

LIGA DE ENSINO DO RIO GRANDE DO NORTE
FACULDADE NATALENSE PARA O DESENVOLVIMENTO
DO RIO GRANDE DO NORTE

VICENTE DE PAULA MELO FILHO

ANÁLISE DA APLICABILIDADE DE PADRÕES DE PROJETOS EM UMA
ARQUITETURA DE DESENVOLVIMENTO DE SISTEMAS WEB UTILIZANDO A
TECNOLOGIA JEE.

NATAL/RN

2007

LIGA DE ENSINO DO RIO GRANDE DO NORTE
FACULDADE NATALENSE PARA O DESENVOLVIMENTO
DO RIO GRANDE DO NORTE

VICENTE DE PAULA MELO FILHO

ANÁLISE DA APLICABILIDADE DE PADRÕES DE PROJETOS EM UMA
ARQUITETURA DE DESENVOLVIMENTO DE SISTEMAS WEB UTILIZANDO A
TECNOLOGIA JEE.

NATAL/RN

2007

VICENTE DE PAULA MELO FILHO

**ANÁLISE DA APLICABILIDADE DE PADRÕES DE PROJETOS EM UMA
ARQUITETURA DE DESENVOLVIMENTO DE SISTEMAS WEB UTILIZANDO A
TECNOLOGIA JEE.**

Monografia apresentada ao Curso de Especialização em Desenvolvimento de Sistemas Corporativos da Faculdade Natalense para o desenvolvimento do Rio Grande do Norte como requisito obrigatório para obtenção do título de Especialista em Sistemas Corporativos.

Orientador:

Gleydson Lima, M.Sc.

NATAL/RN

2007

Catálogo na Publicação – Biblioteca da FARN

Setor de Processos Técnicos

Melo Filho, Vicente de Paula

Análise da aplicabilidade de padrões de projetos em uma arquitetura de desenvolvimento de sistemas Web utilizando a tecnologia JEE / Vicente de Paula Melo Filho. – Natal: FARN, 2007.

86 f.

Orientador: Gleydson de Azevedo Ferreira Lima (M. Sc.)

Monografia (Especialização em Sistemas Corporativos) – Faculdade Natalense para o Desenvolvimento do Rio Grande do Norte.

1. Padrões de Projeto – Arquitetura Web – Monografia. 2. Projetos de Software – Monografia. 3. Padrões JEE – Monografia. I. Título.

RN/FARN/BC

CDU 004.4

RB- 30.719
FARN
BIBLIOTECA
Nº Reg 10009110
16.04.08

VICENTE DE PAULA MELO FILHO

**ANÁLISE DA APLICABILIDADE DE PADRÕES DE PROJETOS EM UMA
ARQUITETURA DE DESENVOLVIMENTO DE SISTEMAS WEB UTILIZANDO A
TECNOLOGIA JEE.**

Monografia apresentada ao Curso de Especialização em Desenvolvimento de Sistemas Corporativos da Faculdade Natalense para o desenvolvimento do Rio Grande do Norte, aprovada em ____ de ____ de _____, pela banca examinadora composta pelos seguintes membros:



Gleydson Lima, M.Sc.

Orientador

NATAL/RN

2007

RESUMO

O uso de padrões de projeto (*design patterns*) é considerado uma das mais valiosas técnicas para produzir projetos software com qualidade. Nesse trabalho é feita uma análise de padrões de projeto que são utilizados no desenvolvimento de um sistema WEB utilizando a tecnologia JEE (*Java Enterprise Edition*). Esses padrões de projeto são propostos para garantir melhor reuso e manutenibilidade dos componentes do sistema. Será utilizado como estudo de caso o projeto Hercules da empresa Natalsoft Ltda.

As principais APIs da tecnologia Java que são utilizadas no projeto serão apresentadas, de forma resumida, para que se tenha uma fundamentação teórica básica dos assuntos que serão abordados. Os principais padrões de projeto que foram usados no projeto serão apresentados de duas formas: os padrões GoF serão apresentados de forma resumida e os padrões JEE serão apresentados com maiores detalhes.

A arquitetura do sistema Hércules será mostrada no fim do trabalho. Serão exibidas as principais classes e que padrões foram utilizados para resolver determinados tipos de problema.

Palavras chave: Plataforma Web. Padrões de Projeto. JEE.

ABSTRACT

The use of design patterns is considered the most valuable technique to make software projects with quality. In this paper is made an analysis of design patterns that are used in the development of a WEB system using the JEE (*Java Enterprise Edition*) technology. Those design patterns are proposed to guarantee better reusability and maintainability of system's components. It will be used as case study the Project Hercules of the company Natalsoft Ltda.

The main APIs of the technology Java that are used in the project will be presented, in a summarized way, to do a basic theoretical foundations of the subjects that will be approached. The main project patterns that were used in the project will be presented in two ways: the patterns GoF will be presented of summarized form and the patterns JEE will be presented with more details.

The architecture of the system Hercules will be shown at the end of the work. The main classes will be exhibited and that patterns were used to solve certain problem types.

Key words: Web Platform. Design Patterns. JEE.

LISTA DE FIGURAS

Figura 1 - O conceito do contêiner	15
Figura 2 - Catálogo GoF organizados em famílias e escopo	22
Figura 3 - Modelo de cinco camadas para divisão lógica dos aspectos	25
Figura 4 - Relacionamentos dos padrões JEE	28
Figura 5 - Estrutura de classes do padrão Template Method	29
Figura 6 - Diagrama de classes do Padrão Singleton	31
Figura 7 - Diagrama de Classes do Padrão Adapter (Class Adapter).....	32
Figura 8 - Diagrama de Classes do Padrão Adapter (Object Adapter).....	33
Figura 9 - Diagrama de Classes do Padrão Decorator.....	33
Figura 10 - Diagrama de classes do padrão Proxy	35
Figura 11 - Diagrama de classes do padrão Template Method	36
Figura 12 - Diagrama de classes do padrão Command	37
Figura 13 - Diagrama de classes do padrão Intercepting Filter.....	39
Figura 14 - Diagrama de seqüência do Intercepting Filter	40
Figura 15 - Diagrama de classes do Front Controller.....	42
Figura 16 - Diagrama de seqüência do Front Controller.....	44
Figura 17 - Diagrama de Classes do View Helper	45
Figura 18 - Diagrama de seqüência do View Helper.....	46
Figura 19 - Diagrama de seqüência simples do View Helper	46
Figura 20 - Diagrama de classes do Composite View.....	48
Figura 21 - Diagrama de seqüência do Composite View	49
Figura 22 - Diagrama de classes do padrão Business Delegate	52
Figura 23 - BusinessDelegate com um diagrama de seqüência ID	54
Figura 24 - Diagrama de Classe Session Facade.....	58
Figura 25 - Diagrama de sequencia Session Facade.....	59
Figura 26 - Diagrama de classes Service Locator	62

Figura 27 - Diagrama de Seqüência Service Locator	62
Figura 28 - Data Access Object (DAO).....	66
Figura 29 - Camadas do Sistema Hércules	74
Figura 30 - Definição de Templates para páginas do sistema	75
Figura 31 - Pacote br.com.natalsoft.arq.jsf.....	78
Figura 32 - Pacote: br.com.natalsoft.arq.dao.....	79
Figura 33 - Diagrama de classes do pacote br.com.natalsoft.arq.negocio.....	81
Figura 34 - Diagrama de seqüência para execução de comandos	81
Figura 35 - Diagrama de classes do pacote br.com.natalsoft.arq.erros	82

ÍNDICE DE LISTAGENS

Listagem 1 - Código de Exemplo do Padrão Singleton.....	31
Listagem 2 - Código-fonte da Interface Componente	34
Listagem 3 - Código-fonte da classe Decorator	34
Listagem 4 - Código-fonte do DecoradorContretoUm.....	34
Listagem 5 - Trecho do arquivo web.xml.....	76
Listagem 6 - Código-fonte do HerculesFacesControler	78

SUMÁRIO

1	INTRODUÇÃO	11
2	FUNDAMENTAÇÃO TEÓRICA	13
2.1	JAVA ENTERPRISE EDITION (JEE)	13
2.1.1	Contêiner JEE.....	14
2.1.2	Servlets.....	17
2.1.3	Java Server Pages (JSP)	17
2.1.4	Java ServerFaces (JSF)	18
2.1.5	Enterprise Java Beans (EJB)	19
2.1.6	Java Message Service (JMS).....	19
2.1.7	Java Persistence API (JPA).....	20
3	PADRÕES DE PROJETO DE SOFTWARE	21
3.1	O QUE SÃO PADRÕES?	21
3.2	CATÁLOGO DE PADRÕES GoF	22
3.3	CATÁLOGO DE PADRÕES JEE	23
3.3.1	Como utilizar o catálogo de padrões JEE?.....	25
3.4	O ENFOQUE NAS CAMADAS.....	25
3.4.1	Camada de Cliente	26
3.4.2	Camada de apresentação	26
3.4.3	Camada de negócios.....	26
3.4.4	Camada de integração	26
3.4.5	Camada de recursos.....	26
3.5	RELACIONAMENTOS ENTRE PADRÕES J2EE	27
4	PADRÕES GOF	29
4.1	PADRÕES DE CRIAÇÃO	29
4.1.1	Factory Method	29
4.1.2	Singleton	30
4.2	PADRÕES ESTRUTURAIS	31
4.2.1	Adapter	31
4.2.2	Decorator.....	33
4.2.3	Proxy	34
4.3	PADRÕES DE COMPORTAMENTO.....	36

4.3.1	Template Method	36
4.3.2	Command	37
5	PADRÕES JEE	38
5.1	PADRÕES DA CAMADA DE APRESENTAÇÃO	38
5.1.1	Intercepting Filter	38
5.1.2	Front Controller	41
5.1.3	View Helper	44
5.1.4	Composite View	47
5.2	CAMADA DE NEGÓCIOS	50
5.2.1	Business Delegate	50
5.2.2	Session Facade	56
5.2.3	Service Locator	60
5.3	CAMADA DE ACESSO A DADOS	64
5.3.1	Data Access Object (DAO)	64
5.4	OUTROS PADRÕES IMPORTANTES	69
5.4.1	Padrão Open Session in view	69
5.4.2	Padrão EJB Command	69
6	ARQUITETURA DO SISTEMA HÉRCULES	73
6.1	INTRODUÇÃO	73
6.2	CAMADAS	73
6.2.1	Camada de Apresentação	74
6.2.2	Camada de Persistência	78
6.2.3	Camada de Negócio	79
7	CONCLUSÕES	84

1 INTRODUÇÃO

O processo de desenvolvimento de software é difícil, lento e caro. Ainda não há tecnologia para se criar softwares grandes, desde o início, de forma rápida e que possua poucos defeitos (*bugs*). Há algumas soluções para minimizar esse problema e praticamente todas se baseiam em reusabilidade. A proposta da reusabilidade é aproveitar componentes de *software* já existentes e funcionais, mas isso não é uma tarefa fácil e geralmente é um investimento de longo prazo, pois exige mais *expertise* dos desenvolvedores de software. Mesmo utilizando o paradigma de programação orientado a objetos ocorrem alguns problemas que dificultam o reuso e a manutenibilidade dos *softwares*. Para ajudar na resolução desses problemas, foram catalogadas algumas técnicas que podem ser consideradas “boas idéias de projeto de software”. Essas técnicas são chamadas de *design patterns*, como será visto adiante.

Para aumentar a complexidade, o desenvolvimento de aplicações para a WEB modifica uma série de conceitos em relação ao desenvolvimento de tradicionais aplicações *Desktop*. Uma aplicação WEB executa num ambiente distribuído, onde cada parte que compõe o programa pode estar localizada em uma máquina diferente. As aplicações desenvolvidas utilizando o paradigma cliente-servidor já vivenciam hoje uma parte desta experiência, onde a parte relativa à interface com o usuário reside na estação do cliente e a parte de acesso aos dados, no servidor de banco de dados. A lógica da aplicação pode ficar dividida entre o cliente e o servidor. Esta é uma arquitetura em duas camadas.

As aplicações WEB utilizam uma arquitetura multicamadas, onde as funções executadas pelas aplicações podem estar distribuídas por uma rede de computadores. Elas fazem uso de uma infra-estrutura de rede que é o padrão atual adotado pela Internet.

Segundo a Sun (2007) a plataforma *Java Enterprise Edition* (JEE) define um padrão para o desenvolvimento de aplicações multicamadas. Nesta arquitetura, a camada que contém as regras de negócio, normalmente implementada utilizando Enterprise Javabeans, pode ficar concentrada no servidor de aplicações, sendo compartilhada com diversas aplicações clientes. As aplicações clientes não contêm a lógica do negócio, atendo-se somente à camada de apresentação. Na camada de apresentação, normalmente são utilizadas as tecnologias de *Servlets*, *Java Server Pages* e *Java ServerFaces*.

Segundo Marinescu (2002), sem um conjunto de boas práticas de modelagem, o desenvolvimento utilizando a arquitetura multicamadas JEE, pode se tornar muito difícil.

Segundo Alur (2002), uma boa maneira de adquirir experiência em projeto é pela utilização de padrões de projeto que se constituem em um moderno e importante mecanismo para a elaboração de projetos orientados a objetos.

Esse trabalho está organizado em seis capítulos. Segue abaixo um resumo de cada um deles:

O capítulo 1 é apenas a introdução do trabalho. Será feita uma breve apresentação sobre algumas dificuldades para desenvolver software.

O capítulo 2 apresenta, de forma bem resumida, o que é a tecnologia JEE e quais são as principais APIs (*Application Programming Interface*) que são utilizadas no projeto Hércules da Natalsoft. Nesse capítulo são descritas as vantagens de se utilizar essa tecnologia.

O capítulo 3 mostra o que são padrões de projetos na área de desenvolvimento de software, quais são as vantagens ao utilizá-los e apresenta dois catálogos de padrões: catálogo GoF e JEE.

O capítulo 4 mostra, de forma resumida, quais são os principais padrões de projeto do catálogo GoF que são utilizados no projeto Hércules da Natalsoft.

O capítulo 5 faz uma avaliação detalhada dos padrões de projeto do catálogo GoF que são utilizados no projeto Hércules da Natalsoft.

O capítulo 6 exhibe a arquitetura de software que está sendo utilizada no sistema Hércules da Natalsoft e como os padrões de projeto exibidos nos capítulos anteriores estão sendo utilizados.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Java Enterprise Edition (JEE)

Segundo Alur (2002, p.4), JEE é uma plataforma para desenvolvimento de aplicativos empresariais distribuídos. A linguagem Java, desde a sua implementação, teve uma grande aceitação e crescimento. Mais e mais tecnologias tornaram-se parte da plataforma Java e novas APIs e padrões foram desenvolvidos para atender as várias necessidades.

A plataforma JEE oferece inúmeras vantagens para as empresas que a adotam, tais como:

1. Estabelece padrões para todas as áreas da empresa que necessitam de computação, como conectividade de banco de dados, componentes de negócio da empresa, *middleware* orientado para mensagens, componentes relativos à WEB, protocolos de comunicação e interoperabilidade.
2. Promove as melhores implementações baseadas em padrões abertos, protegendo o investimento tecnológico.
3. Proporciona uma plataforma padrão para a construção de componentes de software que são portáteis através de implementações de fornecedor, evitando depender exclusivamente de um fornecedor.
4. Diminui o tempo necessário para o lançamento do produto no mercado, já que grande parte da estrutura é fornecida por produtos de fornecedores que são implementados de acordo com a especificação JEE padrão. As empresas de tecnologia de informação podem agora sair dos negócios de *middleware* e se concentrar na criação de aplicativos para seus negócios.
5. Aumenta a produtividade do programador, pois os programadores Java podem aprender, com relativa facilidade, a usar as tecnologias JEE baseadas na linguagem Java. Todo o desenvolvimento de software empresarial pode ser realizado usando a plataforma JEE e usando Java, como linguagem de programação.
6. Promove a interoperabilidade em ambientes heterogêneos existentes.

Segundo Ahmed (2002, p.13) o JEE também oferece algumas outras vantagens. Essas permitem aos desenvolvedores se concentrarem no desenvolvimento das suas regras de negócio, na implementação do sistema sem um conhecimento detalhado anterior do ambiente de execução e na criação de sistemas que podem ser portados mais facilmente entre plataformas e sistemas operacionais.

O desenvolvimento de software empresarial é uma tarefa complexa e pode requerer um grande conhecimento de muitas áreas diferentes. Por exemplo, um esforço de desenvolvimento típico de aplicações empresariais poderia requerer que o desenvolvedor fosse familiarizado com as questões de comunicação entre os processos, as questões de segurança, as consultas de acesso específicas do banco de dados etc. O JEE inclui um suporte predefinido e largamente transparente para esses e outros serviços similares. Como resultado, os desenvolvedores são capazes de se concentrar na implementação do código das regras de negócio ao invés do código que suporta a infra-estrutura da aplicação básica.

Talvez uma das maiores vantagens do JEE seja seu suporte para os componentes. Segundo Ahmed (2002, p.13) o software baseado em componentes tem numerosas vantagens sobre o desenvolvimento do software personalizado e tradicional:

1. **Produtividade mais alta:** Menos desenvolvedores podem conseguir mais reunindo uma aplicação a partir de componentes pré-definidos e testados previamente em vez de implementar uma solução personalizada a partir do zero.
2. **Desenvolvimento rápido:** Os componentes existentes podem ser reunidos rapidamente para criar novas aplicações.
3. **Qualidade mais alta:** Em vez de testar as aplicações inteiras, os desenvolvedores da aplicação baseadas em componentes podem se concentrar em testar a integração e a funcionalidade geral da aplicação conseguida através dos componentes pré-definidos.
4. **Manutenção mais fácil:** Como os componentes são independentes para começar, a manutenção como, por exemplo, as atualizações de componentes individuais, é muito mais fácil e com um custo menor.

2.1.1 CONTÊINER JEE

Para compreender as tecnologias, JEE será necessário entender o papel do contêiner na arquitetura JEE. Todas as tecnologias atuais no JEE contam com este conceito simples, porém eficiente.

Um contêiner é uma entidade do software executada no servidor e é responsável por gerenciar os tipos específicos de componentes. Fornece o ambiente de execução para os componentes JEE desenvolvidos. É através de tais contêineres que a arquitetura JEE é capaz de fornecer a independência entre o desenvolvimento e a distribuição e de fornecer a portabilidade entre os diversos servidores de *middleware*.

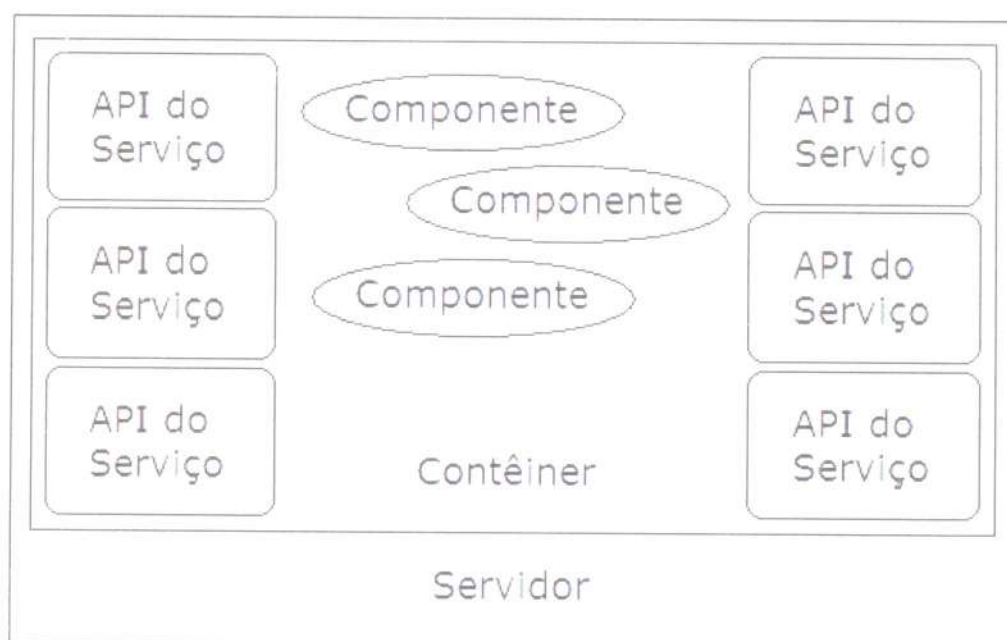


Figura 1 - O conceito do contêiner

Fonte: Dados da Pesquisa (2007)

Um contêiner também é responsável por gerenciar o ciclo de vida dos componentes distribuídos nele e por coisas como o *pool* de recursos e aplicar a segurança. Por exemplo, pode-se limitar a capacidade em acessar um método específico a um pequeno grupo de chamadas. O contêiner então aplicaria essa restrição interceptando as solicitações para esse método e assegurando que a entidade que solicita o acesso esteja na lista privilegiada.

Dependendo do tipo de contêiner, poderá também fornecer acesso a algumas ou a todas as APIs JEE.

Todos os componentes JEE são distribuídos e executados em algum tipo de contêiner. O JEE tem três tipos diferentes de contêineres.

1. **EJB (*Enterprise Java Beans*)**: um dos contêineres mais conhecidos da arquitetura. Especifica um ambiente de execução para enterprise beans como serviços de segurança, concorrência, gerenciamento de ciclo de vida, transação, entre outros serviços. Os beans rodam no servidor JEE.
2. **WEB**: trata-se da implementação de contrato de componente da Web com a arquitetura J2EE, fornecendo ambiente de execução para os componentes do tipo web que incluem desde segurança até gerenciamento de ciclo de vida e transação. Inclui *Servlets* (classes Java que podem ser carregadas dinamicamente e executar sob um servidor web) disponibilizadas por serviços de rede referentes aos pedidos e respostas enviados, com

uso obrigatório do protocolo HTTP. E também engloba JSPs (*Java Server Pages* - tecnologia Web que utiliza, por exemplo, linguagens de scripting e objetos Java no servidor para retorno dinâmico a um cliente).

3. **Cliente da aplicação:** Um contêiner que aceita e gerencia a execução de componentes do cliente da aplicação.

Algumas das principais APIs e suas funcionalidades da plataforma JEE são:

1. **Enterprise JavaBeans:** API que padroniza componentes J2EE do lado servidor, com regras de desenvolvimento definidas como implementação de interfaces responsáveis por tarefas definidas ou regras de herança. Tem como vantagem o tratamento automático de segurança, persistência, transação dentre outros serviços. Hoje a tecnologia EJB tornou-se um padrão para desenvolvimento de aplicações corporativas distribuídas e multicamadas, realizando o encapsulamento da lógica de negócio de uma aplicação.
2. **Java Servlet / Java Server Pages:** Um servlet estende as funcionalidades de um webserver e pode fazer uso de qualquer outra API. Um JSP faz as funções de um script em uma página web e é compilado como servlet em seu container. Uma página JSP contém dois tipos de texto: estático, representado por marcações HTML, WML, ou XML, e elementos dinâmicos JSP.
3. **Java Message Service API (JMS):** API que permite o acesso de forma padronizada serviços de mensagens na interação com outros tipos de programas, com uso de mecanismos de criação, envio, recebimento e leitura de mensagens.
4. **Java Transaction API (JTA):** Especifica uma interface para as aplicações utilizarem um serviço de suporte transacional, por exemplo, gerenciamento de aplicações distribuídas.
5. **JavaMail API / JavaBeans Activation Framework (JAF):** São implementações de suporte ao envio e recebimento de e-mails em Java.
6. **J2EE Connector Architecture (JCA):** As APIs que formam esse tipo de arquitetura têm como funções prover integrações entre sistemas de forma corporativa, ou seja, padronizar conexões entre sistemas Java e EIS (*Enterprise Information Systems*), como por exemplo, ERPs e CRMs.
7. **JDBC:** API de acesso a bancos de dados. Sua importância está no fato de permitir o acesso a diferentes tipos de bancos de dados usando a mesma API.

8. **Java Naming and Directory Interface (JNDI):** Tem como objetivo fornecer uma interface padrão para localização de usuários, máquinas, objetos, redes e serviços. JNDI pode ser utilizado tanto para localização de um computador em uma rede, quanto para busca de objetos Java.
9. **Java API for XML Processing (JAXP):** Trata-se de um conjunto padrão de APIs para XML adicionados nas versões mais recentes de J2SE e J2EE. Suporta os analisadores padrão SAX e DOM, assim com XSLT, dando assim maior controle sobre a apresentação de dados e capacitando a conversão de dados. Essas bibliotecas também estão disponíveis separadamente (JAX Pack), contendo outras APIs específicas para uso de XML e *Web Services*, como JAX-RPC ou JAXR.

2.1.2 SERVLETS

Segundo Ahmed (2002, p.15) servlets são componentes Web capazes de gerenciar o conteúdo dinâmico. Eles fornecem um mecanismo eficiente para a interação entre a lógica comercial baseada no servidor e o cliente baseado na WEB.

Os servlets são executados dentro do contêiner do servlet em um servidor WEB. O contêiner do *servlet* gerencia o ciclo de vida de um servlet e converte as solicitações do cliente WEB, feitas através de protocolos como o *Hypertext Transfer Protocol* (HTTP), nas solicitações baseadas em objeto. Do mesmo modo, o contêiner converte a resposta de um servlet e mapeia o objeto da resposta para o devido protocolo WEB.

As aplicações baseadas no *Servlet* geram conteúdo dinâmico (normalmente HTML) e interagem com os clientes, utilizando o modelo request/response. Os *servlets* normalmente utilizam o protocolo HTTP, apesar de não serem restritos a ele.

2.1.3 JAVA SERVER PAGES (JSP)

Os JSPs são outro tipo de componente WEB JEE e evoluíram a partir da tecnologia do *servlet*. Na verdade, as partes do JSPs são compiladas nos *servlets* que então são executados dentro do ambiente do contêiner do servlet.

Os JSPs acabaram facilitando para os membros de uma equipe WEB manter partes do sistema que suportam a apresentação da página WEB sem requerer que sejam programadores tradicionais. Os não programadores geralmente mantêm o código da apresentação HTML. Isso é mais difícil de fazer quando essa HTML é gerada pelas instruções Java que estão contidas nos *servlets*.

Algumas tecnologias Java já existentes há algum tempo, como o *javabeans*, também usam os JSPs. Ajudam a tornar menos complicado exibir quantidades maiores de dados para coisas como tabelas nas páginas WEB.

2.1.4 JAVA SERVERFACES (JSF)

Java ServerFaces surgiu em 2002 e é um *framework* MVC para o desenvolvimento de aplicações Web, que permite o desenvolvimento de aplicações para a internet tal como era feito com Delphi ou Visual Basic, ou seja, arrastando e soltando os componentes na tela (JSP), definindo propriedades dos mesmos, etc.

O JSF é atualmente considerado por muitos como a última palavra em termos de desenvolvimento de aplicações Web, resultado da experiência e maturidade adquiridas com o JSP/Servlet (Model1), Model2 (MVC) e Struts.

2.1.4.1 Características

1. Permite que o desenvolvedor crie UIs através de um conjunto de componentes UIs pré-definidos;
2. Fornece um conjunto de tags JSP para acessar os componentes;
3. Reusa componentes da página;
4. Associa os eventos do lado cliente com os manipuladores dos eventos do lado do servidor (os componentes de entrada possuem um valor local representando o estado no lado servidor);
5. Fornece separação de funções que envolvem a construção de aplicações Web.

JSF inclui:

1. Suporte a internacionalização e acessibilidade;
2. Um conjunto padrão de componentes de interface de usuário que possibilitam validação padronizada;
3. Duas bibliotecas de etiqueta ("*tag libraries*") especiais do *Java Server Pages* (JSP) para expressar a interface do *Java ServerFaces* dentro de uma página JSP;
4. Um modelo de eventos do lado servidor ("*server-side event model*");
5. Gerência de estados;
6. *Managed Beans* (*Javabeans* criados com injeção de dependência ("*dependency injection*");

7. Linguagem de Expressão Unificada ("*Unified Expression Language*") para JSP 2.0 e JSF 1.2.

2.1.5 ENTERPRISE JAVA BEANS (EJB)

Basicamente EJB é uma arquitetura de componentes multi-plataforma para o desenvolvimento de aplicações Java, multi-tier, distribuídas, escaláveis e orientadas a objetos.

O objetivo da arquitetura EJB é facilitar o trabalho do desenvolvedor para que ele não tenha que se preocupar com aspectos de infra-estrutura.

Existem três tipos de EJBs:

1. **Session Bean** - é o tipo mais simples de EJB, pode ter estado (*stateful*) ou não ter (*stateless*)
2. **Entity Bean** - mapeam tabelas de um banco de dados relacional através de um arquivo de mapeamento. Na prática cada objeto *entity* representa uma linha de uma tabela. Existe uma linguagem de *query* específica para buscar *entitys* chamada EQL. Atualmente é recomendado a utilização de classes persistentes utilizando JPA (será visto mais adiante).
3. **MDB** - são consumidores assíncronos de mensagens de filas / tópicos JMS

2.1.6 JAVA MESSAGE SERVICE (JMS)

É uma API da linguagem Java para middleware orientado à mensagens. Através da API JMS duas ou mais aplicações podem se comunicar por mensagens.

A Versão 1.1 do JMS API na plataforma J2EE 1.4 tem as seguintes características:

1. *Message-driven beans* - permite consumo assíncrono de mensagens JMS.
2. O Envio e recebimento de mensagens podem participar de transações *Java Transaction API* (JTA).
3. *J2EE Connector Architecture interface* que permite implementações JMS de diferentes fornecedores de software serem externamente conectadas a um servidor aplicação.

A arquitetura do contêiner EJB, por outro lado, reforça a API JMS de duas maneiras:

1. Autorizando o consumo concomitante de mensagens
2. Ao prover suporte para transações distribuídas, a fim de que atualizações de dados, processamento de mensagens e conexões para sistemas EIS usando o *J2EE Connector Architecture* possam participar do mesmo contexto de transação.

2.1.7 JAVA PERSISTENCE API (JPA)

A maioria das aplicações corporativas precisa criar, armazenar e pesquisar dados de alguma forma. Em outras palavras: usar um banco de dados. Isto é frequentemente chamado de CRUD (*Create, Read, Update e Delete*). O banco de dados será utilizado para persistir dados.

A persistência de dados refere-se ao armazenamento "eterno" de dados, enquanto o dispositivo físico de armazenamento dure. Quando se grava um dado no disco, por exemplo, ele está sendo "eternizado", ou seja, deixa de ficar volátil na memória RAM e passa a ser escrito num dispositivo que armazena a informação de modo que ela não desapareça facilmente.

No mercado atual a maioria dos bancos de dados utiliza o modelo relacional para armazenar os dados. Surge então, a necessidade de ter mecanismos que mapeiem objetos em memória para meios persistentes relacionais e que forneçam transparência ao usuário. Este processo (mapeamento objeto relacional) não é imediato e as linguagens OO não fornecem atualmente suporte direto para o mesmo.

JPA é a especificação padrão para o gerenciamento de persistência e mapeamento objeto relacional, surgida na plataforma Java EE 5.0 (Java Enterprise Edition) na especificação do EJB 3.0 (Enterprise Java Beans 3.0). Introduzida no intuito de substituir os Entity Beans (que foram descontinuados) e simplificar o desenvolvimento de aplicações Java EE e Java SE que utilizam persistência de dados. Possui uma completa especificação para realizar mapeamento objeto relacional, utilizando anotações da linguagem Java (Java SE 5.0 ou superior). Também dá suporte a uma rica linguagem de consulta, semelhante à SQL, permitindo consultas estáticas ou dinâmicas.

Um dos principais conceitos relacionados à API JPA é o de entidade. Uma entidade corresponde a um objeto que pode ser gravado na base de dados a partir de um mecanismo de persistência. A classe que implementa a entidade persistente é um POJO, que basicamente contém um construtor padrão sem argumentos e uma chave primária e também não precisa derivar de nenhuma interface ou classe, nem implementar qualquer método em especial.

Quando se decide utilizar o JPA, é necessária a escolha de um provedor JPA. Como esta API é uma especificação para frameworks de persistência, existe a necessidade de provedores JPA. Por padrão, a implementação de referência é o Oracle Toplink Essentials. Também existem outros provedores JPA no mercado, como o Hibernate Entity Manager, Bea Kodo e o ApacheJPA.

No projeto que está sendo referenciado nesse trabalho foi utilizado o provedor Hibernate.

3 PADRÕES DE PROJETO DE SOFTWARE

3.1 O que são padrões?

O conceito de padrão de projeto foi criado na década de 70 pelo arquiteto Christopher Alexander. Em seus livros *Notes on the Synthesis of Form*, *The Timeless Way of Building* e *A Pattern Language*, ele estabeleceu que um padrão deve ser descrito em cinco partes:

1. **Nome:** uma descrição da solução, mais do que do problema ou do contexto.
2. **Exemplo:** uma ou mais figuras, diagramas ou descrições que ilustrem um protótipo de aplicação.
3. **Contexto:** a descrição das situações sob as quais o padrão se aplica.
4. **Problema:** uma descrição das forças e restrições envolvidos e como elas interagem.
5. **Solução:** relacionamentos estáticos e regras dinâmicas descrevendo como construir artefatos de acordo com o padrão, freqüentemente citando variações e formas de ajustar a solução segundo as circunstâncias. Inclui referências a outras soluções e o relacionamento com outros padrões de nível mais baixo ou mais alto.

Os padrões de projeto de software, também conhecidos como *Design Patterns*, visam uma melhor reutilização de software. Estes padrões tornam mais fáceis reutilizar projetos e arquiteturas bem sucedidas. Eles ajudam a escolher alternativas de projeto que tornam um sistema reutilizável e a evitar alternativas que comprometam a reutilização. Segundo Marinescu (2002), um padrão de projeto é definido como uma solução desenvolvida utilizando boas práticas para um problema comum que ocorre várias vezes. Um padrão de projeto documenta e explana um problema importante que pode ocorrer no projeto ou implementação de uma aplicação e então discute a melhor solução prática para o problema.

Em termos de orientação a objetos, padrões de projeto identificam classes, instâncias, seus papéis, colaborações e a distribuição de responsabilidades. São, portanto, descrições de classes e objetos que se comunicam, implementados a fim de solucionar um problema comum em um contexto específico (SCHNEIDE, 1999). As vantagens de se utilizar padrões em um projeto são listadas por Alur (2002):

1. **Foram testados:** refletem a experiência e conhecimento dos desenvolvedores que utilizaram estes padrões com sucesso em seu trabalho;
2. **São reutilizáveis:** fornecem uma solução pronta que pode ser adaptada para diferentes problemas quando necessário;

3. **São expressivos:** formam um vocabulário comum para expressar grandes soluções sucintamente;
4. **Facilitam o aprendizado:** reduzem o tempo de aprendizado de uma determinada biblioteca de classes. Isto é fundamental para o aprendizado dos desenvolvedores novatos;
5. **Diminuem retrabalho:** quanto mais cedo são usados, menor será o retrabalho em etapas mais avançadas do projeto.

3.2 Catálogo de padrões GoF

O movimento ao redor de padrões de projeto ganhou popularidade com o livro *Design Patterns: Elements of Reusable Object-Oriented Software*, publicado em 1995. Os autores desse livro são Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, conhecidos como a "Gangue dos Quatro" (*Gang of Four*) ou simplesmente "GoF".

Os padrões "GoF" são organizados em famílias de padrões: de criação, estruturais e comportamentais. Os padrões de criação são relacionados à criação de objetos, os estruturais tratam das associações entre classes e objetos e os comportamentais das interações e divisões de responsabilidades entre as classes ou objetos.

Um padrão "GoF" também é classificado segundo o seu escopo; de classe ou de objeto. Nos padrões com escopo de classe os relacionamentos que definem este padrão são definidos através de herança e em tempo de compilação. Nos padrões com escopo de objeto o padrão é encontrado no relacionamento entre os objetos definidos em tempo de execução.

Escopo		Propósito		
		Criação	Estrutura	Comportamento
		Factory Method	Class Adapter	Interpreter Template Method
	Classe			
	Objeto	Abstract Factory Builder Prototype Singleton	Object Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Figura 2 - Catálogo GoF organizados em famílias e escopo

Fonte: Dados da pesquisa (2007)

3.3 Catálogo de padrões JEE

A plataforma JEE surgiu com o objetivo de padronizar e simplificar a criação de aplicações empresariais. Para isso, propõe um modelo onde componentes JEE (páginas JSP, Servlets, EJB's, etc) escritos pelos usuários da plataforma, podem fazer uso de serviços providos por esta, os quais simplificam sua implementação e possibilitam maior foco no negócio. Um diferencial significativo na arquitetura proposta para a plataforma J2EE foi a iniciativa de enfatizar a utilização de padrões de projetos.

O catálogo de padrões JEE inclui atualmente 15 padrões. A tabela abaixo lista os padrões incluídos no catálogo.

Camada de Apresentação	Intercepting Filter Front Controller View Helper Composite View Service to Worker Dispatcher View
Camada de Negócios	Business Delegate Value Object Session Facade Composite Entity Value Object Assembler Value List Handler Service Locator
Camada de Integração	Data Access Object Service Activator

Quadro 1 – Catálogo de padrões JEE
Fonte: ALUR (2002)

O catálogo de padrões de projeto aqui apresentados foi dividido de acordo com as camadas a que eles pertencem. A camada de apresentação contém os padrões relacionados aos Servlets e páginas JSP. Os padrões da camada de negócios são relacionados à tecnologia de EJB e os padrões da camada de integração estão relacionados à comunicação com recursos e sistemas externos.

Os padrões da camada de apresentação, apresentados por Alur (2002) são:

1. *intercepting filter*: intercepta solicitações e respostas, e aplica algum processamento na requisição;
2. *front controller*: fornece um controlador centralizado para manter a lógica de processamento que ocorre na camada de apresentação e que, erroneamente é colocado nas visões (*view*);

3. *view helper*: encapsula a lógica que não esteja relacionada à formatação da apresentação em componentes auxiliares;
4. *composite view*: cria uma visão através da composição de outras visões;
5. *service to worker*: combina um componente distribuidor (*dispatcher*) com os padrões *Front Controller* e *View Helper*. Neste padrão a responsabilidade de recuperar o conteúdo para apresentar é do controlador;
6. *dispatcher view*: semelhante ao padrão *Service to Worker*, mas, neste padrão a recuperação do conteúdo é responsabilidade das visões.

Os padrões da camada de negócios (ALUR, 2002) são:

1. *business delegate*: reduz o acoplamento entre o cliente e a camada de negócios;
2. *data transfer object*: facilita o intercâmbio de dados entre as camadas;
3. *data transfer object factory*: facilita e centraliza a criação de data transfer objects;
4. *session facade*: fornece uma interface unificada para um sistema distribuído;
5. *composite entity*: representa uma prática mais recomendada para criar entity beans de granulação grossa, agrupando os objetos dependentes em um único entity bean;
6. *value list handler*: gerencia o processamento e armazenamento em cache de consultas que retornam uma grande quantidade de informações;
7. *service locator*: encapsula a complexidade de localização e criação de serviços de negócio e localiza os objetos *Home* dos EJB.

Os padrões da camada de integração descritos por Alur (2002) são:

1. *data access object*: abstrai e encapsula o acesso aos dados;
2. *service activator*: facilita o processamento assíncrono para *Message-driven beans*.

As descrições detalhadas destes padrões podem ser encontradas em Alur (2002), Marinescu (2002) e Sorroche (2002).

Os padrões dos catálogos GoF e JEE que foram utilizados no projeto Hércules da Natalsoft serão descritos nos capítulos posteriores.

3.3.1 COMO UTILIZAR O CATÁLOGO DE PADRÕES JEE?

Um dos desafios durante a utilização de qualquer conjunto de padrões é compreender como combinar os padrões da melhor maneira possível. Como Christopher Alexander (1977) diz em seu livro, *A Pattern Language*:

“Em poucas palavras, nenhum padrão é uma entidade isolada. Cada padrão só pode existir no mundo na medida em que for suportado por outros padrões: os padrões maiores nos quais ele é incorporado, os padrões de mesmo tamanho e os menores que são incorporados a ele.”

Algumas vezes, o padrão depende de outro padrão do catálogo JEE ou de outros padrões descritos no catálogo de padrões GoF.

3.4 O enfoque nas camadas

Como o catálogo de padrões JEE deve ser utilizado nessa plataforma que é um sistema em camadas múltiplas, visualizaremos o sistema em termos de camadas. Uma camada equivale a um dos particionamentos lógicos dos diversos aspectos tratados em um sistema. A cada camada é atribuída sua responsabilidade distinta, ou única, no sistema. Cada camada não está estritamente acoplada com a camada adjacente.

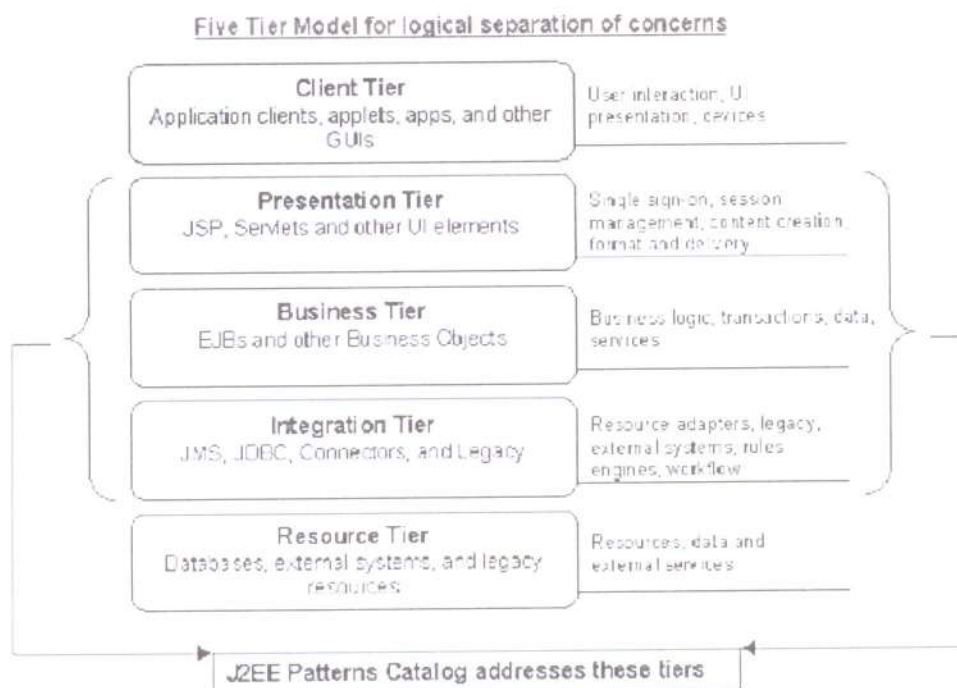


Figura 3 - Modelo de cinco camadas para divisão lógica dos aspectos
Fonte: ALUR (2002)

3.4.1 CAMADA DE CLIENTE

Esta camada representa todos os dispositivos ou clientes do sistema que acessam a aplicação. Um cliente pode ser um navegador WEB, uma aplicação Java ou outra aplicação, um applet Java, um telefone WAP, uma aplicação de rede ou algum dispositivo introduzido no futuro. Poderia ser ainda um processo em lotes.

3.4.2 CAMADA DE APRESENTAÇÃO

Essa camada encapsula toda a lógica de apresentação exigida para servir os clientes que acessam o sistema. A camada de apresentação intercepta as solicitações dos clientes, fornece um único início de sessão, conduz o gerenciamento de sessão, controla o acesso a serviços dos negócios, constrói as respostas e as entrega aos clientes. Os *servlets*, JSPs e JSFs residem nessa camada, pois apesar de não serem elementos de interface com o usuário propriamente dito, produzem esses elementos.

3.4.3 CAMADA DE NEGÓCIOS

Essa camada fornece os serviços de negócios necessários aos clientes das aplicações. A camada contém os dados e a lógica de negócios. Normalmente, a maior parte do processamento de negócios para a aplicação está centralizada nessa camada. É possível que, devido a sistemas legados, alguns processamentos de negócios possam ocorrer na camada de recursos. Os componentes enterprise beans são a solução usual e mais indicada para implementar objetos de negócios na camada de negócios.

3.4.4 CAMADA DE INTEGRAÇÃO

Essa camada é responsável pela comunicação com os recursos e sistemas externos como armazenamento de dados e aplicações legadas. A camada de negócios é acoplada com a camada de integração quando os objetos de negócio exigem dados ou serviços que residam na camada de recursos. Os componentes nessa camada podem utilizar tecnologia de conector JDBC ou algum middleware exclusivo para trabalhar com a camada de recursos.

3.4.5 CAMADA DE RECURSOS

Essa é a camada que contém os dados de negócios e recursos externos como mainframes e sistemas legados, sistema de integração de comércio eletrônico entre empresas e serviços como autorizações de cartões de crédito.

3.5 Relacionamentos entre padrões J2EE

De acordo com Christopher Alexander, um padrão não existe isoladamente e precisa do suporte de outros padrões para trazer significado e utilidade. Virtualmente todo padrão possui um relacionamento com outros padrões. Compreender esses relacionamentos ao projetar e arquitetar uma solução ajuda das seguintes maneiras:

1. Permite considerar que novos problemas podem ser apresentados quando é aplicado um padrão para resolver um problema. Esse é o efeito dominó: Que novos problemas são apresentados quando um determinado padrão é introduzido na arquitetura? É crucial identificar esses conflitos antes de começar a codificação.
2. Permite revisar os relacionamentos entre padrões para determinar soluções alternativas. Após os possíveis problemas serem identificados deve-se retornar aos relacionamentos de padrões e coletar as soluções alternativas. Talvez os novos problemas possam ser resolvidos selecionando um padrão diferente ou utilizando outro padrão em combinação com o padrão já escolhido.

A Figura 4 - Relacionamentos dos padrões JEE os relacionamentos entre os padrões do catálogo JEE.

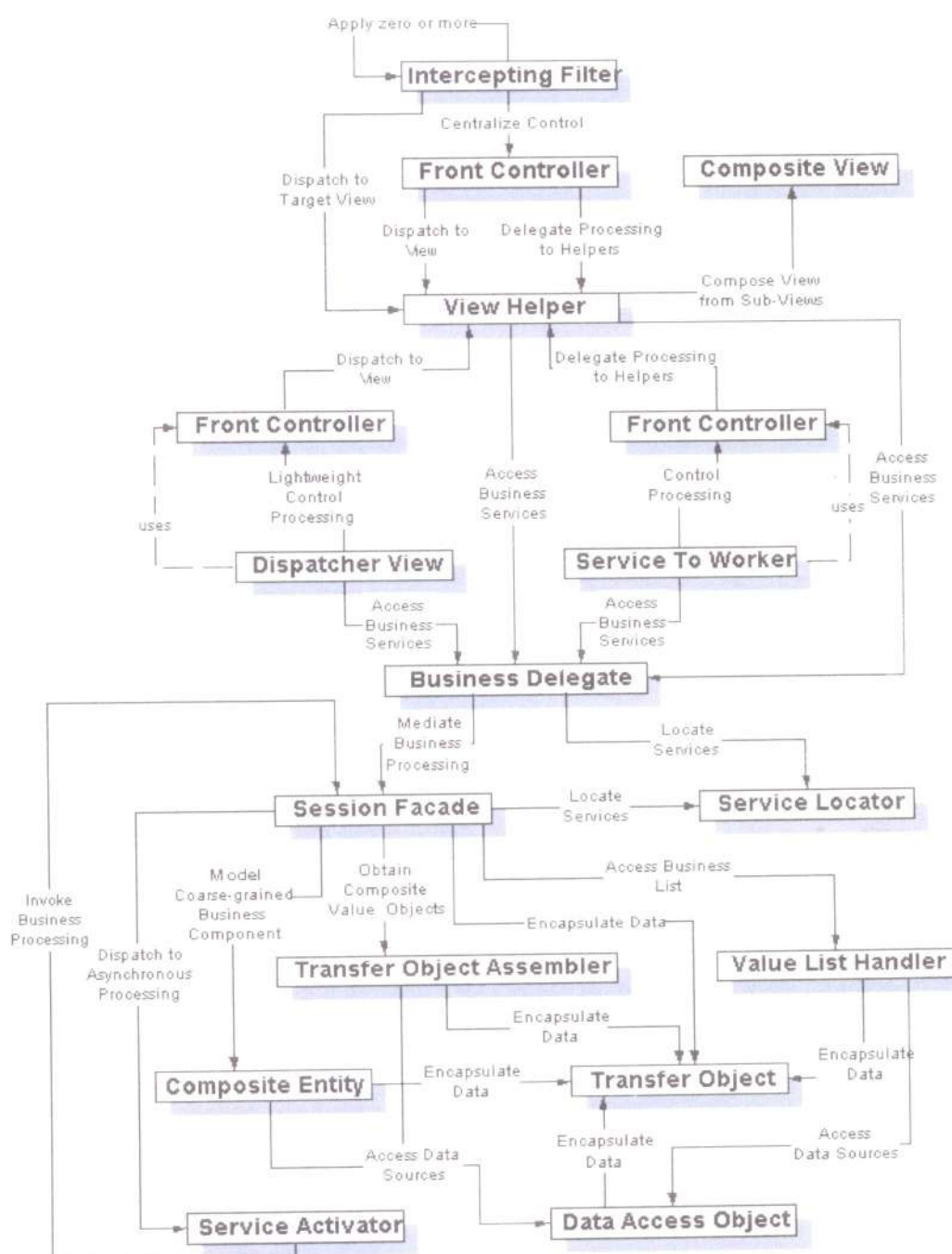


Figura 4 - Relacionamentos dos padrões JEE
Fonte: ALUR (2002)

4 PADRÕES GOF

4.1 Padrões de Criação

4.1.1 FACTORY METHOD

Segundo Gamma (1998) o objetivo desse padrão é definir uma interface para criar objetos de forma a deixar que uma classe decida qual subclasse instanciar. *Factory Method* deixa que subclasses façam a instanciação.

Uma aplicação pode necessitar instanciar vários objetos dentro da mesma hierarquia de classes ou interfaces. Essa hierarquia de classes pode sofrer alteração e novas classes serem inseridas e outras classes deixam de existir. Se a aplicação instanciar os objetos de uma mesma hierarquia diretamente ocorrerá um acoplamento entre essas classes e a aplicação, pois será necessário conhecer os detalhes de instanciação desses objetos.

Duas motivações para utilizar esse padrão são:

1. Uma classe deve ser capaz de iniciar a criação de objetos sem ter qualquer dependência na classe do objeto criado.
2. O conjunto de classes que uma classe pode instanciar deve ser dinâmico de acordo com as novas classes que ficam disponíveis.

Uma boa solução seria criar uma fábrica de objetos que delegue a operação de instanciação para os próprios objetos que se pretende criar.

A figura abaixo representa o diagrama de classes do padrão *Factory Method*.

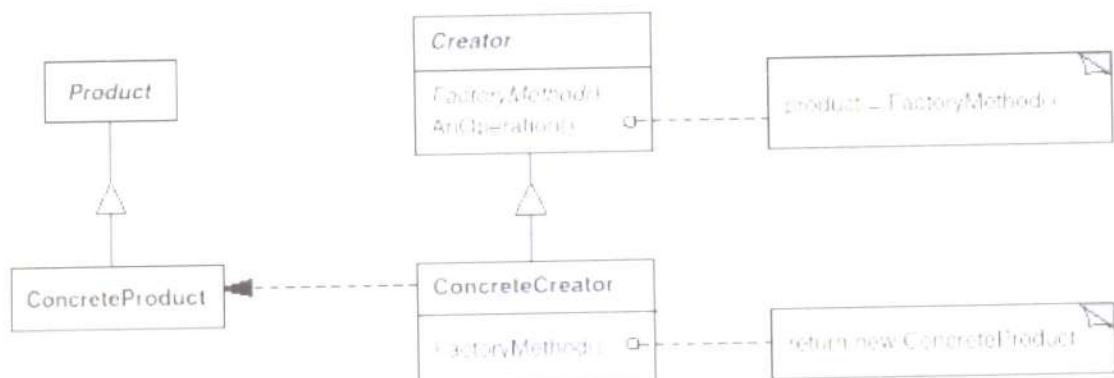


Figura 5 - Estrutura de classes do padrão Template Method
Fonte: GAMMA (1998)

Os participantes do diagrama são:

Product: Define a interface dos objetos criados pelo *Factory Method*.

ConcreteProduct: Implementa a interface *Product*.

Creator: Declara o *Factory Method* que retorna um objeto do tipo *Product*. Às vezes, o *Creator* não é apenas uma interface, mas pode envolver uma classe concreta que tenha uma implementação default para o *Factory Method* para retornar um objeto com algum tipo *ConcreteProduct* padrão. Pode chamar o *Factory Method* para criar um *ConcreteProduct* do tipo *Product*.

ConcreteCreator: Faz override do *Factory Method* para retornar uma instância de *ConcreteProduct*.

4.1.1.1 Consequências

Elimina a necessidade de colocar classes específicas da aplicação no código

O código só lida com a interface *Product*. O código pode, portanto funcionar com qualquer classe *ConcreteProduct*.

Conjunto de classes a ser criado pode mudar dinamicamente

Podem ser criados ou excluídos classes do tipo *Product* apenas alterando a implementação na fábrica de objetos.

4.1.2 SINGLETON

Segundo Gamma (1998) o objetivo desse padrão é garantir que uma classe só tenha uma única instância, e prover um ponto de acesso global a ela.

Uma aplicação pode ter a necessidade de ter apenas uma única instância de uma classe e essa pode ser acessada por toda a aplicação. Exemplos disso são objetos para acessar dados, comunicação com dispositivos, objeto que contém a configuração de um programa, etc. Para solucionar essa necessidade, deve-se criar uma classe responsável pela manutenção de uma instância única e que esta instância seja facilmente acessível através de ponto global.

Uma forma de garantir que exista apenas uma única instância é não permitir chamadas diretamente ao construtor do método. O código abaixo mostra um exemplo de código do padrão *singleton*.

```
public class Config implements ConfigIF {
    private static ConfigIF instancia = null;

    private Config() {
    } // o compilador não vai gerar um construtor default público

    public static synchronized ConfigIF getInstance() {
        if (instancia == null) {
```

```

    // "lazy instantiation"
    instancia = new Config();
}
return instancia;
}
}

```

Listagem 1 - Código de Exemplo do Padrão Singleton
Fonte: Dados da Pesquisa (2007)

O instanciamento do objeto pode ser feito quando a classe for carregada ou quando o método de criação for chamado pela primeira vez. Neste caso, é preciso garantir que outros objetos não tentarão criar outro *Singleton* declarando o bloco crítico com *synchronized*, como pode ser observado no código acima.

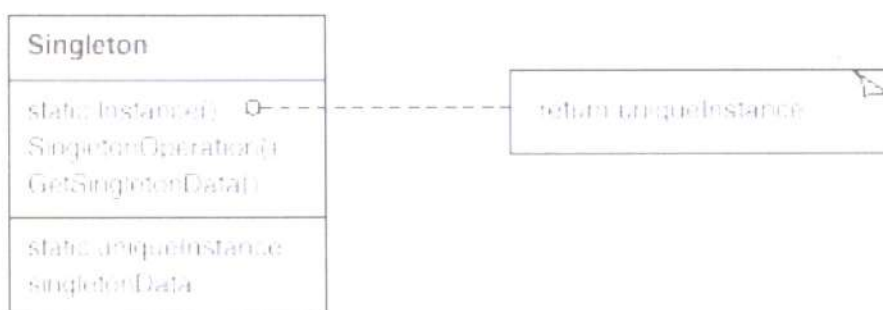


Figura 6 - Diagrama de classes do Padrão Singleton
Fonte: GAMMA (1998)

Os clientes acessam a instância apenas através da operação *getInstance()* do *Singleton*.

4.2 Padrões Estruturais

4.2.1 ADAPTER

Segundo Gamma (1998) o objetivo desse padrão é converter a interface de uma classe em outra interface esperada pelos clientes. Adapter permite a comunicação entre classes que não poderiam trabalhar juntas devido à incompatibilidade de suas interfaces.

Pode-se ter a necessidade de usar uma classe existente, mas a sua interface não corresponde à interface de que se necessita ou pode-se querer utilizar de uma forma uniforme duas classes ou mais classes com interfaces diferentes.

Existem duas opções de utilização do padrão Adapter:

1. **Class Adapter:** usa herança múltipla (herdando a classe existente e implementando interfaces)
2. **Object Adapter:** usa composição.

4.2.1.1 Class Adapter

Utiliza herança múltipla, onde o cliente é a aplicação que colabora com objetos aderentes a interface *target* que define a interface requerida pelo cliente. *Adaptee* é a interface que requer adaptação e o Adaptador (*Adapter*) é quem adapta a interface do recurso à interface *target*. *Class Adapter* será utilizado quando houver uma interface que permita a implementação estática.

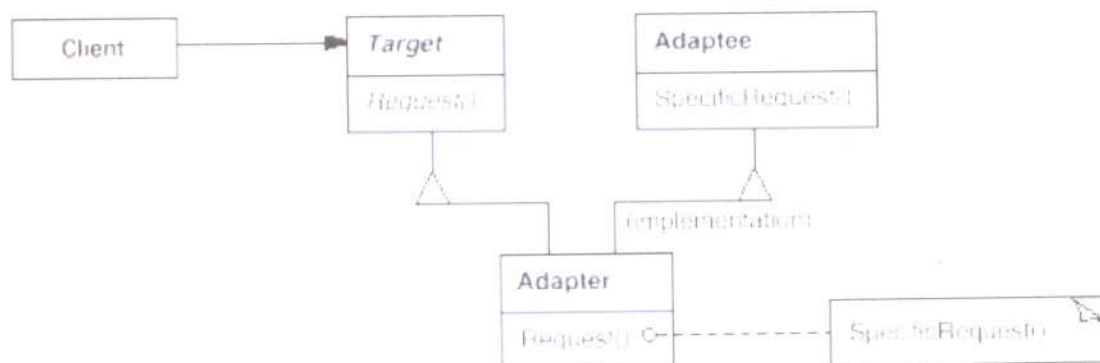


Figura 7 - Diagrama de Classes do Padrão Adapter (Class Adapter)
Fonte: GAMMA (1998)

1. *Client*: aplicação que colabora com objetos aderentes à interface *Target*
2. *Target*: define a interface requerida pelo *Client*
3. *Adaptee*: interface que requer adaptação
4. *Adapter*: adapta a interface do Recurso à interface *Target*

4.2.1.2 Object Adapter

Utiliza composição, onde uma única solução se o *target* não for uma interface Java. O Adaptador possui referência para o objeto que terá sua interface adaptada (instância de uma classe que implemente *Adaptee*). Cada método de *target* chama os métodos correspondentes na interface adaptada. *Object Adapter* será utilizado quando o menor acoplamento for desejado, o cliente não usa interface Java ou classe abstrata que possa ser estendida.

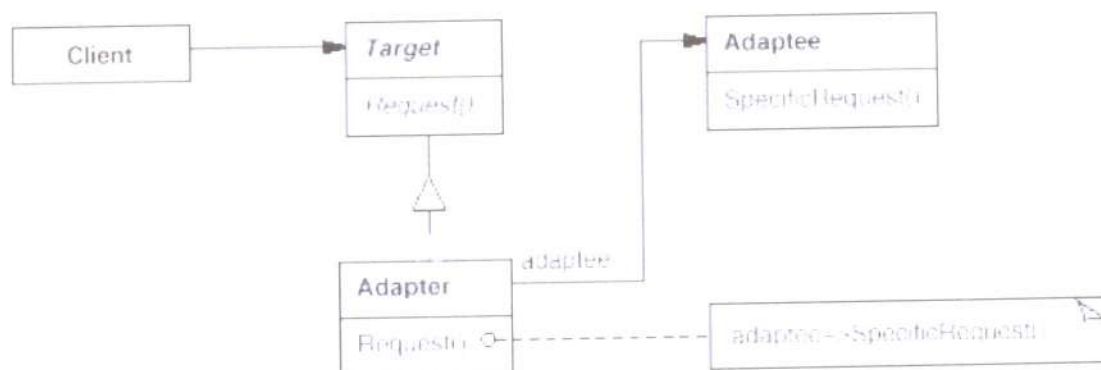


Figura 8 - Diagrama de Classes do Padrão Adapter (Object Adapter)
 Fonte: GAMMA (1998)

4.2.2 DECORATOR

De acordo com Gamma (1998) o objetivo desse padrão é anexar responsabilidades adicionais a um objeto dinamicamente. *Decorators* oferecem uma alternativa flexível ao uso de herança para estender uma funcionalidade.

Em uma aplicação às vezes é necessário adicionar responsabilidades adicionais a um objeto em tempo de execução. O padrão *Decorator* fornece uma opção para estender as funcionalidades de um objeto e não de uma classe inteira.

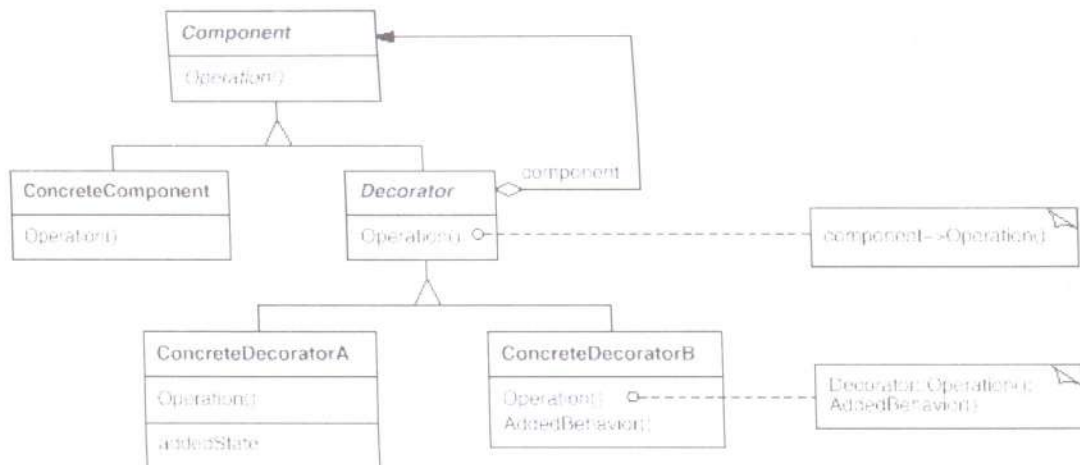


Figura 9 - Diagrama de Classes do Padrão Decorator
 Fonte: GAMMA (1998)

1. **Component:** Define a interface de objetos que podem ter responsabilidades adicionadas dinamicamente.
2. **ConcreteComponent:** Define um objeto ao qual se pode adicionar responsabilidades adicionais.
3. **Decorator:** Mantém uma referência a um objeto *Component* e define uma interface igual à do *Component*.
4. **ConcreteDecoratorA e ConcreteDecoratorB:** Adicionam responsabilidades ao *Component*.

O padrão *Decorator* tem a vantagem de ser mais flexível que a herança estática, pois pode adicionar responsabilidades dinamicamente, mas possui um fator negativo que é a utilização de muitos objetos pequenos.

Os códigos abaixo mostram a implementação de alguns participantes do padrão *Decorator*.

```
public interface Componente {
    Object getDados();

    void operacao(Object arg);
}
```

Listagem 2 - Código-fonte da Interface Componente
Fonte: Dados da Pesquisa (2007)

```
public abstract class Decorador implements Componente {
    private Componente componente;

    public Decorador(Componente componente) {
        this.componente = componente;
    }

    public Object getDados() {
        return componente.getDados();
    }

    public void operacao(Object arg) {
        componente.operacao(arg);
    }
}
```

Listagem 3 - Código-fonte da classe Decorator
Fonte: Dados da Pesquisa (2007)

```
public abstract class DecoradorConcretoUm extends Decorador {
    private Object estado;

    public DecoradorConcretoUm(Componente comp, Object estado) {
        super(comp);
        this.estado = estado;
    }

    public void operacao(Object arg) {
        // ... comportamento adicional
        super.operacao(estado);
        // ...
    }
}
```

Listagem 4 - Código-fonte do DecoradorConcretoUm
Fonte: Dados da Pesquisa (2007)

4.2.3 PROXY

De acordo com Gamma (1998) o objetivo desse padrão é prover um substituto ou ponto através do qual um objeto possa controlar o acesso a outro.

Um *proxy* possui a mesma interface e a mesma funcionalidade da classe que ele encapsula. Isto não parece muito útil a primeira vista. Entretanto, os *proxies* servem a um

importante propósito de controlar o acesso a outros objetos. Isto é particularmente importante se estes objetos devam ser acessados de uma maneira estilizada ou complicada. Por exemplo, se um objeto está em uma máquina remota, então para acessá-lo é necessário utilizar vários recursos de rede. Neste caso, é mais fácil criar um *proxy* local que entenda a rede e realize as operações necessárias, e então retorne o resultado. Isto simplifica o cliente localizando o código específico da rede em outro lugar.

Como outro exemplo, um objeto pode necessitar de uma trava se ele puder ser acessado por múltiplos clientes. A trava representa o direito de ler e/ou atualizar um objeto; sem a trava, atualizações concorrentes podem levar o objeto a um estado inconsistente, ou uma leitura no meio de uma seqüência de atualizações pode resultar na observação de um estado inconsistente. Um proxy pode se responsabilizar por bloquear um objeto antes de uma operação ou uma seqüência de operações, e então depois desbloqueá-lo. Isto é menos propenso a erros do que requerer que os clientes implementem corretamente o protocolo de travamento.

Outra variedade de *proxy* é o *proxy* de segurança. Ele pode operar corretamente se quem o chamar tiver as credenciais corretas (como um certificado Kerberos válido), mas lança um erro se um usuário não autorizado tentar realizar operações.

Um último exemplo de *proxy* é o *proxy* virtual. Se criar um objeto é caro (por causa da latência computacional ou de rede), então ele pode ser representado por um *proxy* ao invés disso. Este *proxy* pode começar imediatamente a criar o objeto, em uma tarefa em background, na esperança de que ele esteja pronto até o momento que a primeira operação seja invocada, ou ele pode retardar a criação do objeto até que uma operação seja invocada. No primeiro caso, o resto do sistema pode prosseguir sem esperar; no último caso, o trabalho de criação do objeto não precisará nunca ser realizado se ele nunca for usado. Nos dois casos, as operações são postergadas até que o objeto esteja pronto.

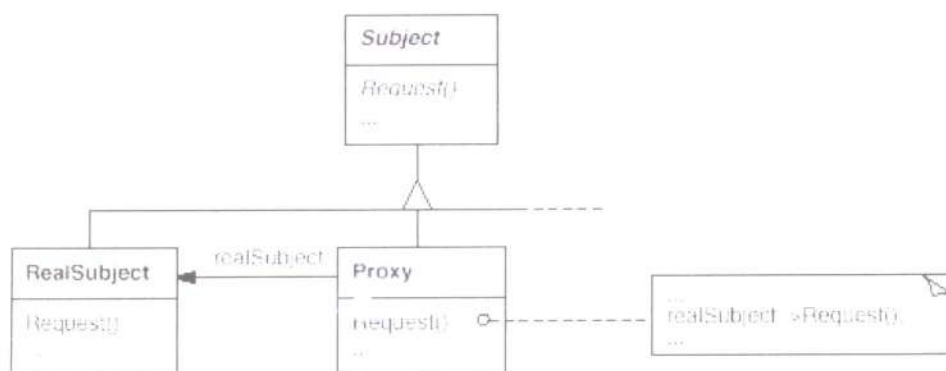


Figura 10 - Diagrama de classes do padrão Proxy

Fonte: GAMMA (1998)

1. **Proxy**: Mantém uma referência que permite ao *proxy* acessar o objeto real. Pode referenciar um *Subject* se as interfaces de *RealSubject* e *Subject* forem as mesmas. Fornece uma interface idêntica a de *Subject*, de modo que o *proxy* possa substituir o objeto real. Controla o acesso ao objeto real e pode ser responsável pela sua criação e exclusão.
2. **Subject**: Define uma interface comum para *RealSubject* e *Proxy*, de maneira que um *Proxy* possa ser usado em qualquer lugar em que um *RealSubject* é esperado.
3. **RealSubject**: Define o objeto real que o *proxy* representa

4.3 Padrões de Comportamento

4.3.1 TEMPLATE METHOD

De acordo com Gamma (1998) o objetivo desse padrão é definir o esqueleto de um algoritmo dentro de uma operação, deixando alguns passos a serem preenchidos pelas subclasses. Template Method permite que suas subclasses redefinam certos passos de um algoritmo sem mudar sua estrutura.

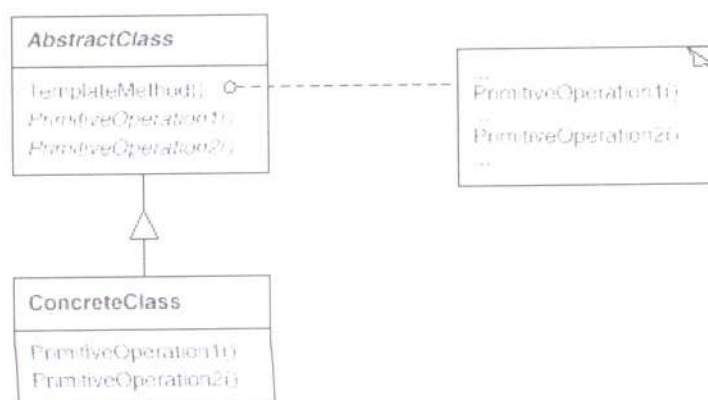


Figura 11 - Diagrama de classes do padrão Template Method
Fonte: GAMMA (1998)

1. **AbstractClass**: Define operações primitivas abstratas que as subclasses concretas definem para implementar passos de um algoritmo. O método template invoca operações primitivas.
2. **ConcreteClass**: Implementa as operações primitivas.

4.3.2 COMMAND

De acordo com Gamma (1998) o objetivo desse padrão é encapsular uma requisição como um objeto, permitindo que clientes parametrizem diferentes requisições, filas ou requisições de log, e suportar operações reversíveis.

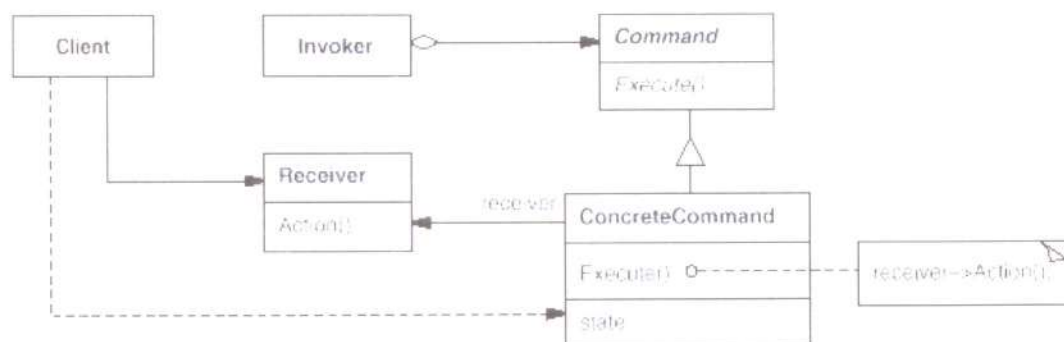


Figura 12 - Diagrama de classes do padrão Command
Fonte: GAMMA (1998)

1. **Command**: Declara uma interface para a execução de uma operação.
2. **ConcreteCommand**: Define um vínculo entre um objeto Receiver e uma ação. Implementa *execute()* através da invocação da(s) correspondente(s) operação(ões) no Receiver.
3. **Client**: Cria um objeto *ConcreteCommand* e estabelece o seu receptor (*Receiver*).
4. **Invoker**: Solicita ao *Command* a execução da solicitação.
5. **Receiver**: Sabe como executar as operações associadas a uma solicitação. Qualquer classe pode funcionar como um *Receiver*.

No próximo capítulo serão exibidos os principais padrões de projeto do catálogo JEE que foram utilizados no projeto Hércules da Natalsoft. Alguns dos padrões de projeto do catálogo GoF servem como base para esse catálogo específico da tecnologia JEE.

5 PADRÕES JEE

5.1 Padrões da camada de apresentação

5.1.1 INTERCEPTING FILTER

O mecanismo de tratamento de solicitação da camada de apresentação recebe muitos tipos diferentes de solicitações, que exigem tipos variados de processamentos. Algumas solicitações são simplesmente enviadas ao componente de tratamento apropriado, enquanto outras solicitações devem ser modificadas, examinadas ou descompactadas antes de serem processadas mais adiante.

A solicitação e a resposta na WEB de um cliente exigem o processamento prévio e posterior. Quando uma solicitação entra em uma aplicação WEB, ela deve frequentemente passar por alguns testes de entrada do estágio de processamento principal. Por exemplo:

1. O cliente foi autenticado?
2. O cliente tem uma sessão válida?
3. O endereço IP do cliente é de uma rede confiável?
4. O caminho da aplicação viola alguma restrição?
5. Que codificação o cliente utiliza para enviar os dados?
6. Há suporte ao tipo de navegador do cliente?

Algumas dessas verificações são testes, resultando em uma resposta positiva ou negativa que determina se o processamento prosseguirá. Outras verificações tratam o fluxo de entrada para uma forma adequada para o processamento.

5.1.1.1 Motivações

1. O processamento comum, como verificar o esquema de codificação de dados ou informações de acesso sobre cada solicitação, é concluído por solicitação.
2. É desejável a centralização de lógica comum
3. Os serviços devem ser fáceis de adicionar e remover em segundo plano sem afetar os componentes existentes, para que possam ser utilizados em várias combinações como:
 - a. Acesso a autenticação
 - b. Depuração e transformação de saída para um cliente específico

- c. Descompactação e conversão do esquema de conversão de entrada.

5.1.1.2 Solução

Criar filtros conectáveis para processar serviços comuns de uma maneira padrão sem exigir alterações de código de processamento de solicitação principal. Os filtros interceptam solicitações de entrada e respostas de saída, permitindo o processamento prévio e posterior. Os filtros podem ser adicionados e removidos em segundo plano, sem exigir alterações em códigos existentes. Pode-se decorar o processamento principal com uma variedade de serviços comuns, como segurança, acesso, depuração e assim por diante.

5.1.1.3 Estrutura

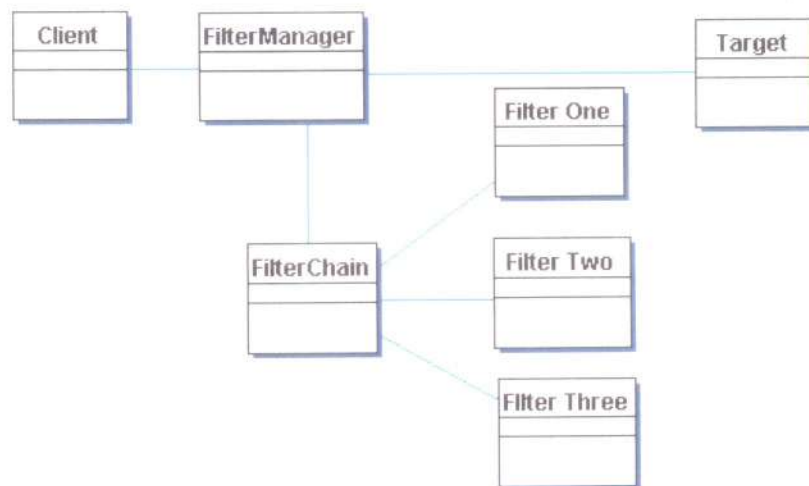


Figura 13 - Diagrama de classes do padrão Intercepting Filter
Fonte: ALUR (2002)

5.1.1.4 Participantes e responsabilidades

FilterManager

Gerencia o processamento de filtros. Cria o *FilterChain* com os filtros apropriados, na ordem correta e inicia o processamento.

FilterChain

É uma combinação ordenada de filtros independentes.

Filter One, Filter Two, Filter Three

São filtros individuais que estão mapeados para um destino. O *FilterChain* coordena seus processamentos.

Target

É o recurso solicitado pelo cliente.

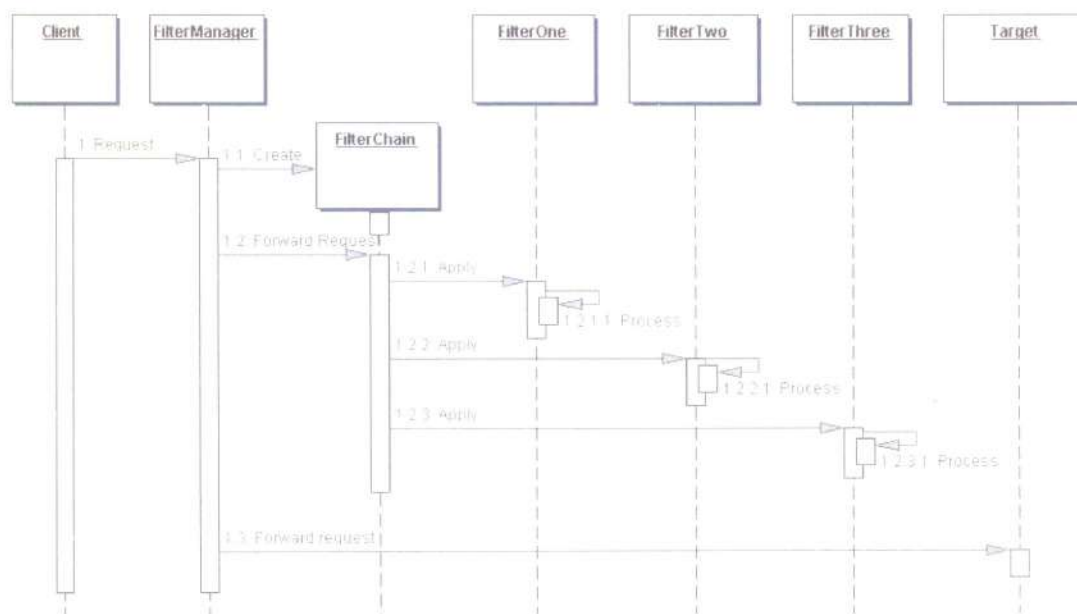


Figura 14 - Diagrama de sequência do Intercepting Filter
Fonte: ALUR (2002)

5.1.1.5 Consequências

Centraliza o controle com manipuladores acoplados livremente

Os filtros fornecem um lugar central para tratar o processamento através de diversas solicitações e são mais indicados para refinar solicitações e respostas para o tratamento final de um recurso de destino, como um controlador. Além disso, um controlador geralmente reúne o gerenciamento de inúmeros serviços comuns não relacionados, como autenticação, acesso, criptografia e assim por diante, embora filtrar permita muito mais manipuladores acoplados livremente, que podem ser acoplados de várias maneiras.

Melhora a capacidade de reutilização

Os filtros favorecem o particionamento da aplicação depuradora e encorajam a reutilização. Esses interceptores (*interceptors*) conectáveis são adicionados ou removidos de forma transparente a partir do código existente e devido a suas interfaces padrões, funcionam em qualquer combinação e são reutilizáveis para apresentações variadas.

Configuração declarativa e flexível

Inúmeros serviços são combinados em permutações variadas sem uma única recompilação da base de código principal.

O compartilhamento de informações é ineficiente

Compartilhar informações entre filtros pode ser ineficiente, visto que por definição cada filtro é acoplado livremente. Se grandes quantidades de informações devem ser compartilhadas entre os filtros, então, esse enfoque pode provar ser pouco vantajosa.

5.1.2 FRONT CONTROLLER

O mecanismo de tratamento de solicitação da camada de apresentação deve controlar e coordenar o processamento de cada usuário através de solicitações múltiplas. Dessa maneira os mecanismos de controle podem ser gerenciados de forma centralizada ou descentralizada.

O sistema exige um ponto de acesso centralizado de tratamento de solicitação da camada de apresentação para suportar a integração de serviços do sistema, recuperação de conteúdo, gerenciamento e navegação de visualização. Quando o usuário acessa a visualização diretamente sem operar através de um mecanismo centralizado, dois problemas podem ocorrer:

1. Cada visualização é solicitada a fornecer seus próprios serviços do sistema, resultando freqüentemente em códigos duplicados.
2. A navegação de visualização está fora das visualizações. Isso pode resultar em conteúdo e navegação de visualização misturados.

Além disso, o controle distribuído é mais difícil de manter, visto que as alterações precisarão freqüentemente ser feitas em diversos locais.

5.1.2.1 Motivações

1. O processamento de serviço de sistemas comuns é concluído por solicitação. Por exemplo, o serviço de segurança conclui as verificações de autenticação e autorização.
2. A lógica que é mais bem tratada em um local central, é replicada em inúmeras visualizações.
3. Pontos de decisão existem com respeito à recuperação e ao tratamento de dados.
4. As visualizações múltiplas são utilizadas para responder a solicitações de negócio semelhantes.
5. Um ponto centralizado de contato para tratar uma solicitação pode ser útil, por exemplo, para controlar e registrar o progresso de um usuário através do site.

6. Os serviços do sistema e a lógica de gerenciamento de visualização são relativamente sofisticados.

5.1.2.2 Solução

Utilizar um controlador como ponto inicial de contato para tratar uma solicitação. O controlador gerencia o tratamento da solicitação, incluindo chamar serviços de segurança como autenticação e autorização, delegar o processamento de negócios, gerenciar a seleção de uma visualização apropriada, tratar erros e gerenciar a seleção de estratégias de criação de conteúdos.

O controlador fornece um ponto de entrada centralizado que controla e gerencia o tratamento de solicitação da WEB, além disso, ajuda a reduzir a quantidade de código Java, denominado *scriptlets*, incluídos na JSP.

5.1.2.3 Estrutura

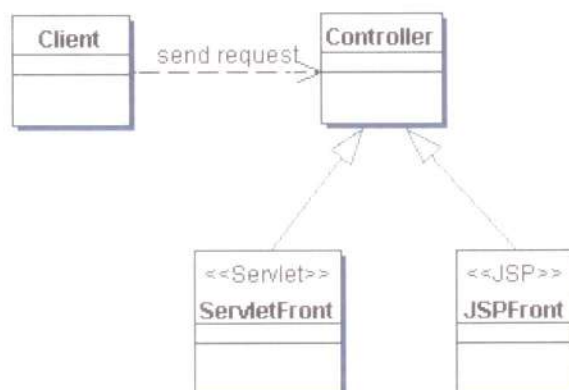


Figura 15 - Diagrama de classes do Front Controller
Fonte: ALUR (2002)

5.1.2.4 Participantes e responsabilidades

Controlador (*Controller*)

O controlador é o ponto de contato inicial para tratar todas as solicitações do sistema. O controlador pode permitir a um auxiliar concluir a autenticação e autorização de um usuário.

Distribuidor (*Dispatcher*)

Um distribuidor é o responsável pelo gerenciamento e navegação da visualização, gerenciar a seleção da próxima visualização e apresentar ao usuário e fornecer o mecanismo para distribuir o controle a esse recurso.

Um distribuidor pode ser encapsulado em um controlador ou pode ser um componente separado trabalhando em combinação. O mecanismo de distribuição pode ser estático ou dinâmico.

O distribuidor utiliza o objeto *RequestDispatcher* e encapsula alguns processamentos adicionais.

Auxiliar (*Helper*)

Um auxiliar é responsável em ajudar uma visualização ou um controlador a concluir seu processamento. Os auxiliares podem ter inúmeras responsabilidades, incluindo reunir dados exigidos pela visualização e armazenar esse modelo intermediário, formatar dados como conteúdo da Web e etc.

Uma visualização pode trabalhar com qualquer número de auxiliares, que normalmente são implementados como *JavaBeans* (JSP 1.0) e tags personalizadas (JSP 1.1). Além disso, um auxiliar pode representar um objeto *Command*, um representante (*delegate*) ou um *XSL Transformer*, que é utilizado em combinação com uma folha de estilo para adaptar e converter o modelo na forma apropriada.

Visualização (*view*)

Uma visualização representa e exibe informações para o cliente. A visualização recupera informações de um modelo. Os auxiliares suportam visualizações encapsulando e adaptando o modelo de dados essenciais para utilizar na exibição.

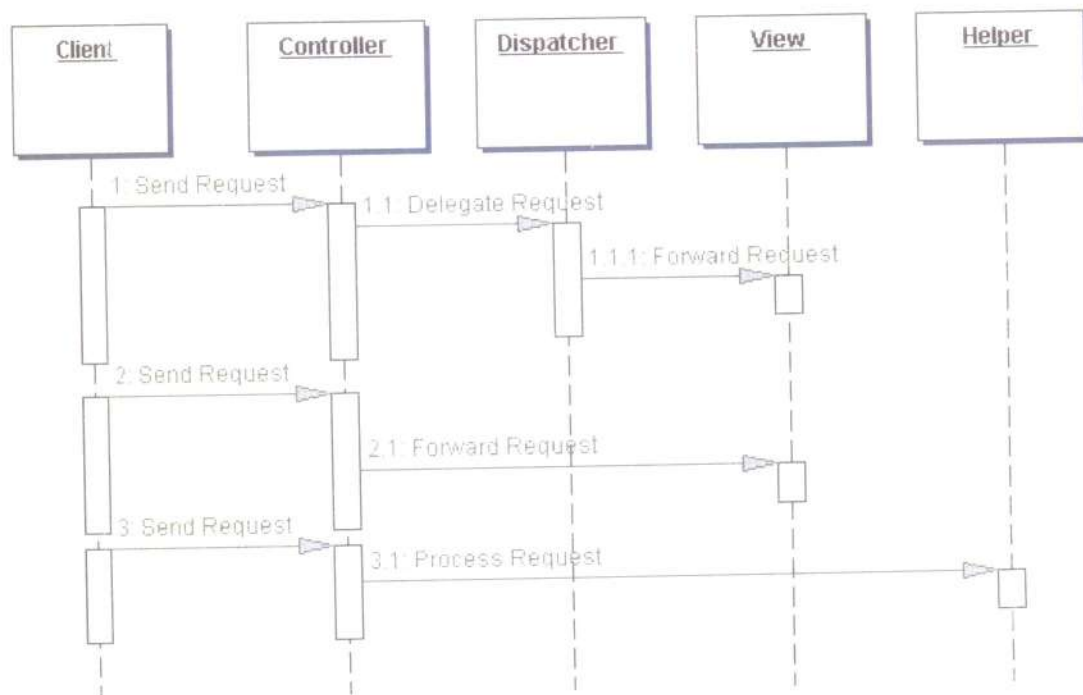


Figura 16 - Diagrama de sequência do Front Controller
 Fonte: ALUR (2002)

5.1.2.5 Conseqüências

Centraliza o controle

Um controlador fornece um local central para tratar os serviços do sistema e a lógica de negócios através de solicitações múltiplas. O acesso centralizado para uma aplicação significa que as solicitações são facilmente rastreadas e acessadas.

Melhora a capacidade de gerenciamento de segurança

Um controlador centraliza o controle, fornecendo um ponto de restrição para tentativas de acesso ilícitas na aplicação Web. Examinar uma única entrada na aplicação exige menos recursos que distribuir verificações de segurança através de todas as páginas.

5.1.3 VIEW HELPER

O sistema cria o conteúdo da apresentação, que exige o processamento de dados de negócios dinâmicos. As alterações da camada de apresentação ocorrem frequentemente e são difíceis de desenvolver e manter quando a lógica de acesso aos dados de negócios e a lógica de formatação da apresentação são combinadas. Isso torna o sistema menos flexível, menos reutilizável e geralmente menos acessível para alterar. Além disso, fornece uma fraca divisão de papéis entre a equipe de desenvolvedores de software e designers Web.

5.1.3.1 Motivações

1. As exigências de assimilação de dados de negócios não são triviais.
2. Embutir a lógica de negócios na visualização desenvolve um tipo de reutilização copie-e-cole. Isso causa problemas de manutenção e erros porque um componente de lógica é reutilizado em uma visualização diferente simplesmente duplicando-o no novo local.
3. O mais adequado desenvolver uma separação nítida de tarefas tendo indivíduos diferentes que realizem os papéis de desenvolvedor de software e de membro de equipe de design Web.
4. Uma visualização é utilizada normalmente para responder a uma determinada solicitação de negócios.

5.1.3.2 Solução

Uma visualização contém códigos de formatação, delegando suas responsabilidades de processamento para suas classes *helper*, implementados como *JavaBeans* ou tags personalizadas. Os auxiliares também armazenam o modelo de dados intermediários de visualização e servem como adaptadores de dados de negócios.

Encapsular a lógica de negócios em um auxiliar em vez de em uma visualização torna nossa aplicação mais modular e facilita a reutilização de componentes. Clientes múltiplos, como controladores e visualizações, podem estimular o mesmo auxiliar a recuperar e adaptar o estado de modelo similar para a apresentação em diversas maneiras.

5.1.3.3 Estrutura

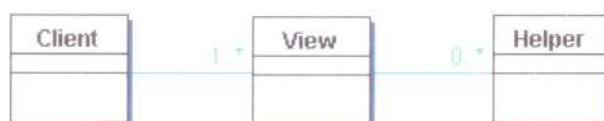


Figura 17 - Diagrama de Classes do View Helper
Fonte: ALUR (2002)

5.1.3.4 Participantes e responsabilidades

Um controlador geralmente serve de mediador entre o cliente e a visualização. Em alguns casos, porém, um controlador não é utilizado e a visualização se torna o ponto de contato inicial para tratar a solicitação.

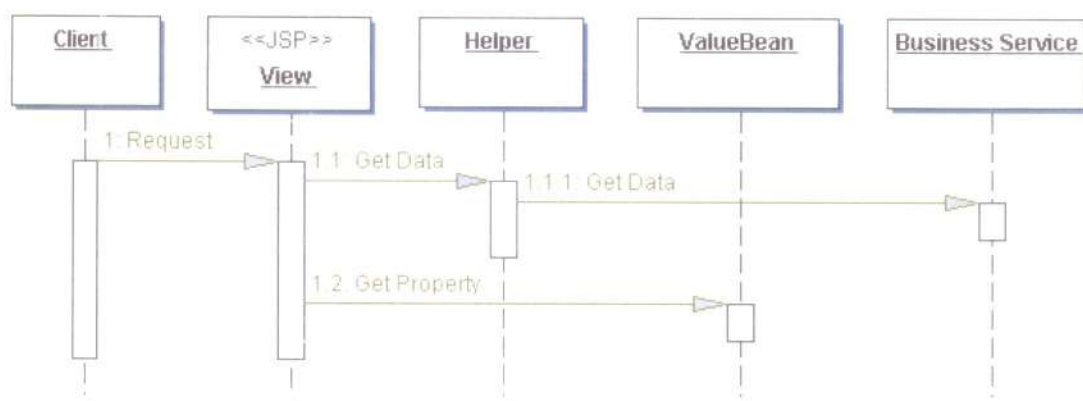


Figura 18 - Diagrama de sequência do View Helper
Fonte: ALUR (2002)

Como observado no diagrama de classes, podem não haver auxiliares associados a uma visualização. Nesse caso simples, a página pode ser inteiramente estática ou incluir quantidades muito pequenas de códigos *scriptlet in-line*.

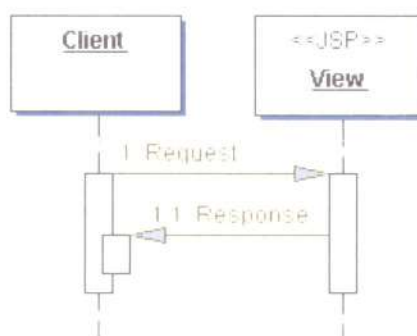


Figura 19 - Diagrama de sequência simples do View Helper
Fonte: ALUR (2002)

Visualização (*view*)

Representa e exibe as informações para o cliente. Os auxiliares suportam visualizações encapsulando e adaptando um modelo para utilizar em uma exibição.

Auxiliar (*helper*)

É responsável em ajudar uma visualização ou controlador a concluir seu processamento. Assim, os auxiliares têm inúmeras responsabilidades, incluindo reunir dados exigidos pela visualização e armazenar esse modelo intermediário, caso esse, onde o auxiliar às vezes é introduzido como um *value bean*. Além disso, os auxiliares podem adaptar esse modelo de dados para utilizar através da visualização. Os auxiliares podem manter solicitações de dados na visualização simplesmente fornecendo acesso para os dados não processados ou formatando os dados como conteúdo da Web.

Value Bean

É outro nome para um auxiliar que é responsável em manter o estado de modelo intermediário para utilizar através de uma visualização.

Serviço de negócios (*BusinessService*)

É um papel desempenhado pelo serviço que o cliente está procurando acessar. Normalmente, o serviço de negócios é acessado através de um representante de negócios (*business delegate*). O papel do representante de negócios é fornecer controle e proteção para o serviço de negócios.

5.1.3.5 Consequências

Melhora o particionamento, a reutilização e capacidade de manutenção da aplicação

Utilizar os auxiliares resulta em uma nítida divisão da visualização do processamento de negócios em uma aplicação. A lógica de negócios que é representada fora das JSPs é reutilizada nos *JavaBeans* e nas tags personalizadas, reduzindo a duplicação e facilitando a manutenção.

Melhora a separação de papéis

Separar a lógica de formatação da lógica de negócios da aplicação reduz as dependências que os indivíduos desempenhando papéis diferentes poderiam ter nos mesmos recursos.

5.1.4 COMPOSITE VIEW

As páginas Web sofisticadas apresentam conteúdos de inúmeras origens de dados, utilizando diversas visualizações secundárias que abrangem uma única página de exibição. Além disso, uma variedade de indivíduos com conjuntos de habilidades diferentes contribuem para o desenvolvimento e manutenção dessas páginas Web.

Em vez de fornecer um mecanismo para combinar de forma modular, pequenas parcelas de uma visualização em um todo da composição, as páginas são criadas embutindo o código de formatação diretamente em cada visualização. A modificação para o layout de diversas visualizações é difícil e propensa a erros, devido à duplicação de código.

5.1.4.1 Motivações

1. Pequenas parcelas de conteúdo da visualização se alteram frequentemente.
2. Diversas visualizações compostas utilizam visualizações secundárias similares, como uma tabela de inventário de cliente. Essas pequenas parcelas são decoradas com textos

templates de delimitações diferentes ou aparecem em um local diferente dentro da página.

3. As alterações de layout são mais difíceis de gerenciar e o código mais difícil de manter quando as visualizações secundárias são diretamente embutidas e duplicadas em diversas visualizações.
4. Embutir frequentemente parcelas de alterações de textos templates diretamente nas visualizações também afeta potencialmente a disponibilidade e administração do sistema.

5.1.4.2 Solução

Utilize visualizações compostas (*composite views*) que sejam compostas de varias visualizações secundárias menores. Cada componente do template pode ser incluído dinamicamente no todo e o layout da página pode ser gerenciado independentemente do conteúdo.

Essa solução favorece a reutilização de pequenas parcelas da visualização estimulando o projeto modular. As desvantagens são a sobrecarga de tempo de execução e dificuldade de gerenciamento e desenvolvimento, visto que há mais artefatos para manter e um nível de procedimentos de implementação para compreender.

5.1.4.3 Estrutura

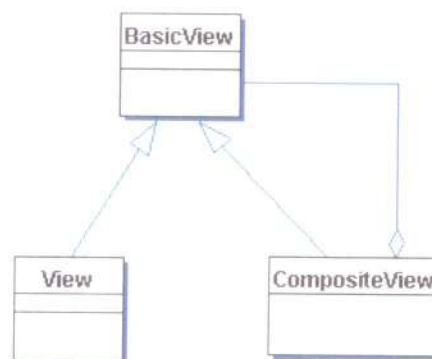


Figura 20 - Diagrama de classes do Composite View
Fonte: ALUR (2002)

5.1.4.4 Participantes e responsabilidades

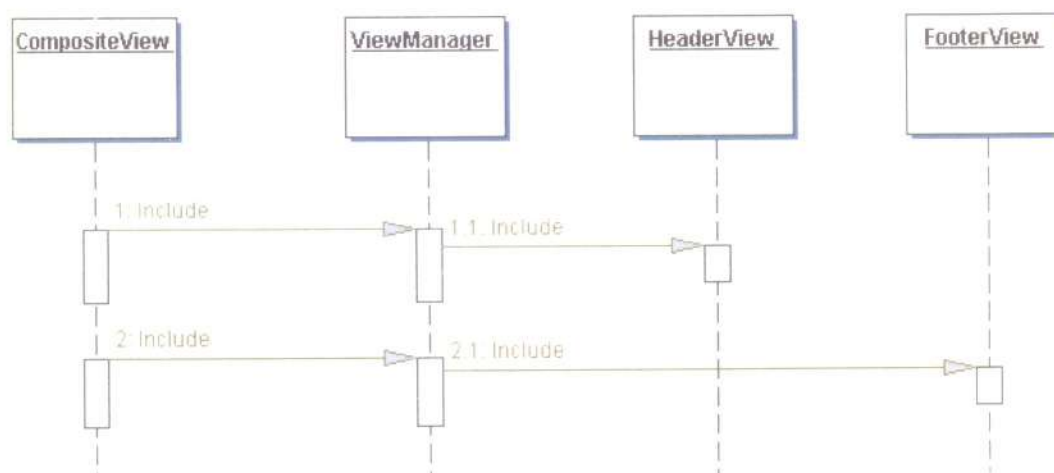


Figura 21 - Diagrama de sequência do Composite View
Fonte: ALUR (2002)

Composite view

É um agregado de diversas visualizações secundárias.

View Manager

Gerencia a inclusão de parcelas de fragmentos de templates na visualização composta. O *view manager* pode ser parte de um motor no momento da execução da JSP padrão, na forma da JSP padrão de incluir tag (`<jsp:include>`) ou pode ser encapsulado em um auxiliar *JavaBean* ou auxiliar de tag personalizada para fornecer uma funcionalidade mais robusta.

Included View

É uma visualização secundária de um pequeno pedaço de uma visualização integral maior. Essa visualização incluída provavelmente também poderia ser uma visualização composta, se incluísse diversas visualizações secundárias.

5.1.4.5 Conseqüências

Melhora a modularidade e a reutilização

O padrão estimula o projeto modular. É possível reutilizar pequenas parcelas de um template, como uma tabela de cotas de estoque, em inúmeras visualizações e para decorar essas parcelas reutilizadas com informações diferentes.

Aumenta a flexibilidade

Uma implementação sofisticada pode eventualmente incluir fragmentos de templates de visualização baseado em decisões no momento da execução, como o papel do usuário ou política de segurança.

Aumenta a capacidade de manutenção e de gerenciamento

É muito mais eficiente gerenciar alterações para parcelas de um template quando o template não estiver inserido no código diretamente na marcação da visualização. Quando mantidas separadas da visualização, é possível modificar as parcelas modulares de conteúdo do template independente do layout do template.

Reduz a capacidade de gerenciamento

Reunir pequenas partes da exibição para criar uma única visualização introduz a possibilidade de erros de exibição, visto que as visualizações secundárias são fragmentos de páginas. Essa é uma limitação que pode se tornar um problema na capacidade de gerenciamento. Por exemplo, de uma página JSP estiver gerando uma página HTML utilizando uma página principal que inclua três visualizações secundárias e cada uma delas inclua a tag aberta e fechada HTML (isto é, `<HTML>` e `</HTML>`), então, a página composta será inválida.

Impacto de desempenho

Gerar uma exibição que inclua diversas visualizações secundárias pode retardar o desempenho. A inclusão no momento da execução de visualizações secundárias resultará em uma demora sempre que a página for servida ao cliente.

5.2 Camada de negócios

5.2.1 BUSINESS DELEGATE

Um sistema distribuído de multicamadas requer chamadas de método remotas para enviar e receber dados através das camadas. Os clientes são expostos à complexidade de lidar com os componentes distribuídos.

Os componentes da camada de apresentação interagem diretamente com os serviços de negócios. Essa interação direta expõe, para a camada de apresentação, os detalhes de implementação da camada de negócios. Como resultado, os componentes da camada de apresentação ficam vulneráveis às alterações na implementação dos serviços de negócios: quando a implementação dos serviços de negócios é alterada, o código da implementação exposto na camada de apresentação também precisa ser alterado.

5.2.1.1 Motivações

1. Os clientes da camada de apresentação precisam acessar os serviços de negócios.
2. Diferentes clientes, como os dispositivos móveis, clientes Web e clientes Desktop precisam acessar os serviços de negócios
3. As API dos serviços de negócios podem alterar de acordo com a evolução das exigências de negócios.
4. É desejável minimizar o acoplamento entre os clientes da camada de apresentação e o serviço de negócios, ocultando, dessa maneira, os detalhes de implementação por trás do serviço, como a pesquisa e o acesso.
5. Os clientes podem precisar implementar mecanismos de cache para informações de serviços de negócios.
6. É desejável reduzir o tráfego da rede entre o cliente e os serviços de negócios.

5.2.1.2 Solução

Utilize um *Business Delegate* para reduzir o acoplamento entre os clientes da camada de apresentação e os serviços de negócios. O *Business Delegate* oculta os detalhes de implementação por trás do serviço de negócios, como detalhes de pesquisa e acesso da arquitetura EJB.

Este padrão atua como uma abstração dos negócios no lado do cliente. Utilizá-lo reduz o acoplamento entre os clientes da camada de apresentação e os serviços de negócios do sistema.

O *Business Delegate* também trata as exceções geradas pelos serviços de negócios, como as exceções *java.rmi.Remote*, as exceções JMS e assim por diante. Ele pode interceptar tais exceções de nível de serviço e gerar exceções no nível de aplicação, que são mais fáceis de serem tratadas pelos clientes e podem ser amigáveis para os usuários. O *Business Delegate* também pode, de modo transparente, executar qualquer operação necessária de tentativa ou recuperação caso haja uma falha no serviço, sem expor o cliente ao problema até que seja determinado que o problema não é solucionável.

Outro benefício é que o padrão pode colocar em cache resultados e referências para serviços de negócios remotos. Isso pode melhorar significativamente o desempenho, porque ele limita transmissões desnecessárias e potencialmente

O *Business Delegate* utiliza um componente denominado *Lookup Service* que é responsável por ocultar os detalhes de implementação por trás do código de pesquisa de serviços

de negócios. O *Lookup Service* pode ser escrito como parte do *Delegate*, mas nós recomendamos que ele seja implementado como um componente separado, como será resumido no padrão *Service Locator*.

Deve ser observado que esse padrão pode ser utilizado para reduzir o acoplamento entre outras camadas, não se restringindo às camadas de apresentação e de negócios.

5.2.1.3 Estrutura

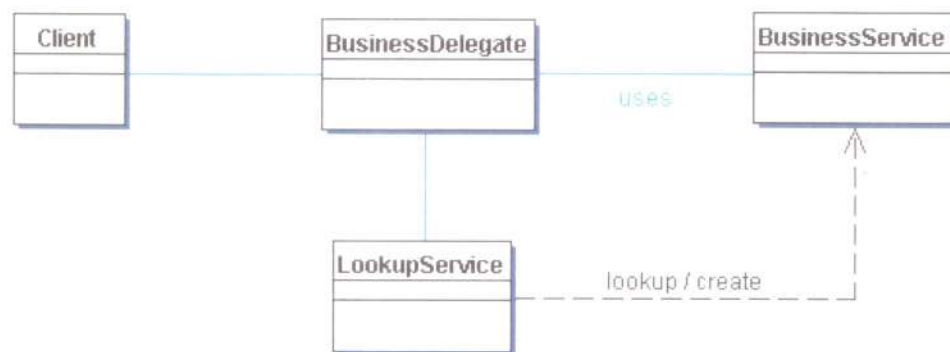


Figura 22 - Diagrama de classes do padrão Business Delegate
Fonte: ALUR (2002)

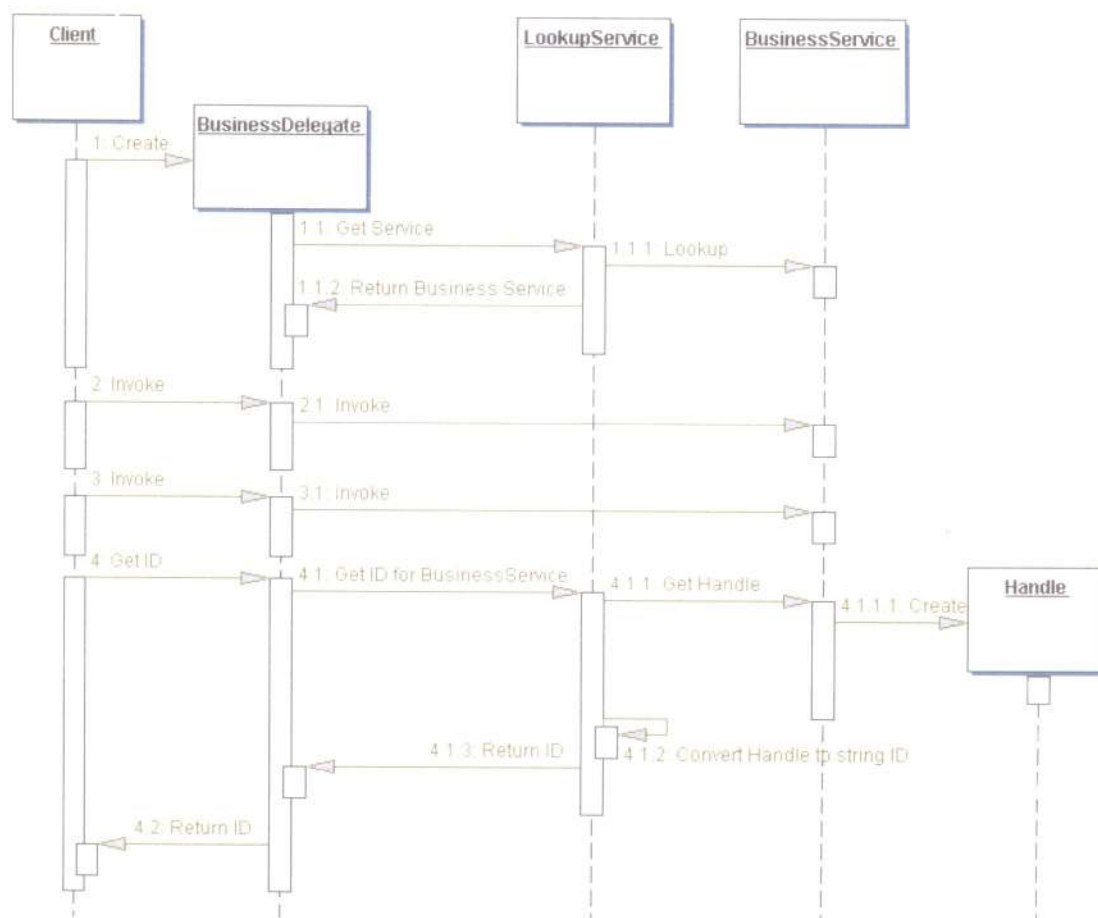


Figure 5-1 - Diagrama de sequência do padrão Business Delegate

Fonte: ALUR (2002)

Observe na figura que o cliente solicita que o *Business Delegate* proporcione acesso a serviços de negócios subjacentes. O *Business Delegate* utiliza um *Lookup Service* para localizar o componente do *BusinessService* solicitado.

5.2.1.4 Participantes e responsabilidades

Business Delegate

A função do *Business Delegate* é proporcionar controle e proteção para o serviço de negócios. Ele pode expor dois tipos de construtores para o cliente. Um tipo de solicitação instancia o *Business Delegate* sem um ID, enquanto o outro tipo instancia com um ID, onde o ID é uma versão *String* da referência para um objeto remoto, como o *EJBHome* ou o *EJBObject*.

Quando inicializado sem um ID, o *Business Delegate* solicita o serviço a partir do *Lookup Service*, normalmente implementado como um *ServiceLocator*, que retorna o *Service Factory*,

como um *EJBHome*. O *Business Delegate* solicita que o *Service Factory* localize, crie ou remova um *BusinessService*, como um *enterprise bean*.

Quando inicializado com uma *string ID*, o *Business Delegate* utiliza a *string ID* para reconectar ao *Business Service*. Dessa maneira, o *Business Delegate* protege o cliente dos detalhes de implementação por trás da nomeação e pesquisa do *Business Service*. Além disso, o cliente da camada de apresentação nunca faz diretamente uma chamada remota em um *BusinessSession*. Em vez disso, ele utiliza o *Business Delegate*.

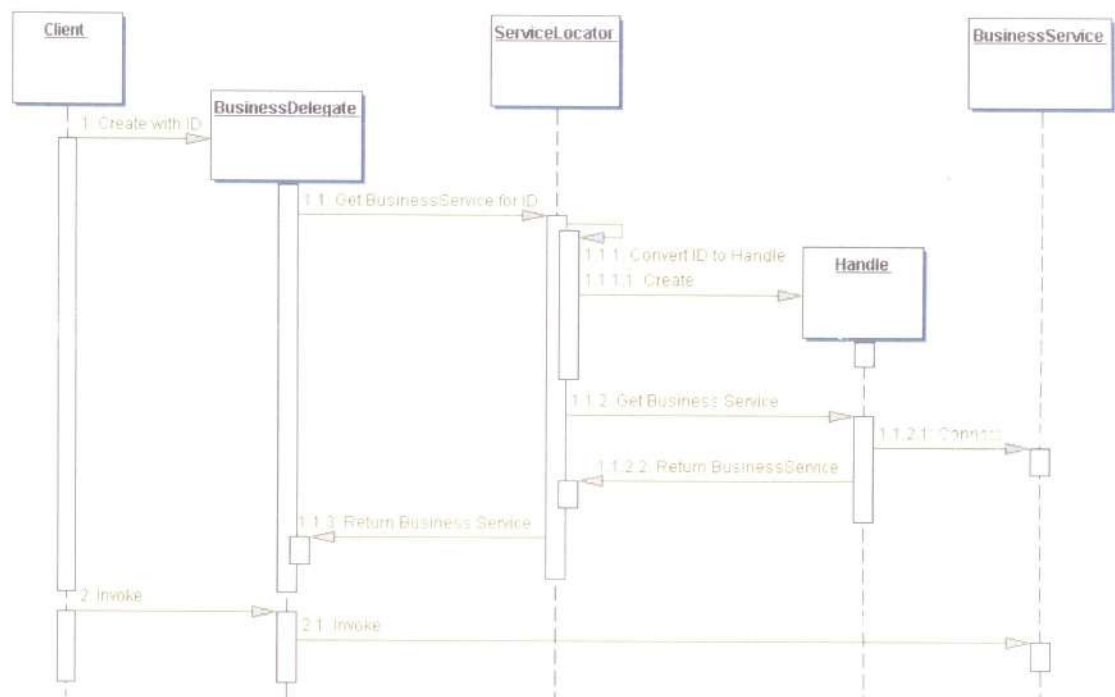


Figura 23 - BusinessDelegate com um diagrama de sequência ID
Fonte: ALUR (2002)

LookupService

O *Business Delegate* utiliza o *LookupService* para localizar o *BusinessService*. O *LookupService* encapsula os detalhes de implementação de pesquisa do *BusinessService*.

BusinessService

É um componente da camada de negócios, como um EJB ou JMS, que fornece o serviço solicitado para o cliente.

5.2.1.5 Conseqüências

Reduz o acoplamento e melhora a gerenciabilidade

O *Business Delegate* reduz o acoplamento entre as camadas de apresentação de negócios ocultando todos os detalhes de implementação da camada de negócios. Fica mais fácil gerenciar as alterações porque elas estão centralizadas em um único lugar, o *Business Delegate*.

Traduz as exceções do serviço de negócios

O *Business Delegate* é responsável por traduzir quaisquer exceções de rede ou relacionadas à infraestrutura em exceções de negócios, protegendo os clientes de implementação subadjacentes.

Implementa recuperação de falhas e sincronização de *thread*

O *Business Delegate* ao encontrar uma falha do serviço de negócios pode implementar os recursos de recuperação automática sem expor o problema ao cliente. Se a recuperação for bem sucedida, o cliente não precisa ter conhecimento da falha, caso contrário, o *Business Delegate* precisará informar ao cliente.

Expõe uma interface mais simples e uniforme para a camada de negócios

O *Business Delegate*, para melhor atender a seus clientes, pode proporcionar uma variante da interface fornecida pelos *enterprise beans* subadjacentes.

Afeta o desempenho

O *Business Delegate* pode oferecer serviços de cache para a camada de apresentação, para as solicitações de serviços comuns.

Introduz uma camada adicional

O *Business Delegate* pode ser visto como uma camada desnecessária entre o cliente e o serviço, introduzindo, dessa maneira, uma complexidade adicional e diminuindo a flexibilidade. Os benefícios do padrão normalmente superam essas desvantagens.

Oculta a localização remota

Embora a transparência da localização seja um dos benefícios desse padrão, um diferente problema pode surgir devido ao fato do desenvolvedor tratar um serviço remoto como se ele fosse um serviço local. Isso pode acontecer se o desenvolvedor do lado do cliente não compreender que o *Business Delegate* é um *proxy* do cliente para um serviço remoto e passar a fazer inúmeras chamadas de método para resolver uma tarefa, aumentando dessa forma o tráfego da rede.

5.2.2 SESSION FACADE

Os *enterprise beans* encapsulam a lógica de negócios e os dados de negócios e expõem suas interfaces e, dessa maneira, a complexidade de serviços distribuídos, para a camada de cliente.

Em um ambiente de aplicação JEE de múltiplas camadas, surgem os seguintes problemas:

1. Um forte acoplamento, que leva à dependência direta entre os clientes e os objetos de negócios
2. Muitas chamadas de métodos entre o cliente e o servidor, levando a problemas de desempenho da rede.
3. Falta de uma estratégia uniforme de acesso ao cliente, expondo os objetos de negócios a um emprego incorreto.

Uma aplicação JEE de múltiplas camadas possui inúmeros objetos do servidor que são implementados como *enterprise bean*. Os clientes da aplicação precisam acessar esses objetos de negócio. Os clientes podem interagir diretamente com esses objetos de negócios porque eles expõem suas interfaces. Os clientes precisam compreender e ser responsáveis por relacionamentos de objetos de dados de negócios, além de serem capazes de lidar com o fluxo de processo de negócios.

Entretanto, a interação direta entre o cliente e os objetos de negócios leva a um forte acoplamento entre os dois. Tal acoplamento torna o cliente diretamente dependente da implementação dos objetos de negócio. A complexidade de interação entre os vários objetos de negócio aumenta à medida que as exigências dos clientes aumentam. Dessa forma o cliente torna-se muito suscetível às alterações na camada do objeto de negócios. Além disso, ele é exposto desnecessariamente à complexidade subjacente do sistema.

Ao acessarem os *enterprise beans*, os clientes interagem com objetos remotos. Podem aparecer problemas no desempenho da rede se o cliente interagir diretamente com todos os objetos de negócios participantes porque o alto volume de chamadas de método remotas aumenta a quantidade de interações ao longo da camada de rede.

Além disso, surge um problema quando um cliente interage diretamente com os objetos de negócios, pois não existe uma estratégia unificada para acessar tais objetos. Sem essa estratégia de acesso ao cliente uniforme os objetos de negócios são expostos para os clientes e isso pode reduzir uma utilização consistente.

5.2.2.1 Motivações

1. Fornecer uma interface mais simples para os clientes ocultando todas as complexas interações entre os componentes de negócios.
2. Reduzir o número de objetos de negócios que estão expostos para o cliente na camada de serviço ao longo da rede.
3. Ocultar do cliente as interações subadjacentes e as interdependências entre os componentes de negócios. Isso proporciona uma melhor gerenciabilidade, centralização das interações (reponsabilidade), maior flexibilidade e maior habilidade para lidar com as alterações.
4. Fornecer uma camada de serviço uniforme de granulação grossa para separar a implementação do objeto de negócios da remoção do serviço de negócios.
5. Evitar a exposição dos objetos de negócios diretamente para manter o acoplamento entre as duas camadas no menor grau possível.

5.2.2.2 Solução

Utilizar um bean de sessão como um facade para encapsular a complexidade das interações entre os objetos de negócios participantes em um fluxo de trabalho. O *Session Facade* gerencia os objetos de negócios e fornece uma camada de acesso a serviços uniforme e de granulação grossa para os clientes.

O *Session Facade* remove as interações de objeto de negócios subadjacentes e proporciona uma camada de serviços que expõe apenas as interfaces necessárias. Dessa maneira, ele oculta da visualização do cliente as interações complexas entre os participantes. O *Session Facade* gerencia as interações entre os dados de negócios e os objetos de serviço de negócios que participam do fluxo de trabalho. O *bean* de sessão (representando o *Session Facade*) também gerencia o ciclo de vida desses participantes criando, localizando, modificando e excluindo conforme for exigido pelo fluxo de trabalho.

5.2.2.3 Estrutura

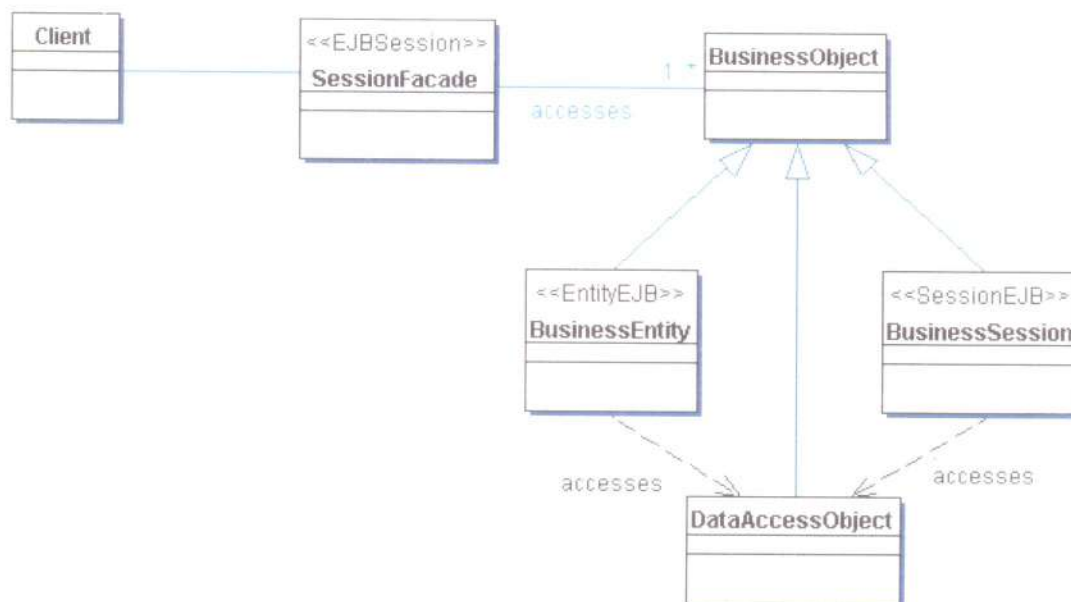


Figura 24 - Diagrama de Classe Session Facade
Fonte: ALUR (2002)

5.2.2.4 Participantes e colaborações

Cliente

Representa o cliente do *Session Facade*, que necessita acessar o serviço de negócios. Esse cliente pode ser outro bean de sessão na mesma camada de negócios ou um representante de negócios em outra camada.

Session Facade

É implementado como um *bean* de sessão. Ele gerencia os relacionamentos entre inúmeros *BusinessObjects* e proporciona uma abstração de nível mais alto para o cliente. O *Session Facade* oferece acesso de granulação grossa para o *BusinessObject* participante representado pela chamada *Invoke* para o *bean* de sessão.

BusinessObject

É um objeto de função que facilita a aplicação de diferentes estratégias, como os *beans* de sessão e de entidade e um DAO. Um *BusinessObject* fornece dados e/ou algum serviço no diagrama de classe. A *Session Facade* interage com várias instâncias do *BusinessObject* para proporcionar o serviço.

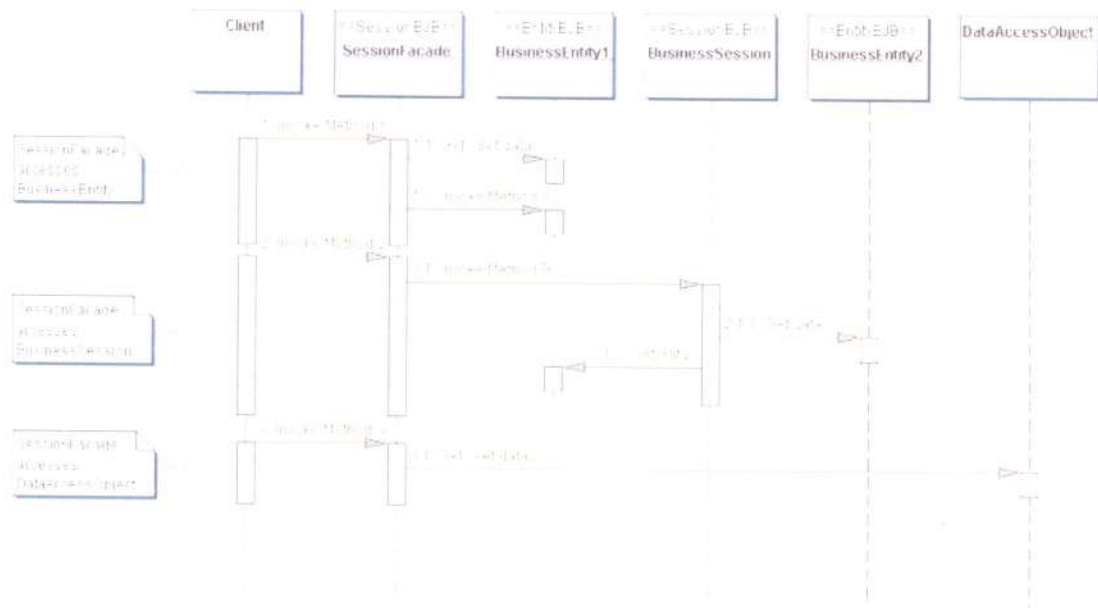


Figura 25 - Diagrama de sequencia Session Facade
Fonte: ALUR (2002)

5.2.2.5 Conseqüências

Introduz uma camada *Controller* (controlador) da camada de negócios

Um *Session Facade* abrange as interações entre o cliente e os componentes de negócios. Tanto em uma aplicação sofisticada como em uma aplicação simples, utilizar um *Session Facade* logo de início produzirá benefícios em um estágio mais avançado.

Expõe a interface uniforme

Utiliza-se um *Session Facade* para criar o design de uma camada de serviços mais simples para o sistema. Dessa maneira, um facade proporciona uma camada de acesso de granulação grossa e uniforme para todos os tipos de clientes além de proteger e ocultar os componentes de negócios participantes subadjacentes.

Reduz o acoplamento, aumenta a gerenciabilidade

Utilizar um *Session Facade* desacopla os objetos de negócios dos clientes, reduzindo, dessa maneira, o forte acoplamento e a dependência do cliente em relação aos objetos de negócios. É melhor utilizar um *Session Facade* para gerenciar o fluxo de trabalho entre os objetos de negócios, em vez de fazer com que esses objetos tomem consciência uns dos outros. Isso proporciona uma maior gerenciabilidade, centralização de interações (responsabilidade e fluxo de trabalho), maior flexibilidade e maior habilidade para lidar com as alterações.

Melhora o desempenho, reduz os métodos de granulação fina

O *Session Facade* reduz a sobrecarga da rede entre o cliente e o servidor porque sua utilização elimina a interação direta entre o cliente, os dados de negócios e os objetos de serviço de negócios. Em vez disso, todas as interações são roteadas através do *Session Facade* de uma maneira de granulação grossa. O *Session Facade* e os seus participantes estão mais próximos entre si, tornando mais eficiente para o *facade* o gerenciamento das interações entre os objetos participantes.

Fornece acesso de granulação grossa

Um *Session Facade* é projetado para ser uma abstração de granulação grossa do fluxo de trabalho. Determina-se a granularidade ótima de um *Session Facade*, particionando a aplicação em subsistemas lógicos e fornecendo um *Session Facade* para cada subsistema.

Centraliza gerenciamento de segurança

As políticas de segurança para a aplicação podem ser gerenciadas no nível de *Session Facade*, visto que essa é a camada apresentada para o cliente. Devido ao acesso de granulação grossa do *Session Facade*, é mais fácil e mais gerenciável definir as políticas de segurança nesse nível, em vez de defini-las no nível dos componentes de negócios participantes.

Centraliza o controle de transação

Como o *Session Facade* representa o fluxo de trabalho para casos de uso, é mais lógico aplicar o gerenciamento de transação no nível do *Session Facade*. O controle de transação centralizado possui vantagens similares à segurança centralizada. Além disso, ter o acesso direto do cliente ao enterprise *bean* em vez de utilizar o *Session Facade* coloca a carga da demarcação da transação sobre o cliente e pode produzir resultados indesejados.

Expõe menos interfaces remotas para o cliente

Os clientes que interagem diretamente com os dados de negócios e os objetos de serviço de negócios provocam um aumento no ruído de rede entre o cliente e o servidor. O *facade* apresenta um mecanismo de acesso de granulação grossa para os componentes de negócios, isso reduz o número desses componentes que são expostos para o cliente.

5.2.3 SERVICE LOCATOR

Os clientes da JEE interagem com os componentes de serviço, como os componentes EJB e JMS, que fornecem serviços de negócios e capacidades persistentes. Para interagir com esses componentes, os clientes precisam localizar o componente de serviço ou criar um novo componente. O serviço de pesquisa e criação envolve interfaces complexas e operações de rede.

Todos os clientes das aplicações JEE utilizam o recurso comum JNDI para procurar e criar componentes EJB e JMS. A API JNDI permite aos clientes obter um objeto de contexto inicial que mantenha o nome do componente para vinculações de objeto. O cliente inicia obtendo o contexto inicial para um objeto local do bean. O contexto inicial permanece válido enquanto a sessão do cliente for válida. O cliente fornece o nome do JNDI registrado para o objeto requerido para obter uma referência a um objeto administrado.

Localizar um objeto de serviço administrado pelo JNDI é comum para todos os clientes que precisam acessar esse objeto de serviço, por isso, é fácil ver que muitos tipos de clientes utilizam repetidamente o serviço JNDI e esse código aparece muitas vezes entre os clientes. Isso resulta em uma duplicação desnecessária de código nos clientes que precisam procurar serviços.

5.2.3.1 Motivações

1. Os clientes EJB precisam utilizar a API JNDI para procurar objetos EJB-Home utilizando o nome JNDI registrado no enterprise bean.
2. Clientes JMS precisam utilizar a API JNDI para procurar componentes JMS usando nomes registrados para componentes JMS.
3. A fábrica de contexto usada para a criação do contexto inicial JNDI é fornecida pelo provedor do serviço e é, portanto, dependente do fornecedor. A fábrica de contexto também depende do tipo de objeto que está sendo pesquisado.
4. A pesquisa e a criação de componentes de serviço podem ser complexas e utilizadas repetidamente em múltiplos clientes na aplicação.
5. A criação de contexto inicial e as pesquisas de objetos de serviço, se frequentemente requeridas, podem utilizar muitos recursos e talvez afetem o desempenho da aplicação. Isso é verdadeiro especialmente se os clientes e serviços estiverem localizados em diferentes camadas.
6. Os clientes EJB talvez precisem reestabelecer a conexão para uma instância de *enterprise bean* previamente acessada, possuindo apenas seu objeto Handle.

5.2.3.2 Solução

Utilizar um objeto *Service Locator* para remover toda utilização JNDI e ocultar as complexidades da criação de contexto inicial, pesquisa de objeto *EJBHome* e recriação de objeto EJB. Vários clientes podem reutilizar o objeto *Service Locator* para reduzir a complexidade do código, fornecer um único ponto de controle e melhorar o desempenho fornecendo o recurso do armazenamento em cache.

Esse padrão reduz a complexidade do cliente que resulta da dependência do cliente e da necessidade de executar processos de pesquisa e criação, o que utiliza muitos recursos. Para eliminar esses problemas, esse padrão proporciona um mecanismo para remover todas as dependências e detalhes de rede no *Service Locator*.

5.2.3.3 Estrutura

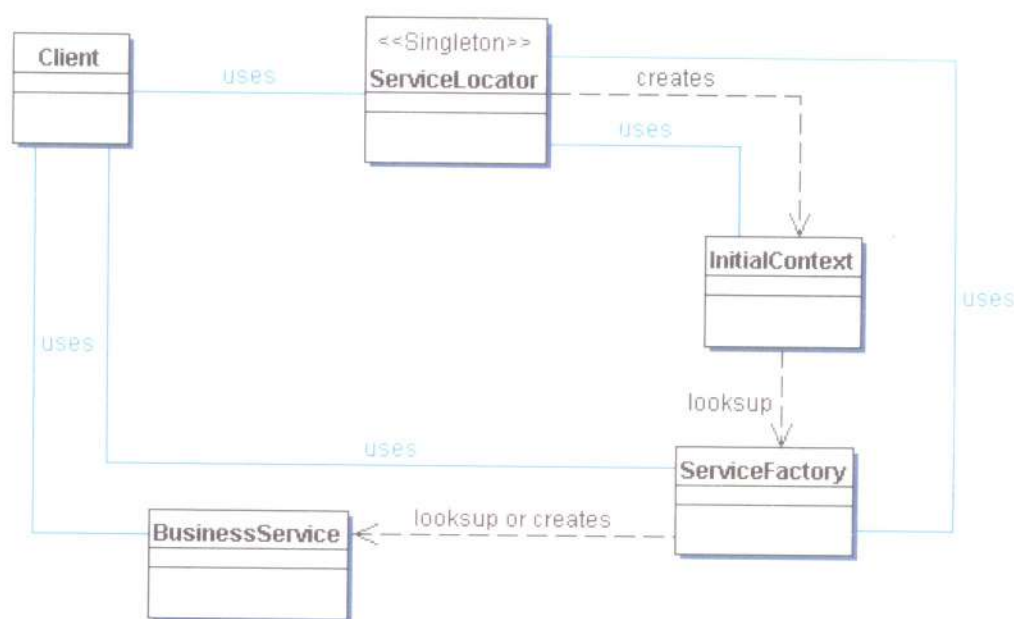


Figura 26 - Diagrama de classes Service Locator
Fonte: ALUR (2002)

5.2.3.4 Participantes e colaborações

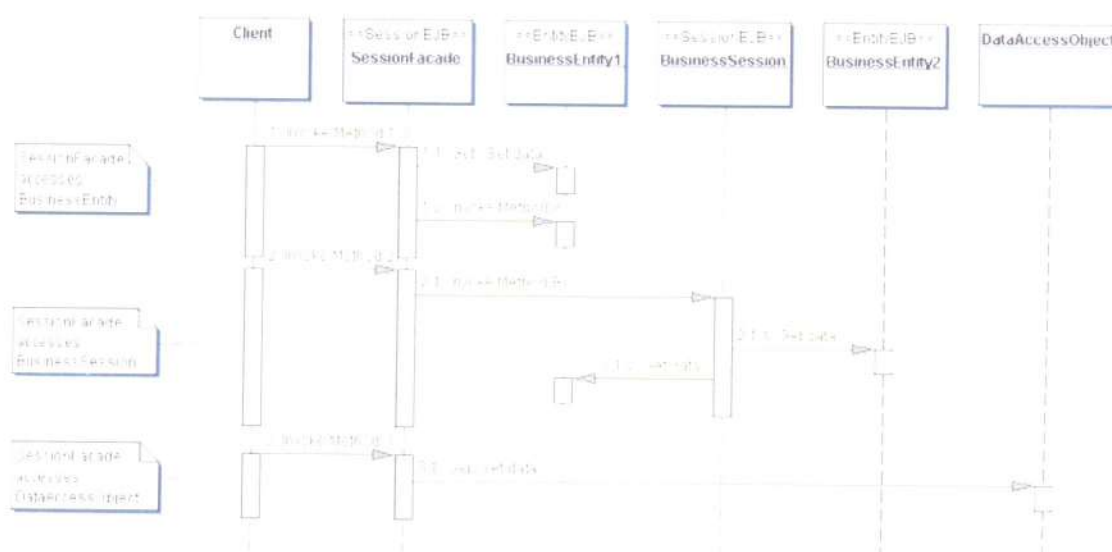


Figura 27 - Diagrama de Sequência Service Locator

Fonte: ALUR (2002)

Cliente

Esse é o cliente do *Service Locator*. O cliente é um objeto que normalmente requer acesso a objetos de negócios tais como um *Business Delegate*.

Service Locator

Remove os serviços de pesquisa da API, dependências de fornecedor, complexidades de pesquisa e criação de objetos de negócios e proporciona uma interface simples para o cliente. Isso reduz a complexidade do cliente. Além disso, o mesmo cliente ou outros podem reutilizar o *Service Locator*.

InitialContext

É o ponto inicial em um processo de criação e pesquisa. Os provedores de serviços proporcionam o objeto contexto, que varia dependendo do tipo de objeto de negócios fornecido pelo serviço de criação e pesquisa do *Service Locator*. Um *Service Locator* que proporciona serviços para múltiplos tipos de objetos de negócios utiliza vários tipos de objetos contexto, cada um obtido de um diferente provedor.

Service Factory

Representa um objeto que proporciona gerenciamento de ciclo de vida para os objetos *BusinessObject*.

Business Service

É uma função que é preenchida pelo serviço que o cliente está buscando acessar. O objeto *BusinessService* é criado, pesquisado ou removido pelo *ServiceFactory*.

5.2.3.5 Conseqüências

Remove a complexidade

Esse padrão remove a complexidade desse processo de pesquisa e criação e a mantém oculta do cliente. O cliente não precisa lidar com a pesquisa de objetos da fábrica de componentes pelo fato do *Service Locator* estar incumbido dessas responsabilidades.

Proporciona aos clientes acesso uniforme a serviço

Esse padrão proporciona uma interface muito útil e precisa que pode ser utilizada por todos os clientes. A interface padrão garante que todos os tipos de clientes na aplicação acessarão uniformemente os objetos de negócios, em termos de pesquisa e criação. Essa uniformidade reduz sobrecarga no desenvolvimento e na manutenção.

Facilita a adição de novos componentes de negócios

Como os clientes dos *enterprise beans* não estão cientes dos objetos *EJBHome*, é possível adicionar novos objetos *EJBHome* para *enterprise beans*, desenvolvidos e distribuídos posteriormente, sem que isso afete os clientes. Os clientes JMS não estão cientes diretamente das fabricas de conexão JMS, então, novas fabricas de conexão podem ser adicionadas sem afetá-los.

Melhora o desempenho da rede

Os clientes não estão envolvidos na pesquisa JNDI e na criação de objetos *factory/home*. Como o *Service Locator* realiza esse trabalho, ele pode agregar as chamadas de rede necessárias para procurar e criar objetos de negócios.

Melhora o desempenho do cliente armazenado em cache

O *Service Locator* pode armazenar em cache os objetos de contexto inicial e referências a objetos de fábrica para eliminar atividades JNDI desnecessárias que ocorrem durante a obtenção do contexto inicial e de outros objetos. Isso melhora o desempenho da aplicação.

5.3 Camada de Acesso a Dados

5.3.1 DATA ACCESS OBJECT (DAO)

O acesso a dados varia dependendo da origem dos dados. O acesso ao armazenamento persistente, como a um banco de dados, varia muito dependendo do tipo de armazenamento (bancos de dados relacionais, bancos de dados orientados a objetos, arquivos planos e assim por diante) e da implementação do fornecedor.

Muitas aplicações JEE implementam o armazenamento persistente com mecanismos diferentes e há diferenças marcantes nas APIs utilizadas. Outras aplicações talvez precisem acessar dados que residem em sistemas separados. Outro exemplo é onde os dados são fornecidos pelos serviços através de sistemas externos como sistemas de integração de comércio eletrônico entre empresas, serviço central de cartão de crédito e assim por diante.

As aplicações podem utilizar a API JDBC para acessar dados residentes em um sistema de gerenciamento de banco de dados relacional (RDBMS, *relational database management system*). A API JDBC permite às aplicações JEE utilizarem declarações SQL, que são meios padrões para acessar as tabelas do RDBMS, a sintaxe e o formato reais das declarações SQL podem variar dependendo do produto de banco de dados específico.

Origens de dados diferentes oferecem desafios à aplicação e podem potencialmente criar uma dependência direta entre o código da aplicação e o código de acesso aos dados. Incluir a

conectividade e o código de acesso de dados dentro de componentes de negócios introduz um acoplamento estrito entre os componentes e a implementação da origem de dados. Assim, as dependências de códigos em componentes tornam difíceis e cansativas a migração da aplicação de um tipo de origem de dados para outro. Quando a origem de dados muda, os componentes precisam ser alterados para tratar o novo tipo de origem de dados.

5.3.1.1 Motivações

1. Componentes como beans de entidade gerenciados pelo *bean* (*bean managed entity beans*), *beans* de sessão (*session beans*), *servlets* e outros objetos como auxiliares JSPs precisam recuperar e armazenar informações de armazenamento persistentes e outras origens de dados como sistemas legados, B2B, LDAP e assim por diante.
2. As API de armazenamento persistente variam dependendo do fornecedor do produto. Essas APIs também variam dependendo do tipo de armazenamento – RDBMS, sistemas de bancos de dados orientados a objetos (OODBMS, *object-oriented database management system*), documentos de XML, arquivos planos e assim por diante.
3. Os componentes normalmente utilizam APIs exclusivas para acessar sistemas externos e/ou legados para recuperar e armazenar dados.
4. A portabilidade dos componentes é diretamente afetada quando mecanismos e APIs de acesso específicos são incluídos nos componentes.
5. Os componentes precisam ser transparentes ao armazenamento persistente real ou à implementação de origem de dados para fornecer migração fácil a produtos de fornecedores diferentes, tipos de armazenamentos diferentes e tipos de origem de dados diferentes.

5.3.1.2 Solução

Utilizar um *Data Access Object* (DAO) para extrair e encapsular todos os acessos à origem de dados. O DAO gerencia a conexão com a origem de dados para obter e armazenar dados.

O DAO implementa o mecanismo de acesso exigido para trabalhar com a origem de dados. Ele oculta completamente os detalhes da implementação da origem de seus clientes. Como a interface exposta pelo DAO a seus clientes não se altera quando a implementação da origem de dados subjacente se altera, esse padrão permite que o DAO se adapte a esquemas diferentes de armazenamento sem afetar seus clientes ou componentes de negócios. Essencialmente, o DAO age como um adaptador entre o componente e a origem de dados.

5.3.1.3 Estrutura

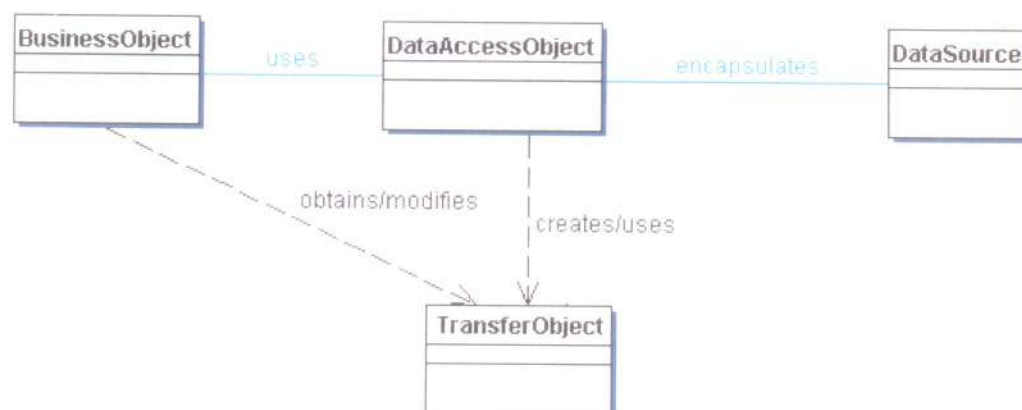


Figura 28 - Data Access Object (DAO)
Fonte: ALUR (2002)

5.3.1.4 Participantes e responsabilidades

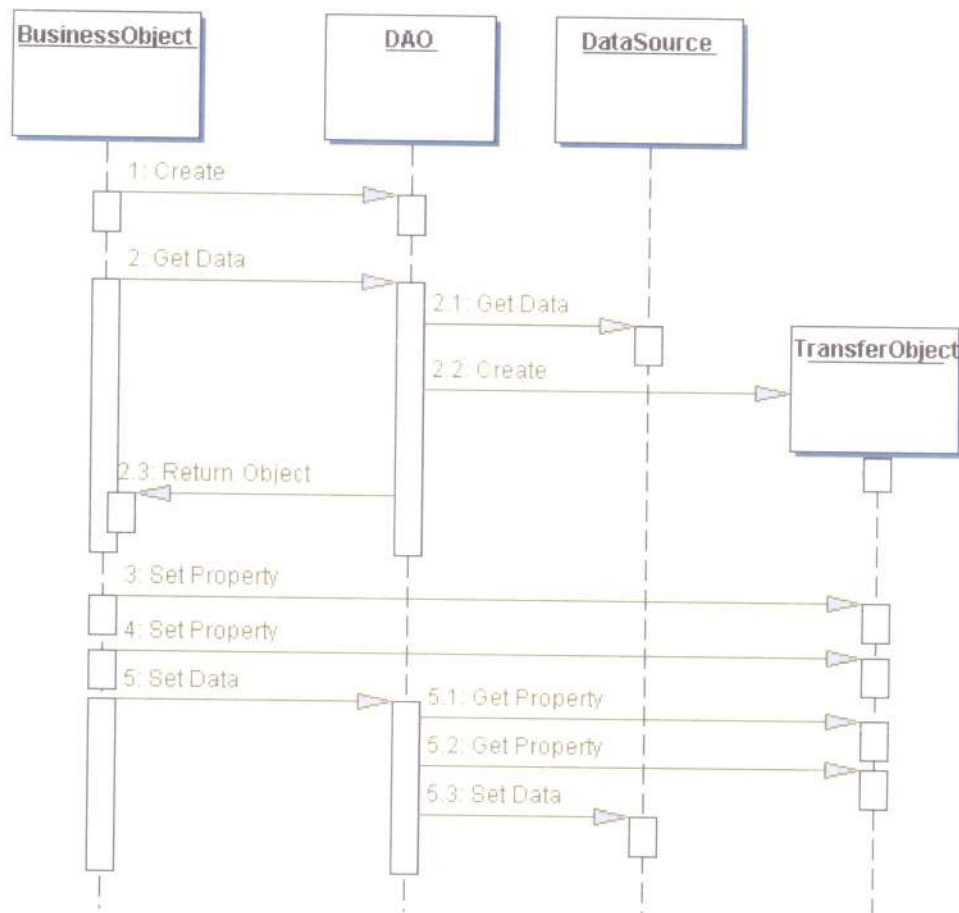


Figure 5-2 - Diagrama de sequência do Data Access Object (DAO)

Fonte: ALUR (2002)

BusinessObject

Representa o cliente de dados. É o objeto que requer acesso à origem de dados para obter e armazenar dados. Um *BusinessObject* pode ser implementado como um *bean* de sessão, um *bean* de entidade ou algum outro objeto do Java, incluindo um *servlet* ou um *bean* auxiliar que acesse a origem de dados.

DataAccessObject (DAO)

É o objeto principal desse padrão. Ele extrai a implementação de acesso de dados subadjacente do *BusinessObject* para permitir o acesso transparente à origem de dados. O *BusinessObject* também delega as operações de carregar e armazenar dados ao *DataAccessObject*.

DataSource

Representa uma implementação de origem de dados. Uma origem de dados poderia ser um banco de dados como um RDBMS, OODBMS, repositório de XML, sistema de arquivos planos e assim por diante. Uma origem de dados também pode ser outro sistema (legado/mainframe), serviço (serviço B2B ou central de cartão de crédito) ou algum tipo de repositório (LDAP).

ValueObject

Representa um objeto de dados (*value object*) utilizado como um transportador de dados. O *DataAccessObject* pode utilizar um objeto de dados para retornar dados ao cliente.

5.3.1.5 Conseqüências

Permite a transparência

Os objetos de negócio podem utilizar a origem de dados sem conhecer os detalhes específicos da implementação da origem de dados. O acesso é transparente porque os detalhes da implementação estão ocultos dentro do DAO.

Permite a migração mais fácil

Uma camada de DAO torna mais fácil para uma aplicação migrar para uma implementação de banco de dados diferente. Os objetos de negócios não tem nenhum conhecimento da implementação de dados subjacente.

Reduz a complexidade dos códigos nos objetos de negócios

Como os DAOs gerenciam todas as complexidades de acesso a dados, simplifica os códigos nos objetos de negócio e outros clientes de dados que utilizem os DAOs. Todas as implementações relativas a códigos (como declarações SQL) estão contidas no DAO e não nos objetos de negócios. Isso melhora a legibilidade dos códigos e a produtividade do desenvolvimento.

Centraliza todo o acesso de dados em uma camada separada

Como todas as operações de acesso a dados estão agora delegadas aos DAOs, a camada de acesso a dados separada pode ser visualizada como a camada que pode isolar o resto da aplicação da implementação de acesso a dados. Essa centralização torna a aplicação mais fácil de manter e gerenciar.

Adiciona camada extra

Os DAOs criam uma camada adicional de objetos entre o cliente de dados e a origem de dados que precisa ser atribuída e implementada para aumentar os benefícios desse padrão. Mas o benefício liberado escolhendo essa abordagem compensa o esforço adicional.

5.4 Outros padrões importantes

Apesar desses dois padrões que serão exibidos agora não estarem no catálogo de padrões JEE, eles são padrões dessa tecnologia e foram utilizados no projeto Hércules da empresa Natalsoft Ltda.

5.4.1 PADRÃO OPEN SESSION IN VIEW

Em uma aplicação WEB típica, após a solicitação ser completada e a visualização ser renderizada, a sessão é fechada e a transação com o banco de dados é finalizada. Se o desenvolvedor acessar objetos detached que foram carregados na sessão dentro da JSP, estará certamente acessando uma coleção que não está inicializada e será disparada uma exceção.

Pode-se utilizar o *pattern Open Session in View*, para permitir que sejam carregados relacionamentos do tipo *LAZY*, mesmo após a transação ter sido finalizada.

O problema é quando as *Entities* carregadas do banco de dados são passadas para a camada de apresentação em seu estado detached os atributos anotados com *fetch=LAZY* podem estar indisponíveis. Para resolver esse problema pode-se utilizar um *filter* para que não tenha que ser criada uma nova *Hibernate Session* para carregar esses dados. O *filter* faz com que existe apenas uma *Hibernate Session* para cada *thread* (no caso das aplicações WEB, um *Hibernate Session* para cada *request*). Essa abordagem provê de imediato duas vantagens:

1. Otimização de recursos
2. Fechamento de *Hibernate Session* garantido

5.4.2 PADRÃO EJB COMMAND

Um cliente EJB necessita executar as regras de negócio para completar um caso de uso. Como um desenvolvedor pode implementar as regras de negócio de um caso de uso de uma maneira leve, desacoplando o cliente do EJB e executando o caso de uso em uma transação e chamada de rede?

Uma decisão crítica numa arquitetura de software é onde colocar as regras de negócio. Colocá-las nos clientes (*Servlets*, *applets*, etc) tem conseqüências negativas, afetando o desempenho e manutenibilidade como explicado no padrão *Session Facade*. Poderíamos então, utilizar um *Session Facade* para resolver esse problema, mas chamá-lo diretamente de um cliente

poderia causar um acoplamento entre camadas que seria maléfico à manutenibilidade de acordo com o que foi visto no padrão *Business Delegate*.

5.4.2.1 Motivações

O padrão *Session Facade* em combinação com o padrão *Business Delegate* é a melhor prática para escrever regras de negócio de forma a desacoplar o cliente dos detalhes de implementação do servidor e permite a execução do caso de uso em uma única chamada de rede e em uma única transação. Essa combinação também tem os seus problemas:

1. **Processo de desenvolvimento lento** – Como a regra de negócio (que muda constantemente) é executada em um *session bean*, quando é necessário mudar um caso de uso (adicionar parâmetros para um método ou retornar um atributo extra), pode ser necessário mudar o método do *session bean*. O processo de alterar um *session bean* não é trivial – a alteração requer alteração no código fonte e redistribuir no *EJB Server* (possivelmente será necessário reiniciar o servidor). Possivelmente também será necessário alterar o *Business Delegate* que encapsula o *session bean* alterado no cliente.
2. **Divisão do trabalho em um grande projeto é mais difícil** – Dependendo das estratégias usadas para particionar o trabalho dos desenvolvedores em um projeto, o *Session Facade* é um gargalo que diferentes times de desenvolvedores precisam passar, por isso, ele pode ser alvo de freqüentes mudanças no progresso do projeto.

Resumidamente, utilizar o padrão *Session Facade* em conjunto com *Business Delegate* pode resultar em um longo processo de codificar, distribuir e testar e isso pode ser um gargalo em um grande projeto. A raiz desse problema é que as regras de negócio estão sendo colocadas na camada de sessão EJB e pode ser pesado o seu desenvolvimento.

5.4.2.2 Solução

Utilizar o padrão *Command* para envolver as regras de negócio em um *command bean* leve que desacople o cliente do EJB, execute em uma única chamada de rede e atue como um *Facade* para a camada do EJB.

Um *command bean* é uma classe Java com *gets*, *sets* e um método *execute*, como descrito no padrão *Command* (Gamma, et al., 1995). Aplicado em EJB, o padrão *Command* fornece uma solução leve para atingir os mesmos benefícios dos padrões *Session Facade* e *Business Delegate*. Isso é alcançado porque o *Command* fornece clientes com classes locais, mas seus comandos são executados dentro de um servidor EJB remoto.

São utilizados comandos para encapsular unidades de trabalho individuais. A interação de um cliente com um comando é muito simples, deve-se criá-lo e atribuir todos os seus atributos até que ele possua todos os dados necessários para executar o caso de uso. Nesse ponto o cliente chama o método de execução do comando, após isso, capturam-se todos os dados resultantes da execução do comando/caso de uso.

Quando um cliente executa um comando, coisas interessantes acontecem nos bastidores. Ao invés de ser executado localmente, o comando é transferido para um servidor EJB remoto e sua execução será feita na JVM desse servidor. Quando a execução do comando é finalizada, ele retorna para o cliente que pode capturar seus dados através de métodos *gets*. Pelo fato do comando ser executado em um servidor EJB, um caso de uso pode ser executado em apenas uma transação.

5.4.2.3 Participantes e responsabilidades

Há muitas maneiras de implementar o padrão EJB Command, mas todas elas possuem os mesmos 3 elementos:

1. **Command beans** – Uma classe Java simples com *gets*, *sets* e um método *execute* que contém a regra de negócio do caso de uso. Os command beans são apenas uma parte da implementação do padrão *Command*, os outros componentes serão descritos abaixo e serão reusados em outros projetos.
2. **Client-side routing logic** – geralmente um *framework* de classes que é responsável por enviar um comando para um servidor EJB remoto. A rotina de roteamento geralmente não é visível para o cliente e é disparada quando o método *execute* é chamado. Esse componente é genérico e pode ser reutilizado em outros projetos.
3. **Remot Command Server** – é um serviço que simplesmente aceita os comandos e executa-os. Aplicados para EJB, a classe *command Server* é um *stateless session bean* que aceita um comando como parâmetro e executa-o localmente. Esse componente também é genérico e pode ser reutilizado em outros projetos.

5.4.2.4 Consequências

1. **Facilita o desenvolvimento rápido de aplicações devido a um processo de desenvolvimento e distribuição leves** – escrever um caso de uso com um *command bean* é consideravelmente mais fácil e rápido para distribuir e testar que utilizar método um *session bean*. Alterações freqüentes podem ser feitas em classe Java básica, mas o mesmo não pode ser feito em uma classe EJB.

2. **Separação das regras de negocio da lógica de apresentação** – Comandos atuam como uma facade encapsulando regras de negocio dentro de comandos e expondo apenas uma interface para os clientes usarem. Essa separação permite que a camada cliente e servidor evoluam separadamente.
3. **Força a execução do caso de uso em uma única viagem de ida e volta** – Como o comando é executado em um servidor EJB, somente uma chamada de rede é necessária para completar um caso de uso.
4. **Desacopla o cliente do EJB** – Clientes são completamente desacoplados dos detalhes de implementação do servidor – tudo que eles vêem são os *commands beans*, que aparecem como classes locais.
5. **Comandos podem ser executados localmente ou produzir dados de teste** – Comandos de teste podem ser criados no início do projeto, permitindo os desenvolvedores da camada de apresentação escrever, compilar e testar seu código independentemente da equipe de desenvolvedores das regras de negócio.
6. **Controle de transações de granulação grossa** – como os comandos são apenas classes Java, não há uma forma de configurar um comando para executar uma transação em uma configuração em particular ou os níveis de isolamento, como podem ser feitos com os métodos dos session beans. Todos os comandos executam com as configurações do servidor EJB.
7. **Comandos são Stateless** – O objeto *Command* não pode armazenar nenhum estado na sessão que o executa.
8. **Dificuldade para gerenciamento de erros** – Como o *framework* do padrão *command* é genérico, somente um *CommandException* pode ser disparado por um *command*. Isso significa que as exceções da aplicação devem ser capturadas e embrulhadas dentro de um *CommandException*.

6 ARQUITETURA DO SISTEMA HÉRCULES

6.1 Introdução

Para lidar com sistemas grandes e complexos deve-se utilizar uma arquitetura de software que sirva de estrutura para o entendimento dos componentes dos sistemas e seus inter-relacionamentos. Essa arquitetura deve permitir o reuso em outros sistemas de mesmo contexto, por isso, boas práticas de design devem ser utilizadas para maximizar o reaproveitamento de seus componentes, aumentar a produtividade e diminuir os custos de manutenção.

A arquitetura de software que será abordada nessa sessão está sendo utilizada no projeto Hércules da empresa Natalsoft Ltda. Ela é uma arquitetura em camadas e provê uma infraestrutura de software para a construção de sistemas WEB utilizando a tecnologia JEE. Essa arquitetura pode ser considerada como um framework para construção de sistemas utilizando a tecnologia Java.

A utilização dessa arquitetura tem um papel importante no sucesso do projeto Hércules, pois trás algumas vantagens imprescindíveis. As vantagens da utilização dessa arquitetura são:

1. **Comunicação entre os envolvidos** - O projeto da arquitetura é um ponto de referência para discussão entre os responsáveis por cada uma das partes do sistema além de servir como modelo para padronização dos demais sistemas;
2. **Requisitos não funcionais** - A arquitetura pode atender a alguns requisitos não funcionais do sistema, como autorização de acesso e controle de log;
3. **Reuso** - A estrutura básica da arquitetura pode ser reutilizada em sistemas com tecnologia semelhante. Temos com isso a redução de tempo e custo de desenvolvimento e manutenção do software;
4. **Desempenho** - Disposição das funcionalidades de forma a minimizar a comunicação entre os subsistemas e suas diversas camadas;
5. **Segurança** - Uso de arquitetura em camadas nas quais os elementos críticos residem nas camadas mais internas;
6. **Facilidade de Manutenção** - Uso de elementos simples e auto-contidos;

6.2 Camadas

A arquitetura do sistema Hércules está modelada nas camadas de apresentação, persistência e negócio. A Figura 29 exhibe as camadas do sistema Hércules e quais as tecnologias, *frameworks* e APIs que são usados em cada uma delas.

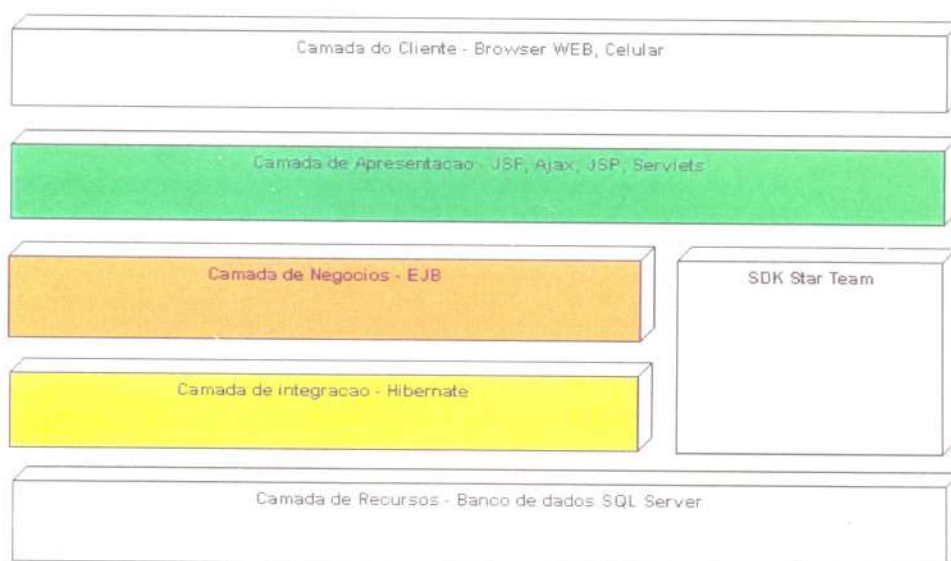


Figura 29 - Camadas do Sistema Hércules
 Fonte: Dados da Pesquisa (2007)

6.2.1 CAMADA DE APRESENTAÇÃO

A camada de visualização da arquitetura tem a função de abstrair e/ou estender algumas funcionalidades do *framework* JSF. De uma forma geral, a arquitetura contribui na camada de visualização com as seguintes funcionalidades:

1. Abstração de managed beans (padrão JSF): Utiliza-se o padrão GoF *TemplateMethod*.
2. Controlador de requisições centralizado: Utiliza-se o padrão JEE *FrontController*.
3. Implementação de *Tag-Libraries* Personalizadas.
4. Compactação das requisições e respostas: Utiliza-se o padrão *Intercepting Filter*.

Os seguintes padrões de projeto são utilizados na camada de apresentação da arquitetura:

6.2.1.1 Composite View

As telas do sistema foram definidas de acordo com o padrão de projeto *Composite View*, uma *template* para as páginas JSP. Esta *template* é uma página JSP que especifica o layout de todas as páginas do sistema. A *template* define cinco seções em uma página e utiliza *custom tags* para incluir outras páginas JSP nestas seções. A Figura abaixo apresenta o layout produzido pela *template*.

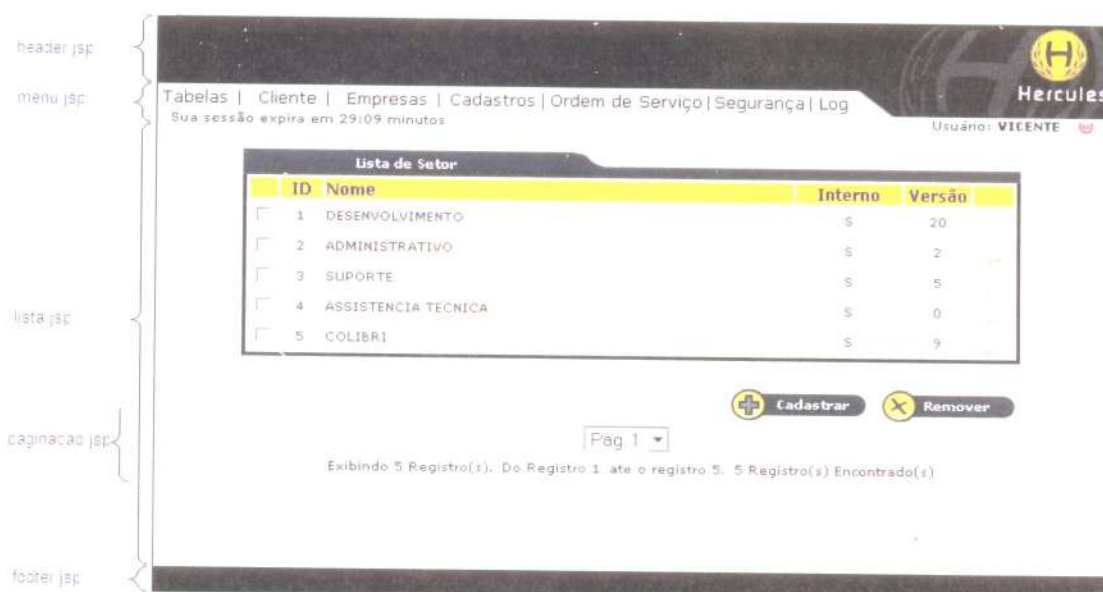


Figura 30 - Definição de Templates para páginas do sistema
Fonte: Dados da Pesquisa (2007)

Utilizando esta template, o conteúdo das seções cabeçalho, identificação, menu, paginação e rodapé, foram definidos somente uma vez para todo o sistema, como páginas JSP, sendo reutilizadas várias vezes.

6.2.1.2 Intercepting Filter

O sistema utiliza um filtro denominado *GZIPFilter* que é implementado de acordo com padrão de projeto *Intercepting Filter*. Esse filtro responsável por compactar o conteúdo no formato GZIP. Essa compactação é suportada por praticamente todos os *browsers*. Para aqueles que não suportarem esse tipo de compactação a informação não será modificada. A compressão de conteúdo possui uma utilidade muito grande em aplicações WEB, pois permite minimizar a utilização da largura de banda e por isso tornará a aplicação mais rápida.

O código abaixo exibe um trecho do arquivo `web.xml` que é utilizado do projeto Hércules da empresa Natalsoft Ltda. Esse trecho é referente ao mapeamento do filtro *GZIPFilter*. A classe `br.com.natalsoft.arq.web.GZIPFilter` é responsável por filtrar as requisições de páginas com a extensão `.jsp`, `.html` e `.jsf`.

```

...
<filter>
  <filter-name>Compress</filter-name>
  <filter-class>br.com.natalsoft.arq.web.GZIPFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>Compress</filter-name>
  <url-pattern>*.jsp</url-pattern>

```

```

</filter-mapping>

<filter-mapping>
    <filter-name>Compress</filter-name>
    <url-pattern>*.html</url-pattern>
</filter-mapping>

<filter-mapping>
    <filter-name>Compress</filter-name>
    <url-pattern>*.jsf</url-pattern>
</filter-mapping>

```

Listagem 5 - Trecho do arquivo web.xml
Fonte: Dados da Pesquisa (2007)

6.2.1.3 Front Controller

Todas as solicitações (com extensão “.jsf”) da camada de apresentação são controladas por um front controller, a classe *HerculesFacesController*, que estende a classe *MyFacesServlet* e centraliza o controle de autenticação, tratamento de exceções, log de operações de navegação e gerenciamento do ciclo de vida das sessões do Hibernate.

Código fonte do *HerculesFacesController*

```

public class HerculesFacesController extends MyFacesServlet {
    public void service(ServletRequest req, ServletResponse res)
        throws IOException, ServletException {
        HttpServletRequest request = (HttpServletRequest) req;
        String reqUrl = request.getRequestURL().toString();

        int pos = 7; // posição do início da URL (http://)
        if (req.isSecure()) {
            pos = 8;
        }

        Usuario user = (Usuario) request.getSession().getAttribute(
            "usuarioLogado");

        if (user == null
            && !reqUrl.endsWith("/login.jsf")
            && !reqUrl
                .endsWith("/portais/discente/menu_discente_externo.jsf")) {

            HttpServletResponse response = (HttpServletResponse) res;
            response.sendRedirect("/hercules/login.jsf");
            return;
        }

        // long initOperacao = System.currentTimeMillis();
        long initOperacao = System.currentTimeMillis();
        try {

```

```

System.out.println("Iniciando Requisição do Controlador");

super.service(req, res);
System.out.println("Fim do Service");
if (user != null) {

    UserAgent.logaOperacao(request.getRequestURL().toString(),
        System.currentTimeMillis() - initOperacao, 0,
        ErrorUtils.parametrosToString(request));

}
} catch (Exception e) {
    res.getWriter().println("<pre>Erro!");
    if (e instanceof ServletException) {
        e = ErrorUtils.traceFacesException((ServletException) e);
        if (e instanceof SegurancaException) {
            request.getSession().setAttribute("urlAcesso",
                request.getRequestURL());
            request.getSession().setAttribute("codAcessoNegado",
                (SegurancaException) e).getCodAcesso());

            ((HttpServletResponse) res).sendRedirect("/hercules/include/acesso.jsf"
);
            return;
        }
    }

    e.printStackTrace(res.getWriter());
    res.getWriter().println("</pre>");

    UserAgent.logaOperacaoErro(request.getRequestURL().toString(),
        System.currentTimeMillis() -
            initOperacao, 0, ErrorUtils
                .parametrosToString(request), e, 0);

}

finally {
    Session session = (Session)
        req.getAttribute(Database.SESSION_ATTRIBUTE);
    if (session != null && session.isOpen()) {
        session.clear();
        try {
            session.connection().close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        session.close();

        System.out.println("Fez ou sessão no controlador");
    } else {
        try {
            if (session.connection() != null)
                session.connection().close();
        } catch (Exception e) {

```

```

    // e.printStackTrace();
  }
  System.out.println("Sessão já fechada ou nula");
}
}
}
}

```

Listagem 6 - Código-fonte do HerculesFacesController
Fonte: Dados da Pesquisa (2007)

Segue abaixo um breve resumo das classes utilizadas na camada de apresentação do projeto Hércules e seu respectivo diagrama de classes.

1. **AbstractController** - Classe que implementa comportamentos que todos os controladores devem ter, tais como controle de paginação, redirecionamento de páginas, etc.
2. **AbstractControllerCadastro** - Controlador abstrado para CRUD (*create, read, update e delete*) de entidades. Estende a classe *AbstractController*.

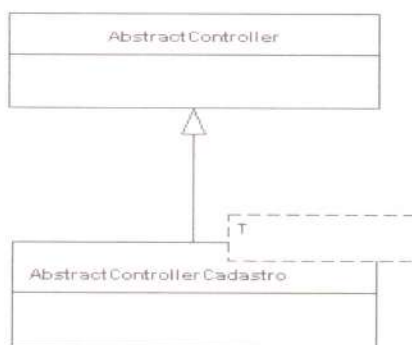


Figura 31 - Pacote br.com.natalsoft.arq.jsf
Fonte: Dados da Pesquisa (2007)

6.2.2 CAMADA DE PERSISTÊNCIA

6.2.2.1 DAO

O padrão de projeto DAO abstrai a camada de acesso aos dados dos desenvolvedores. No projeto foi utilizado o *framework* Hibernate para fazer acesso aos dados. Esse *framework* faz o mapeamento objeto-relacional. Como foi utilizado o padrão DAO, esse *framework* poderá ser substituído futuramente por outro sem causar um impacto muito grande nos casos de uso.

Segue abaixo um breve resumo das classes utilizadas na camada de persistência do projeto Hércules e seu respectivo diagrama de classes.

1. **Database** – Fábrica de sessions. No projeto há duas sessions: uma para o log da aplicação e outra para as entidades restantes. Essa classe utiliza os padrões *Singleton* e *FactoryMethod*.
2. **GenericDAO** – Interface genérica que todos os DAOs devem implementar.
3. **GenericDAOImpl** – Implementação da interface *GenericDAO*. Essa implementação depende do Hibernate.
4. **PagingInformation** - Classe usada para informações relativas a paginação.
5. **LogInterceptor** - Classe utilizada para ser um *interceptor* do Hibernate e logar as operações de modificação no banco de dados.

A classe *LogInterceptor* implementa a interface *org.hibernate.Interceptor*. Essa classe inspeciona os eventos que ocorrem no framework Hibernate. Essa inspeção ocorre antes dos valores serem escritos e depois elas são lidas a partir do banco de dados. O *LogInterceptor* é criado na classe *Database* e associado à sessão de dados que está sendo criada ou utilizada no momento.

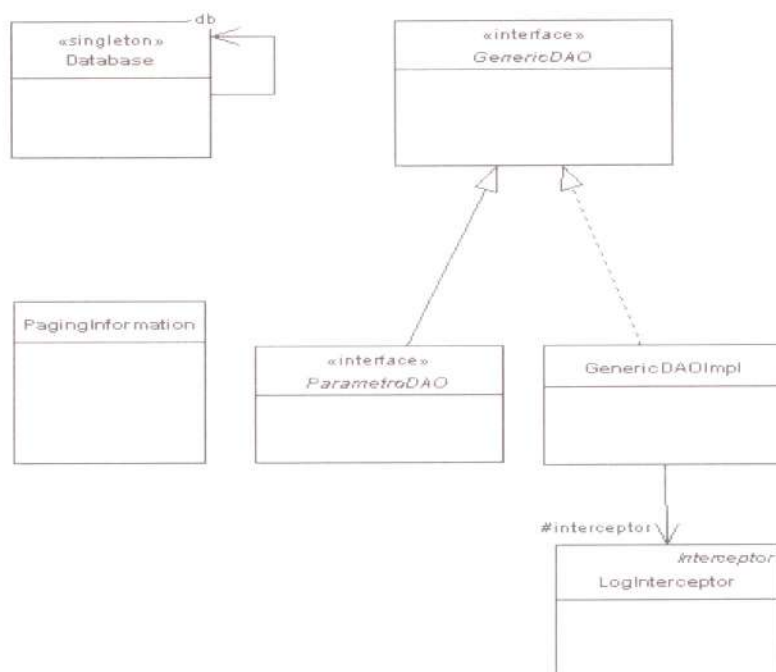


Figura 32 - Pacote: *br.com.natalsoft.arq.dao*
Fonte: Dados da Pesquisa (2007)

6.2.3 CAMADA DE NEGÓCIO

A camada de negócio fornece um conjunto de classes e interfaces que provê a padronização e infra-estrutura para efetivação das operações de negócio.

A arquitetura foi modelada usando o padrão de projeto *EJB Command*. Através desta padronização, todo caso de uso possui um código de comando que é executado por um processador. Os processadores são as classes responsáveis pela implementação da lógica de negócio. Todas elas devem ser *Enterprise JavaBeans*. Não é obrigatório que toda e qualquer lógica de negócio esteja contida em processadores, pois, em várias situações existem lógicas de negócio acessórias que fazem parte de uma ou várias operações de negócio. Nestes casos, a arquitetura recomenda a utilização de classes ajudantes (*Helper Classes*) para implementação dessas lógicas de negócio auxiliares.

Através do *EJB Command*, os clientes não sabem que classe irá executar a operação, somente o código do comando que é enviado juntamente com o ativador de comando (classe que herda *AbstractMovimento*). Para suportar a localização e ativação dos comandos, a arquitetura implementa uma fachada de ativação

Segue abaixo um breve resumo das classes utilizadas na camada de negócio do projeto Hércules e seu respectivo diagrama de classes.

1. **ProcessadorComando** - Interface que todo processador de comando deve ter.
2. **Abstract Processador** - Classe pai de todos os processadores do Sistema. Implementa funções de interesse de todos os processadores, como a autorização dos usuários e verificar se houveram erros de validação, por exemplo.
3. **ProcessadorComando** - Processador para as operações básicas de CRUD.
4. **SessionAdapter** - A classe SessionAdapter é uma implementação do design pattern Class Adapter que adapta a interface do EJB a Object. Será usado para tirar lógica de rede.
5. **ArqFacadeBean** - Implementa o Pattern Session Façade.
6. **MovimentoLocator** - Implementa os patterns Service Locator e Singleton. Esta classe recebe o código do comando e retorna o processador responsável pelo seu processamento.
7. **Processador** - Interface remota para o processador de Comandos.
8. **ProcessadorHome** - Interface Home para todos os processadores.
9. **FacadeDelegate** - Implementa o padrão de projeto Bussiness Delegate e é usada para ativar o Session Facade.
10. **ArqListaComando** - Possui a listagem de comandos da arquitetura.

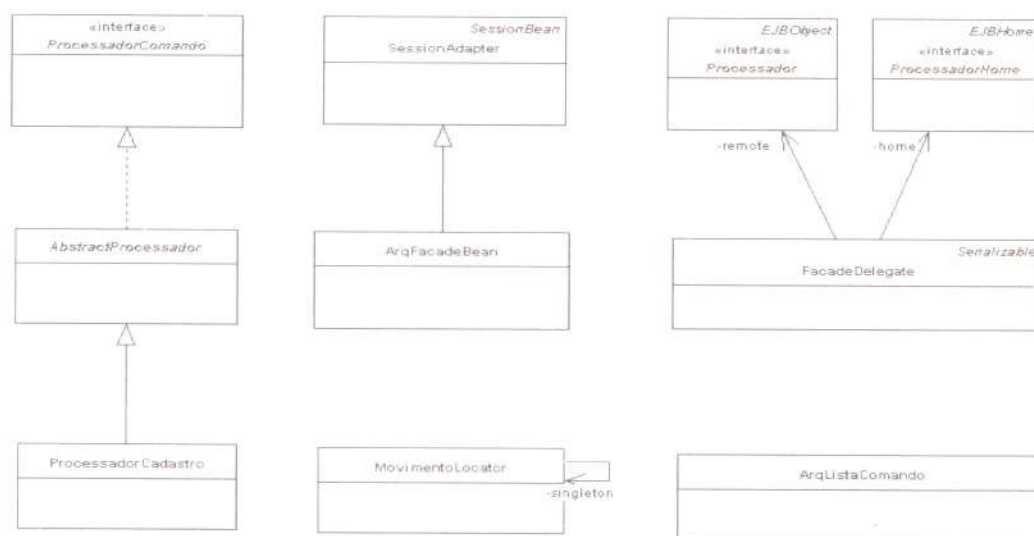


Figura 33 - Diagrama de classes do pacote br.com.natalsoft.arq.negocio
Fonte: Dados da Pesquisa (2007)

O diagrama de sequência de execução dos comandos é mostrado no diagrama abaixo.

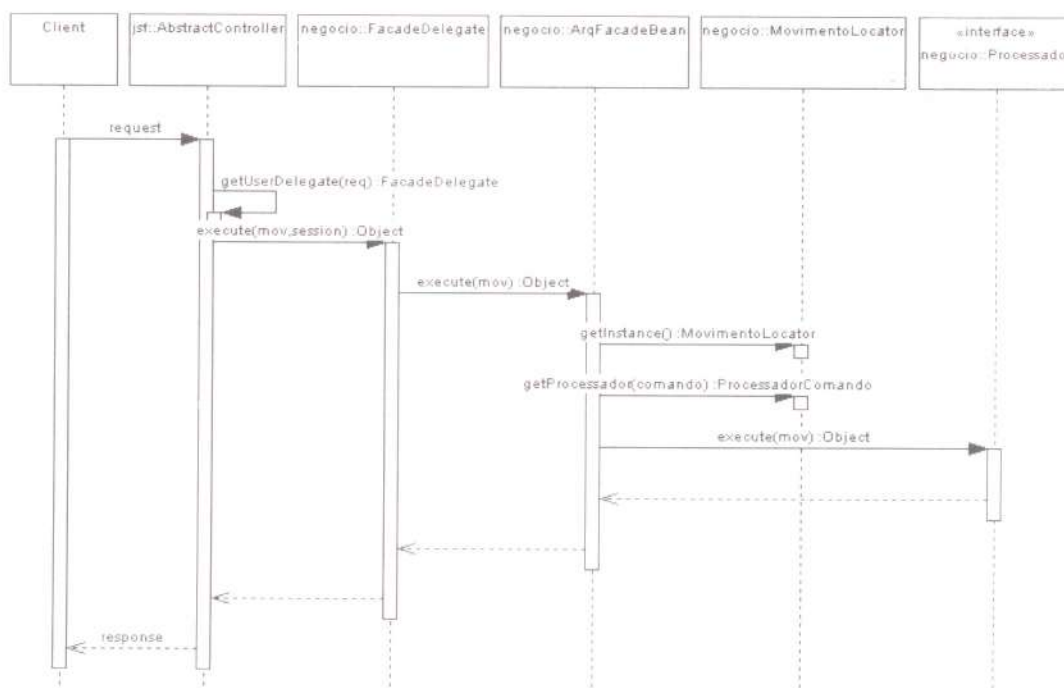


Figura 34 - Diagrama de sequência para execução de comandos
Fonte: Dados da Pesquisa (2007)

O cliente faz uma requisição e uma subclasse de *AbstractController* irá executar o comando solicitado. Para fazer isso ela inicialmente irá capturar ou criar o *FacadeDelegate* (padrão *Bussines Delegate*). O *FacadeDelegate* irá solicitar ao *ArqFacadeBean* (padrão *Session Facade*) que execute o comando. O *ArqFacadeBean* irá solicitar ao *MovimentoLocator* que

capture o processador responsável pela efetiva realização do comando. Após isso, *ArqFacadeBean* irá solicitar ao processador que execute o comando solicitado.

6.2.3.1 Tratamento de erros

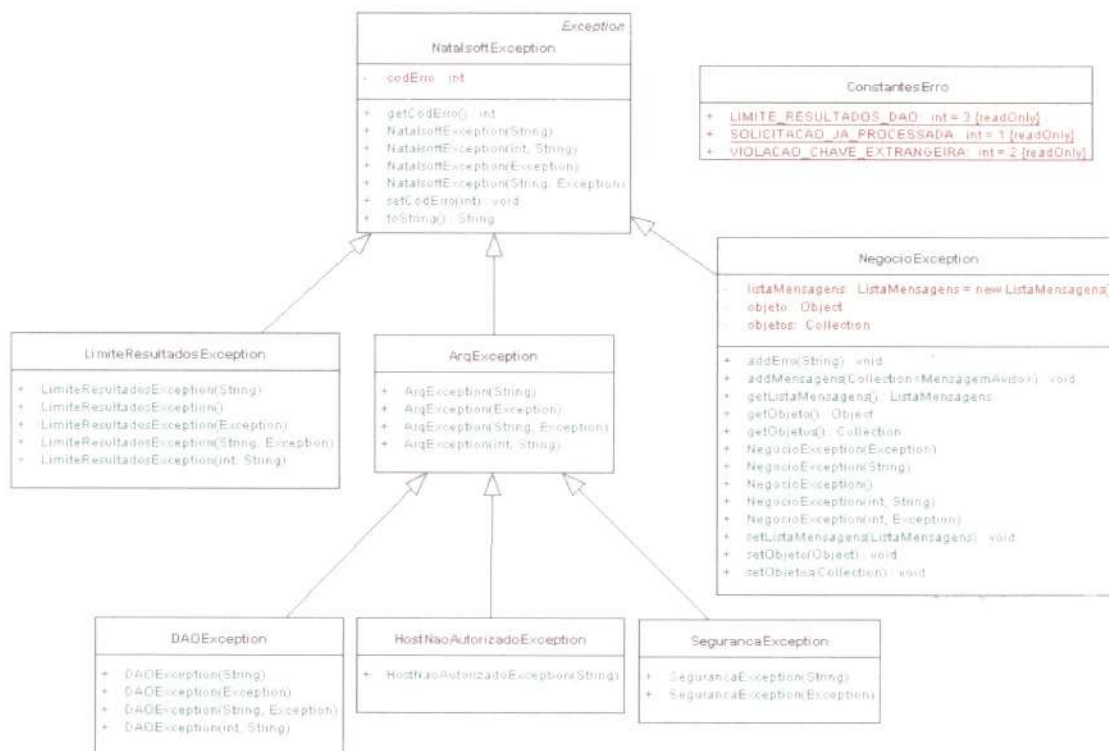


Figura 35 - Diagrama de classes do pacote br.com.natalsoft.arq.erros

Fonte: Dados da Pesquisa (2007)

O diagrama acima ilustra as exceções padronizadas na arquitetura do sistema Hércules. O *NatalsoftException* é a classe mãe de todas as exceções da arquitetura. Ela possui duas especializações: o *ArqException* que representa os erros gerados pela arquitetura e o *NegocioException*, que são erros de lógica de negócio.

Em muitas situações emitimos um tipo de exceção para diversos tipos de erro. Por exemplo, erros de negócio podem acontecer inúmeros, porém, apenas uma exceção é enviada. Nas situações, onde é desejável informar que tipo específico de erro a exceção carrega, usamos o atributo *codErro*. Este atributo deve, necessariamente, assumir um valor que consta na classe *ConstantesErro*.

A exceção da arquitetura possui três especializações: *SegurancaException*, emitida quando houver uma falha da autorização; *DAOException*, emitida quando houver falha no mapeamento objeto relacional e *HostNaoAutorizadoException*, emitida quando houver a tentativa de um host não autorizado acessar o sistema.

Na notificação de erros, algumas vezes, desejamos retornar um conjunto de informações mais elaboradas que uma simples mensagem ou código de erro. Por exemplo, o lançamento de uma ordem de serviço só deve ocorrer se todas as informações obrigatórias forem preenchidas e validadas. Caso alguma informação obrigatória não tenha sido informada ou validada, um *NegocioException* é emitido. No entanto, se desejarmos notificar ao cliente quais informações não foram informadas ou validadas, poderemos utilizar uma coleção de mensagens que está disponível na exceção.

7 CONCLUSÕES

A utilização de padrões de projetos no desenvolvimento WEB pode prover um framework que facilita e padroniza o desenvolvimento em uma equipe de trabalho. Os desenvolvedores não precisam ficar preocupados com todos os detalhes de implementação, pois a infraestrutura necessária está fornecida em uma arquitetura que utiliza os padrões de projeto, fazendo com que o desenvolvedor tenha mais tempo nas tarefas inerentes a aplicação.

Outra vantagem apresentada na utilização de padrões de projeto é o baixo acoplamento aliado à facilidade de utilização. Essa componentização fornece ao desenvolvedor um aumento de produtividade e facilidade de manutenção. Inclusive, a arquitetura pode ser reutilizada em outros projetos Java, diminuindo custos e também gerando aumento de produtividade.

A arquitetura ou o próprio sistema poderão ser extendidos quando houver a necessidade de acrescentar novas classes ou outras funcionalidades, mas deve-se, sempre que possível, adotar os padrões de projetos para que ocorra aumento de produtividade e facilidade de manutenção.

Os padrões GoF que foram apresentados nesse trabalho (*Factory Method*, *Singleton*, *Adapter*, *Decorator*, *Proxy*, *Template Method* e *Command*) foram bastante utilizados na implementação do sistema Hércules. Sempre que possível devem ser utilizados, pois fornecem soluções com baixo acoplamento e alta coesão, duas características importantes no desenvolvimento de software, pois facilitam o trabalho da manutenção do sistema.

Os padrões de projeto JEE que foram apresentados nesse trabalho contribuíram bastante na implementação do sistema Hércules, da seguinte forma:

Intercepting Filter: compactação das requisições e respostas, com isso o sistema torna-se mais rápido e utiliza uma largura de banda menor.

Front Controler: centralização das requisições. Todo o mecanismo de log das requisições e autenticação de usuários fica centralizado em uma única classe.

Composite View: torna mais fácil a manutenção das páginas .JSP.

Business Delegate, *Session Facade*, *Service Locator* e *EJB Command*: fazem com que o uso de EJBs fique mais fácil para fazer manutenção, mais rápido para execução e muito mais fácil de entender para novos desenvolvedores pois toda complexidade fica abstraída na arquitetura.

Data Access Object: torna o acesso ao banco de dados mais fácil e padronizado. Qualquer necessidade de mudar a tecnologia de acesso aos dados não terá um impacto muito grande por causa da adoção desse padrão.

Como se pode verificar são inúmeras as vantagens da utilização de padrões de projeto na arquitetura do sistema Hércules. Sem o uso dessa arquitetura haveria um custo bem maior no desenvolvimento e manutenção do sistema.

REFERÊNCIAS

AHMED, Khawar Zaman e UMRYSH, Cary E. **Desenvolvendo aplicações comerciais em Java com J2EE e UML**. Tradução Eveline Vieira Machado. Rio de Janeiro: Editora Ciência Moderna Ltda, 2002.

ALUR, Deepak; CRUPI, John; MALKS, Dan. **Core j2ee patterns**: as melhores práticas e estratégias de design. Tradução Altair Dias Caldas de Moraes, Cláudio Belleza Dias, Guilherme Dias Caldas Moraes. Rio de Janeiro: Campus, 2002.

ALEXANDER, Christopher. et al. **A Pattern Language**. New York: Oxford University Press, 1977.

GAMMA, E. , HELM, R., JOHNSON, R. , VLISSIDES, J. **Design Patterns**: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1998.

JAVA ServerFaces. In: WIKIPÉDIA. Disponível em:

<http://pt.wikipedia.org/wiki/JavaServer_Faces>. Acesso em 15 set. 2007.

JEEBrasil – O site da comunidade Java EE no Brasil. Disponível em:

<<http://www.jeebrasil.com.br/mostrar/76>>. Acesso em: 27 out. 2007.

INTERCEPTOR (Hibernate API Documentation). Disponível em:

<http://www.hibernate.org/hib_docs/v3/api/org/hibernate/Interceptor.html>. Acesso em: 02 nov. 2007.

MARINESCU, Floyd. **EJB design patterns**: advanced patterns, processes, and idioms. New York: John Wiley & Sons, 2002.

PADRÕES de projeto de software. In: WIKIPÉDIA. Disponível em:

<http://pt.wikipedia.org/wiki/Padr%C3%B5es_de_projeto_de_software>. Acesso em: 14 out. 2007.

SUN, Sun Microsystems. **Java EE at a Glance**. Disponível em: <<http://java.sun.com/javaee/>>. Acesso em: 18 jun. 2007.

SUN, Sun Microsystems. **Java Message Service API Overview**. Disponível em:

<<http://java.sun.com/products/jms/overview.html/>>. Acesso em: 21 out. 2007.