade, visando a qualidade nas aplicações (MARAFON, 2006).

Segue abaixo a Figura 4.18, que mostra o funcionamento do ciclo de vida JSF, divido em suas seis fases:

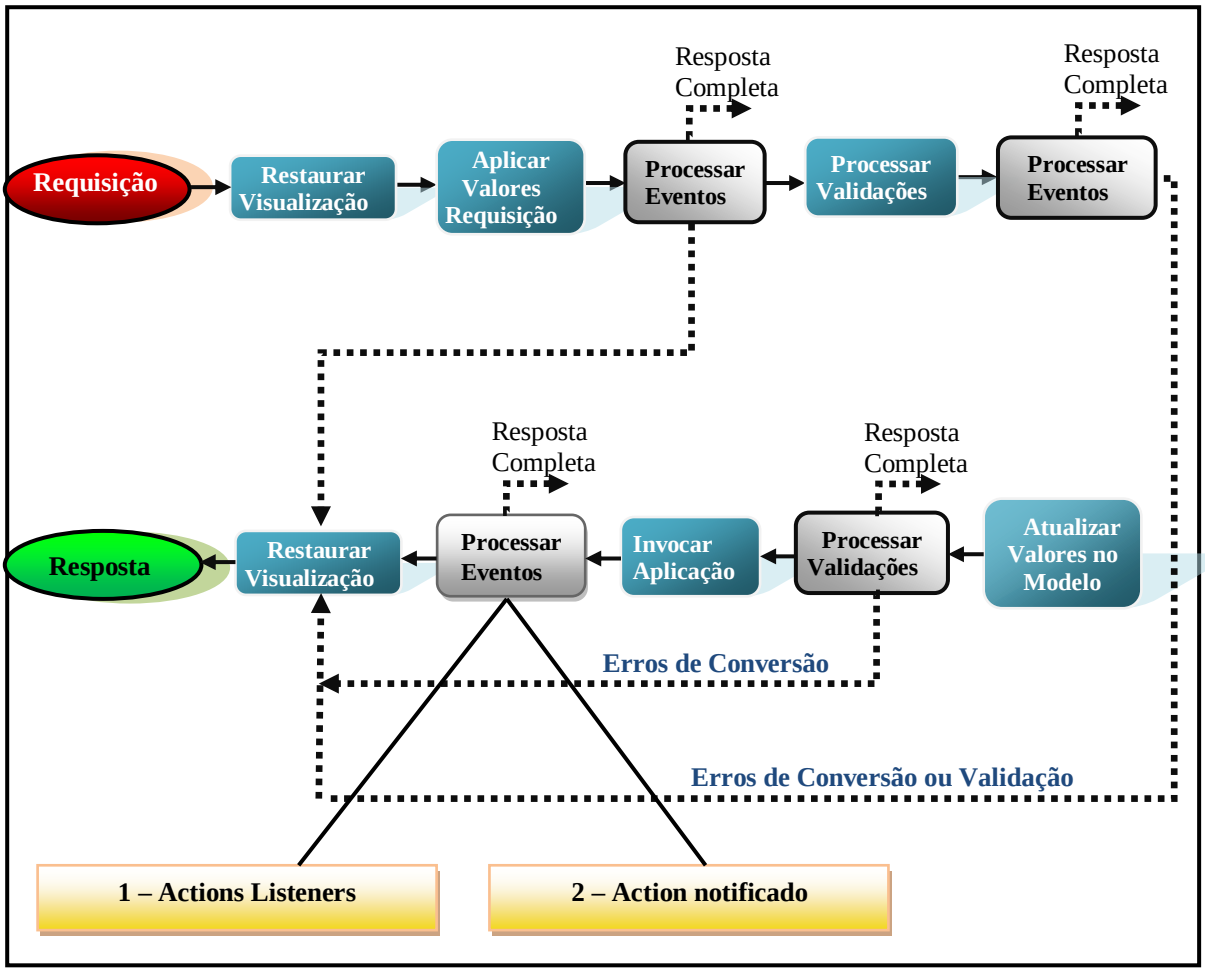

Figura 4.18: Ciclo de Vida do JSF (ORACLE, 2011)

4.2.1 Restauração da Visualização

Quando uma requisição é disparada ao JSF, esta é a primeira fase a ser executada pelo *FacesServlet*. Como citado anteriormente, o JSF armazena todas as informações relativas à uma página em sua *View ID*. Desse modo, nesta fase o *FacesServlet* analisa a requisição e a página solicitante para verificar se já existe uma *View ID* da página solicitante no *FacesContext*, caso contrário, é criada uma nova *View ID* para ela.

Logo, a esta fase é dada a responsabilidade de carregamento das informações preexistentes, ou seja, carregar a *View ID* gerada na última resposta da página, ou então gerar uma nova para o contexto da aplicação JSF. Assim, vale destacar que dentre os dados obtidos, estão à árvore de componentes, que contém todos os objetos da visualização e o estado da mesma.

4.2.2 Aplicação dos Valores da Requisição

Após a árvore de componentes ser restaurada, cada componente recebe o seu novo valor contido nos parâmetros da requisição, valor este, armazenado na própria instância do componente. Caso a conversão do valor falhar para o tipo de componente, é enfileirada no *FacesContext* uma mensagem de erro associada ao componente, que será exibida na fase de resposta *Render*.

Caso algum evento for enfileirado nesta fase, o JSF transmite os eventos para o *listeners* (ouvidores) associados.

Pode ainda, existir componentes com o atributo "immediate", definido com o valor booleano *true*. Quando isto acontece, o componente tem a validação, conversão e tratamento de eventos associados executados durante esta fase.

4.2.3 Processamento de Validações

Nesta fase o JSF executa as validações de todos os componentes contidos na árvore de componentes. A implementação examina os atributos que definem regras de validação nos componentes e compara se o valor armazenado localmente no componente atende estas regras.

Caso seja constatado um valor inválido, a aplicação JSF adiciona uma mensagem de erro à instância do *FacesContext*, e o ciclo de vida pode avançar para a fase de resposta *Render* para que a página seja recarregada e as mensagens de erro sejam exibidas. Além disso, se houver erros de conversão nos valores, serão exibidas mensagens para estes também.

Quando métodos de validação ou ouvidores de evento (*listeners*) chamam o método *renderResponse* da instância do *FacesContext*, é realizado um avanço para a fase de resposta *Render*. Entretanto, se a aplicação requerer recursos diferentes como páginas sem componentes JSF, deve ser chamado o método *FacesContext.responseComplete()*.

4.2.4 Atualização dos Valores no Modelo

Como essa fase é posterior à fase de validação dos valores, pode-se concluir que os valores locais de cada componente estão prontamente atualizados. Desse modo, cada componente pertencente à árvore, é atualizado com o seu valor local, o valor do atributo no *ManagedBean*/*BackingBean* o qual este referencia. São atualizados apenas os atributos do *BackingBean* referenciados. Além disso, do mesmo modo que na fase Processamento de

Validações, se os tipos de dados do componente forem incompatíveis com as propriedades no modelo, o ciclo de vida avança para a fase de resposta *Render*, para a exibição dos erros.

4.2.5 Invocação da Aplicação

Nesta fase é realizado o tratamento de eventos no nível da aplicação: lógica de interface e lógica de negócios nos *ManagedBeans*. Como na fase anterior, são carregados os valores dos componentes, nos atributos vinculados do *ManagedBean*, o mesmo pode então, realizar os processamentos sobre os dados, recebidos pela requisição.

Ainda nesta fase, podem ser definidas as páginas para qual para a requisição irá redirecionar, com base no retorno dos métodos e nas regras de navegação definidas no arquivo *faces-config.xml*.

4.2.6 Renderização da Resposta

No decorrer desta fase, a implementação JSF constrói a visualização e despacha aos recursos apropriados o comando para renderizar a página. Logo, a fase é composta de duas operações: renderizar uma resposta para o cliente e salvar o estado de cada componente dessa resposta na *View ID* associada.

4.3 AJAX

O AJAX traz grandes benefícios, pois permite que algumas partes do sistema sejam modificadas, sem que necessite de total carregamento de página novamente. Com isso, as operações no sistema ficam mais rápidas, o que torna o sistema mais dinâmico (MARAFON, 2006).

Asynchronous JavaScript com XML (AJAX) é uma tecnologia para atualização de regiões bem definidas nas páginas Web, sem necessidade de submissão do formulário inteiro e posterior renderização da resposta. É embutido na página Web código JavaScript para comunicar com o servidor e prover mudanças incrementais no estrutura da página (GEARY e HORSTMANN, 2010).

Algumas vantagens do Ajax se destacam (ORACLE, 2011):

a) validação dos dados nos campos do formulários em tempo real, sem a enviar todo o formulário para análise;

b) funcionalidades melhoradas para páginas Web, com modelo semelhante aos aplicativos *desktop*;

c) atualização parcial do conteúdo Web, evitando processamento e trafego de dados desnecessariamente.

Geary e Horstmann (2010) afirmam ainda, que no JSF 2.0 o desenvolvedor não necessita conhecer as complexidades da tecnologia Ajax para seu uso. Contudo, abaixo da versão 2.0 da tecnologia JSF esta tecnologia era fornecida por bibliotecas de componentes, um exemplo delas, é a Ajax4Jsf, que será detalhada na próxima seção. Esta biblioteca de suporte a Ajax forneceu um modelo de programação para que a tecnologia estivesse presente no JSF 2.0 da forma como está hoje, ou seja, uso transparente de uma tecnologia complexa (LOBO FILHO, 2010).

Na tecnologia JavaServer Faces o suporte a Ajax pode ser incorporada através de uma biblioteca de recursos JavaScript, que é fornecida no núcleo de bibliotecas. Sendo que esta incorporação pode ser feita das seguintes formas (ORACLE, 2011):

a) usando a *tag* f:ajax, embutida com outro componente padrão em um aplicativo JSF. Esse elemento adiciona suporte a Ajax em qualquer componente da interface do usuário sem configurações adicionais;

b) ou então, usando o método `jsf.ajax.request()` da API JavaScript diretamente na aplicação JSF, fazendo com que o comportamento do componentes sejam personalizados com Ajax.

4.4 Bibliotecas de Componentes

4.4.1 Introdução

Focada no modelo de programação baseada em componentes, a implementação JSF trouxe maior produtividade no desenvolvimento de interface Web, uma vez que cada um dos componentes padrões do HTML possui um componente representante no JSF. Estes componentes fornecem uma base inicial para a construção de aplicações. Contudo, as necessidades podem ir além desse conjunto básico de componentes, o que leva os desenvolvedores a criarem seus próprios componentes personalizados (CONCEIÇÃO, 2008).

Isso, no entanto, faz com que existam desenvolvedores dedicados à criação e manutenção de componentes. Para solucionar esse problema, tem-se o surgimento das bibliotecas de componentes JSF, deixando à disposição uma gama de componentes genéricos e reutilizáveis (OLIVEIRA e PAULA, 2009). Sendo que, estas são desenvolvidas por grandes empresas, o que garantem evolução, confiabilidade e qualidade dos componentes.

4.4.2 Richfaces

Richfaces é um *framework* livre que provê suporte AJAX nos componentes JSF. Além disso, contém um conjunto rico de componentes visuais, para auxiliar os desenvolvedores.

Este *framework* tem sua origem no framework Ajax4jsf e na biblioteca de componentes Richfaces. O Ajax4jsf foi desenvolvido por Alexander Smirnov em 2005, enquanto Jesse James Garrett estabelecia o conceito de Ajax. Alexander decidiu criar um projeto no "sourceforge.net"[4], onde uniu o JSF, que estava em crescente utilização, com a tecnologia Ajax. Em outubro do mesmo ano, Smirnov Exadel uniu-se a Alexander para o desenvolvimento do *Framework (Exadel apud* (GUTIERREZ, 2009)).

O objetivo de Smirnov era que o *framework* pudesse ser usado em diferentes bibliotecas de componentes. A primeira versão foi lançada em 2006, e não era independente, era parte de um produto chamado Exadel Richfaces. Sendo que, no mesmo ano houve a separação do produto em Richfaces e Ajax4jsf. O Richfaces era uma biblioteca de componentes para os mais diversos usos. Já o Ajax4jsf era um *framework* que dava suporte Ajax as páginas JSF (KATZ, 2008).

Segundo o autor Smirnov, desse momento em diante o Richfaces tornou-se uma biblioteca comercial de componentes JSF, e o Ajax4jsf tornou-se um projeto *open-source*. Contudo, em 2007 tanto o Ajax4jsf quanto o Richfaces passaram a ser mantidos pela JBoss como projetos livres. Além disso, a mesma empresa decidiu integrá-los, para evitar problemas de incompatibilidades de versões.

O Richfaces contém mais de cem componentes com suporte a Ajax à disposição. Várias inovações do Richfaces foram reunidas na especificação do JSF 2 para beneficiar os desenvolvedores. Em sua versão 4.0, o Richfaces além de dar suporte ao JSF, estendeu vários recursos deste para prover melhorias como: inclusão de usabilidade, aperfeiçoamento de

[4] O portal sourceforge.net é o maior repositório de projetos livres do mundo (sourceforge).

desempenho, recursos dinâmicos, *Skinning* e desenvolvimento de componentes (JBossRichfaces).

Alguns dos benefícios que o *framework* Richfaces trouxe se destacam, como: não despender tempo na escrita de código JavaScript evitando manutenção desses códigos e maior compatibilidade entre navegadores e manutenção de código JavaScript.

4.4.2.1 Usando o Richfaces

Nas versões anteriores à versão 4 do Richfaces, eram necessárias algumas configurações no arquivo descritor de *deploy* (web.xml) para utilizar o *framework*. Nesta última versão, assim que adicionado os arquivos JAR da implementação e suas dependências, o desenvolvedor já pode utilizá-lo. Contudo, se o mesmo desejar personalizar as configurações padrões do Richfaces, pode fazer isso nos arquivos web.xml e faces-config.xml.

Através da Figura 4.19 verifica-se a declaração das bibliotecas de componentes Richfaces, estas são necessárias em cada página XHTML onde ocorrer o uso dos mesmos.

```
1.  <ui:composition
2.      xmlns:a4j="http://richfaces.org/a4j"
3.      xmlns:rich="http://richfaces.org/rich">
4.      ...
5.  </ui:composition>
```

Figura 4.19: Modelo de declaração das “taglibs” do Richfaces, para uso dos componentes

A Figura 4.20 exemplifica como é modificada a Skin padrão dos componentes visuais do Richfaces. A declaração da figura, altera o nome da Skin padrão para a “blueSky”, uma das Skins disponíveis por padrão no *framework*. Todavia, esta pode ser criada pelo desenvolvedor e chamada da mesma forma. Além disso, no termo “blueSky” poderia ser substituído por uma EL do tipo, #{nomeManagedBean.nomeSkin}, onde o nome da Skin poderia ser modificado em tempo de execução.

```
1.  <context-param>
2.      <param-name>org.richfaces.skin</param-name>
3.      <param-value>blueSky</param-value>
4.  </context-param>
```

Figura 4.20: Exemplo de personalização da Skin do Richfaces

Vale destacar ainda, que as propriedades dos estilos numa Skin, não seguem apenas CSS padrão. Na sua criação pode ser adicionado arquivo ECSS, que são arquivos CSS com suporte a uso de EL. Com isso, há um aumento significativo das possibilidades de modificações dinâmicas dos componentes de interface em termo de execução.

Como citado na seção anterior, o Richfaces possui um conjunto muito grande de componentes, porém, será mostrados apenas o componente rich:dataTable na sequência, onde os demais seguem o mesmo princípio lógico.

O componente a4j:dataTable é um componente que objetiva a exibição de dados em tabela, assim como faz o componente padrão do JSF, h:dataTable. No entanto, este componente possui muito mais recursos, como: uso de Skin na tabela e em seus filhos, ordenação e filtragem de colunas, uso de Ajax para atualização dos dados e paginação embutida.

O a4j:dataTable para exibir os dados, acessa através do seu atributo "value", uma lista de objetos presente no *managedBean*. Esta lista é iterada, e cada objeto é enviado no momento do processamento do componente, para uma variável de nome declarado no atributo "var" deste mesmo componente. Para acessar as propriedades de cada objeto da lista de objetos, usa-se uma EL com o formato #{nomeVariavel.nomePropriedade}.

A Figura 4.21 exemplifica isso. Onde o componente irá iterar os objetos da lista "listaUsuarios" jogando cada componente na variável "usuario". Apenas uma coluna será exibida através da *tag* rich:column, que possui internamente a declaração de seu cabeçalho através da linha 3 e na linha 6 que represente a propriedade a ser exibida, no caso o nome do usuário presente no modelo.

```
1.  <rich:dataTable value="#{gerenciaUsuariosBean.listaUsuarios}" var="usuario">
2.      <rich:column>
3.          <f:facet name="header" >
4.              <h:outputText value="Nome:" ></h:outputText>
5.          </f:facet>
6.          <h:outputText value="#{usuario.nome}" ></h:outputText>
7.      </rich:column>
8.  </rich:dataTable>
```

Figura 4.21: Trecho de código que exibe o nome de usuários numa tabela com o componente a4j:dataTable do Richfaces

Outro aspecto importante é o uso do Ajax através do Richfaces. No Richfaces versão 4, a *tag* que prove suporte do Ajax aos componentes JSF, é a "a4j:ajax". Como expresso na Figura 4.22, o componente padrão do JSF, "inputText", referencia um propriedade "nome" no *managedBean* "usuarioBean". Este componente, no fluxo padrão do JSF, ou seja, sem uso de Ajax, faria a submissão do valor do campo na página para a propriedade "nome", apenas após um *Submit* do formulário inteiro. Entretanto, adicionando o código da linha três, determina-se qual evento (event="") do componente irá disparar a submissão do valor para a propriedade.

Além disso, na mesma *tag* através do atributo *render*, é determinado através de Ids quais componentes serão renderizados novamente ao final da requisição Ajax, no caso da Figura 4.22, o componente "outputText" com Id "saidaTexto", mostra o valor da propriedade #{usuarioBean.nome} após a requisição Ajax.

```
1.  <h:form id="form1" >
2.      <h:inputText id="entradaNome" value="#{usuarioBean.nome}">
3.          <a4j:ajax event="keyup" render=" saidaTexto " />
4.      </h:inputText>
5.      <h:outputText id="saidaTexto" value="#{usuarioBean.nome}" />
6.  </h:form>
```

Figura 4.22: Trecho que demonstra como é dado aos componentes JSF o suporte a Ajax

Capítulo 5

Implementação de Estudo de Caso

Feita a exposição do problema na reutilização de interfaces JSF, se faz necessário uma maneira de avaliar a aplicação de *framelets* num sistema que resolva essa alta dependência advinda do uso do *framework* JSF em sistemas Web.

Assim, optou-se pelo desenvolvimento de um estudo de caso, que consiste na construção de uma estrutura desenvolvida na linguagem Java, mas que possa se adaptar dinamicamente às configurações realizadas pelo reutilizador da mesma.

Inicialmente surgiram duas áreas de domínio que esta estrutura poderia atender. A primeira seria a construção de uma implementação que gerasse interfaces Web e adaptasse as regras de negócio para as seguintes operações básicas em banco de dados relacionais: criação, consulta, atualização e remoção de dados. Contudo, o desenvolvimento desta solução foi descartado, devido à sua amplitude, que poderia resultar em dificuldades fora do contexto deste trabalho. Mais detalhes sobre a mesma serão mostrados na seção 7.1.

Já a segunda área de domínio, baseia-se no desenvolvimento de uma estrutura que será chamada neste trabalho de **Módulo de Relatório Reutilizável (MRR)**. Este módulo oferece a geração de uma interface Web voltada para exibição dos resultados de uma consulta SQL em banco de dados, ou seja, um relatório completo da consulta é gerado através do módulo. Dentre seus recursos então: funcionalidades de filtragem de dados, exibição de operações de totalização, paginação dos resultados e seções do relatório configuráveis pelo reutilizador.

Ambas as áreas são muito frequentes nos mais diversos sistemas, logo, implementar reuso nelas pode resolver um problema contínuo de desenvolvimento de uma mesma funcionalidade para diferentes aplicações.

O desenvolvimento do estudo de caso teve por objetivo elencar quais os pontos benéficos e difíceis de desenvolver esta estrutura reutilizável em JSF. Com isso, buscando mostrar as limitações encontradas para o desenvolvimento da mesma, a reutilização na prática e principalmente como as características dos *framelets* foram vantajosas para a construção do **Módulo de Relatório Reutilizável**.

O projeto Web do **Módulo de Relatório Reutilizável** foi desenvolvido na linguagem Java com uso dos *frameworks* JSF 2.0 e JBoss Richfaces 4.0 sobre a IDE NetBeans 7.0 e com uso do servidor de aplicação GlassFish 3.1. O módulo de persistência vinculado foi projetado para suportar os mais diversos bancos de dados relacionais, pois, utiliza a tecnologia JDBC. Neste estudo de caso, fez-se uso apenas do SGBD Postgres 9, visto que, não é foco do trabalho analisar o módulo sobre diferentes SGBDs.

Desse ponto em diante, o termo **Módulo de Relatório Reutilizável** será referenciado como **MRR**, para não tornar muito repetitiva as citações do nome completo.

5.1 Metodologia de Reutilização Aplicada

Existem duas formas dos componentes da implementação JSF se comunicar com as regras de negócios de uma aplicação. Isso pode ocorrer através das expressões de linguagem, já citadas na seção 4.1.6 (Expressões de Linguagem (EL)), ou então, os dados podem ser acessados diretamente por uma classe Java, pois, cada componente JSF possui uma instância de classe associada conforme seu tipo. Desse modo, tais componentes são mantidos na memória do servidor enquanto uma página (*View*) está sendo acessada.

Com base nestas duas formas, pode-se também elencar duas estratégias para implementar reutilização de interfaces de usuário em JSF: personalização de páginas prontas ou a adaptação de componentes criados em regras de negócios.

Na primeira estratégia, nas aplicações construídas com *frameworks* Java para Web as telas são criadas em meio a código HTML e uso de *tags* que representam os componentes do *framework*. No caso do JSF ocorre o mesmo, cada componente pode ser acessado através do objeto *FacesContext*, citado na seção 4.1.4, que engloba todos os componentes da tela JSF.

Portanto, para realizar as adaptações dinâmicas necessárias para cada diferente domínio, cada componente adaptado deve ter um *id* pré-definido, que será usado como chave para buscar o objeto que o representa. Uma desvantagem é que os componentes são primeiramente carregados, e na sequência ocorrem os processamentos para adaptá-los ao contexto aplicado.

O que também torna esta tarefa complexa é o fato de que todos os *ids* devem ser conhecidos e as configurações da interface devem ser feitas sobre os componentes, e não sobre as regras de negócio que os geram.

Além disso, aplicar mudanças numa página pronta pode acarretar em uma baixa flexibilidade, visto que cada elemento da interface Web, deve ser mapeado para um elemento do domínio em que será reutilizado.

Uma segunda estratégia baseia-se num recurso que o *framework* JSF prove, que é o chamado "*binding*". O *binding* é um atributo dos componentes JSF que permite realizar uma ligação dinâmica entre um componente da interface Web, com uma propriedade do *managedBean*. A Figura 5.1 mostra a declaração de um formulário do JSF (h:form) que faz referência a um método que gera o componente em tempo de execução. Já a Figura 5.2 mostra a classe "ControladorMG" e a declaração do método "gerarForm", responsável por criar o componente UIForm, e posteriormente mudar do estilo CSS da largura para "400px".

```
1.
2.      <h:form  binding="#{controladorMG.gerarForm}" ></h:form>
3.
```

Figura 5.1: Exemplo de binding de um componente JSF h:form

```
1.      import  javax.faces.component.UIForm ;
2.
3.      public class ControladorMG {
4.              private UIForm formulario;
5.
6.              public  UIForm gerarForm() {
7.                      this.formulario = new UIForm();
8.                      this.formulario.setStyle("width: 400px; ");
9.
10.                     return this.formulario;
11.             }
12.     }
```

Figura 5.2: *ManagedBean* ControladorMG exemplificando o *binding* de componentes JSF

Por este ser um recurso padrão da implementação, com o *binding* todos os componentes gerados e personalizados nas classes Java podem ser implantados em qualquer aplicação cliente que esteja usando o *framework* JSF.

O recurso principal para consolidar a afirmação anterior é o "append" (anexo) de componentes, ou seja, cada componente a quem é permitido ter outros componentes internos, possuem uma lista de componentes filhos, obtida pelo método "getChildren". Esta lista pode ser preenchida com os elementos criados nas regras de negócio de um "módulo gerador da interface", a maior vantagem dessa operação é que cada parte da interface Web a ser reutilizada, implementa a interface UIComponent, assim pode ser adicionada em interfaces JSF diretamente.

No estudo de caso deste trabalho, utilizou-se a segunda estratégia de reutilização, buscando uma maior flexibilidade na implementação e os seguintes resultados: possibilidade de aplicar padrões de projeto, modularização da estrutura reutilizável e uma maior facilidade na manutenção e evolução da interface Web.

5.2 Requisitos Funcionais e Casos de Uso

O MRR fornece os recursos de geração e visualização de dados em relatório. Esta seção objetiva mostrar quais os requisitos funcionais desta estrutura e suas relações com os submódulos dos *framelets*.

Os requisitos funcionais estão dispostos em categorias, pois, o relatório é dividido visual e estruturalmente em: cabeçalho, conteúdo e rodapé. Da mesma forma, o MRR se utiliza de *framelets* para a geração e manipulação destas partes dos relatórios.

5.2.1 Cabeçalho

Um cabeçalho se localiza no topo de um relatório, e mostra informações que geralmente auxiliam na identificação dos dados expressos no mesmo. O MRR possui um *framelet* chamado FrameletCabecalho, que é responsável pela geração do cabeçalho e de suas colunas, bem como a exibição de operações de totalização e o tratamento das operações de ordenação e filtragem.

Tabela 5.1: Requisito Funcional **Título para Coluna**

RF-01 Especificar título para coluna	
Descrição	O Módulo de Relatório Reutilizável deve permitir ao reutilizador especificar um título personalizado para cada coluna da consulta SQL.
Pré-condições e Entradas	Esta configuração deve ser feita num *managedBean* que herdou a classe principal do MRR, NucleoRelatorio.
Saída	Para cada coluna configurada aparece o título especificado no topo da página.

Tabela 5.2: Requisito Funcional **Campo de Filtragem para Coluna**

RF-02 Configuração de Campo de Filtragem para Coluna	
Descrição	O MRR deve permitir ao reutilizador especificar um tipo de filtro pré-configurado para que os dados de uma coluna do relatório possam ser filtrados em tempo de execução.
Pré-condições e Entradas	Esta configuração deve ser feita num *managedBean* que herdou a classe principal do MRR, NucleoRelatorio. Especificação da SQL de consulta, e a correta configuração da coluna.
Saída	Cada diferente filtro possui seus campos de filtragem modelados. Logo, a coluna que configurar um determinado filtro vai possui os campos correspondentes dele logo abaixo do seu título.

Tabela 5.3: Requisito Funcional **Totalização de Dados**

RF-03 Configurar Operações de Totalização de Dados	
Descrição	É muito comum em relatórios, operações de totalização de dados, como a soma, média, menor ou maior valor, entre outras. O MRR deve possibilitar ao reutilizador definir operações de totalização e um título para elas, para exibição no cabeçalho.
Pré-condições e Entradas	Esta configuração deve ser feita num *managedBean* que herdou a classe principal do MRR, NucleoRelatorio. Especificação da SQL de consulta, e a correta configuração da coluna.
Saída	A coluna que possui operações configuradas irá exibir o título da operação seguido do resultado dela, na barra de operações abaixo da barra de filtros.

Tabela 5.4: Requisito Funcional **Ordenação na Coluna**

RF-04 Configurar Ordenação de Dados na Coluna	
Descrição	O MRR deve permitir ao reutilizador especificar se a coluna configurada poderá ser ordenada no relatório.

Pré-condições e Entradas	Esta configuração deve ser feita num *managedBean* que herdou a classe principal do MRR, NucleoRelatorio. Especificação da SQL de consulta, e a correta configuração da coluna.
Saída	Cada coluna com ordenação habilitada irá possuir um conjunto de três botões abaixo da barra de operações. O primeiro é para ordenação ascendente, o botão do centro é para limpar ordenação da coluna e o último tem como função a ordenação descendente.

Tabela 5.5: Requisito Funcional **Estilos e Classes CSS personalizados para Colunas**

RF-05 Estilos e Classes CSS, personalizados para Colunas	
Descrição	O MRR deve permitir ao reutilizador especificar estilos e classes em CSS, para que o cabeçalho e o conteúdo de uma coluna possam ser alterados visualmente. Ex.: plano de fundo, cor da borda, tamanho de fonte.
Pré-condições e Entradas	Esta configuração deve ser feita num *managedBean* que herdou a classe principal do MRR, NucleoRelatorio. Esta funcionalidade é dependente da renderização final das partes do relatório.
Saída	Existem estilos e classes padrão no sistema, mas as colunas com estilos personalizados modificam estas configurações pré-definidas.

5.2.2 Conteúdo

O conteúdo mostra os dados obtidos na consulta SQL ao banco de dados, na mesma ordem das colunas presentes no cabeçalho, e com um número máximo de resultados (linhas), que também pode ser alterado em tempo de execução. O MRR se utiliza de um *framelet* chamado FrameletConteudo, que realiza as consultas de geração dos resultados e ainda faz a criação da representação visual dos mesmos no relatório.

Tabela 5.6: Requisito Funcional **Selecionar Linha do Relatório**

RF-06 Selecionar de linha do relatório	
Descrição	O usuário do relatório pode selecionar uma linha do mesmo apenas clicando sobre ela, e retirar a seleção clicando novamente sobre a

	mesma. Está operação é útil quando se deseja destacar uma linha em um relatório com muitas colunas.
Pré-condições e Entradas	Esta funcionalidade não possui configurações.
Saída	O usuário ao clicar em uma linha, a mesma deve ser destacada com a cor verde limão. Para desmarcar a seleção, deve-se clicar sobre a linha.

Tabela 5.7: Requisito Funcional **Ocultação de linha do relatório**

RF-07 Ocultação de linha do relatório	
Descrição	O usuário do relatório pode querer ocultar determinadas linhas do relatório, para isso, na última coluna do relatório existe um botão para a ocultação de determinada linha. Para exibir todas as linhas ocultadas, buscar clicar sobre a frase "Mostrar Tudo" na coluna onde os botões estão presentes.
Pré-condições e Entradas	Esta funcionalidade não possui configurações.
Saída	O usuário ao clicar no botão ocultar, a linha deve sumir da visualização, quando executado o comando para mostrar novamente, a linha deve aparecer, mas, também ser destacada (cor verde limão) como seleção.

5.2.3 Rodapé

O rodapé, por sua vez, é composto por uma barra de controle, onde é possível alterar o número de resultados por página, visualizar os contadores de resultados mostrados e navegar entre as páginas do relatório. Para a construção da interface e manipulação das operações do rodapé, o MRR se utiliza de um *framelet* chamado FrameletRodape.

Tabela 5.8: Requisito Funcional **Adicionar elementos personalizados**

RF-08 Adição de elementos personalizados	
Descrição	É permitido ao reutilizador adicionar elementos personalizados que implementam

	a interface “IRepresentacao”, no início da barra de rodapé, as opções são: texto simples, código HTML ou componente JSF. Com isso, é possível, por exemplo, adicionar um título no relatório ou então um botão de ação.
Pré-condições e Entradas	Esta configuração deve ser feita num *managedBean* que herdou a classe principal do MRR, NucleoRelatorio, com a adição dos códigos de elementos extras do rodapé.
Saída	Cada elemento personalizado deve aparecer antes dos elementos básicos do rodapé (resultados e paginação).

Tabela 5.9: Requisito Funcional **Alterar número de resultados por página dinamicamente**

RF-09 Alterar número máximo de resultados por página	
Descrição	O relatório irá prover ao usuário uma opção para aumentar ou diminuir a quantidade de resultados exibidos na página.
Pré-condições e Entradas	Esta funcionalidade não possui configuração.
Saída	O rodapé deve conter um *menu* de seleção que exiba o atual número máximo de resultados na página, e que permita alterar esse valor.

Tabela 5.10: Requisito Funcional **Alterar página atual do relatório**

RF-10 Alterar a página atual do relatório	
Descrição	O relatório deve exibir ao usuário opções para mudança de página como: ir para a primeira página, ir para a anterior, avançar uma página e ir para a última página.
Pré-condições e Entradas	Esta funcionalidade não possui configuração.
Saída	O rodapé deve conter botões expressos por imagens que identificam claramente à quais das operações anteriores os botões estão vinculados.

No estudo de caso foram desenvolvidos todos os casos de uso que referenciam os requisitos funcionais ilustrados acima.

A Figura B.1 ilustra o diagrama de casos de uso que o MRR buscou atender. Como pode ser visto cada ator representa um segmento com função bem definida em um relatório. Pelo fato do MRR ser um módulo reutilizável e não uma aplicação de uso final, buscou-se uma representação onde os atores são entidades que intermediam suas funcionalidades com a aplicação cliente, mas para isso, os mesmos comunicam-se com o MRR para a execução dos casos de uso. Além disso, os casos de uso do MRR foram representados de duas formas, casos de uso de acesso direto (cor clara) e casos de uso interno (cor escura).

A primeira são as funcionalidades que os usuários terão acesso no relatório gerado pela aplicação cliente, isto é, as operações finais de um relatório. Agora, a segunda forma representa os casos de uso internos, que são as operações que um ator pode realizar para consolidar outras operações dependentes ou relacionadas à mesma. Então, um exemplo disso, é o caso de uso Ordenar Coluna que está relacionado ao Exibir Dados, quando o mesmo é acionado é necessário reexibir os dados do conteúdo, mas não existe uma operação e nenhum comando de acesso direto para esta operação.

Considerou-se também a criação do ator Relatório, por meio do qual as aplicações clientes realizam a configuração do relatório e a geração da interface final, sendo este ator dependente de todos os demais.

5.3 Implementação

Em larga escala, uma aplicação é construída utilizando *framelets* como componentes caixa-preta. Já em pequena escala, cada *framelet* é um pequeno *framework* caixa branca (PREE, 1999). Neste trabalho, o estudo de caso representa um pequeno domínio de aplicação, ou seja, a família de *framelets* utilizada no mesmo está voltada apenas para a construção de relatórios Web para uma consulta SQL específica.

Os *frameworks* geralmente são construídos sobre uma arquitetura flexível orientada a objetos. No planejamento da arquitetura do estudo de caso, buscou-se que o MRR preservasse a compatibilidade com o *framework* JSF, pois assim poderia ser utilizado de maneira natural nas aplicações.

Esta seção, objetiva mostrar detalhes arquiteturais da estrutura criada para estudo de caso, assim como quais estratégias foram utilizadas para o desenvolvimento de uma estrutura reutilizável em diferentes domínios, que busca diminuir o acoplamento com a aplicação cliente.

5.3.1 Planejamento da Implementação

Como já citado acima, foi escolhida a segunda forma ("*binding*") para reutilização no JSF, então, se fez necessário encontrar uma melhor forma de projetar a arquitetura dos elementos da implementação, para o efetivo uso na aplicação cliente.

Existem duas maneiras conhecidas de reutilização de objetos. A primeira é através da delegação, onde é declarado numa classe, objetos das classes reutilizadas, então são delegadas operações aos métodos delas. A segunda forma é através da herança de objetos, onde uma classe herda (reutiliza) de uma classe "mãe", todos os métodos e atributos desta (GURP e BOSCH, 2001).

Foram aplicadas ambas as estratégias em diferentes regiões da implementação. Para a utilização do MRR é necessário a criação de um *managedBean* na aplicação cliente, e este objeto pode herdar a classe NucleoRelatorio, que centraliza todas as funcionalidades do MRR. Ao invés disso, o *managedBean* criado pode se utilizar de delegação declarando um objeto de classe que herde a classe NucleoRelatorio.

Já no interior da classe NucleoRelatorio, cada grande funcionalidade foi representada por um *framelet*, como cada um deles implementam uma mesma interface, isto buscou permitir maior escalabilidade e flexibilidade da estrutura. Cada um dos *framelets* se acoplou ao módulo por meio de delegação, isto é, os módulos que estão habilitados no sistema são adicionados na classe principal como *framelets*, e seus tipos definem quais seções (cabeçalho, conteúdo e rodapé) eles podem assumir. Para permitir os relacionamentos entre os *framelets* e conseqüentemente entre as funcionalidades do relatório, cada um deles, ao serem criados recebem uma instância da classe NucleoRelatorio, a principal do MRR, delegando assim operações do relatório aos *framelets*.

Para permitir um maior nível de reutilização dos componentes embutidos no relatório em aplicações JSF, cada tipo de elemento do relatório (texto, html (estático ou dinâmico) e JSF) implementa a interface UIComponent do JSF. Desse modo, os mesmos podem ser

adicionados facilmente em qualquer outro componente padrão da implementação JSF, onde se buscou aumentar o grau de compatibilidade MRR com interfaces de outras aplicações.

Portanto, o MRR foi estruturado em sete *framelets*, onde cada um é responsável por uma seção do sistema ou por fornecer serviços a elas. Na sequência serão citadas as funcionalidades dos *framelets* e descrito as principais classes e relacionamentos entre elas com base nos diagramas de classe de cada um.

5.3.1.1 FrameletRelatorio

Este *framelet* é composto pelas classes NucleoRelatorio e Coluna, e pela interface IFramelet. A classe NucleoRelatorio centraliza todas as configurações do MRR e as operações do relatório, nela também são instanciados os *framelets* de cabeçalho, conteúdo e rodapé. Além disso, cada *framelet* recebe uma instância dessa classe, para que assim se comuniquem entre si.

A classe NucleoRelatorio possui uma lista de módulos (*framelets*) onde é adicionado cada um deles. Posteriormente é reconhecido qual o tipo de serviço que eles fornecem. Todo esse processo ocorre sobre as interfaces (IFramelet, ICabecalho, IConteudo e IRodape), como mostrado na Figura C.1.

A interface IFramelet é a base para a construção de cada estrutura das partes do sistema, através da implementação do método `public UIComponent processarColunas()` em cada *framelet* existe uma única forma de renderizar os componentes da interface. Também é mostrado na figura o relacionamento que expressa que um objeto NucleoRelatorio pode possuir várias Colunas, sendo que cada uma tem à disposição uma gama de atributos a ser configurados conforme mostrado no modelo.

A classe UnidadeOrdenacao é a última deste modelo e representa uma configuração de ordenação de coluna, que pode ser efetuada configurando ordenação fixa em determinadas colunas na classe do relatório, ou então, em tempo de execução ao efetuar uma ordenação de dados;

5.3.1.2 FrameletRepresentaDados

No projeto do relatório que é gerado pelo MRR criou-se uma estrutura que representa diferentes tipos de dados. Como o relatório é voltado para sistemas Web é essencial que

represente: valores simples, código HTML estático e dinâmico, e os componentes padrões do *framework* JSF.

Nesse contexto, o FrameletRepresentaDados é um *framelet* para a integração desse tipo de representações nos módulos do relatório. O mesmo é composto pela interface IRepresentaDados, e por suas implementações RepresentaTexto, RepresentaHTML, RepresentaHTMLDinâmico e RepresentaJSF, e uma última classe DescritorValorDinamico, usada em representações dinâmicas para repassar dados ao código gerador.

O diagrama de classe da Figura C.2 mostra que a interface IRepresentaDados determina que o método `public UIComponent retornarValor()` seja implementado em cada classe concreta. Da mesma forma que foi citado no tópico anterior, aqui tem-se uma única forma para a renderização de diferentes tipos de representação. Pode-se notar também que as classes com função mais simples (RepresentaTexto e RepresentaHTML), apenas recebem os valores estáticos e fazem a geração final.

Porém, as classes RepresentaHTMLDinamico e RepresentaJSF, possuem uma lista de objetos do tipo DescritorValorDinamico. Sendo que, na geração dos dados, ou seja, na chamada do método `retornarValor`, esta lista é interada e obtidos os valores das colunas descritas nesta configuração na consulta SQL.

Posteriormente, o código de geração de HTML ou JSF que foi criado pelo reutilizador do MRR, será processado e as "lacunas" do mesmo serão preenchidas com os valores obtidos na consulta anterior;

5.3.1.3 FrameletFiltro

Assim como o *framelet* anterior, para o segmento de filtros do relatório, o MRR é estruturado de forma que podem ser criados vários filtros através de um artefato base. No caso, a classe abstrata AFiltro desempenha este papel, pois possui os atributos básicos de um filtro e os métodos de tratamento do valor filtrado, além da adição do mesmo na SQL de consulta para a efetiva filtragem.

A Figura C.3 mostra que a classe AFiltro especifica quais os tipos de filtros por meio de constantes e também, contém métodos básicos para verificar se o filtro está ativado, para desabilitá-lo, para construir a SQL de filtragem e o método `public UIComponent`

`representacaoFiltro(Nucleo nucleo)`, que é responsável pela criação de cada tipo de filtro.

O MRR possui três implementações de filtro conforme mostrado no diagrama, a seguir são mostradas as classes e funções correspondentes a cada um deles:

a) FiltroIlike – busca o texto digitado pelo usuário;

b) FiltroSelecaoUnica – seleção de um único valor dentre todos os presentes na tabela do banco de dados;

c) FiltroSelecaoMultipla – semelhante ao anterior, mas permite múltiplos dados selecionados.

5.3.1.4 FrameletPersistencia

Cada seção de um relatório em diferentes momentos e finalidades deve ter acesso ao banco de dados, seja para exibir os dados principais, ou então, para operações adicionais como filtragem, ordenação e paginação.

Nesse sentido, o FrameletPersistencia é um *framelet* voltado para acessos ao banco de dados. No mesmo foram utilizados os padrões Factory e Singleton (GAMMA, 1995), para a configuração e geração da conexão, e para definição de apenas uma fábrica de conexões (classe FabricaConexao), respectivamente.

Através do diagrama de classe (Figura C.4) do frameletPersistencia verifica-se que uma conexão é representada pela interface AConexao, sendo que esta classe possui um objeto de conexão (java.sql.Connection) e um objeto de transação (java.sql.PreparedStatement), que são manipulados em suas implementações.

Para a criação de uma conexão o objeto da classe ConfiguracaoBD deve ser configurado e enviado ao objeto da classe FabricaConexao que fará a geração de uma instância que implemente AConexao.

No caso deste trabalho, a conexão é representada pela instância da classe PostgresConexao;

5.3.1.5 FrameletCabecalho

Este *framelet* é um integrador de funcionalidades do cabeçalho, onde são criados os componentes de interface com base nas configurações feitas no FrameletRelatorio. Isso abrange a criação dos títulos, criação dos filtros (frameletFiltro), exibição das operações (classe Operacao) e criação dos ordenadores (classe UnidadeOrdenacao).

O diagrama de classe deste *framelet* representado na Figura C.5, mostra que o cabeçalho é representado pela interface ICabecalho (que herda a interface IFramelet), e sua classe concreta, Cabecalho. No entanto, o relacionamento com os demais *framelets* do MRR não são diretos, sendo a classe NucleoRelatorio o único ponto de acesso a eles.

O cabeçalho é composto ainda pelas classes Operacao e UnidadeOrdenacao. A primeira é responsável pelo processamento das operações e também por exibir os resultados delas. O objetivo da segunda classe é a criação dos botões de ordenação vinculados as colunas com ordenação habilitada.

Este *framelet* é acionado apenas uma vez para a construção do cabeçalho e dos componentes do mesmo. Por exemplo, caso seja desenvolvido em um trabalho futuro, uma funcionalidade que altere a ordem das colunas, cabe a criação de um método de reprocessamento do cabeçalho nesse *framelet*.

Tanto neste diagrama de classe, quanto nos outros dois que serão citados abaixo, foram adicionadas as interfaces dos demais *framelets* para evidenciar a separação que existe entre eles;

5.3.1.6 FrameletConteudo

Conforme mostrado na Figura C.6 este *framelet* é constituído pela interface IConteudo (que herda a interface IFramelet) e a classe concreta Conteudo.

O mesmo é o efetivo responsável pela geração da interface que possui os resultados de cada coluna, após acionar o FrameletPersistencia que efetuará a consulta SQL. Estes resultados podem ser representados de diferentes formas conforme a escolha do tipo do item no FrameletRepresentaDados.

Depois de gerados os dados do conteúdo, este *framelet* atualiza as informações de resultados mostrados e paginação, através da classe Paginador, para na sequência propagar as mudanças no rodapé comunicando-se com o FrameletRodape.

Importante salientar que, na chamada ao método `public UIComponent processarColuna()` deste *framelet* são criados os elementos básicos do conteúdo e executado a geração dos resultados. Contudo, um relatório pode mudar frequentemente em decorrência de operações como ordenação ou filtragem, então, este *framelet* é invocado várias vezes para a reconstrução dos resultados;

5.3.1.7 FrameletRodape

Este *framelet* contém as operações para construção e manipulação do rodapé. Como mostrado no diagrama de classe da Figura C.7, o mesmo é constituído pela interface IRodape, e pelas classes Rodape e Paginador.

A interface IRodape é o ponto de acesso à implementação do rodapé, e também é derivada da interface IFramelet. A primeira classe (Rodape), é responsável pela geração dos elementos personalizados do rodapé (código HTML e componentes JSF) e também por tratar as ações dos componentes do rodapé, invocando a reexibição dos resultados através do FrameletConteudo. Já a classe Paginador, é responsável por consultar o banco de dados através do FrameletPersistencia, para calcular o valor dos indicadores de resultados e de paginação.

Os métodos invocados através do FrameletRodape sempre modificam o conteúdo do relatório, logo, a cada invocação deles também é invocada uma nova geração dos componentes do conteúdo por meio do FrameletConteudo.

5.3.2 Diagrama de Componentes

Além do Diagrama de Classes do estudo de caso, foi desenvolvido um Diagrama de Componentes do mesmo. Este diagrama tem por objetivo ajudar na compreensão das estruturas do módulo em alto nível e também dos serviços fornecidos e requisitados por meio de interfaces. Nos diagramas de classe desenvolvidos para os módulos do MRR, ficam evidente que para um *framelet* interagir com os demais, primeiramente este deve fazer uma requisição ao FrameletRelatorio para obtenção das interfaces correspondentes.

Contudo, para diminuir a complexidade do diagrama de componentes buscou-se uma abordagem diferente da real organização do MRR, em que as ligações entre os componentes associados do módulo, são efetuadas diretamente. Isso facilitou a visualização das

dependências, relações, e demais ligações entre os *framelets*, pois, mostra exatamente quais os envolvidos em determinada operação.

Como ilustrado na Figura D.1 o componente FrameletRelatorio se relaciona com a interface IFramelet que é a base para a definição dos *framelets* no MRR. Neste mesmo componente estão presentes três partes: cabeçalho, conteúdo, rodapé. Cada parte é dependente de um *framelet* para sua consolidação, isto envolve construção da interface e execução das operações, que são realizadas através das interfaces que estão ligadas as portas de interface deste componente.

O componente FrameletPersistencia representa internamente as interligações entre as partes para exemplificar os padrões de projeto Factory e DAO (GAMMA et al., 1995) aplicados à criação de conexões com um SGBD. Ainda no contexto de SGBD, existe o componente ConstrutorSQL, que é o responsável por gerar todas as SQL de consultas das funcionalidades e de outras partes do MRR, servindo como um componente utilitário.

Para representar cada uma das três partes do componente FrameletRelatorio, existem os componentes FrameletCabecalho, FrameletConteudo e FrameletRodape. Suas funcionalidades já foram elucidadas anteriormente, então, em relação ao diagrama de componentes será descrita apenas uma operação do cabeçalho onde o conceito será extendido para as demais do MRR.

Portanto, será exemplificada a operação descrita de Filtragem de Colunas no componente FrameletCabecalho. Como pode ser visto na Figura D.1 a caixa que define esta operação é dependente da porta de interface “Desenhar e Processar Filtro”, que por sua vez, é dependente da porta de interface “InterfaceFiltro” do componente FrameletFiltro, para inicialmente criar a interface dos Filtros.

Outra função que é desempenhada pela porta de interface “Desenhar e Processar Filtro” é o processamento dos filtros escolhidos pelo usuário no relatório. Para isso, a mesma interage com o a porta de interface “ExecutarFiltro” do componente FrameletConteudo, onde é acessado o componente ConstrutorSQL para a construção da SQL global, e o componente FrameletFiltro para a construção da SQL específica de cada filtro ativo no relatório.

Nota-se que para a porta de interface “Executar Filtro” executar por completo, sua função deve invocar a parte “Exibir Dados das Colunas”.

Capítulo 6

Análise e Avaliação do Estudo de Caso

Neste capítulo serão apresentadas as análises e as avaliações em relação ao desenvolvimento do Estudo de Caso e sua aplicação. Na Tabela 6.1 abaixo são mostrados os critérios utilização neste capítulo.

Tabela 6.1: Critérios usados na análise do estudo de caso

Seção	**Critério**
Seção 6.1	Testando o MRR em uma aplicação
Seção 6.2	Utilização do MRR em sistemas
Seção 6.3	Compatibilidade
Seção 6.4	Manutenabilidade e Extensibilidade
Seção 6.5	Dificuldades
Seção 6.6	Vantagens
Seção 6.7	Limitações
Seção 6.8	Uso de framelets no MRR

6.1 Testando o MRR em uma aplicação

Através da Figura 6.1, pode ser visto como é configurada a SQL base do relatório requerido na aplicação cliente. As constantes do tipo `ConstrutorSQL.padrao<Tipo>`, funcionam como macros para que o MRR consiga definir em que parte da consulta SQL um determinado *framelet* fará as operações.

```
1.          String SQL = "SELECT " + ConstrutorSQL.padraoColunas +
2.                       + "FROM " +
3.                       + "          pessoa " +
4.                       + " WHERE " +
5.                       + "     " + ConstrutorSQL.padraoFiltros
6.                       + "     ORDER BY " + ConstrutorSQL.padraoOrdenacao + " "
7.                       + ConstrutorSQL.padraoPaginacao + " ;  ";
8.
9.          this.setSQLConsulta(SQL);
10.
```

Figura 6.1: Exemplo de SQL sendo configurada no MRR

A Figura 6.2, mostra um relatório gerado através da SQL da Figura 6.1, que utilizou-se de praticamente todas as funcionalidades do MRR. Na primeira coluna, por exemplo, foi escolhido um filtro do tipo ILike (valor à ser digitado) que pode ser desabilitado com o botão na lateral do campo de busca. Também foram incluídas operações de soma e média de todos os ids e habilitado a ordenação (ascendente e descendente) da coluna por meio dos botões na sequência.

Além disso, utilizou-se a funcionalidade de seleção de linhas, que visa dar destaque a determinado resultado e no rodapé, para exemplificar o uso de elementos personalizados (HTML ou JSF), utilizou o código fonte da Figura 6.3, para a criação de um título e de um botão de ação nesta região.

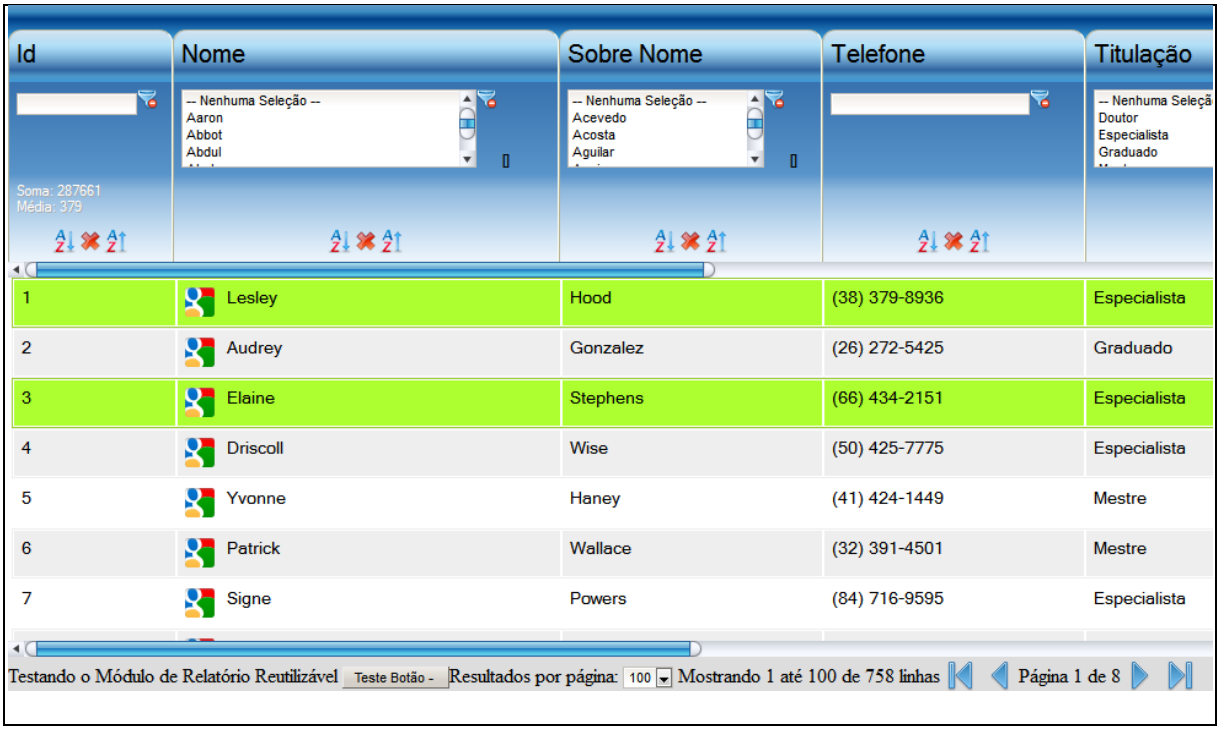

Figura 6.2: Captura de tela do MRR em uso

Ainda em relação à figura anterior, pode ser observado que na coluna “Nome”, antes de cada nome existe um botão em HTML criado através de elementos personalizados, este pode realizar chamadas a funções JavaScript ou para componentes JSF, métodos no *managedBean* definidos pelo reutilizador.

No caso deste exemplo, para cada valor gerado na consulta criou-se uma chamada a função que realiza uma busca pelo valor em um portal de busca.

```
1.                  addElementoDoRodape(new RepresentaHTMLDinamico() {
2.
```

```
3.                          @Override
4.                          public String getValor() throws Exception {
5.                              String html = "";
6.                              html += "<label id=\"teste\" style=\"min-width: 100px; \">" +
7.                                      "Testando o Módulo de Relatório Reutilizável </label>";
8.
9.                              return html;
10.                         }
11.                     });
12.                     addElementoDoRodape(new RepresentaJSF() {
13.
14.                         @Override
15.                         public ArrayList<UIComponent> getValor() throws Exception {
16.                             ArrayList<UIComponent> lista = new ArrayList<UIComponent>();
17.
18.                             UICommandButton c = new UICommandButton();
19.                             c.setValue("Teste Botão - ");
20.                             lista.add(c);
21.
22.                             return lista;
23.                         }
24.                     });
25.
```

Figura 6.3: Configuração de elementos personalizados no rodapé

6.2 Utilização do MRR em sistemas

Desde o início do projeto do MRR buscou-se construir uma arquitetura que facilitasse a reutilização do módulo. Logo, como pode ser visto no diagrama de classes presente na Figura C.1, a classe NucleoRelatorio é a responsável por gerar e centralizar todas as funcionalidades do relatório.

Como já citado, para que uma interface JSF possa executar determinada regra de negócio é necessária a presença de um *managedBean* na aplicação. Como o MRR cria um relatório que utiliza componentes JSF, as interações da mesma com as regras de negócio que manipulam o relatório, devem acontecer por meio um *managedBean* declarado na aplicação cliente.

O *managedBean* que será responsável pela criação do relatório deverá obrigatoriamente herdar a classe principal do MRR, NucleoRelatorio, e no seu construtor adicionar a linha "`super("nomeMG");`", para que o MRR identifique qual *managedBean* será a ponte de comunicação entre a interface e suas estruturas internas.

Além disso, existem outros quatro requisitos que um reutilizador deve cumprir para utilizar o módulo: criar um componente JSF container, configurar as colunas do MRR, configurar elementos do rodapé e processar o relatório. O primeiro requisito não mais é do que a declaração de um componente container, ou seja, que servirá como um agrupador de

elementos. Este componente depois de criado no *managedBean*, deverá ser referenciado através do *binding* de objetos JSF no arquivo de interfaces onde o relatório será acoplado.

O segundo requisito é a configuração do relatório, que nada mais é do que definir quais serão os componentes do mesmo.

A configuração mais importante é a especificação da consulta SQL a ser realizada. A Figura 6.4 representa um exemplo de configuração da SQL para uma consulta na tabela "pessoa". Nota-se nesta a utilização de macros, presentes na classe ConstrutorSQL, para definir em que região da *String* estão as colunas, filtros, ordenação e paginação. É por meio delas que a classe ConstrutorSQL fará as alterações na SQL de consulta para atender cada um dos segmentos do MRR.

```
1.      String SQL = "" +
2.                  "SELECT " + ConstrutorSQL.padraoColunas + " " +
3.                   + "FROM " +
4.                   + "    pessoa " +
5.                   + " WHERE " +
6.                   + "    " + ConstrutorSQL.padraoFiltros
7.                   + "    ORDER BY " + ConstrutorSQL.padraoOrdenacao + " " +
8.      ConstrutorSQL.padraoPaginacao + " " + ";";
9.
10.     this.setSQLConsulta(SQL);
```

Figura 6.4: Configuração da consulta SQL no uso do MRR

O passo seguinte é a configuração das colunas do relatório, em que é definido o nome da coluna na tabela e seu tipo de dados, e outros elementos como: identificador da coluna utilizado na cláusula *where*, adição de operações de totalização, personalização de estilos e classes CSS, definição de tipo de filtro, ativação da ordenação de valores, definição de largura, e outros.

Na Figura 6.5 é mostrado um exemplo de configuração de uma coluna, como pode ser observado, cada parte do relatório pode ser configurada em relação às necessidades do reutilizador. No entanto, caso se deseje apenas uma coluna básica, ou seja, apenas o título no cabeçalho e os dados no conteúdo, são necessárias configurações do título, do nome e do tipo SQL da coluna na tabela do banco de dados.

```
1.  this.addColuna(new Coluna().setTitulo("Código").
2.                  setColunaSQL("cod_pes").
3.                  setColunaWhere("cod_pes").
4.                  setTipoSQL(java.sql.Types.INTEGER).
5.                  adicionarOperacao(new Operacao("Soma", "sum(cod_pes)", java.sql.Types.INTEGER)).
6.                  adicionarOperacao(new Operacao("Média", "avg(cod_pes)", java.sql.Types.INTEGER)).
7.                  estilosCSSConteudo(" height: 100%; ").
```

```
8.                  setFiltro(AFiltro.FILTRO_ILIKE).
9.                  setOrdenar(Boolean.TRUE).
10.                 classesCSSRodape("classeRodape").
11.                 classesCSSCabecalho("classeCabecalho").
12.                 classesCSSConteudo("classeConteudo").
13.                 setLargura(110));
```

Figura 6.5: Configuração de Coluna no uso do MRR

O terceiro requisito abrange a configuração de elementos do rodapé do relatório. Na Figura 6.6 é ilustrada a configuração que permite ao reutilizador adicionar um botão de ação na barra de rodapé, usando internamente o FrameletRepresentaDados, que possui a possibilidade de tratar elementos como: HTML estático, HTML dinâmico e componentes JSF. Sendo que, podem ser adicionados tanto no conteúdo, através do método `addElementoDoConteudo`, quanto no rodapé, permitindo maior flexibilidade no relatório.

O quarto e último requisito é o passo final para processar e adicionar o relatório no componente de interface de apresentação, na linha 15 da Figura 6.6, é mostrado o comando para adicionar o relatório na página, em que o método `processarRelatorio` retorna um componente JSF que é adicionado como filho do componente container criado no primeiro requisito.

```
1.  addElementoDoRodape(new RepresentaJSF() {
2.
3.                  @Override
4.                  public ArrayList<UIComponent> getValor() throws Exception {
5.                      ArrayList<UIComponent> lista = new ArrayList<UIComponent>();
6.
7.                      UICommandButton c = new UICommandButton();
8.                      c.setValue("Botão de Teste ");
9.                      lista.add(c);
10.
11.                     return lista;
12.                 }
13.             });
14.
15. form.getChildren().add(this.processarRelatorio());
```

Figura 6.6: Configurando elementos do rodapé e processando o relatório

6.3 Compatibilidade

No projeto do MRR buscou-se uma integração natural com o *framework* JSF e isso foi possível utilizando componentes que implementam a interface padrão do JSF, a UIComponent, em todas as partes do relatório. Contudo, a compatibilidade deste módulo

pode ir muito além deste caso, como vários *frameworks* da plataforma JEE se integram com o JSF, o MRR pode ser extendido para uso em sistemas feitos com estas tecnologias.

Além disso, com o desenvolvimento do estudo de caso, foi desenvolvida uma metodologia para adaptação de componentes de um *framework* Web. Esta poderá ser difundida em outros *frameworks* que não tenham necessariamente o recurso de *binding* de componente. Portanto, uma alternativa para este recurso é a utilização de métodos nativos de *frameworks* que permitem acessar os objetos que descrevem cada componente da página em exibição.

6.4 Manutenabilidade e Extensibilidade

O desenvolvimento baseado em *framelets* facilitou a manutenabilidade, visto que, cada funcionalidade pertencente a determinado ramo é de responsabilidade de um dos *framelets*. Portanto, ao utilizar a estratégia dividir para conquistar como neste estudo de caso, evitou-se o entrelaçamento entre os módulos e camadas, o que torna mais direta a localização de problemas.

A metodologia utilizada para estruturar os *framelets* permitiu a extensibilidade através do FrameletRelatorio, que é o integrador do MRR. Através dele é possível adicionar outro *framelet* com uma especialidade nova, por exemplo, geração de gráficos ou exportação dos dados do relatório. Nesta situação, o FrameletRelatorio irá definir o relacionamento com as interfaces correspondentes às novas especialidades, e os demais *framelets* envolvidos na nova funcionalidade vão se integrar com o novo módulo somente por meio destas.

6.5 Dificuldades

A primeira dificuldade que surgiu no projeto do estudo de caso deste trabalho foi: que problema poderia ser tratado, para ilustrar a aplicação de *framelets* num módulo reutilizável para sistema Web? Como citado na introdução do capítulo 5, o contexto de relatório para sistemas, se mostrou uma solução compatível com a proposta do trabalho. Escolhido o problema, outra grande dificuldade, foi determinar qual a granularidade das partes do sistema para definir os *framelets* e a construção arquitetural para integrar os mesmos em um módulo reutilizável com o mínimo acoplamento possível.

Também existiram dificuldades decorrentes do uso do *framework* JSF, mesmo com a evolução da versão 2.0. Através do desenvolvimento programático envolvido no MRR notou-

se que apesar da robustez do JSF, este não facilita esse tipo de desenvolvimento, visto que, seus componentes nem sempre podem ser personalizados ao "gosto" do desenvolvedor. Um exemplo disso, é o componente panelGroup, que após renderizado gera uma *tag* Div do HTML, a mesma possui suporte ao evento de movimentação da barra de rolagem (*onscroll*), mas o componente não permite manipulá-lo.

Contudo, esta dificuldade foi contornada através de um código na linguagem JavaScript, que após o carregamento da página, adiciona um *listener* no evento *onscroll* das Divs que necessitam ser manipuladas.

Outra restrição imposta pelo *framework* JSF envolve a tecnologia AJAX. O AJAX permite uma maior interatividade dos usuários com os sistemas Web (seção 4.3). Assim, o projeto do MRR previa a utilização desta tecnologia nas operações do relatório. No entanto, determinar que um dado componente da implementação utilize AJAX, não é uma operação prevista no conjunto de métodos destes. Nesta situação, utilizou-se do mapeamento nos eventos dos componentes de funções JavaScritpt que são geradas pelo uso *tag* que introduz AJAX (f:ajax) em componentes das páginas JSF.

Foram realizadas pesquisas a fim de solucionar os problemas que surgiram com o "desenvolvimento programático". Nem todos os problemas foram solucionados completamente ou tiveram a melhor solução, como se notou nos parágrafos anteriores. Logo, a falta desse tipo de ajuda por parte da comunidade de desenvolvedores, pode ter relação com o fato da grande parte deles desenvolverem utilizando-se de *tags* de componentes, e profissionais que desenvolvem programaticamente, não buscam um maior aprofundamento nessa abordagem.

6.6 Vantagens

O MRR buscou atender uma necessidade recorrente de vários sistemas, a geração de relatório. Como mostrado anteriormente o uso do MRR é facilitado, pois utiliza a linguagem SQL e as configurações são parametrizadas. Com isso, a necessidade de treinamento de equipe e desenvolvimento de documentação é reduzida. Além disso, pelo fato do MRR não ser um *framework* a mais no sistema e por ser desenvolvido com os próprios componentes do JSF, evita-se incompatibilidades resultantes da Inversão de Controle, que geralmente estes *frameworks* necessitam.

A reusabilidade de um artefato pode estar ligada diretamente à facilidade de compreensão do seu funcionamento e à facilidade de uso deste, além de outros fatores. O estudo de caso buscou garantir esse fator, através da combinação da linguagem SQL e das configurações parametrizadas. Desse modo, tem-se um módulo para geração de relatório de dados de propósito geral e reutilizável em diferentes domínios de aplicação.

Uma última vantagem a destacar no MRR, é a utilização de diferentes tipos de representação de dados, em que o FrameletRepresentaDados faz o processamento e geração de texto simples, código HTML ou componente JSF. Assim, o reutilizador tem maior liberdade para personalizar os dados exibidos, por exemplo, supondo que o resultado de uma consulta a uma entidade fictícia chamada "Pessoa" retorne várias "tuplas". Neste caso, podem ser definidos durante as configurações do MRR vários botões de comando (HTML ou JSF) para uma coluna de "Ação", sendo gerados para cada resultado, e o valor desse resultado pode ser passado por parâmetro para definir qual registro está sendo manipulado pela ação disparada.

6.7 Limitações

Nas seções anteriores foram expostos os fatores de sucesso advindos da análise dos pontos de vista como: a reutilização, coesão e acoplamento, manutenabilidade, e outros aspectos que mostraram bons resultados.

No entanto, como qualquer outro artefato reutilizável, a análise do estudo de caso deste trabalho resultou em limitações no próprio MRR e também nas abordagens utilizadas para seu desenvolvimento.

A primeira delas tem relação com o desenvolvimento programático das interfaces, esta tarefa é muito complexa de projetar e executar, pois, tanto os aspectos visuais, quanto os estruturais, estão completamente ligados. Logo, a execução de um projeto como o MRR obriga que a equipe de trabalho tenha domínio sobre esses aspectos, caso contrário o desenvolvimento não será produtivo. Além disso, o foco da equipe deve ser concentrado em cada serviço fornecido pelos *framelets* de um módulo reutilizável.

Essa abordagem de programação gera conflitos entre a produtividade e o controle sobre a implementação reutilizada, ou seja, num contexto em que a implementação de um módulo reutilizável precisa ser evoluída, por um lado esta abordagem traz maior domínio do que está

sendo desenvolvido, por outro se tem a criação das interfaces de usuário através de regras de negócio internas ao módulo reutilizável, o que limita a produtividade.

Já em relação ao estudo de caso, nem todas as funcionalidades estão completamente disponíveis a um reutilizador. Os filtros, a conversão de dados, e personalização das estruturas do relatório gerado pelo MRR, se enquadram nessa situação. O exemplo dos filtros e a conversão de dados se assemelham, pois, a cada diferente tipo de dados contido na classe java.sql.Types, deve existir uma implementação correspondente de filtro, da mesma forma, para os mesmos devem também existir configurações prévias que definem como cada um será exibido no conteúdo do relatório final.

6.8 Uso de *framelets* no MRR

Os *framelets* foram essências no MRR para distribuir as responsabilidades do problema para diferentes estruturas, pois semelhante à idéia dos *frameworks*. Os *framelets* envolvidos fornecem serviços reutilizáveis. Utilizou-se no estudo de caso uma das heurísticas do projeto do *framework* AOCS, que é o mapeamento de um conjunto de requisitos para um *framelet*.

A aplicação do desenvolvimento baseado em *framelets* no MRR evitou que as várias classes da estrutura se comunicassem diretamente entre si. Pree e Koskimies (1999) defendem a arquitetura baseada em unidades pequenas e flexíveis, com interfaces conhecidas. Tal combinação de caixa-branca e caixa-preta sobre a aspecto arquitetural pode resolver os conhecidos problemas da utilização de *frameworks* complexos.

Portanto, esta abordagem diminuiu a complexidade de desenvolvimento e execução do projeto do MRR, visto que, com a separação de funcionalidades em diferentes *framelets* buscou-se uma maior coesão nas estruturas internas deles. Como resultado disso, houve a diminuição do alto acoplamento que uma solução convencional teria.

Sob o ponto de vista de reutilização, *framelets* por si só não garantem o reuso, visto que seu objetivo envolve a separação de especialidades em pequenas unidades e o fornecimento de certos serviços ou esqueletos de um domínio específico. A reutilização de uma estrutura que os utilize fica a cargo de um projeto de generalização dos elementos de um domínio.

Uma das expectativas deste trabalho era que, a implementação do estudo de caso não possuísse funcionalidades inutilizadas ou fora do serviço prestado, assim como os *frameworks*. Isso foi garantido no MRR integrando no mesmo apenas as funcionalidades

essenciais de um relatório, uma nova funcionalidade que se integre a ele deve abrir margem para a construção de outra estrutura de integração dos módulos.

A proposta neste trabalho mostrou que uma equipe de desenvolvimento não precisa necessariamente de um *framework* para atender uma pequena necessidade como um gerador de relatório, ao invés disso, pode ser desenvolvida a própria ferramenta reutilizável e para um segmento específico.

Entretanto, Pree (1999) afirma que, este tipo de desenvolvimento implica em mudanças organizacionais, isto é, têm-se mais um projeto a ser mantido pela equipe. Com isso, fatores como dimensão do escopo do problema, necessidades constantes de evolução, disponibilidade de profissionais e artefatos comerciais de reuso, prazo de entrega, dentre outros, são decisivos para a criação de uma solução genérica.

Além disso, o fato dos *framelets* possuírem uma granularidade baixa faz com que em determinados modelos sua utilização se torne muito onerosa, pois serão combinadas muitas unidades destes para a consolidação de um módulo útil. Uma possível alternativa para este problema seria a utilização de herança nos pontos fixos do *framelet*. Porém, a herança não é uma maneira bem sucedida de reutilização, pois estas relações entre as classes não podem ser desfeitas em tempo de execução (GURP, 2000).

Capítulo 7

Considerações Finais

Existem grandes dificuldades para se projetar sistemas Web, fato que da complexidade desse paradigma. Porém, existem muitas ferramentas para auxiliar os desenvolvedores sendo o *framework* uma das principais. O *framework* JSF utilizado no estudo de caso trouxe grande produtividade no desenvolvimento de sistemas Web comerciais.

Contudo, existem necessidades específicas de desenvolvimento que podem não ser atendidas pelos componentes prontos disponibilizados pelo *framework*. Nesta situação, um projetista pode simplesmente, mudar de tecnologia ou pesquisar melhores formas de suprir essas carências.

Desse modo, este trabalho buscou através do desenvolvimento de um estudo de caso, validar a integração da metodologia de programação baseada em *framelets* com o desenvolvimento programático de um Módulo de Relatório Reutilizável (MRR) em diferentes domínios de aplicação.

A adaptação dinâmica das subpartes dos relatórios gerados pelo MRR foi garantida por meio do recurso "binding" dos componentes JSF, onde cada *framelet* representante de uma das subpartes do modelo de relatórios, internamente adapta e cria os componentes conforme as configurações escolhidas por um reutilizador.

As análises do estudo de caso resultaram em vários critérios que devem pesar na decisão de desenvolvimento de uma nova solução reutilizável.

Como pontos positivos, destacaram-se a compatibilidade com a plataforma JEE advinda do uso do *framework* JSF e o fluxo de controle apenas em estruturas internas aos *framelets*, a manutenabilidade e extensibilidade proporcionadas pela arquitetura dos *framelets*. Como principais vantagens, a utilização do MRR baseada em configurações simplificadas e personalização dos relatórios com diferentes tecnologias por meio do FrameletRepresentaDado.

No entanto, também resultaram pontos negativos das análises do MRR, tanto no desenvolvimento do estudo de caso, quanto no confronto com os objetivos do trabalho. No contexto do JSF, pode-se notar que este *framework* não é uma estrutura flexível e preparada para alterações dinâmicas como as requeridas neste trabalho.

Já em relação aos *framelets* a afirmação defendida por Gurp (2000), anteriormente, se faz pertinente, uma vez que, frequentemente os sistemas necessitam de soluções que suportam vários modelos, o que não é o caso dos *framelets*. Portanto, estas necessidades são supridas com a integração de vários *framelets*, o que podem em contrapartida tornar a estrutura resultante desorganizada e de difícil manutenção.

Sobre outro ponto de vista, não é correta a integração desordenada de *framelets*, exemplo disso, seria ao invés de integrar as funcionalidades de cabeçalho no FrameletCabecalho, criar um *framelet* por funcionalidade e integrar posteriormente. Neste tipo de caso se faz necessário um novo projeto arquitetural para definir qual o nível de granularidade das implementações.

Seguindo essa linha de pensamento, o MRR foi usado para representar o conhecimento arquitetural de um segmento de relatórios, e combinou em seu desenvolvimento o uso de padrões de projeto para a construção de um módulo de arquitetura reutilizável.

Destaca-se que o desenvolvimento baseado em *framelets* pode ser extendido para outros sistemas e tecnologias, como uma metodologia em que um problema específico é segmentado e os módulos resultantes disso podem ser representados pelos *framelets*. O principal retorno desse fato é a diminuição do acoplamento entre as partes de um módulo, em que a busca por estruturas genéricas podem necessitar de muitos relacionamentos entre as classes internas.

O uso de *framelets* fez o projeto mais gerenciável e tornou mais fácil a extensão da estrutura MRR para outros domínios de aplicação, sendo que alguns dos *framelets* podem ser utilizados para outros projetos com outro foco sem qualquer alteração. A abordagem de desenvolvimento com *framelets* e adaptação dinâmica mostrou-se uma solução adequada na busca de soluções reutilizáveis em diferentes domínios de aplicação.

7.1 Trabalhos Futuros

a) Integrar várias entidades de um sistema no MRR de forma automática com uso de Reflexão Computacional;

b) Uso de Skins para padronização e personalização da interface do relatório através de um *framelet* especifíco;

c) Desenvolver um estudo de caso que implemente um módulo para geração de um CRUD (Criar, Recuperar, Atualizar e Excluir) reutilizável;

d) Substituir os componentes da implementação JSF por outra tecnologia e manter as estruturas de geração de interfaces definidos no MRR;

e) Avaliar a abordagem *framelet* empregada em modelos complexos;

Apêndice A

Primeiras Tecnologias Java para Web

A.1 Servlets

Os Servlets surgiram como uma alternativa para aplicações CGI. Logo, esta tecnologia desenvolvida pela Sun Microsystems em 1996, tornou-se padrão para o desenvolvimento de aplicações Web na linguagem Java. A tecnologia Servlet age como uma interface entre o desenvolvedor Java e o servidor de aplicação, usando serviços oferecidos pelo servidor de aplicação para execução de aplicações para Web (KONO, 2008).

Os Servlets seguem um modelo muito semelhante ao da tecnologia CGI, assim que os mesmos recebem solicitações HTTP como entrada acontece o processamento para posterior envio de uma resposta pelo servidor (FIELDS e KOLB, 2000 (apud FERLIN, 2004)).

Portanto, diferente do CGI, que a cada nova requisição criava necessariamente um novo processo para tratá-la, os Servlets rodam sobre um mesmo processo contido no servidor Web. Logo, o servidor contém uma Java Virtual Machine (JVM), que dá suporte à execução dos componentes contidos no servidor de aplicação, permitindo o encadeamento de processos para tratar requisições nas aplicações (FERLIN, 2004).

Os Servlets se baseiam em fornecer conteúdo dinâmico através de códigos Java, estes constroem documentos em HTML ou outros formatos, se baseando no tratamento de requisições e respostas HTTP. Quanto à construção de páginas HTML, os Servlets permitem que o programador gere as *tags* da linguagem HTML, de forma que os elementos da página podem sofrer alterações dinâmicas, e serem retornados ao browser através do documento construído.

Além disso, é importante destacar as vantagens dos Servlets, que se estende ao JSP (descrito na próxima seção), que podem não ser parte de outras tecnologias concorrentes (KURNIAWAN, 2002):

a) o desempenho é superior ao CGI, pois não são criados novos processos por nova requisição. Cada requisição é gerenciada pelo processo servlet container. Os Servlets ficam na memória aguardando por requisições;

b) a portabilidade está inerente nessa tecnologia, pois eles podem ser implantados em qualquer ambiente com servidor de aplicação que implemente a especificação Servlets;
c) o desenvolvimento é acelerado pelo uso da linguagem Java com seu rico conjunto de classes para diversos fins;
d) a gerência de memória e coleta de lixo fica por conta da Máquina Virtual Java, por conseguinte, facilita o desenvolvimento de aplicações robustas;
e) não há grandes preocupações para desenvolver componentes que atendam às necessidades, pois Java é uma tecnologia de vasta aceitação o que significa que muitas empresas fornecem esses recursos prontos para uso.

A.1.1 Arquitetura de uma Aplicação Servlet

Servlet é uma classe Java que está contida em um *container servlet*, ou seja, um servidor Web voltado à tecnologia Servlets ao carregamento e execução dessas classes (KURNIAWAN, 2002).

Como citado anteriormente, esta tecnologia comunica-se com clientes usando um modelo solicitação/resposta com o protocolo HTTP. Contudo, nada impede que outros protocolos sejam usados pelas classes Servlets, basta que o *container servlet* implemente a especificação do protocolo. Logo, nota-se que nesta tecnologia, as aplicações não fogem da forma de comunicação da tecnologia CGI.

O ciclo de vida básico de uma aplicação Servlet é divido em quatro fases, como mostrado na Figura A.1, são elas (CORRÊA, 2004):

a) requisição enviada do cliente ao servidor;
b) o container servlet para o Servlet que foi solicitado na requisição;
c) o Servlet responde à requisição do cliente, com conteúdo dinâmico;
d) o container servlet retorna a resposta HTTP gerada, para o cliente.

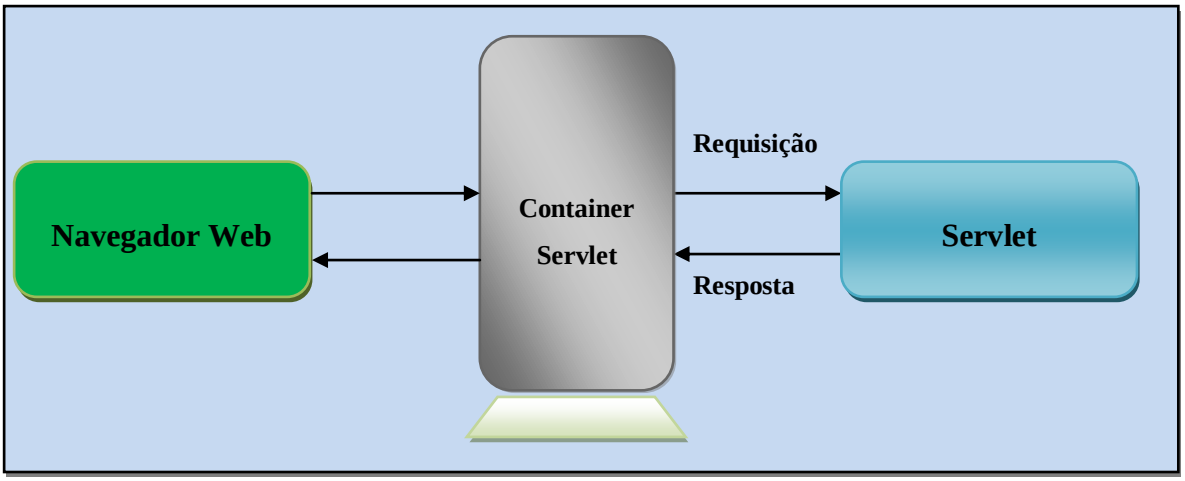

Figura A.1: : Arquitetura Básica Servlet 1 – Ciclo de Vida Básico de um Servlet (GOODWILL, 2002)

O *container servlet* é responsável por gerenciar o ciclo de vida de um servlet. O servidor de aplicação aguarda requisições, estas são repassadas não ao Servlet requerido, mas sim ao *container* onde o Servlet foi implantado. Desse modo, ao *container* é dada a responsabilidade de entrega das solicitações e respostas HTTP (BASHAM et al., 2005) (KONO, 2008)).

A interface Servlet no pacote `javax.servlet` é a base para todas as operações de um servlet. Um servlet deve implementar direta ou indiretamente esta interface, e a mesma contém os métodos: *init*, *service* e *destroy* (KURNIAWAN, 2002).

Estes métodos compõem o ciclo de vida de um Servlet. Quando um servlet é instanciado, é chamado o seu método *init* apenas uma vez, inicializando-o (colocando-o) como serviço. Este método tem como finalidade inicializar o servlet para uso, ou seja, realizar operações onde seus resultados vão persistir nas requisições, como exemplo se pode citar a configuração de driver de banco de dados.

Além disso, vários parâmetros de inicialização do servlet podem ser definidos no arquivo de configuração web.xml, que são repassados ao método *init* na inicialização do servlet (Kurniawan, 2002)(KONO, 2008). Uma vez que o método *init* finalizar sem problemas, o servlet estará pronto para receber e enviar requisições HTTP.

A cada requisição que o servidor recebe para um servlet, o servidor cria uma nova *thread* e chama o método *service*. Este método identifica o tipo de requisição HTTP (GET, POST, PUT, DELETE, etc) e invoca métodos *doGet*, *doPost*, *doPut*, etc.

Portanto, o *container* repassa dois objetos HTTP para o servlet, o ServletRequest (solicitação) e ServletResponse (Resposta). Dessa forma, ao serem executados os métodos de

processamento do servlet, é possível realizar mudanças dinâmicas através deles, pois a comunicação ocorre por meio destes objetos.

Para fechar o ciclo de vida, o método *destroy* pode ser chamado para retirar um servlet da condição de serviço no servidor. Essa chamada acontece quando o *container servlet* é finalizado, reinicializado ou precisa liberar memória. Contrário a figura anterior, a Figura A.2 demonstra o funcionamento dos métodos de um servlet.

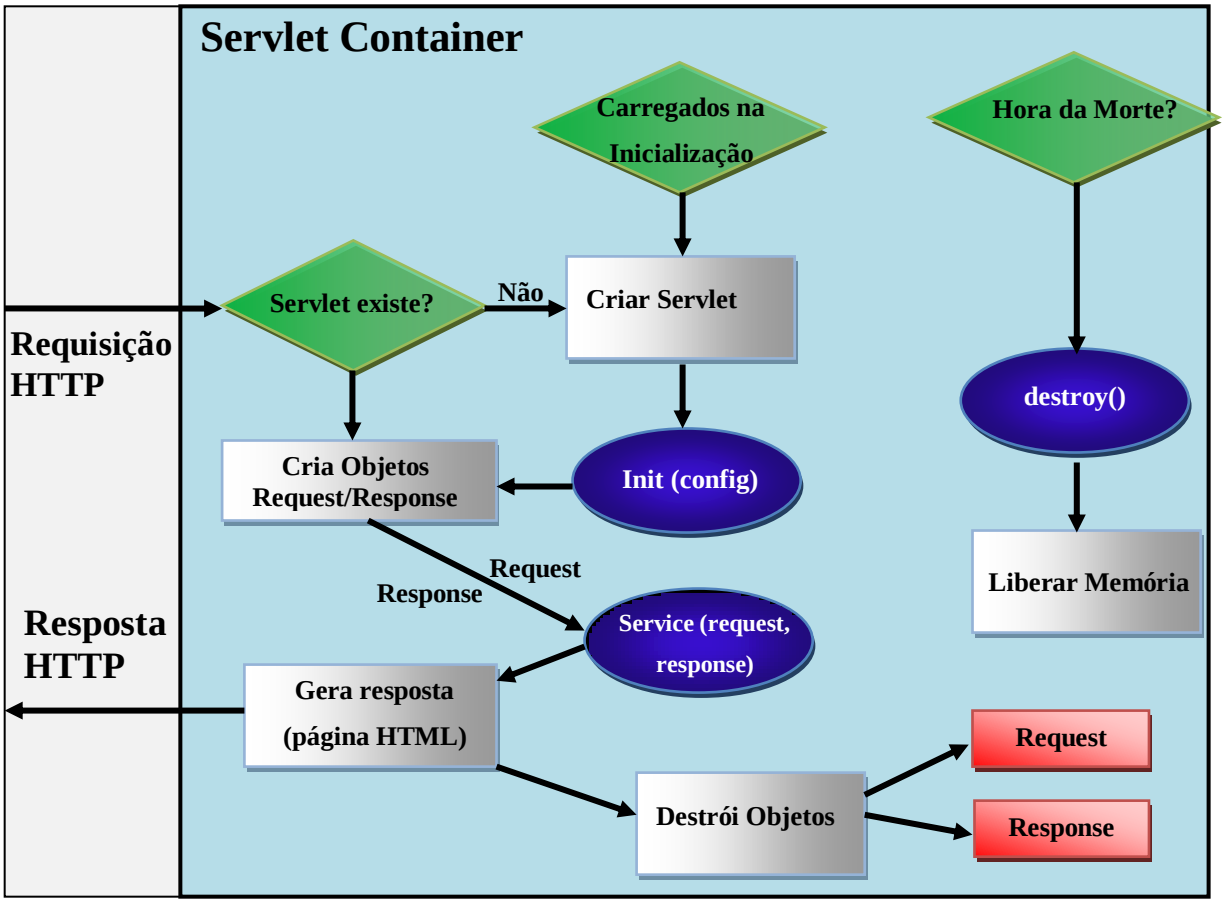

Figura A.2: Ciclo de vida de um Servlet. (KONO, 2008)

A Figura A.3 mostra através da Classe ServletPrimitivo os métodos do ciclo de vida. Cada método ao ser chamado imprime no console de saída o seu nome.

```
1.  import javax.servlet.*;
2.  import java.io.IOException;
3.  public class ServletPrimitivo implements Servlet {
4.
5.      public void init(ServletConfig config) throws ServletException {
6.          System.out.println("init");
7.      }
8.
9.      public void service(ServletRequest request, ServletResponse
10.     response)
11.     throws ServletException, IOException {
12.         response.setContentType("text/html; charset=ISSO-8859-1");
13.         String HTML = "" +
14.             "<html>" +
15.             "       <head>" +
16.             "           <title>Testando Ciclo de Vida</title>" +
17.             "       </head>" +
18.             "       <body>" +
19.             "           <h1>Saída de Texto No Browser </h1>" +
20.             "       </body>" +
21.             "</html>";
22.         PrintWriter saida = response.getWriter();
23.         saida.print(HTML);
24.         saida.close();
25.     }
26.
27.     public void destroy() {
28.         System.out.println("destroy");
29.     }
30.
31.     public String getServletInfo() {
32.         return null;
33.     }
34.
35.     public ServletConfig getServletConfig() {
36.         return null;
37.     }
38. }
```

Figura A.3: Demonstração do Ciclo de Vida de um Servlet (KURNIAWAN, 2002)

A.2 JSP (Java ServerPages)

A tecnologia JSP é uma extensão da tecnologia Servlets (API Servlet 2.1), que permite integrar numa mesma página, HTML estático e código Java. A mesma simplifica o desenvolvimento de páginas, pois permite separar a lógica de apresentação do conteúdo. O JSP é usado na maioria das vezes, quando predomina o conteúdo estático ou se o conteúdo gerado dinamicamente pelo código Java for bem pequeno (H. DEITEL e P. DEITEL, 2005).

Uma página JSP pode ser gerada por profissionais em Web Designer, através de ferramentas de design de páginas, para em seguida receber o código com propriedades dinâmicas. Diferentemente dos *Servlets*, onde apresentação e lógica das páginas encontram-se

no mesmo lugar, nesta, a intercalação de código HTML e Java facilita a divisão dos profissionais no desenvolvimento. Além disso, é mais fácil escrever em HTML nas páginas JSP, do que modificar uma grande quantidade de “println” de HTML na Servlet.

Na primeira chamada à uma página JSP contida no *JSP Conteiner*, é criado, baseado na estrutura estática e nos códigos Java embutidos na página, um *servlet* para gerar a página HTML a ser enviada ao navegador Web. Uma vez que isso aconteça, para requisições futuras não serão necessárias compilações da página JSP se está não se alterou, pois já existe o *servlet* residente em memória para tratar estas (KONO, 2008).

Há duas formas de inserir código dinâmico numa página JSP. A primeira opção cita os *scriptlets*, que é a inserção de código Java diretamente na página. *Scriptlets* iniciam com “<%” e termina, com “%>”, são fáceis de utilizar, rápidos e poderosos. Já na segunda opção, tem-se o uso de *tags* JSP personalizadas; que são muito semelhantes aos comandos HTML (HUSTED et. al, 2003 apud (CORRÊA, 2004)).

A Figura A.4 objetiva contextualizar a tecnologia JSP, onde há a chamada à biblioteca de *tags* chamada “minhaLib-taglib.tlb”, que é direcionada ao prefixo de chamada “minhaLib”.

Além disso, a cor de fundo da página pode ser alterada dinamicamente através do *scriptlet*, pois este a recebe por parâmetro de requisição. Assim, no bloco do *scriptlet* é criada a variável bgColor, que passa seu valor através de expressão de linguagem à propriedade de plano de fundo da página JSP.

Logo na sequência, tem-se a chamada de uma *tag* contida na biblioteca que foi importada no início da página JSP.

```
1.  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
2.  <HTML>  <HEAD>
3.          <%@ taglib uri="minhaLib-taglib.tld" prefix="minhaLib" %>
4.          <TITLE>Testando ScriptLets e Tags Personalizadas</TITLE>
5.          <LINK REL=STYLESHEET  HREF="JSP-Styles.css"
6.      TYPE="text/css">
7.      </HEAD>
8.      <%
9.          String bgColor = request.getParameter("bgColor");
10.         if (bgColor == null) {
11.             bgColor = "WHITE";
12.         }
13.     %>
14.     <BODY BGCOLOR="<%=  bgColor %>">
15.         <minhaLib:example />
16.     </BODY>
17. </HTML>
```

Figura A.4: Exemplo de página JSP, utilizando Scriplets e tags personalizadas

A.2.1 Scriptlets JSP

Com os Scriptlets é possível inserir mais do que simples expressões de linguagem, com eles é possível inserir código Java nas aplicações. Estes têm a seguinte sintaxe: <% Código Java >%.

Scriptlets podem utilizar de forma automática variáveis do ambiente Web e assim interagir gerando mudanças dinâmicas. Além disso, utilizar código Java permite utilizar recursos que não são atendidos apenas por expressões de linguagem, como: saída de texto no console do servidor, executar laços, lógicas condicionais ou operações em banco de dados.

A.2.2 JSP Tags Personalizadas

Um recurso muito interessante na tecnologia JSP, é a possibilidade de criação de *tag's* personalizadas. Estas são constituídas por código Java que é adicionado na página JSP no momento da compilação para gerar conteúdo dinâmico. O código Java contido na *tag* é criado por uma classe Java contida num arquivo de biblioteca. O formato e comportamento da *tag* são definidos pelo programador. Assim, *designers* podem simplesmente importar bibliotecas de *tags* e utilizar seus comandos (JONHSON et. al, 2002 apud (CORRÊA, 2004)).

As *tags* são muito semelhantes às *tags* HTML. Portanto, tornam o código das páginas JSP muito mais legível, visto que, evitam a presença de código Java em meio a código HTML. Outro aspecto, é que as *tags* são mais amigáveis aos *designers*, o que evita que apenas programadores com conhecimento em Java, possam dar manutenção às páginas JSP (SANTOS, 2007).

Corrêa (2004) destaca os principais benefícios das *tags* personalizadas para as aplicações Web, são elas:

a) *tags* personalizadas tem maior grau de reutilização do que *scriptlets*;
b) bibliotecas de *tags* fornecem um alto nível de portabilidade entre os *contêineres* JSP;
c) facilitam a separação de funções no desenvolvimento, ou seja, programadores preocupam-se com a lógica de negócio, enquanto os *designers* com a apresentação;
d) baixo acoplamento entre camada de apresentação e lógica de negócio.

A.2.3 Ciclo de Vida de uma Página JSP

Segundo Booch e Scientist (2003), pelo fato de uma página JSP ser convertida em Servlet, a mesma apresenta um ciclo de vida praticamente equivalente ao apresentado na seção A.1.1, ou seja, inicialização, tratamento/atendimento das requisições e finalização.

A Figura A.5 mostra como são realizadas as chamadas a páginas JSP. Exceto a parte onde o *container* JSP compila a página JSP o restante segue o modelo de chamada do ciclo de vida dos Servlets. Desse modo, cada página tem um servlet correspondente, quando acontece uma requisição à página, é efetuada uma verificação na data de modificação do arquivo jsp, caso tenha uma modificação em relação ao servlet, este arquivo é recompilado.

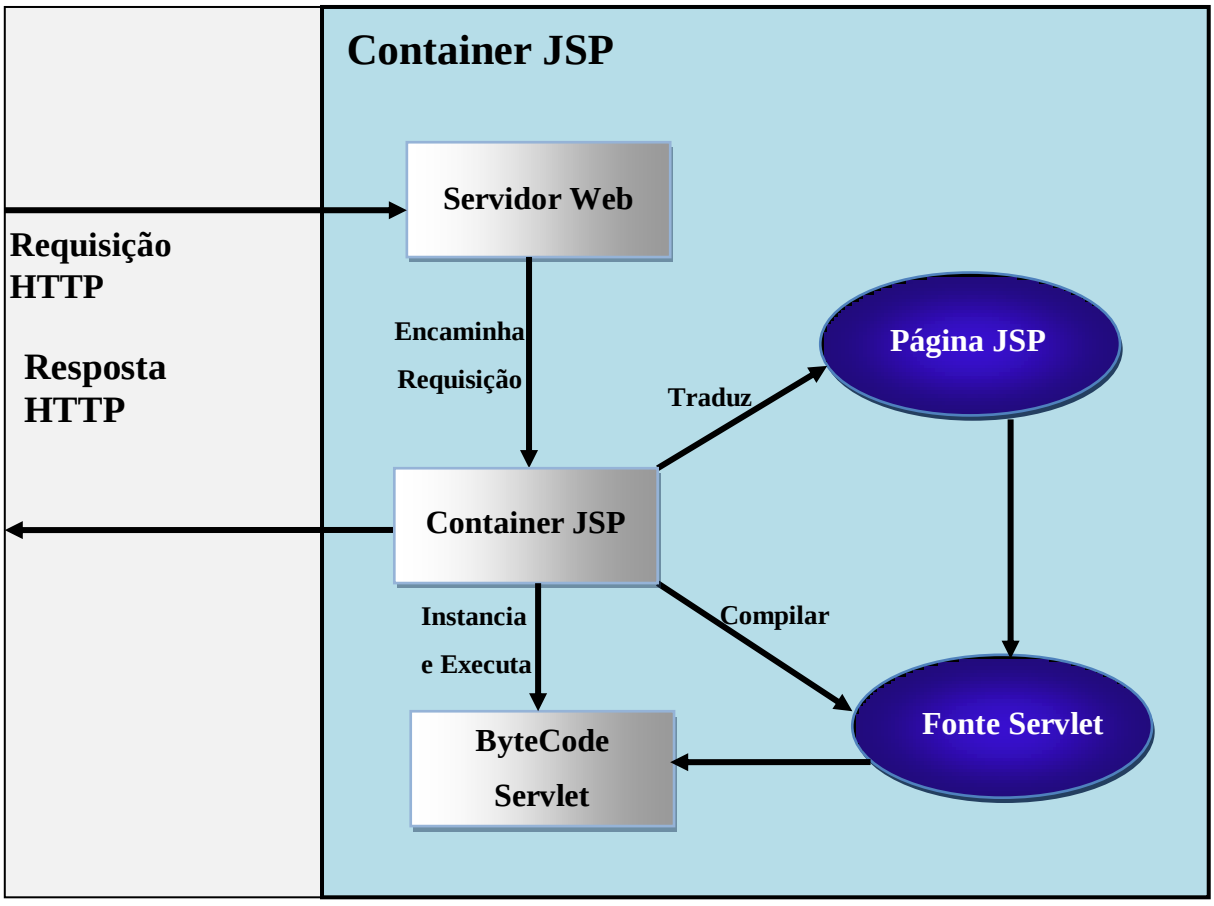

Figura A.5: Ciclo de Vida Básico de Páginas JSP

Apêndice B

Diagrama de Casos de Uso

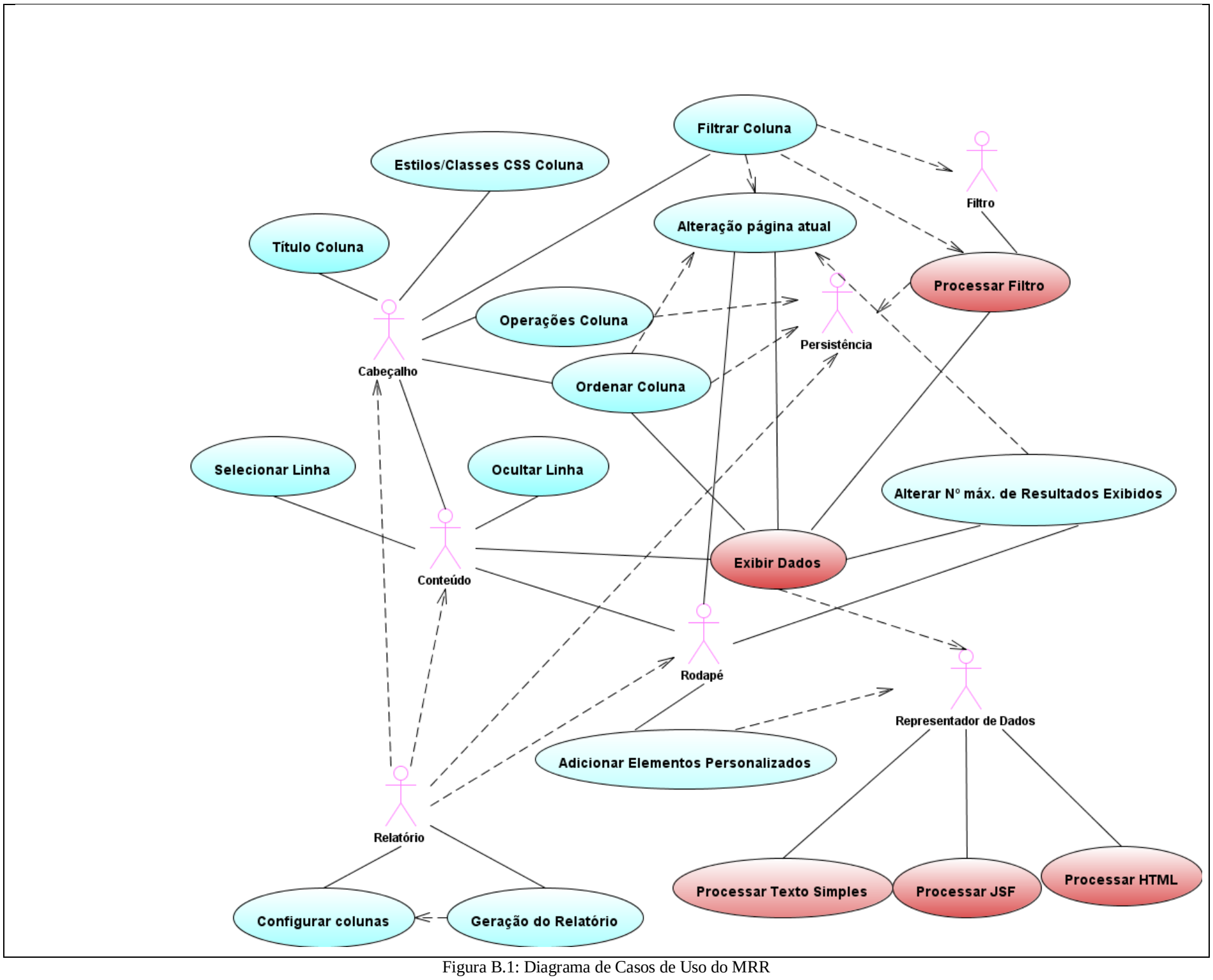

Figura B.1: Diagrama de Casos de Uso do MRR

Apêndice C

Diagramas de Classes

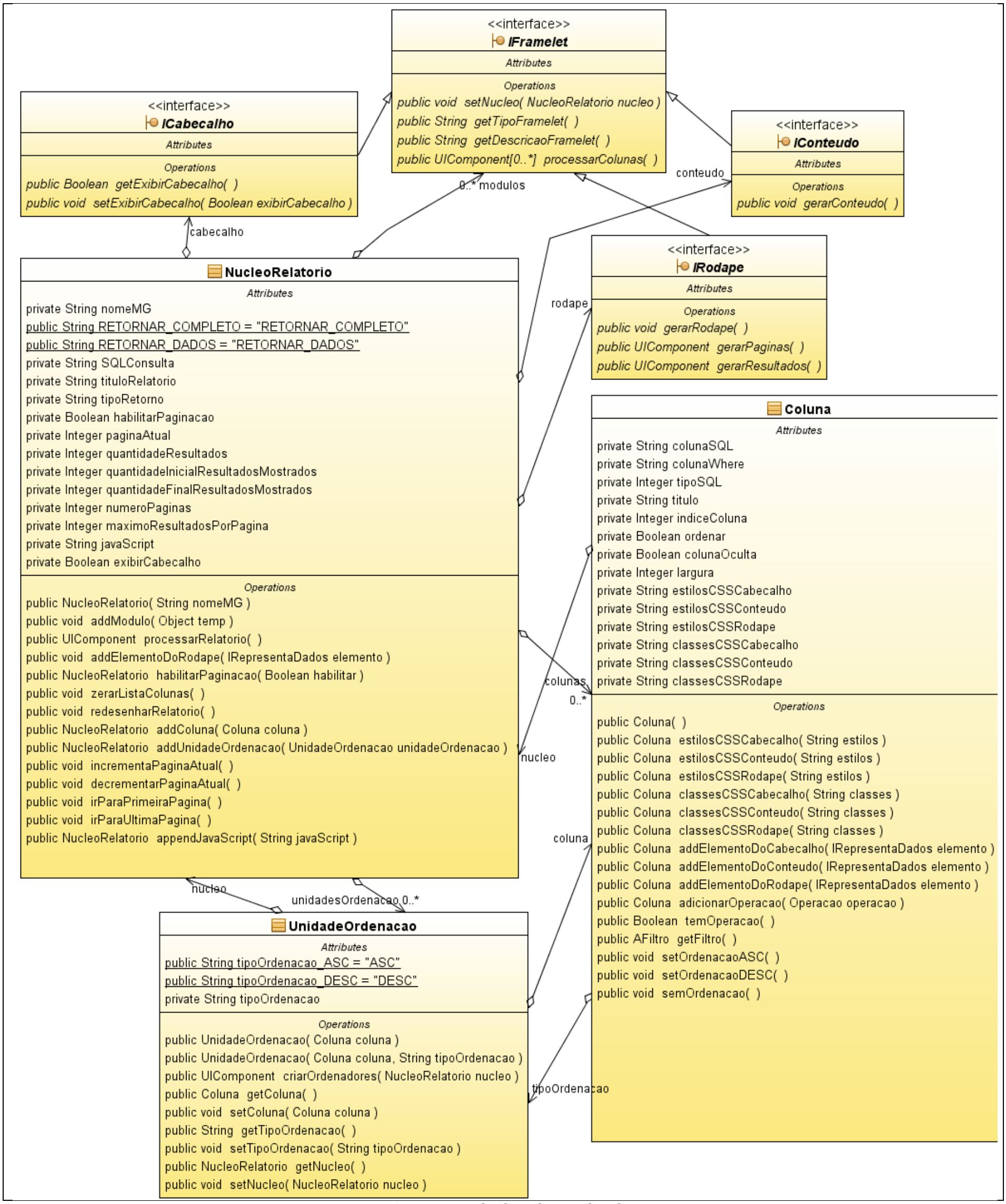

Figura C.1: Diagrama de Classe do FrameletRelatorio

C.1 FrameletRepresentaDados

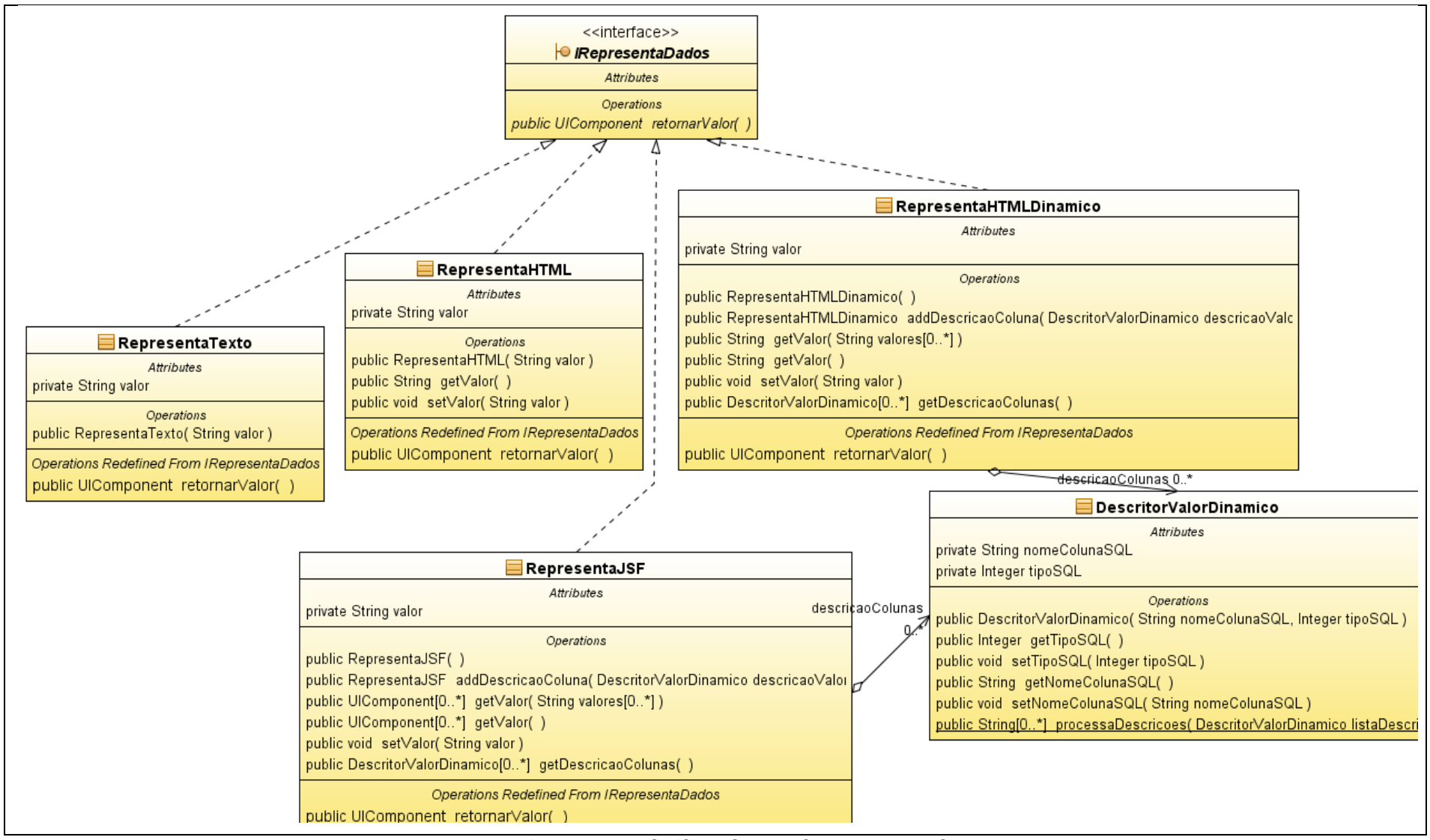

Figura C.2: Diagrama de Classe do FrameletRepresentaDados

C.2 FrameletFiltro

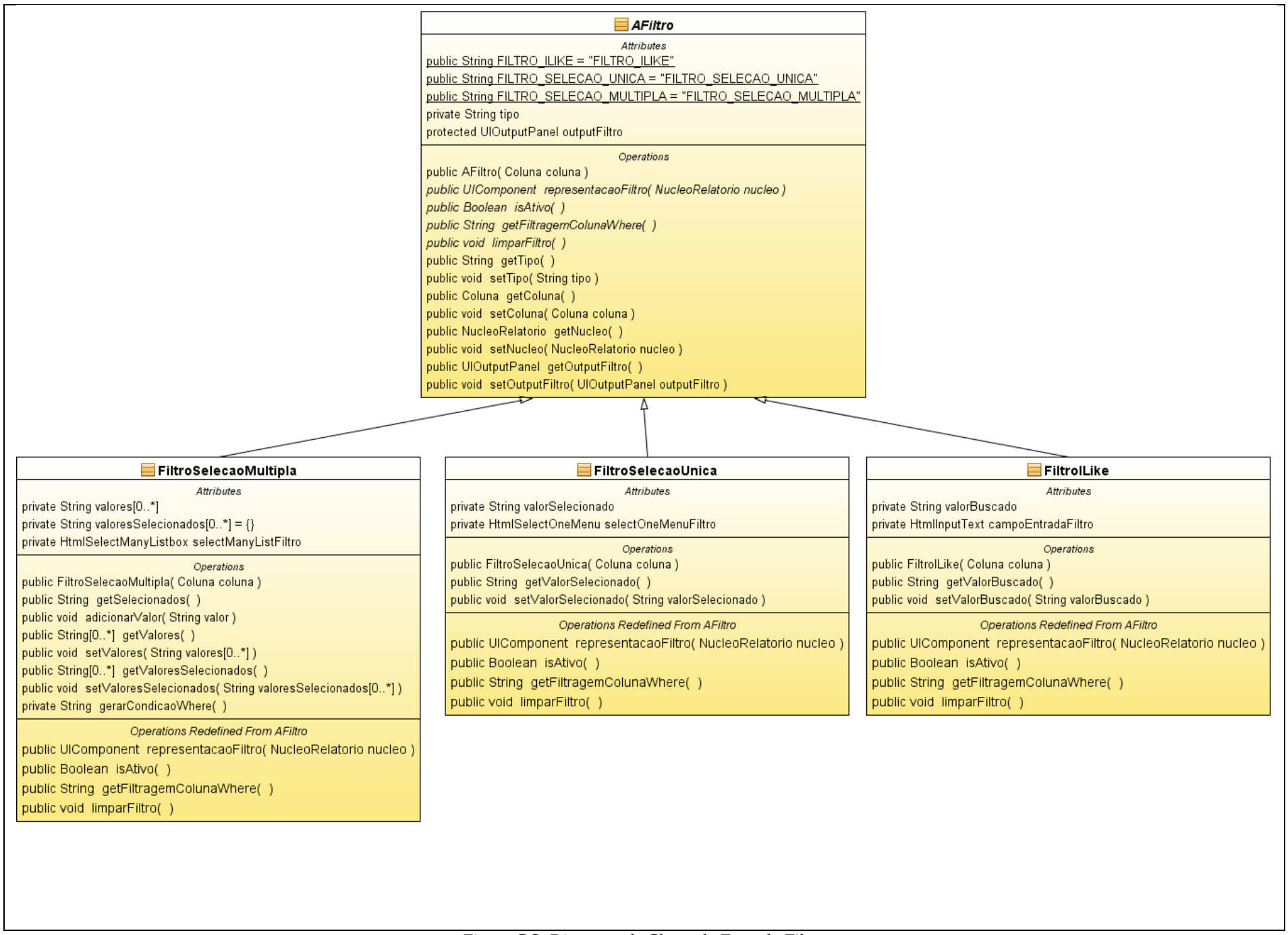

Figura C.3: Diagrama de Classe do FrameletFiltro

C.3 FrameletPersistencia

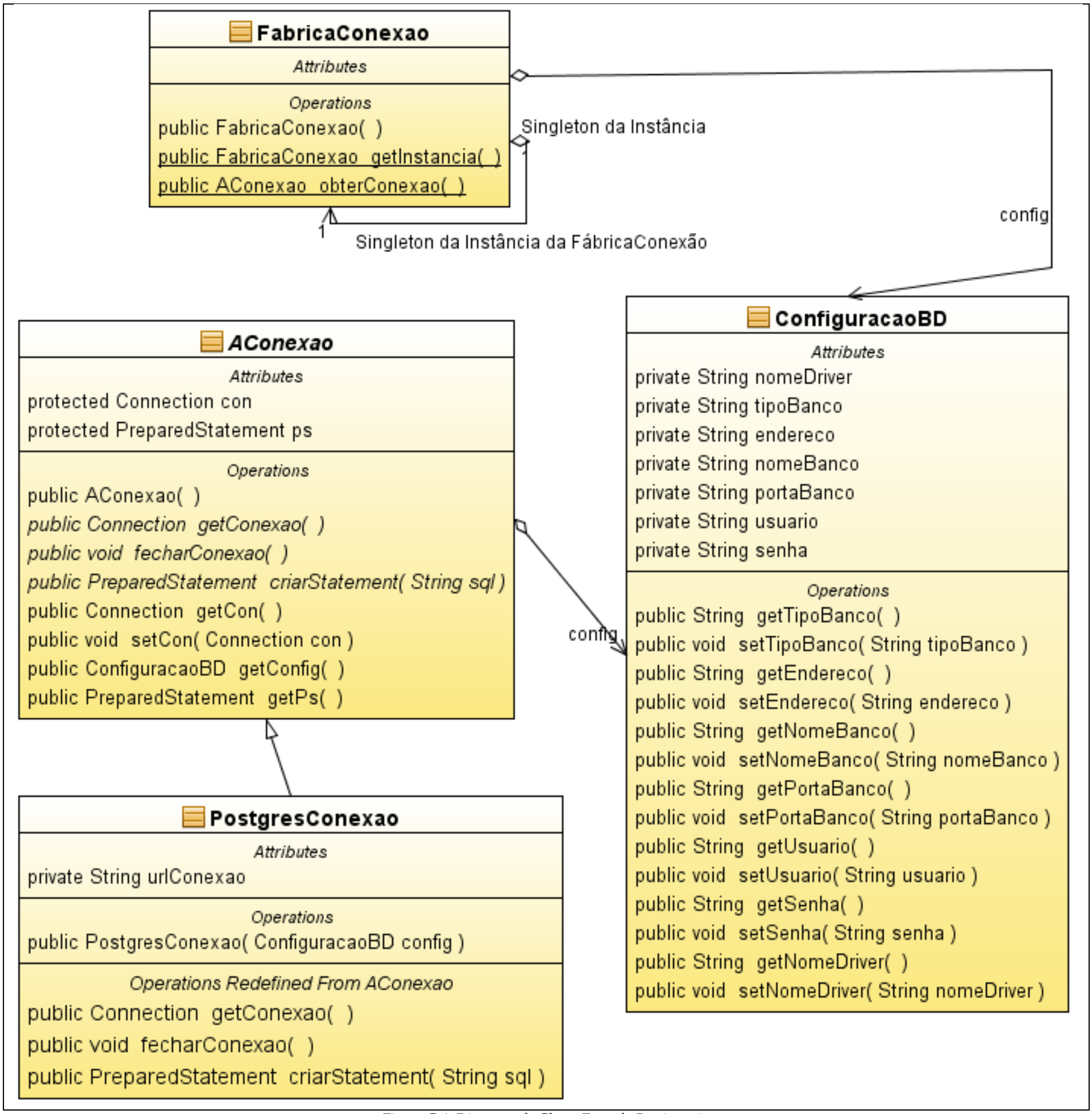

Figura C.4: Diagrama de Classe FrameletPersistencia

C.4 FrameletCabecalho

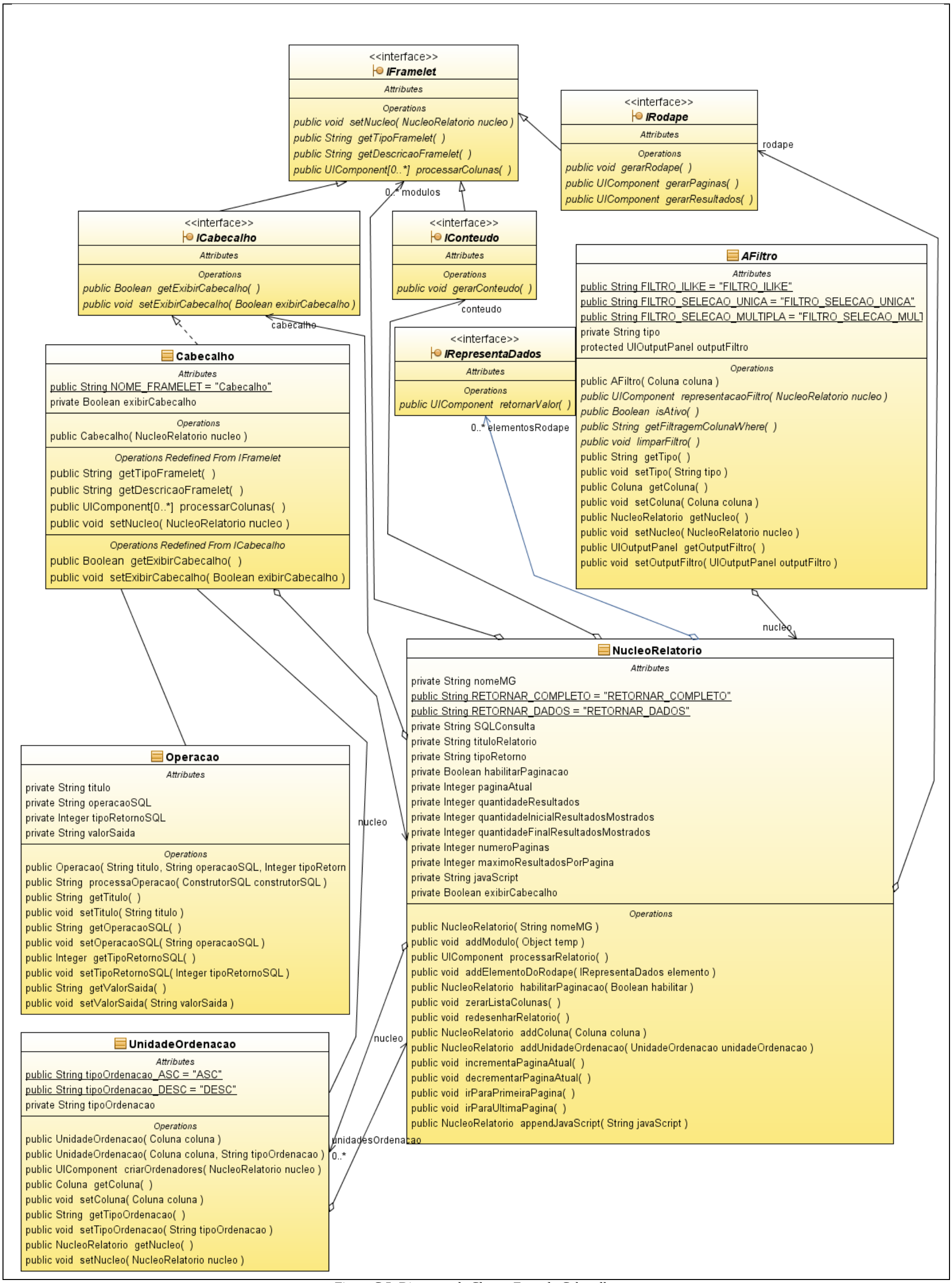

Figura C.5: Diagrama de Classes FrameletCabecalho

C.5 FrameletConteudo

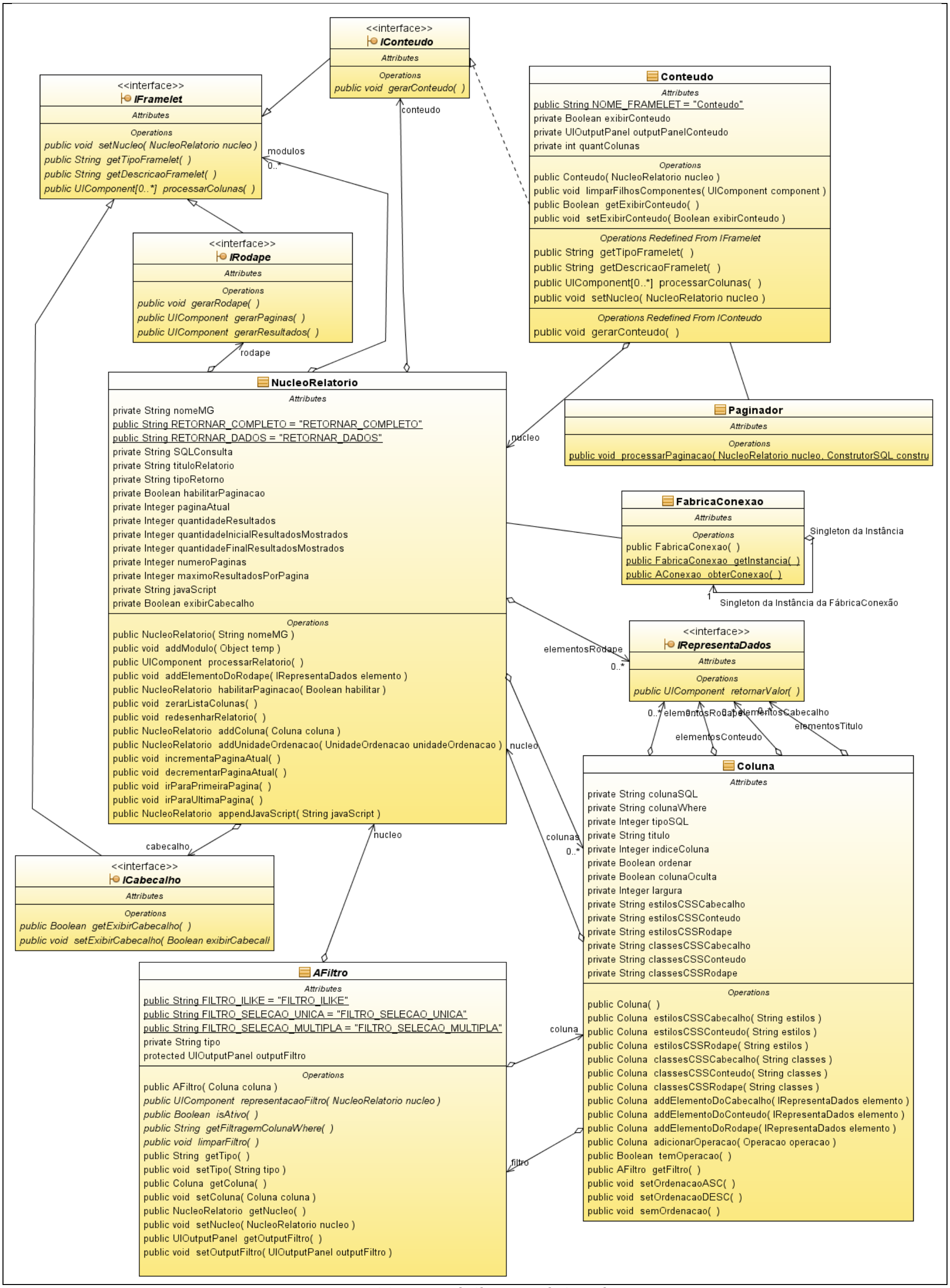

Figura C.6: Diagrama de Classes FrameletConteudo

C.6 FrameletRodape

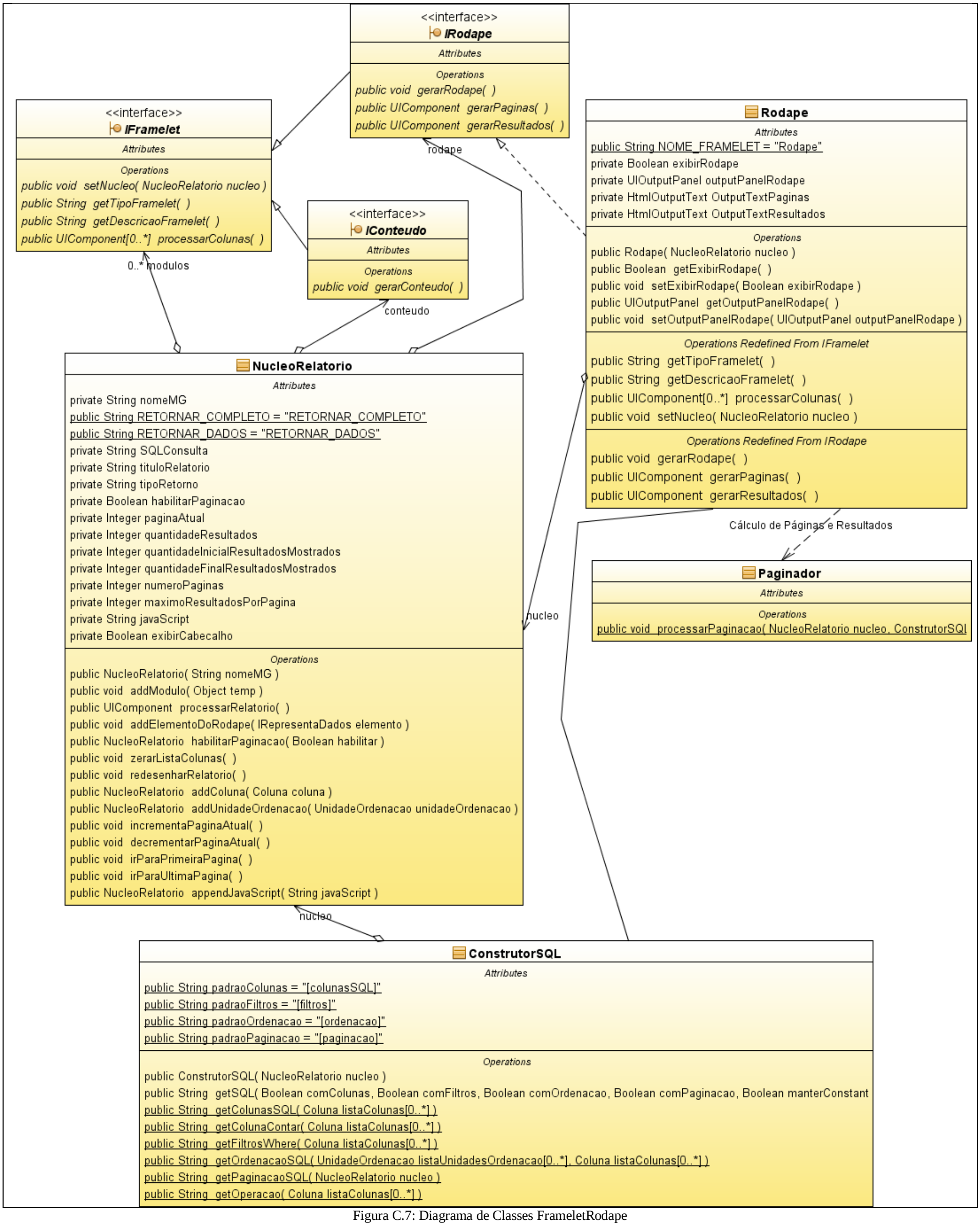

Figura C.7: Diagrama de Classes FrameletRodape

Apêndice D

Diagrama de Componentes

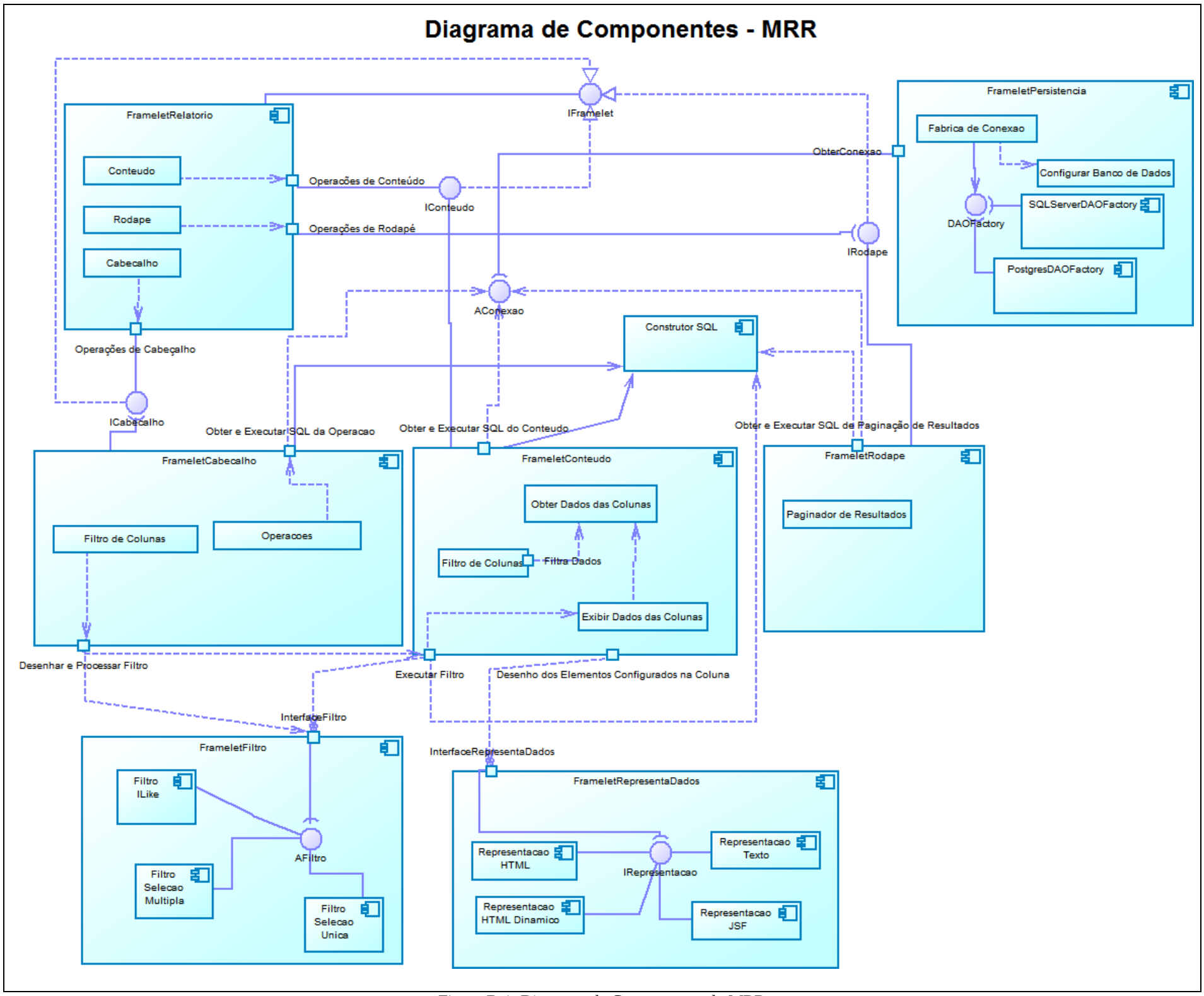

Figura D.1: Diagrama de Componentes do MRR

Referências Bibliográficas

[AHMED e UMRYSH, 2002] AHMED, Khawar Z.; UMRYSH , Cary E. *Desenvolvendo aplicações comerciais em Java em J2EE e UML*. 1. ed. Ciência Moderna, 2002.

[BASHAM et al. 2005] BASHAM, Bryan; SIERRA, Kathy; BATES, Bert. *Use a Cabeça! Servlet e JSP*. 1. ed. Alta Books, 2005.

[BOOCH, 2003] BOOCH, Grady; SCIENTIST, Chief. *Core J2EE Patterns: Best Practices and Designer strategies*. 2. ed. Prentice Hall, 2003.

[BORGES, 2007] BORGES, Guilherme A. P. *Frameworks para o Desenvolvimento WEB.* Monografia – Faculdade de Jaguariúna, Jaguariúna – SP, 2007. Disponível em: <http://bibdig.poliseducacional.com.br/document/?view=66>. Acesso em: 17 mai, 2011.

[BROWN, 1997] BROWN, Alan W., SHORT, Keith. On Components and Objects: The Foundation of Component-Based Development. In: *5º International Symposium on Assessment of Software Tools and Technologies*, p. 112-121, 1997.

[CAMARGO, 2006] CAMARGO, Valter V. *Frameworks transversais: definições, classificações, arquitetura e utilização em um processo de desenvolvimento de software*. Tese (Tese de Doutorado) – ICMC-USP – Instituto de Ciências Matemáticas e de Computação, São Carlos – SP, Agosto 2006. Disponível em: <http://www.teses.usp.br/teses/disponiveis/55/55134/tde-01112006-142356/publico/Tese_Valter_Final.pdf>. Acesso em: 03 jun, 2011.

[CASTILHO, 2007] CASTILHO, Alecindro S. Desenvolvimento do Módulo de Planejamento e Acompanhamento de Frota para a Biblioteca do Projeto Via Digital. Monografia – UFSC - Universidade Federal de Santa Catarina, Florianópolis - SC, Maio 2007. Disponível em:

<projetos.inf.ufsc.br/arquivos_projetos/projeto_676/tccFinalSemestreII.doc>. Acesso em: 11 jul, 2011.

[CECHTICKY e PASETTI, 2002] CECHTICKY, V., PASETTI, Alessandro. *Design and Prototyping of a Software Framework for the AOCS*. Maio 2002. Disponível em: <http://www.pnp-software.com/RealTimeJavaFramework/doc/Documents/SummaryReport.pdf>. Acesso em: 30 ago, 2011.

[CHAPPELL E KAND, 2009] CHAPPELL, Dave, KAND, Khanderao. *SOA & WOA: Article Universal Middleware: What's Happening With OSGi and Why You Should Care*, 17 abr, 2009. Disponível em: <ttp://soa.sys-con.com/node/492519?page=0,2>. Acesso em: 28 jun, 2011.

[CLEMENTS, 2005] CLEMENTS, Paul C. *Faqs: An introduction to software product lines*, Março 2005. Disponível em: <http://www.sei.cmu.edu/library/abstracts/news-at-sei/productlines20053.cfm>. Acesso em: 30 jun, 2011.

[COAD, 1992] COAD, Peter. Object-Oriented Patterns. In: *Communications of the ACM*, V. 35, nº9, p. 152-159, Setembro 1992.

[CONCEIÇÃO, 2008] CONCEIÇÃO, Rodrigo M. JavaServer Faces (JSF): Um Estudo Comparativo entre Bibliotecas de Componentes. Monografia. Unit – Universidade Tiradentes, Aracaju – SE, 2008. Disponível em: <http://pt.scribd.com/doc/9197155/JAVASERVER-FACES-JSF-UM-ESTUDO-COMPARATIVO-ENTRE-BIBLIOTECAS-DE-COMPONENTES> Acesso em: 17 mai, 2011.

[CONTE, MENDES E TRAVASSOS, 2005] CONTE, T.; MENDES, E.; TRAVASSOS, G. H. *Processos de Desenvolvimento para Aplicações Web: Uma Revisão Sistemática*. In: Proceedings of the 11th Brazilian Symposium on Multimedia and Web (WebMedia 2005), v. 1, pp. 107-116, Poços de Caldas - MG. Novembro 2005. Disponível em: <http://lens.cos.ufrj.br:8080/ESEWEB/materials/RSProcessoWeb/2005_10_31_Conte_WebMedia_2005_pubform.pdf>. Acesso em: 14 fev, 2011.

[CORRÊA, 2004] CORRÊA, Alice A. Avaliação do Framework Struts para Implementação de Aplicações Web usando Padrão Modelo-Visão-Controlador. Monografia. UFSC – Universidade Federal de Santa Catarina, Florianópolis – SC, Fevereiro 2004. Disponível em: <http://projetos.inf.ufsc.br/arquivos_projetos/projeto_6/tcc_struts_final_completo.pdf> Acesso em: 15 abr, 2011.

[CSS] CSS, Cascading Style Sheets. Disponível em <http://www.w3.org/Style/CSS>. Acesso em: 10 mai, 2011.

[DONOHOE, 2000] DONOHOE, Patrick. *Software Product Lines*. 1. ed. Springer, 2000.

[D'SOUZA E WILLS, 1998] D'SOUZA, Desmond F.; WILLS, Alan C. *Objects, Components, and Frameworks with UML*. 1. ed. Addison Wesley, 1998.

[DUDNEY et al. 2004] DUDNEY, Bill. LEHR, Jonathan; WILLIS, Bill; MATTINGLY, LeRoy. *Mastering JavaServer Faces*. 1. ed. Wiley, 2004.

[H. DEITEL e P. DEITEL, 2005] DEITEL, H., DEITEL, P. *Java: Como Programar*. 6. ed. Prentice-Hall, 2005.

[FAYAD e SCHMIDT, 1997] FAYAD, M. E., SCHMIDT, D. C. Object-oriented Application frameworks. In: *Communications of the ACM*, Vol. 40, 10 p., 1997.

[FERLIN, 2004] FERLIN, Adriano. Framework para Controle de Concorrência Aplicado a Objetos Compostos. Monografia – UFSC – Universidade Federal de Santa Catarina, Florianópolis - SC, Fevereiro 2004. Disponível em: <http://projetos.inf.ufsc.br/arquivos_projetos/projeto_147/relatorio.pdf> Acesso em: 09 mar, 2011.

[FIELDS e KOLB, 2000] FIELDS, Duane K. e KOLB, Mark. *Desenvolvendo na Web com JavaServer Pages*. Editora Ciência Moderna, 2000.

[FOWLER, 2004] FOWLER, Martin. Inversion of Control Containers and the Dependency Injection Pattern, 24 jan, 2004. Disponível em: <http://www.martinfowler.com/articles/injection.html>. Acesso em: 09 jun, 2011.

[FURLAN, 1998] FURLAN, José D. *Modelagem de objetos através de UML - the unified modeling language*. 1. ed. Markron Books, 1998.

[GAMA et al., 1995] GAMMA, E.; HELM, R. J. R. V. J. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1. ed. Addison Wesley, 1995.

[GARTNER, 2011] GARTNER. Gartner Group. Disponível em: <http://www.gartner.com>. Acesso em: Maio de 2011.

[GEARY e HORSTMANN, 2010] GEARY, David M. HORSTMANN, Cay S. *Core JavaServer faces*. 3. ed. Pearson Education, 2010.

[GINIGE e MURUGESAN, 2001] GINIGE, A.; MURUGESAN, S. Guest Editor's Introduction: The Essence of Web Engineering. In: *IEEE MultiMedia*. Abril, 2001. p. 22 – 25. Disponível em: <http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=917968>. Acesso em: 12 fev, 2011.

[GOODMAN, 2001] GOODMAN, Danny. *JavaScript Bible Gold*. 1. ed. Hungry Minds. 2001.

[GOODWILL, 2002] GOODWILL, James. *Mastering Jakarta Struts*. 1. ed. Wiley, 2002.

[GWT, 2011] Google Web Toolkit (GWT). Disponível em: <http://code.google.com/intl/pt-BR/webtoolkit/>. Acesso em: 06 ago, 2011.

[GROTT, 2003] GROTT, Marcio C. *Reutilização de Soluções com Patterns e Frameworks na Camada de Negócio*. Monografia – FURB – Universidade Regional de Blumenau, Blumenau – SC, Junho 2003. Disponível em: <http://campeche.inf.furb.br/tccs/2003-I/2003-1marciocgrottvf.pdf>. Acesso em: 14 fev, 2011.

[GURP, 2000] GURP, van J., Variability in Software Systems: The Key to Software Reuse, Outubro, 2000. Disponível em: <http://www.rug.nl/informatica/onderzoek/programmas/softwareengineering/old%20search%20website/publications/licentiatethesis.pdf>. Acesso em: 01 jun 2011.

[GURP e BOSCH, 2001] GURP, van J., BOSCH, J. Design, implementation and evolution of object oriented frameworks: concepts and guidelines. In: *Journal Software—Practice & Experience archive*. Março, 2001. v. 31. Disponível em: <http://www.jillesvangurp.com/static/spejvg.pdf>. Acesso em: 01 ago, 2011.

[GUTIERREZ, 2009] GUTIERREZ, Felipe G. *Geração Automática de Interfaces Gráficas a partir de diagramas de classes UML.* Monografia - Universidade Federal de Lavras, Lavras – MG, Julho 2009. Disponível em: <http://www.bcc.ufla.br/monografias/2010/Geracao_Automatica_de_Interfaces_Graficas_a_Partir_de_Diagramas_de_Classes_UML.pdf>. Acesso em: 22 mar, 2011.

[HAMMANT, 2008] HAMMANT, P. Inversion of Control. Fevereiro 2008. Disponível em: <http://www.picocontainer.org/Inversion-of-Control>. Acessado: 01 junho 2011.

[HEINEMAN e COUNCILL, 2001] HEINEMAN, G.; COUNCILL, W. *Component-based software engineering: Putting the pieces together*. 1. ed. Addison-Wesley, 2001.

[HURWITZ et al., 2009] HURWITZ, J.; BLOOR, R.; KAUFMAN, M.; HALPER, F. *Service oriented architecture for dummies*. 2. ed. John Wiley and Sons, 2009.

[HUSTED et al., 2003] HUSTED Ted, DUMOULIN Cedric, FRANCISCUS George et al. *Struts in Action*. Manning Publications. 1. ed. 2003.

[JACYNTHO, 2008] JACYNTHO, Mark A. Processos para Desenvolvimento de Aplicações Web. In: *Pontifícia Universidade Católica do Rio de Janeiro*. Julho 2008. Rio de Janeiro: [s.n.]

[JBOSSRICHFACES] JBOSSRICHFACES. Richfaces Project Page. Disponível em: <http://www.jboss.org/richfaces>. Acesso em: 10 mar, 2011.

[JCP] JCP. Java Community Process. Disponível em: <http://jcp.org>. Acessado em 09 julho 2011.

[JOHNSON e FOOTE, 1988] JOHNSON, R.; FOOTE, B. Designing Reusable Classes. *Journal of Object Oriented Programming – JOOP*, New York – EUA, Vol. 2. n. 1, p. 22 – 35, Julho 1988. Disponível em:

<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.101.8594&rep=rep1&type=pdf>. Acesso em: 08 jul, 2011.

[JONHSON et al., 2002] JONHSON Mark, STEARNS Beth, SINGH Inderjeet. et al. *Desingin Enterprise Applications with the J2EE Platform*. 1. ed. Addison-Wesley, 2002.

[JSR32] JSR 32. Java Java Specification Request. Disponível em: <http://jcp.org/en/jsr/detail?id=32>. Acesso em: 09 jul, 2011.

[KATZ, 2008] KATZ, Max. *Practial RichFaces*. 1. ed. Apress, 2008.

[KONO, 2008] KONO, Anderson T. S. *METAACAD GERENTE: um modelo para indexação e gerência de recursos acadêmicos*. Monografia – UNOESTE – Universidade do Oeste Paulista, Presidente Prudente – SP, Novembro 2008. Disponível em: <http://www2.unoeste.br/~chico/FIPP/projetos/projeto2008/Monografia_Toshio.pdf>. Acesso em: 09 mar, 2011.

[KRUEGER, 1992] KRUEGER, C. Software reuse. In: *ACM Computing Surveys*, v. 24, n. 02, p. 131–183, 1992.

[KUEHNE, 2007] KUEHNE, Bruno T. *Servidores Web*. Monografia – ICMC – USP – Instituto de Ciências Matemáticas e de Computação de São Carlos, São Carlos – SP, Novembro 2007. Disponível em: <http://www.lasdpc.icmc.usp.br/disciplinas/pos-graduacao/sistemas-distribuidos/2007/monografias-seminarios/Monografia_WS_2007%20-%20Bruno.pdf>. Acesso em: 15 abr, 2011.

[KURNIAWAN, 2002] KURNIAWAN, Budi. *Java for the Web with Servlets, JSP, and EJB: A Developer's Guide to J2EE Solutions*. 1. ed. New Riders Publishing, 2002.

[LARMAN, 2002] LARMAN, Graig. *Applying UML and Patterns As Introduction to Object-Oriented Analysis and Design and the Unified Process*. 2. ed. Prentice Hall PRT, 2002.

[LOBO FILHO, 2010] LOBO FILHO, Roberto J. H. *Integração entre HTML5 e JSF 2.0 em Aplicações Web Offline*. Monografia – UFSC – Universidade Federal de Santa Catarina, Florianópolis - SC, Junho 2010. Disponível em:

<http://projetos.inf.ufsc.br/arquivos_projetos/projeto_809/TCC%2BArtigo.pdf>. Acesso em: 14 fev, 2011.

[LUCRÉDIO, 2009] LUCRÉDIO, Daniel. *Uma Abordagem Orientada a Modelos para Reutilização de Software*. Tese (Tese de Doutorado) – ICMC-USP – Instituto de Ciências Matemáticas e de Computação, São Carlos – SP, Julho 2009. Disponível em: <http://www.icmc.usp.br/~lucredio/downloads/files/teseDoutoradoDanielLucredio.pdf>. Acesso em: 10 jun, 2011.

[MANN, 2005] MANN, Kito D. *JavaServer Faces In Action*. 1. ed. Manning, 2005.

[MARAFON, 2006] MARAFON, Diego L. Integração JavaServer Faces e Ajax: Estudo da Integração entre as Tecnologias JSF e Ajax. Monografia – UFSC – Universidade Federal Santa Catarina, Florianópolis – SC, 2006. Disponível em: <http://projetos.inf.ufsc.br/arquivos_projetos/projeto_491/TCC%20-%20Diego%20Luiz%20Marafon.pdf>. Acesso em: 15 mar, 2011.

[METSKER, 2004] METSKER, Steven John. *Padrões de projeto em Java*. 1. ed. Editora Bookman, 2004.

[MURTA, 2006] MURTA, Leonardo G. P. *Gerência de Configuração em Desenvolvimento Baseado em Componentes*. Tese (Tese de Doutorado) – COPPE – Universidade Federal do Rio de Janeiro, Rio de Janeiro – RJ, Outubro 2006. Disponível em: <http://reuse.cos.ufrj.br/prometeus/publicacoes/odyssey-scm.pdf>. Acesso em: 24 jun, 2011.

[NASH, 2003] NASH, M. *Java Frameworks and Components: Accelerate Your Web Application Development*. 1. ed. , Cambridge University, 2003.

[OLIVEIRA e PAULA, 2009] OLIVEIRA, Felipe C.; PAULA, Leonardo L. *Engenharia de Software baseada em componentes: uma abordagem prática em ambientes Web.* Monografia – Unb – Universidade de Brasília, Brasília – DF, Dezembro 2009. Disponível em: <http://monografias.cic.unb.br/dspace/bitstream/123456789/233/1/Monografia.pdf>. Acesso em: 15 fev, 2011.

[ORACLE, 2011] ORACLE. The Java EE 6 Tutorial. Disponível em: <http://download.oracle.com/javaee/6/tutorial/doc/docinfo.html>. Acesso em: 25 mai, 2011.

[ORFALI, HARKEY E EDWARDS, 1999] ORFALI, R.; HARKEY, D; EDWARDS, J. Client/Server Survival Guide. 3. ed. John Wiley, 1999.

[OTTINGER, 2008] OTTINGER, Joseph. The Server Side: What is an App Server?. Setembro 2008 Disponível em: <http://www.theserverside.com/news/1363671/What-is-an-App-Server>. Acesso em: 10 mar, 2011.

[PASETTI, 2002] PASETTI, Alessandro. Software Frameworks and Embedded Control System. 1. ed. Springer, 2002.

[PASETTI, 2011] PASETTI, Alessandro. *Aocs Framework Project. System*. Disponível em: <http://www.pnp-software.com/AocsFramework/ProjectOverview.html>. Acesso em: 10 ago, 2011.

[PASETTI e PREE, 2000] PASETTI, Alessandro, PREE, Wolfgang. Two Novel Concepts for Systematic Product Line Development. In: *Proceedings of the First Software Product Line Conference*, 2000, p. 249-270. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.28.5634&rep=rep1&type=pdf>. Acesso em: 13 mar, 2011.

[PREE, 1999] PREE, Wolfgang. Lean Product-Line Architectures for Client-Server Systems – Concepts & Experience, 1999. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.28.6656&rep=rep1&type=pdf>. Acesso em: 24 mai, 2011.

[PREE E KOSKIMIES, 1999] PREE, Wolfgang; KOSKIMIES, Kai. Framelets – small is beautiful. In: *Building Application Frameworks – Object-Oriented Foundations of Framework Design*, Massachusetts – EUA: Wiley, 1999, p. 411-414. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.28.6945&rep=rep1&type=pdf>. Acesso em: 13 mar, 2011.

[PREE E KOSKIMIES, 2000] PREE, Wolfgang; KOSKIMIES, Kai. Framelets—Small and Loosely Coupled Frameworks. In: *Journal ACM Computing Surveys*. Março de 2000. New York – EUA: [s.n.]. Disponível em:

<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.28.6316&rep=rep1&type=pdf>. Acesso em: 05 mai, 2011.

[PRESSMAN, 2010] PRESSMAN, Roger S. *Engenharia de software*. 6. ed. McGraw Hill, 2010.

[QUADROS, 2002] QUADROS, Claudia I. Uma breve visão histórica do jornalismo on-line. In: XXV Congresso Brasileiro de Ciências da Computação. Salvador – BA, 2002. Disponível em: <http://galaxy.intercom.org.br:8180/dspace/bitstream/1904/18639/1/2002_NP2QUADROS.pdf>. Acesso em: 06 ago, 2011.

[RAMOS, 2008] RAMOS, José Y. A. Comparação entre os Principais Frameworks Java para o desenvolvimento de aplicações WEB 2.0. In: Revista Sistemas de Informação & Gestão de Tecnologia. Belém - PA, 2008. Disponível em: <http://www3.iesam-pa.edu.br/ojs/index.php/sistemas/article/viewFile/502/402 - 502-1774-1-PB.pdf> Acesso em: 25 fev, 2011.

[REIS, 2002] REIS, Rodrigo Q. *APSEE-Reuse: um Meta-Modelo para Apoiar a Reutilização de Processos de Software*. Tese (Tese de Doutorado) – UFRS – Universidade Federal do Rio Grande do Sul, Porto Alegre – RS, Julho 2002. Disponível em: <http://www.lume.ufrgs.br/bitstream/handle/10183/1622/000353745.pdf?sequence=1>. Acesso em: 24 jun, 2011.

[REENSKAUG, 1979] REENSKAUG, Trygve. Models Views Controllers, Relatório Técnico Xerox PARC, Dezembro, 1979. Disponível em: <http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf>. Acesso em: 21 set, 2011.

[REENSKAUG, 1995] REENSKAUG, Trygve. *Working with objects: The OOram Software Engineering Method*, Março, 1995. Disponível em: <http://heim.ifi.uio.no/~trygver/1996/book/WorkingWithObjects.pdf>. Acesso em: 21 set, 2011.

[RIBEIRO et al., 2006] RIBEIRO, Hudson S.; PEREIRA, Jefferson G. F.; NUNES, Luís P. J.; BARRÉRE, Eduardo. Integração de Tecnologias para Desenvolvimento de Sistemas Web,

utilizando a metodologia AJAX. In: *III SEGeT – Simpósio de Excelência em Gestão e Tecnologia.*, 2006. Resende - RJ: [s.n.]. Disponível em: <http://www.info.aedb.br/seget/artigos06/304_Artigo_SEGET.pdf>. Acesso em: 20 jun, 2011.

[SANTOS, 2007] SANTOS, Thiago R. *Análise e Comparação de Frameworks para Desenvolvimento Web em Java*. Monografia – UFSC - Universidade Federal de Santa Catarina, Florianópolis - SC, Julho 2007. Disponível em: <http://projetos.inf.ufsc.br/arquivos_projetos/projeto_669/TCC-ThiagoRobertoSantos-final.pdf>. Acesso em: 24 abr, 2011.

[SANTOS, 2008] SANTOS, Gustavo P. *Aplicação do Padrão de Projeto MVC com JSF*. Monografia – FAJ - Faculdade de Jaguariúna, Jaguariúna - SP, Dezembro 2008. Disponível em: <http://bibdig.poliseducacional.com.br/document/?view=183>. Acesso em: 22 set, 2011.

[SHALLOWAY E TROTT, 2004] SHALLOWAY, Alan; TROTT, James R. *Explicando Padrões de Projeto: uma Nova Perspectiva em Projeto Orientado a Objeto*. 1. ed. Bookman, 2004. Disponível em: <http://books.google.com.br/books?id=6y2jOZX1tFsC&dq=Padr%C3%B5es+de+Responsabilidade+padr%C3%B5es+de+projeto&lr=lang_pt&source=gbs_similarbooks_s&cad=1>. Acesso em: 23 mai, 2011.

[SILVA, 2000] SILVA, Ricardo P. *Suporte ao Desenvolvimento e Uso de Frameworks e Componentes.* Tese. UFRS - Universidade Federal do Rio Grande do Sul, Porto Alegre – RS, Março 2000. Disponível em: <http://www.inf.ufsc.br/~ricardo/download/tese.pdf>. Acesso em: 17 fev, 2011.

[SOMMERVILLE, 2007] SOMMERVILLE, Ian. *Engenharia de software*. 8. ed. Pearson Addison Wesley, 2007.

[SOURCEFORGE] SOURCEFORGE. SourceForge.net: Find, Create, and Publish Open Source software for free. Disponível em: < http://www.sourceforge.net >. Acesso em: 28 mai, 2011.

[SOUZA, 2007] SOUZA, Vitor E. S. *FrameWeb: um método baseado em frameworks para o Projeto de Sistemas de Informação Web*. Dissertação (Dissertação de mestrado) – UFES – Universidade Federal do Espírito Santo, Vitória - ES, Julho 2007. Disponível em: <http://www.inf.ufes.br/~falbo/files/DissertacaoVitorSouza.pdf>. Acesso em: 23 fev, 2011.

[SPAGNOLI e BECKER, 2003] SPAGNOLI, Luciana A.; BECKER, Karin. Um Estudo Sobre o Desenvolvimento Baseado em Componentes. Relatório Técnico. PUCRS – Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre – RS, Maio 2003. Disponível em: <http://www3.pucrs.br/pucrs/files/uni/poa/facin/pos/relatoriostec/tr026.pdf>. Acesso em: 15 fev, 2011.

[SPRING, 2011] SPRINGSOURCE. Spring Home. Disponível em: <http://www.springsource.com>. Acessado em: 06 ago, 2011.

[STRUTS, 2011] Apache Struts. Disponível em: <http://struts.apache.org>. Acessado em: 06 ago, 2011.

[TALIGENT, 2003] TALIGENT Inc. Building Object-Oriented Frameworks. A Taligent White Paper, 2003.

[TEIXEIRA, 2008] TEIXEIRA, Marcelo. *Estudo da Utilização de Frameworks no Desenvolvimento de Aplicações Web*. Monografia – UNIOESTE – Universidade Estadual do Oeste do Paraná, Cascavel – PR, Dezembro 2008.

[TEIXEIRA, 2004] TEIXEIRA, Mário A. M. *Suporte a Serviços Diferenciados em Servidores Web: modelos e algoritmos.* Tese (Tese de Doutorado) – ICMC – USP – Instituto de Ciências Matemáticas e de Computação, São Carlos – SP, Julho 2004. Disponível em: <http://www.deinf.ufma.br/~mario/producao/tese_swds.pdf>. Acesso em: 03 mar, 2011.

[VILLELA, 2000] VILLELA, Regina M. B. *Busca e Recuperação de Componentes em Ambientes de Reutilização de Software.* Tese (Tese de Doutorado) – UFRJ – Universidade Federal do Rio de Janeiro, Rio de Janeiro – RJ, Dezembro 2000. Disponível em: <http://reuse.cos.ufrj.br/files/publicacoes/doutorado/Dou_Regina.pdf>. Acesso em 20 jun, 2011;

[YEAGER e MCGRATH, 1996] YEAGER, N. J.; MCGRATH, R. E. *Web Server Technology: the Advanced Guide for World Wide Web Information Providers*. 1. ed. Morgan Kaufmann, 1996.

[WINCKLER E PIMENTA, 2002] WINCKLER, M.; PIMENTA, M. Avaliação de Usabilidade de Sites Web. In: Nedel, Luciana (Org.) X Escola de Informática da SBC-Sul (ERI2002), Caxias do Sul – RS, Criciúma – SC, Cascavel – PR: [s.n], 2002. p. 85-137. Disponível em: <http://www.irit.fr/~Marco.Winckler/2002-winckler-pimenta-ERI-2002-cap3.pdf>. Acesso em: 15 abr, 2011.