

LIGA DE ENSINO DO RIO GRANDE DO NORTE
FACULDADE NATALENSE PARA O DESENVOLVIMENTO DO RIO GRANDE DO NORTE
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

Ivan Bosnic

ANÁLISE DE PADRÕES DE PROJETOS CORPORATIVOS JEE APLICADOS A UM
SISTEMA DE GERENCIAMENTO DE TAREFAS

IVAN BOSNIC

ANÁLISE DE PADRÕES DE PROJETOS CORPORATIVOS JEE APLICADOS A UM SISTEMA DE GERENCIAMENTO DE TAREFAS

Monografia apresentada ao curso de Bacharelado em Sistemas de Informação da Faculdade Natalense para o Desenvolvimento do Rio Grande do Norte como requisito obrigatório para a obtenção do grau de Bacharel em Sistemas de Informação.

Orientador: prof. Gleydson Lima, M.Sc.

NATAL / 2007

Catálogo na Publicação – Biblioteca da FARN
Setor de Processos Técnicos

Bosnic, Ivan

Análise de padrões de projetos corporativos JEE aplicados a um sistema de gerenciamento de tarefas / Ivan Bosnic. – Natal: FARN, 2007.
52f.

Orientador: Gleydson de Azevedo Ferreira Lima (M. Sc.)
Monografia (Bacharelado em Sistemas de Informação) – Faculdade Natalense para o Desenvolvimento do Rio Grande do Norte.

1. Desenvolvimento de Softwares – Monografia. 2. Arquitetura JEE5 – Monografia. I. Título.

RN/FARN/BC

CDU 004.41

RB- 31461



004.1
5755a
2007
ex 01

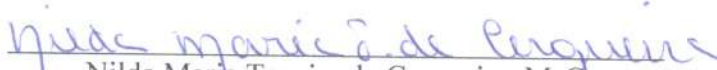
IVAN BOSNIC

ANÁLISE DE PADRÕES DE PROJETOS CORPORATIVOS JEE APLICADOS A UM
SISTEMA DE GERENCIAMENTO DE TAREFAS

Monografia apresentada ao curso de bacharelado em sistemas de informação da Faculdade
Natalense para o Desenvolvimento do Rio Grande do Norte e aprovada em ____ de ____
de ___, pela banca examinadora composta pelos seguintes membros:



Gleydson de Azevedo Ferreira Lima, MsC
Orientador



Nilda Maria Texeira de Cerqueira, MsC
Membro examinador



Aluizio Ferreira da Rocha Neto, MsC
Membro examinador

À minha mãe dedico este trabalho pelo apóio recebido durante toda a minha vida.

À minha esposa Eliene pelo apóio e compreensão constantes.

Às minhas filhas Laura e Lívia pela paciência em momentos da ausência.

SUMÁRIO

1 INTRODUÇÃO.....	1
1.1 PROBLEMA DE PESQUISA.....	4
1.2 OBJETIVOS.....	4
1.2.1 Objetivos gerais.....	4
1.2.2 Objetivos específicos.....	4
1.3 JUSTIFICATIVA.....	5
2 REFERÊNCIAL TEÓRICO.....	6
2.1 PADRÕES GOF.....	6
2.1.1 Abstract Factory.....	6
2.1.2 Adapter.....	7
2.1.3 Builder.....	8
2.1.4 Command	9
2.1.5 Facade.....	10
2.1.6 Factory Method.....	11
2.1.7 Iterator.....	12
2.1.8 Singleton	13
2.2 PADRÕES DE PROJETOS CORPORATIVOS JEE.....	14
2.2.1 EJB(Enterprise Java Beans)	15
2.2.2 Padrões de projetos da camada de apresentação.....	15
2.2.2.1 Intercepting Filter.....	15
2.2.2.2 Front Controller.....	17
2.2.2.3 Composite View.....	19
2.2.3 Padrões de projetos da camada de negócios.....	21
2.2.3.1 Business Delegate.....	21
2.2.3.2 Session Facade.....	23
2.2.3.3 Service Locator.....	25
2.2.4 Padrões de projetos da camada de persistência.....	26
2.2.4.1 Data Access Object.....	26

2.2.4.2 Service Activator.....	28
2.3 JAVA SERVER FACES (JSF).....	29
2.4 DEPENDENCY INJECTION	31
2.5 JPA (JAVA PERSISTENCE API).....	33
3 IMPLEMENTAÇÃO DOS PADRÕES DE PROJETO.....	35
3.1 DIAGRAMA ENTIDADE RELACIONAMENTO DO BANCO DE DADOS.....	35
3.2 IMPLEMENTAÇÃO DE PADRÕES DE PROJETO GOF.....	36
3.2.1 Implementação do padrão de projeto Singleton.....	37
3.2.2 Implementação do padrão de projeto Factory Method.....	37
3.2.3 Implementação do padrão de projeto Command.....	39
3.3 IMPLEMENTAÇÃO DE PADRÕES DE PROJETO DA CAMADA DE APRESENTAÇÃO.....	42
3.3.1 Implementação do padrão de projeto Front Controller.....	42
3.3.2 Implementação do padrão de projeto Composite View.....	43
3.4 IMPLEMENTAÇÃO DE PADRÕES DE PROJETO DA CAMADA DE NEGÓCIO.....	44
3.4.1 Implementação do padrão de projeto Business Delegate.....	44
3.5 IMPLEMENTAÇÃO DE PADRÕES DE PROJETO DA CAMADA DE PERSISTÊNCIA.....	46
3.5.1 Implementação da JPA (Java Persistence API)	46
3.5.2 Implementação do padrão de projeto DAO (Data Access Object).....	48
4 CONCLUSÃO.....	50
REFERÊNCIAS.....	52

LISTA DE FIGURAS

Figura 1 – Diagrama de classes do padrão Abstract Factory	7
Figura 2 – Diagrama de classes do padrão Class Adapter	7
Figura 3 – Diagrama de classes do padrão Object Adapter	8
Figura 4 – Diagrama de classes do padrão Builder	9
Figura 5 – Diagrama de classes do padrão Command	10
Figura 6 – Estrutura do padrão Facade	11
Figura 7 – Diagrama de classes do padrão Factory Method	12
Figura 8 – Diagrama de classes do padrão Iterator	13
Figura 9 – Diagrama de classes do padrão Class Adapter	14
Figura 10 – Arquitetura JEE.....	14
Figura 11 – Diagrama de classes do padrão Intercepting Filter.....	16
Figura 12 – Diagrama de seqüência do padrão Intercepting Filter.....	17
Figura 13 – Diagrama de classes do padrão Front Controller	18
Figura 14 – Diagrama de seqüência do padrão Front Controller.....	19
Figura 15 – Diagrama de classes do padrão Composite View	20
Figura 16 – Diagrama de seqüência do padrão Composite View.....	20
Figura 17 – Diagrama de classes do padrão Business Delegate	22
Figura 18 – Diagrama de sequencia do padrão Business Delegate.....	22
Figura 19 – Diagrama de classes do padrão Session Facade.....	24
Figura 20 – Diagrama de sequencia do padrão Intercepting Filter.....	24
Figura 21 – Diagrama de classes do padrão Service Locator	25
Figura 22 – Diagrama de sequencia do padrão Service Locator	26
Figura 23 – Diagrama de classes do padrão Data Access Object	27
Figura 24 – Diagrama de seqüência do padrão Data Access Object	27
Figura 25 – Diagrama de classes do padrão Service Activator	28
Figura 26 – Diagrama de seqüência do padrão Service Activator	29
Figura 27 – Visão de alto nível do <i>framework</i> JSF.....	30
Figura 28 – Ciclo de vida de JSF.....	31
Figura 29 – Dependências Configuradas sem Dependency Injection	32

Figura 30 – Dependências Configuradas com Dependency Injection	33
Figura 31 – Diagrama entidade relacionamento do projeto.....	36

LISTAGENS

Listagem 1 – Implementação do padrão Singleton.....	37
Listagem 2 – Classe DAOFactory.....	38
Listagem 3 – Criação de objeto para uso com XML	38
Listagem 4 – Criação de objeto para uso com banco de dados.....	38
Listagem 5 – Classe TarefaMovimento.....	39
Listagem 6 – Classe TarefaProcessador.....	40
Listagem 7 – Classe ListaComando.....	40
Listagem 8 – Classe Executor.....	41
Listagem 9 – Chamada do método atribuiTarefa.....	41
Listagem 10 – Arquivo web.xml.....	42
Listagem 11 – Implementação do padrão Business Delegate.....	44
Listagem 12 – Classe Tarefa.....	46
Listagem 13 – Obter todos os registros da tabela TIPOTAREFA.....	48

RESUMO

O surgimento dos padrões de projeto nos anos setenta e sua posterior aplicação no desenvolvimento de software tiveram uma grande colaboração na melhoria da qualidade do processo como todo. Ao longo dos anos, diversos padrões de projeto surgiram, e dentro da plataforma Java se consolidou um grupo de padrões de projeto voltados para aplicações corporativas. O presente trabalho faz uma análise desses padrões de projeto corporativos e a sua aplicação prática em um software-modelo que servirá de referência para aqueles que tiverem o interesse de aplicar essas tecnologias em seus projetos.

Palavras-chave: Padrão de projeto. Arquitetura de software. EJB. JPA. JEE. JSF.

ABSTRACT

Design patterns and its further applications to software development had a big collaboration in the quality improvement of the whole process. Over the years, various design patterns emerged, and inside the Java platform a group of patterns for enterprise applications had been consolidated. This study makes an analysis of these enterprise design patterns and its practical application in a model-software who might serve as a reference for those who have an interest in applying such technologies in their own projects.

Key-words: Design pattern. Software architecture. EJB. JPA. JEE. JSF.

1 INTRODUÇÃO

Por volta do ano 1977, um arquiteto austríaco residente nos Estados Unidos da América chamado Christopher Alexander percebeu que determinados problemas se repetiam a cada projeto arquitetônico que ele executava. Não existiam, no entanto, regras claras de como esses problemas deveriam ser tratados e solucionados. Pensando em aproveitar as experiências anteriores e utilizá-las não apenas nos seus futuros projetos, mas disponibilizá-las também para outros profissionais, ele resolveu documentá-las no seu livro "*A Pattern Language: Towns, Buildings, Construction*".

Desde então surgiu o conceito de *Pattern*, termo que referencia estruturas que podem ser utilizadas para solucionar problemas que ocorrem repetidamente. Apesar de este livro tratar de arquitetura e planejamento urbanos, as idéias que ele traz sobre padrões de projeto podem ser aplicadas em muitas outras áreas, inclusive na de desenvolvimento de software.

Assim como Christopher Alexander percebeu dentro da sua área de atuação, observa-se também que durante o processo de desenvolvimento de software é muito comum que certos problemas se repitam diversas vezes. E todas as vezes que eles ocorrem, cria-se uma solução única, sem o aproveitamento da experiência que foi adquirida em situações anteriores.

Foi com a idéia de padronizar as soluções para os problemas mais comuns que surgiram *Design Patterns* (padrões de projeto). Procura-se com a sua utilização evitar que os problemas comuns sejam tratados de forma diferenciada a cada vez que aparecem. Ao contrário, definem-se regras e soluções claras, previamente testadas e aprovadas por equipes de programadores e que posteriormente são disponibilizadas para a utilização por outros profissionais.

Segundo essa linha de raciocínio e aproximando-a mais para a realidade de análise e programação de sistemas, podemos definir *Design Patterns* como sendo "apenas maneiras convenientes de reutilização de código fonte orientado a objetos entre projetos e programadores". (COOPER, 1998, p.2)

A popularização de *Design Patterns*, no entanto, deu-se de uma forma mais significativa com a publicação do livro "*Design Patterns: Elements of Reusable Object-*

Oriented Software” escrito por Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides em 1992. Estes quatro autores eram conhecidos também como *Gang of Four*, expressão que deu origem a sigla *GoF*, muito utilizada quando se refere aos *Design Patterns* definidos por eles. 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 *Design Pattern* normalmente possui quatro elementos essenciais:

- **Nome do padrão** é a descrição que utilizamos para descrever o problema do padrão, sua solução e suas conseqüências em uma ou duas palavras. Nomeando um padrão logo no começo da sua concepção aumenta o nosso vocabulário e nos permite fazer a modelagem do projeto de um nível de abstração mais alto. Isso torna mais fácil pensar e falar sobre o padrão.
- **Problema** descreve quando o padrão deve ser aplicado. Explica o problema e o seu contexto. Pode descrever algum problema específico como, por exemplo, representar um algoritmo em forma de objeto. Às vezes o problema contém uma lista de condições que precisam ser satisfeitas antes que faça sentido aplicar o padrão em si.
- **Solução** deve conter os elementos que fazem parte do padrão, seus relacionamentos, responsabilidades e colaborações. A solução não descreve uma situação particular de projeto ou implementação porque um padrão de projeto pode e deve ser aplicado em diversas situações. Ao invés disso, procura prover uma descrição mais abstrata do problema e como uma organização dos elementos envolvidos pode resolvê-lo.
- **Conseqüências** são os resultados da aplicação do padrão. Elas são críticas quando avaliadas alternativas de outros *Design Patterns* e também para o entendimento da relação custo-benefício da utilização do padrão.

GoF Patterns trouxeram soluções para os problemas mais comuns da programação orientada a objetos e têm sido utilizados até hoje. Alguns desses padrões mais conhecidos são:

BUILDER

Separa a construção de objetos complexos da sua representação de tal forma que um mesmo processo de construção pode criar representações diferentes (GAMMA et al, 1998).

SINGLETON

Assegura que uma classe tenha apenas uma instância e provê um único ponto global de acesso a ela (GAMMA et al, 1998).

ABSTRACT FACTORY

Provê uma interface para criação de objetos relacionados ou dependentes sem especificação de suas classes concretas (GAMMA et al, 1998).

Ao todo, autores especificam e documentam 23 *Design Patterns* que foram, sem dúvida, de grande importância para a popularização desta abordagem. Porém, para a programação de softwares corporativos, esses padrões não são suficientes, pois não abrangem situações mais complexas que envolvem arquiteturas corporativas atuais, tais como:

- Qual a melhor solução para processar uma requisição do usuário?
- Como fazer uma boa comunicação entre a camada de visualização e a camada de negócios?
- De que forma tratar as transações que atingem diversos bancos de dados em uma requisição apenas?
- E diversos outros problemas que surgem dentro de uma arquitetura corporativa e que serão abordados ao longo deste trabalho.

A arquitetura *JEE (Java Enterprise Edition)*, que envolve diversas tecnologias como: *JSP, Servlets, EJB 3.0, JPA, JSF* entre outras, possui uma vasta coleção de *Design Patterns* que trazem soluções documentadas e testadas e cujo foco é exatamente o desenvolvimento de softwares corporativos.

Alguns dos *JEE design patterns* mais utilizados são:

SESSION FACADE

“... demonstra como separar de forma correta a lógica de negócios dentro do seu sistema para ajudar a minimizar a dependência entre cliente e servidor, enquanto força os casos de uso a executarem dentro de uma transação”. (MARINESCU, 2002, p. 3)

MESSAGE FACADE

“... define quando e como separar a lógica para casos de uso que são assíncronos por sua natureza” (MARINESCU, 2002, p. 3).

O presente trabalho analisa, além dos *Design Patterns* já consagrados dentro da arquitetura *Java*, alguns padrões mais recentes que surgiram com a versão *JEE5*. Esse estudo se dará através de um software-modelo que foi desenvolvido durante esta monografia e disponibilizado ao público de um modo geral sob a licença *GPL* (*General Public License*). O software em questão será uma demonstração de uso dos *patterns* estudados e das boas práticas de programação, mas por outro lado será também funcional para exercer o controle de gerenciamento de tarefas.

1.1 PROBLEMA DE PESQUISA

De que forma os *Design Patterns* de fato podem ajudar na melhoria do software desenvolvido? As soluções implementadas com a sua utilização são mais fáceis de manter e mais adequadas para os ambientes corporativos? Este trabalho pretende, através de estudo e implementação, responder a estas perguntas.

1.2 OBJETIVOS

1.2.1 Objetivos gerais

Análise, estudo e implementação de padrões de projetos *GoF* e *JEE* e das tecnologias *Java* mais recentes como *EJB 3.0*, *JPA*, *JSF*.

1.2.2 Objetivos específicos

Implementação de um software-modelo de gerenciamento de tarefas, de código fonte aberto e com distribuição gratuita utilizando *Design Patterns* da arquitetura *JEE5*.

1.3 JUSTIFICATIVA

Dentro do processo de desenvolvimento de software, muitas vezes os programadores e os analistas de sistemas só se lembram de *design patterns* no momento em que se deparam com um problema para o qual não sabem a solução. Essa abordagem não é a mais indicada, uma vez que não se preocupa com a reutilização de código fonte e também não procura aumentar a produtividade da equipe com a utilização de soluções previamente construídas e testadas.

A proposta de *design patterns* seria um caminho contrário ao citado em cima, ou seja, a preocupação com a modelagem e arquitetura do software deve fazer parte do processo desde a sua concepção e os padrões devem ser aplicados logo no começo. Dessa forma será possível aumentar a produtividade, desenvolver aplicações mais robustas e de fácil manutenção.

O presente trabalho de monografia pretende, além do estudo desses padrões, produzir um software-modelo que irá demonstrar de forma prática as vantagens desse tipo de arquitetura e dessa forma orientar os programadores iniciantes na maneira correta de construir as aplicações.

2 REFERENCIAL TEÓRICO

2.1 PADRÕES *GoF*

Desenvolver softwares orientados a objetos é difícil e desenvolver softwares orientados a objetos reutilizáveis é mais difícil ainda (GAMMA et al, 1998). *Design Patterns* tornam mais fácil a reutilização de projetos e arquiteturas que obtiveram sucesso. Iremos detalhar a seguir alguns dos padrões de projeto mais utilizados que foram documentados no livro *Design Patterns: Elements of Reusable Object-Oriented Software*.

2.1.1 *Abstract Factory*

Provê uma interface para criação de objetos relacionados ou dependentes sem especificação de suas classes concretas (GAMMA et al, 1998, p.77). Este padrão deve ser usado em seguintes situações:

- O sistema deve ser independente de como os seus produtos são criados, compostos e representados.
- O sistema deve ser configurado com uma de múltiplas famílias de produtos.
- A família de objetos relacionados é desenvolvida para ser usada em conjunto e precisa-se garantir essa regra.
- Deseja-se prover uma biblioteca de classes, porém deixando visíveis apenas as suas interfaces e não as suas implementações.

Algumas das conseqüências mais importantes da utilização deste padrão são:

- Isolamento de classes concretas. Isso acontece porque *Factory* encapsula a responsabilidade da criação de objetos.
- Promove consistência junto aos produtos. Quando os objetos de uma família são desenvolvidos para trabalharem juntos, é importante que a aplicação utilize os objetos da mesma família em cada momento.

Estrutura do Design Pattern *Abstract Factory* é apresentada na figura 1.

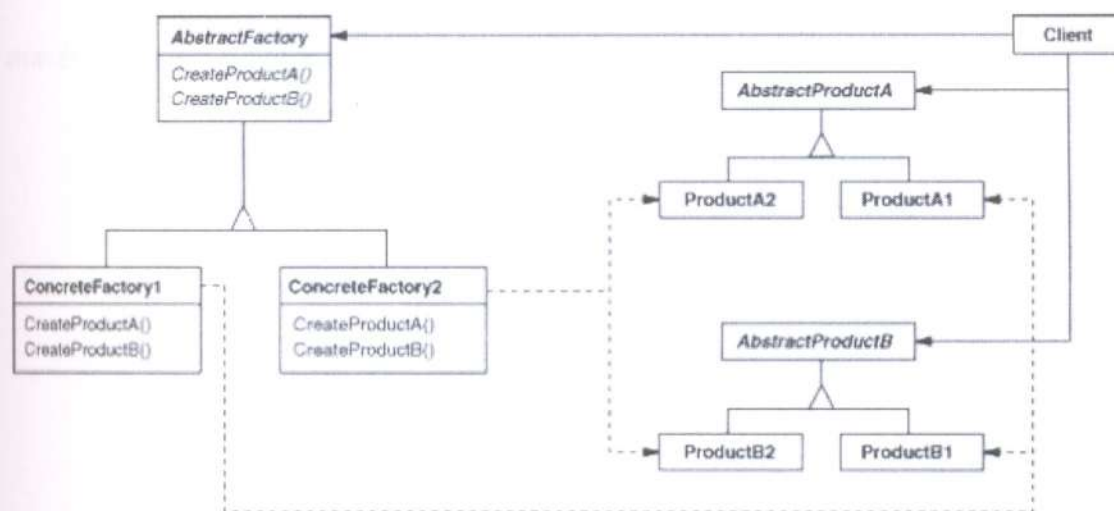


Figura 1 – Diagrama de classes do padrão Abstract Factory

Fonte: GAMMA et al (1998, p.77)

2.1.2 Adapter

Converte a interface de uma determinada classe em uma outra interface que o cliente espera. *Adapter* permite que as classes trabalhem juntas, o que não seria possível de outra forma por causa da incompatibilidade das interfaces (GAMMA et al, 1998, p.102). Deve-se utilizar quando:

- Deseja-se utilizar uma classe, porém a sua interface não é a que se necessita.
- Quando se precisa criar uma classe reutilizável que coopera com outras classes que nem necessariamente tem interfaces compatíveis.

Classe *Adapter* utiliza herança múltipla para adaptar uma interface à outra e a sua estrutura é apresentada na figura 2:

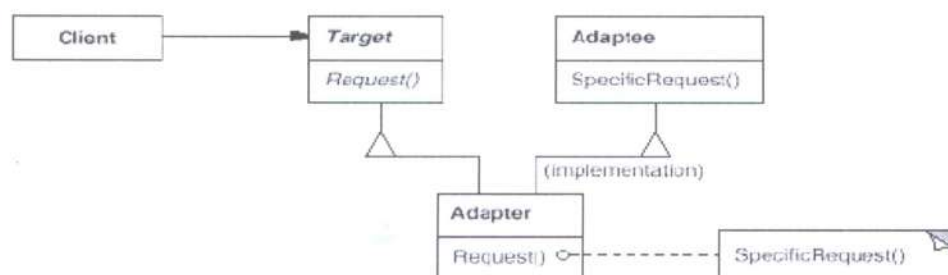


Figura 2 – Diagrama de classes do padrão Class Adapter

Fonte: GAMMA et al (1998, p.122)

Object Adapter se baseia na composição de objetos e a sua estrutura é apresentada através do diagrama a seguir:

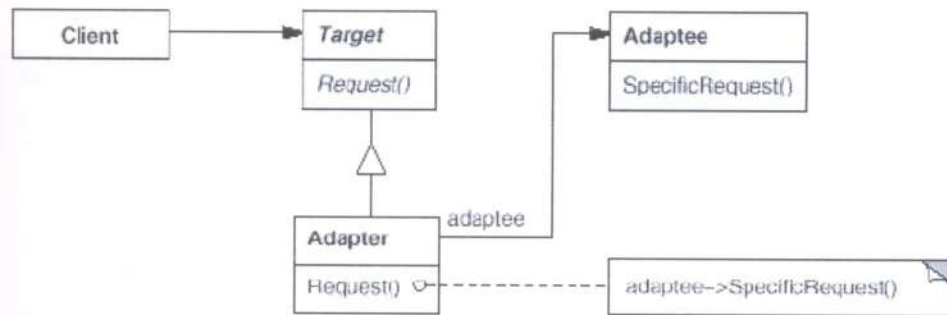


Figura 3 – Diagrama de classes do padrão Object Adapter

Fonte: GAMMA et al (1998, p.122)

2.1.3 Builder

Separa a construção de objetos complexos da sua representação de tal forma que um mesmo processo de construção pode criar representações diferentes (GAMMA et al, 1998, p.85).

Deve ser utilizado em uma das situações a seguir:

- O algoritmo para a criação de objetos complexos deve ser independente das partes que compõem o objeto em si e da forma com elas são juntadas.
- O processo de construção deve permitir representações diferentes para o objeto que é construído.

Conseqüências:

- Permite variar a representação interna do produto uma vez que o Builder provê o diretor com uma interface abstrata para construção de objetos.
- Isola o código fonte da construção e representação.
- Permite um controle mais refinado sobre o processo de construção.

A sua estrutura é representada através da figura 4.

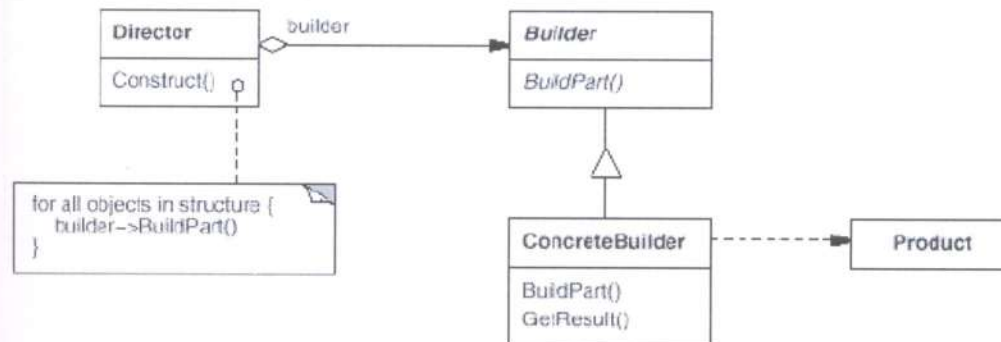


Figura 4 – Diagrama de classes do padrão Builder

Fonte: GAMMA et al (1998, p.86)

2.1.4 Command

Encapsula uma requisição como sendo um objeto, permitindo dessa forma parametrizar clientes com diferentes requisições, pilhas ou requisições de log (GAMMA et al, 1998, p.198).

Deve ser utilizado em seguintes situações:

- Quando se deseja parametrizar um objeto com uma ação que o mesmo deve executar.
- Para implementar recurso 'Desfazer' no software desenvolvido. Objeto *command* pode manter o estado para reverter os seus efeitos.
- Quando se deseja manter o log de atividades do sistema para que as mesmas possam ser refeitas em caso de falha.

Conseqüências:

- Faz o desacoplamento do objeto que invoca a operação daquele que sabe como executá-la.
- *Commands* são objetos de classes independentes. Eles podem ser manipulados e estendidos como quaisquer outros objetos.

A sua estrutura é representada através da figura 5:

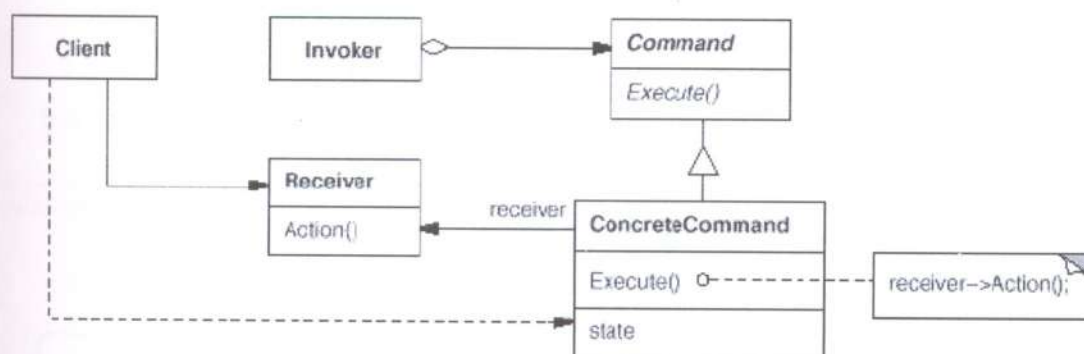


Figura 5 – Diagrama de classes do padrão Command

Fonte: GAMMA et al (1998, p.200)

2.1.5 Facade

Provê uma interface unificada para um conjunto de interfaces dentro de um subsistema. Facade define uma interface de alto nível que facilita a utilização do subsistema (GAMMA et al, 1998, p.157).

Deve ser usado para:

- Prover uma interface simples para um subsistema complexo. Muitos subsistemas se tornam mais complexos à medida que evoluem. Por outro lado, a maioria dos *Design Patterns* quando aplicado resultam em classes menores e mais fáceis de usar. Dessa forma os subsistemas se tornam mais reutilizáveis e fáceis de customizar.
- Abstrair subsistemas. Deve-se então definir um ponto de entrada para cada nível do subsistema.

Conseqüências:

- Abstrai os componentes do subsistema para os clientes, reduzindo número de objetos com os quais clientes têm que lidar.
- Promove um fraco acoplamento entre o subsistema e seus clientes. Isso permite variar os componentes do subsistema sem afetar os clientes.
- Por outro lado não impossibilita a aplicação de usar classes do subsistema se isso se mostrar necessário.

Estrutura do padrão é mostrada na figura 6.

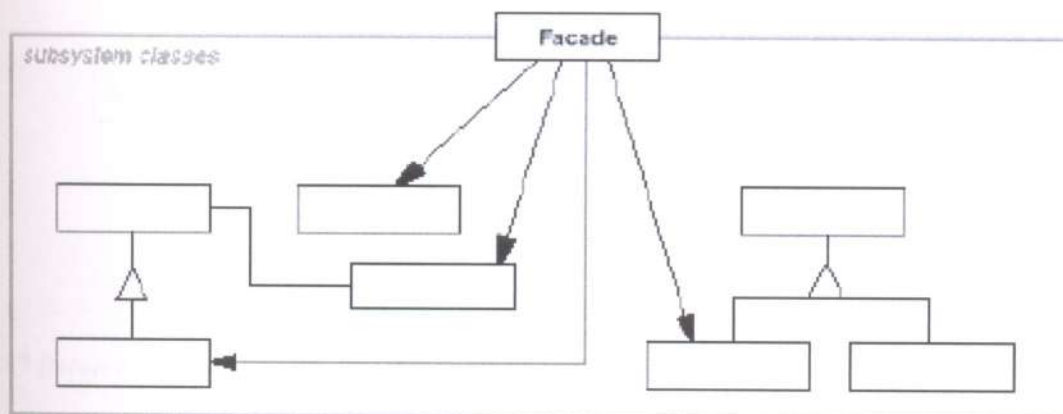


Figura 6 – Estrutura do padrão Facade

Fonte: GAMMA et al (1998, p.157)

2.1.6 Factory Method

Define uma interface para a criação de um objeto, porém deixa para as subclasses decidirem qual classe devem instanciar. *Factory Method* permite à classe repassar instanciamento para a subclasse (GAMMA et al, 1998, p.93).

Deve ser utilizado em uma das situações a seguir:

- Quando uma classe não pode antecipar a classe do objeto que a mesma deve criar.
- Quando a classe deseja que suas subclasses especifiquem o objeto que ela cria.

Conseqüências:

- Elimina a necessidade de criar classes específicas para uma determinada aplicação. O código fonte lida apenas com interfaces e dessa forma pode funcionar com quaisquer classes concretas que implantem essas interfaces.

A sua estrutura é representada através do diagrama da figura 7.

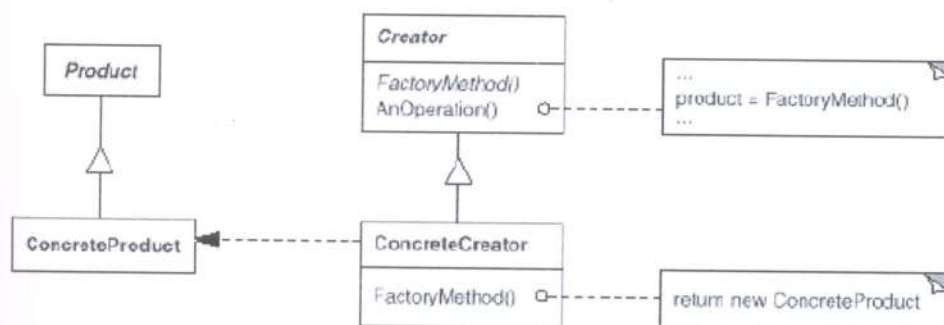


Figura 7 – Diagrama de classes do padrão Factory Method

Fonte: GAMMA et al (1998, p.94)

2.1.7 Iterator

Provê a maneira de acessar sequencialmente elementos de um objeto agregado sem expor a sua representação (GAMMA et al, 1998, p.219).

Deve ser utilizado:

- Para acessar o conteúdo de um objeto agregado sem expor sua representação interna.
- Para suportar múltiplas *traversals* de objetos agregados.
- Para prover uma interface uniforme para percorrer diferentes estruturas agregadas, ou seja, para suportar iteração através de polimorfismo.

Conseqüências:

- Permite variações de formas como deve ser percorrido um objeto agregado. No caso de uma estrutura de árvore, a mesma pode ser percorrida de diversas formas. Utilizando o padrão *Iterator* fica fácil resolver esse problema uma vez que basta substituir a instância do mesmo.
- Simplifica interfaces agregadas.
- Pode percorrer uma coleção diversas vezes ao mesmo tempo porque ele mantém posição do seu próprio estado.

A sua estrutura é representada pela figura 8.

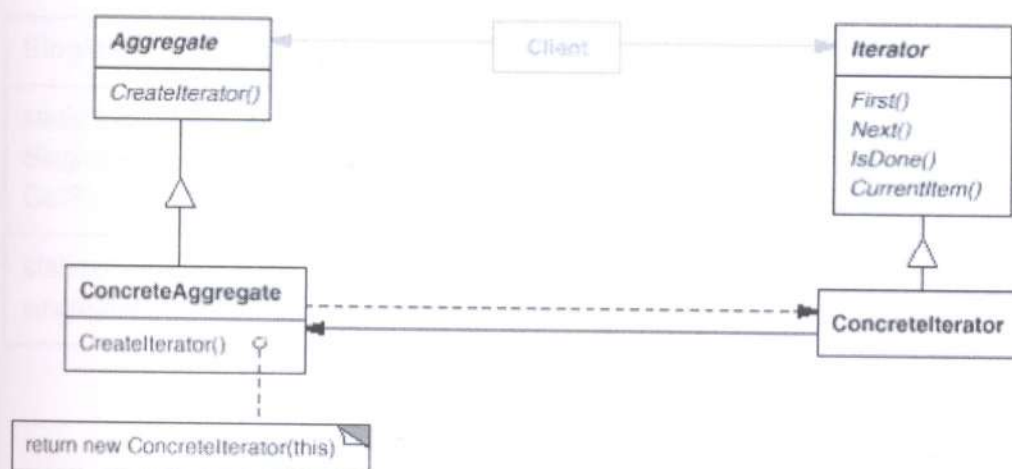


Figura 8 – Diagrama de classes do padrão Iterator

Fonte: GAMMA et al (1998, p.221)

2.1.8 Singleton

Assegura que uma classe tenha apenas uma instância e provê um único ponto global de acesso a ela (GAMMA et al, 1998, p.109).

Deve ser utilizado quando:

- For necessário garantir que haja exatamente uma instância de uma classe e a mesma deve estar acessível para o cliente de pontos de acesso bem conhecido.

Conseqüências:

- Acesso controlado a uma instância única. Como *Singleton* encapsula a única instância da classe é possível ter um controle total sobre como e quando os clientes a acessam.
- Reduz *namespace*. *Singleton* é na verdade uma substituição das variáveis globais.
- Permite um número variado de instâncias. Este padrão pode ser facilmente alterado para permitir mais do que uma instância de uma classe, assim podemos controlar quantas instâncias a aplicação pode utilizar.

A sua estrutura é representada pelo diagrama a seguir:

2.2.1 EJB (Enterprise Java Beans)

São classes Java de negócio que executam dentro de um contêiner EJB, normalmente contido dentro de um servidor de aplicativos. Conhecemos dois tipos de EJB: *Session Beans* e *Entity Beans*.

Session Beans são EJB especializados em processamento ou *workflow*. Eles podem ser *stateful* e *stateless*. *Stateful session beans* mantêm o estado do cliente entre as chamadas de métodos, enquanto *stateless session beans* não mantêm o estado do cliente. Estes últimos são usados quando não há necessidade de se manter o estado entre invocação de métodos e dessa forma sua performance costuma ser melhor. Cada *session bean* pertence a um usuário único e eles não são compartilhados.

Entity beans são utilizados quando um componente de negócios precisa ser persistido e compartilhado entre vários usuários. (ROMAN et al, 2005)

2.2.2 Padrões de projetos da camada de apresentação

Esta camada interage com usuário e mostra as informações vindas do sistema. A plataforma *JEE* suporta diferentes tipos de cliente tais como: clientes HTML, Java *Applets* e aplicações Java *desktop*.

Ela é responsável também por gerar a lógica de apresentação e por aceitar respostas de clientes, tipicamente clientes HTML, Java *Applets* e outros clientes *Web*. Baseado na requisição recebida é gerada então uma resposta adequada e enviada ao cliente. Na plataforma *JEE*, *Servlets* e *JSP* implementam esta camada dentro de um contêiner apropriado (JENDROCK et al, 2006).

2.2.2.1 Intercepting Filter

Este *Design Pattern* trata de um problema muito comum em aplicações corporativas que é pré-processamento e pós-processamento de requisições vindas de um cliente Web.

Quando uma requisição dessas é recebida, é muito comum que tenhamos que validar alguma das questões a seguir:

- Cliente foi autenticado?
- A sessão do cliente está válida?
- Endereço IP do cliente é de uma rede confiável?
- Software navegador do cliente é suportado pela aplicação?
- E diversas outras perguntas que podem surgir neste momento...

Segundo o livro *Core J2EE Patterns*, a solução para este problema se dá através de:

Criar filtros para processar serviços comuns de uma maneira padronizada sem a necessidade de mudar o código central de processamento de requisições. Os filtros interceptam as requisições que entram e respostas que saem, permitindo assim pré-processamento e pós-processamento. Nós somos capazes de adicionar ou remover estes filtros sem necessidade de mudança do nosso código já existente. (ALUR et al, 1999, p. 146).

A figura 11 representa a estrutura deste padrão de projeto.

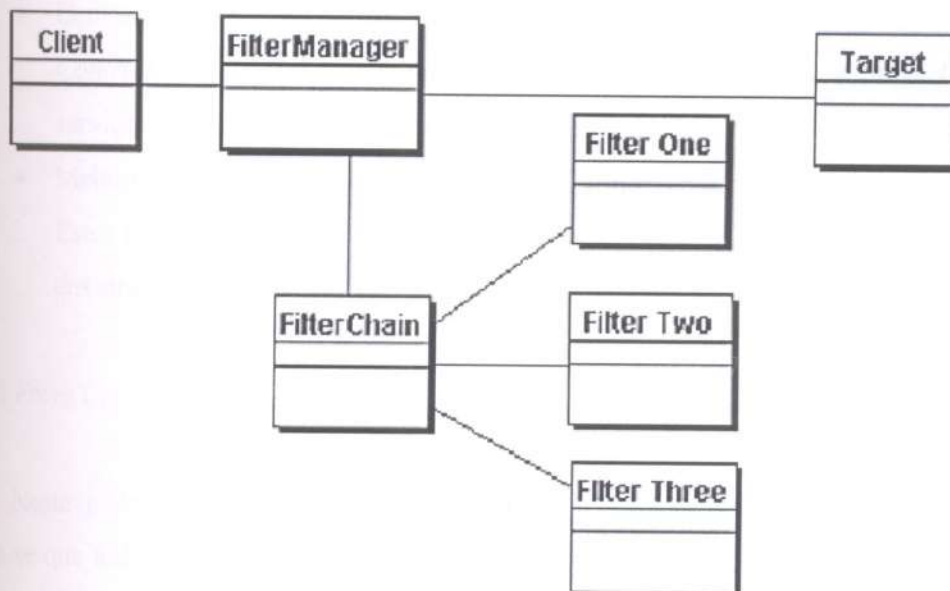


Figura 11 – Diagrama de classes do padrão Intercepting Filter

Fonte: ALUR et al (1999).

O diagrama de seqüência do padrão é apresentado na figura 12.

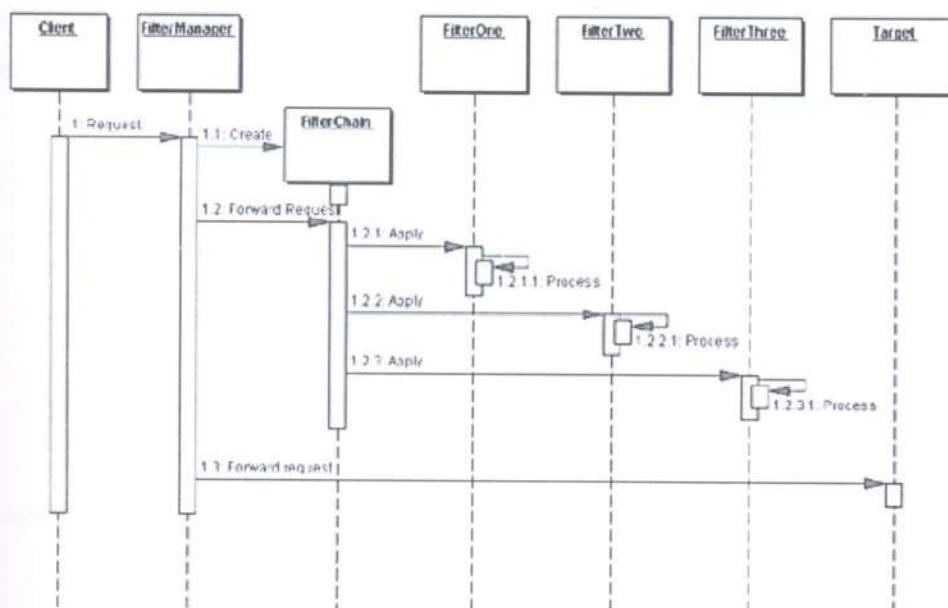


Figura 12 – Diagrama de seqüência do padrão Intercepting Filter

Fonte: ALUR et al (1999).

Conseqüências:

- Centraliza o controle de processamento de requisições, da mesma forma que um *Controller* o faria só que com a vantagem de ser fracamente acoplado a outros serviços, o que o torna mais flexível.
- Melhora reusabilidade e provê um particionamento mais claro da aplicação. Estes filtros são facilmente adicionados ou removidos da aplicação e por causa das suas interfaces padronizadas podem ser usados em quaisquer combinações.

2.2.2.2 Front Controller

Neste padrão trata-se de uma centralização de requisições vindas dos clientes. Sugere-se que todas elas devam passar, em primeiro lugar, por um mecanismo de controle para que sejam evitados alguns problemas comuns que poderiam ocorrer caso o cliente pudesse acessar as visualizações da aplicação diretamente. Os dois problemas mais comuns que ocorrem nessas situações são:

- Cada visualização deve possuir seus próprios serviços, o que geralmente resulta em código duplicado

- O controle da navegação fica a cargo de cada visualização o que pode resultar em redirecionamento inadequado e falho.

A solução adequada que este padrão atinge é:

Usar o controlador como o ponto de contato inicial para tratar a requisição. O controlador gerencia o processamento da requisição, incluindo as chamadas de serviços como os de autorização e autenticação, delegando processamento de negócios, gerenciando a escolha de uma visualização adequada, tratando erros e coordenando a seleção das estratégias de criação de conteúdo. (ALUR et al, 1999, p. 164).

O diagrama de classes a mostrado na figura 13 representa a estrutura do padrão.

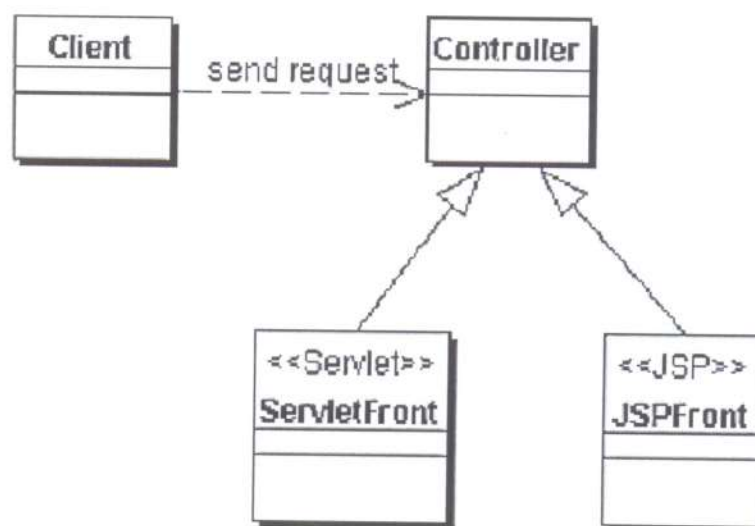


Figura 13 – Diagrama de classes do padrão Front Controller

Fonte: ALUR et al (1999).

O diagrama de sequência do padrão é apresentado na figura 14.

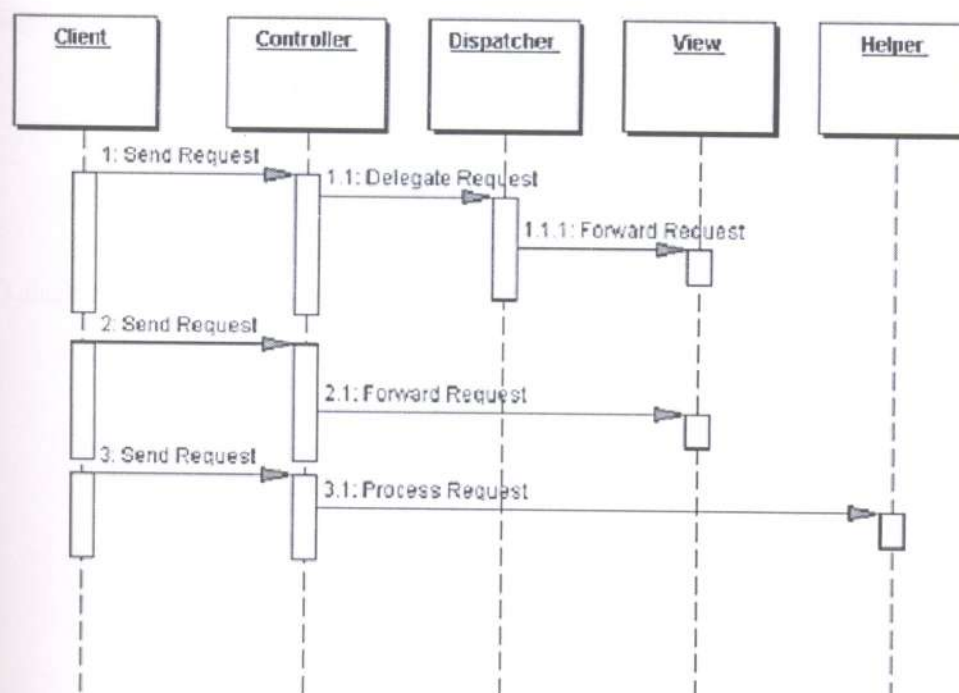


Figura 14 – Diagrama de seqüência do padrão Front Controller

Fonte: ALUR et al (1999).

Conseqüências:

- Controlador provê um mecanismo centralizado de processamento de requisições dos clientes. Como todos os acessos à aplicação passar por esse ponto, torna-se mais fácil implementar log de requisições, por exemplo.
- Melhora o gerenciamento de segurança da aplicação como um todo porque fica mais fácil programar certos recursos em um único ponto do que fazê-lo repetidamente em todas as visualizações.

2.2.2.3 Composite View

Este padrão surge como uma solução para a situação em que uma visualização é composta por outras visualizações, por exemplo, uma página Web que inclui várias outras páginas. Sugere-se que, ao invés de **prover** um mecanismo que combine partes de diferentes visualizações em uma única visualização, as mesmas sejam construídas embutindo-se código já formatado em cada uma delas.

Segundo o livro *Core J2EE Patterns*, a solução para este problema se dá através de:

Utilizar *composite views* compostas de múltiplas visualizações atômicas. Cada componente do padrão pode ser incluído dinamicamente no todo e o layout da visualização pode ser gerenciado independentemente do conteúdo. (ALUR et al, 1999, p. 164).

O diagrama de classes do *Composite View* é apresentado na figura 15.

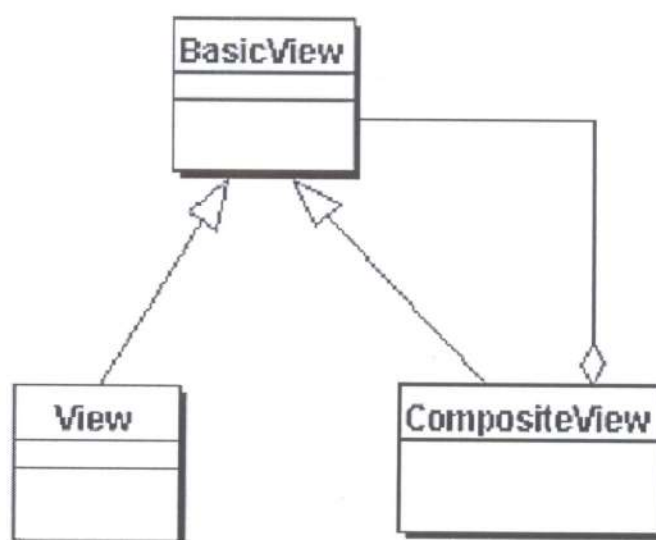


Figura 15 – Diagrama de classes do padrão Composite View

Fonte: ALUR et al (1999).

O diagrama de sequência é demonstrado na figura 16.

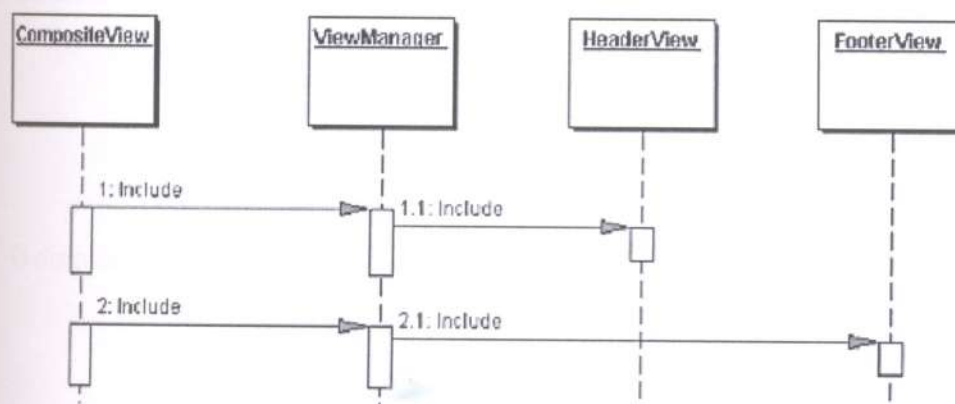


Figura 16 – Diagrama de sequência do padrão Composite View

Fonte: ALUR et al (1999).

Conseqüências:

- Melhora modularidade e reusabilidade uma vez que o padrão em si promove modelagem modulada da aplicação.
- Melhora a flexibilidade.

2.2.3 Padrões de projetos da camada de negócios

Esta camada é responsável por lidar com a lógica de negócios central da aplicação. Camada de negócios provê interfaces necessárias para abstrair componentes de serviços de negócios. Esses componentes são normalmente implementados como EJB (*Enterprise Java Beans*) que por sua vez são executados em um contêiner EJB que facilita o controle de ciclo de vida desses componentes e gerencia a persistência, transações e alocação de recursos.

2.2.3.1 *Business Delegate*

Dentro da arquitetura de um sistema corporativo, componentes da camada de visualização interagem diretamente com os componentes da camada de negócios. Em sistemas mais complexos isso expõe detalhes de implementação da segunda em relação à primeira, o que faz com que a camada de visualização se torne muito vulnerável às mudanças que venham a ocorrer na camada de negócios.

Utilizando o *Design Pattern Business Delegate*, a solução se obtém da seguinte forma:

Utilizar *Business Delegate* para reduzir o acoplamento entre clientes da camada de visualização e serviços de negócios. *Business Delegate* esconde os detalhes de implementação de serviços de negócios, tais como procura e acesso à arquitetura EJB. (ALUR et al, 1999, p. 238).

O diagrama de classe da figura 17 representa a sua estrutura.

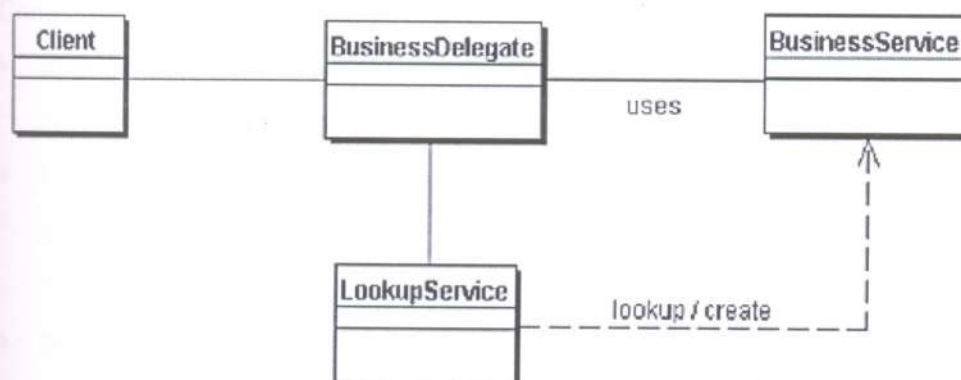


Figura 17 – Diagrama de classes do padrão Business Delegate

Fonte: ALUR et al (1999).

O diagrama de seqüência é representado na figura 18.

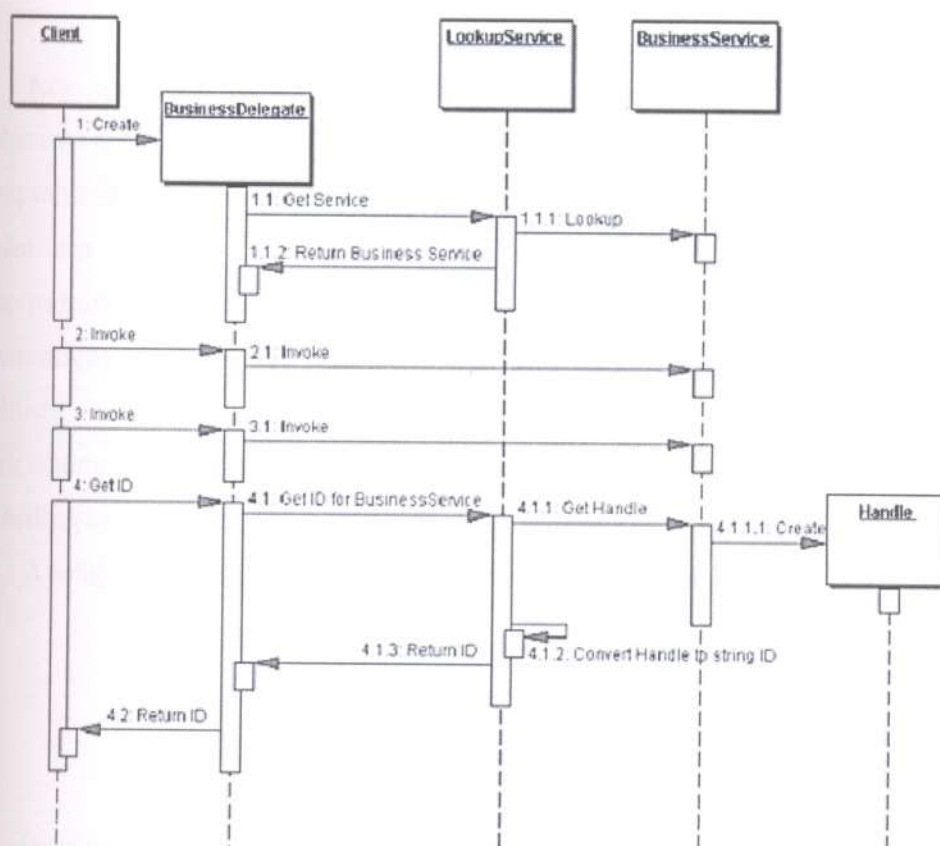


Figura 18 – Diagrama de seqüência do padrão Business Delegate

Fonte: ALUR et al (1999).

Conseqüências:

- Reduz o acoplamento entre camadas de visualização e negócios escondendo os detalhes de implementação da segunda.
- Este padrão é responsável por traduzir quaisquer problemas de infra-estrutura ou de rede em exceções de negócios, escondendo dessa forma dos clientes os detalhes da implementação.
- Melhora o desempenho, pois pode criar serviços de *cache* para as requisições mais comuns dos clientes.
- Em sistemas menos complexos, representa adição de uma camada muitas vezes desnecessária.

2.2.3.2 Session Facade

Nos sistemas multicamada normalmente ocorre uma alta dependência e acoplamento entre objetos de cliente e de negócios. Isso por sua vez provoca problemas de desempenho da rede, pois há demasiadamente muitas chamadas de métodos remotos. Há também uma tendência de uso errôneo de objetos de negócios uma vez que não há um acesso padronizado do lado do cliente. O cliente se torna cada vez mais dependente da implementação de objetos de negócios. A idéia por trás deste padrão tende exatamente a simplificar a interface de negócios para os clientes escondendo a sua complexidade e a reduzir o número de objetos da camada de negócios que são diretamente expostos à camada de visualização.

A solução para estes problemas se dá através do *design pattern Session Facade*:

Utilizar *session bean* como uma fachada para encapsular a complexidade de interação entre objetos de negócios que participam do processo. *Session facade* gerencia objetos de negócios e provê um acesso uniforme para as camadas de clientes. (ALUR et al, 1999).

O diagrama de classes da figura 19 representa a sua estrutura.

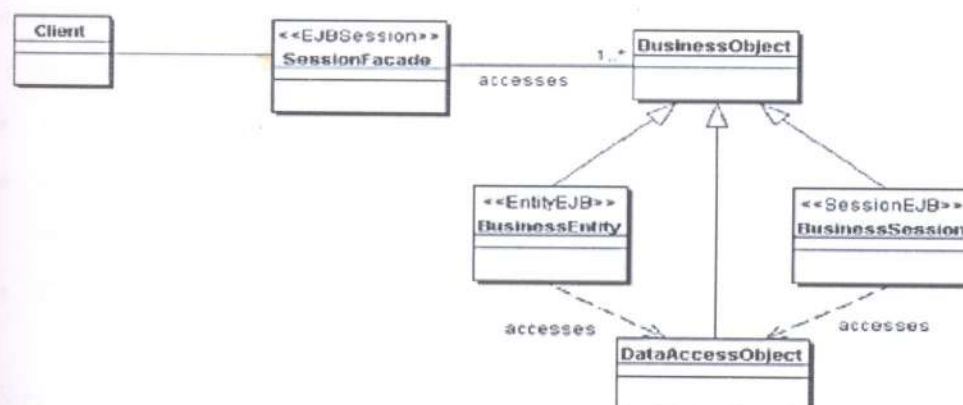


Figura 19 – Diagrama de classes do padrão Session Facade

Fonte: ALUR et al (1999).

O seu diagrama de sequência é representado na figura 20.

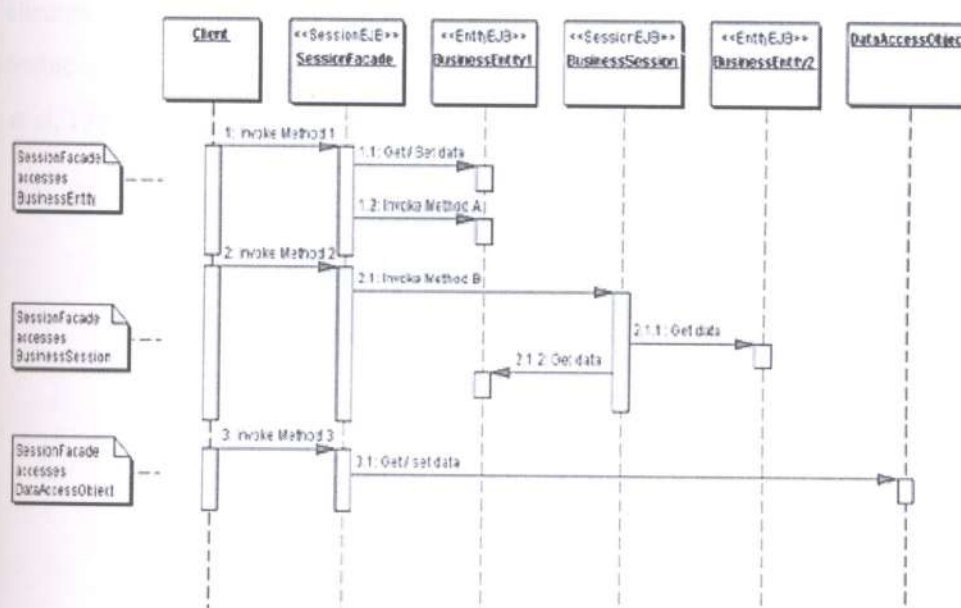


Figura 20 – Diagrama de sequência do padrão Intercepting Filter

Fonte: ALUR et al (1999).

Consequências:

- Introduz o controlador da camada de negócios. Ele simplifica o acesso de clientes aos objetos de negócios.

- Define uma interface uniforme que por sua vez esconde a complexidade das regras de negócios.
- Provê o desacoplamento das camadas de visualização e de negócio.
- Melhora o desempenho do sistema por reduzir o número de chamada de métodos remotos pela rede.
- Centraliza o gerenciamento de segurança e controle de transações.

2.2.3.3 Service Locator

Dentro de uma aplicação *JEE* clientes interagem com componentes *EJB* remotos e também com outros componentes que provêem diversos serviços. Para que essa interação seja possível, cliente precisa de algum mecanismo para localizar o componente ou para criar um novo. Esta operação é conhecida como *lookup operation*. Dentro da arquitetura *JEE*, clientes utilizam *JNDI API* para obterem o contexto inicial que contém o nome do componente que se busca. Todo esse processo pode se tornar bastante complexo e segundo (Alur et al, 1999) o padrão *Service Locator* o soluciona da seguinte forma:

Utilize o *Service Locator* para abstrair toda a utilização de *JNDI* e para esconder toda a complexidade da criação inicial de contexto, busca de objetos *EJB* e recriação de objetos *EJB*. Diversos clientes podem reutilizar um *Service Locator* para reduzir a complexidade do código fonte, prover um único ponto de controle e possibilitar a melhora da performance com utilização de técnicas de *cache*.

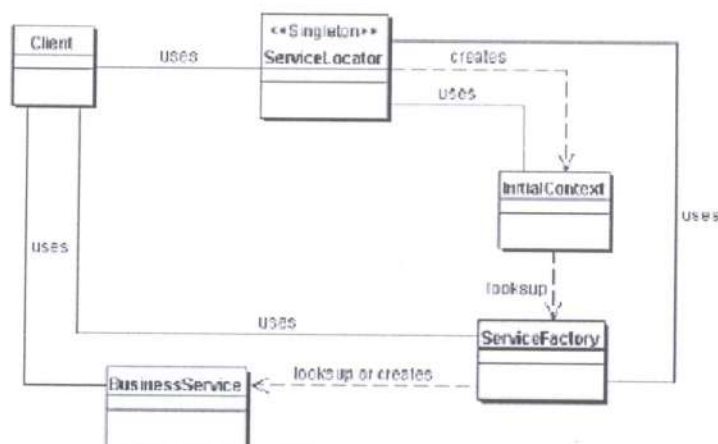


Figura 21 – Diagrama de classes do padrão Service Locator

Fonte: ALUR et al (1999).

Diagrama de sequência é mostrado na figura 22.

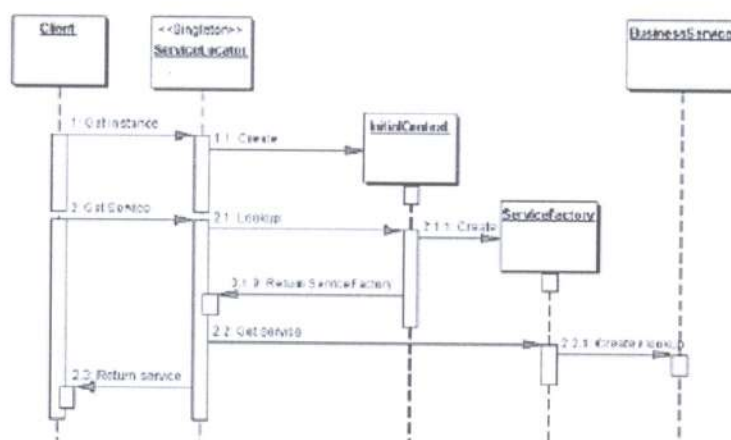


Figura 22 – Diagrama de sequência do padrão Service Locator

Fonte: ALUR et al (1999).

Conseqüências:

- Abstrai a complexidade da procura e criação de objetos e os mantém escondidos dos clientes.
- Provê um serviço de acessos uniforme para os clientes. Isso também facilita a adição de novos objetos na estrutura do sistema.

2.2.4 Padrões de projetos da camada de persistência

É a camada responsável por abstrair o acesso aos bancos de dados físicos. Somente ela é quem deve ter acesso e manipular os bancos de dados. Fazendo esta separação em camadas de um software corporativo, facilita-se a manutenção do mesmo, pois se consegue mudar a tecnologia utilizada em cada camada sem afetar o funcionamento das outras.

2.2.4.1 Data Access Object

A maioria das aplicações JEE, em algum determinado momento, precisa lidar com algum tipo de dados persistentes, como por exemplo: banco de dados relacional, banco de dados orientado a objetos, arquivos etc. Algumas vezes é necessário lidar com diversas fontes de dados dentro de uma mesma aplicação. Para obter acesso a esses dados,

implementações podem variar dependendo do tipo ou do fabricante do produto. É necessário, portanto, que se tenha um mecanismo padronizado de manipulação de dados. Para isso foi desenvolvido o padrão *Data Access Object* que, segundo (Alur et al, 1999) soluciona esse problema da seguinte forma: utilizar *Data Access Object* (DAO) para abstrair e encapsular todos os acessos a fontes de dados. *DAO* gerencia a conexão com fonte de dados para obter e armazenar dados.

Diagrama de classes do padrão é mostrado na figura 23.

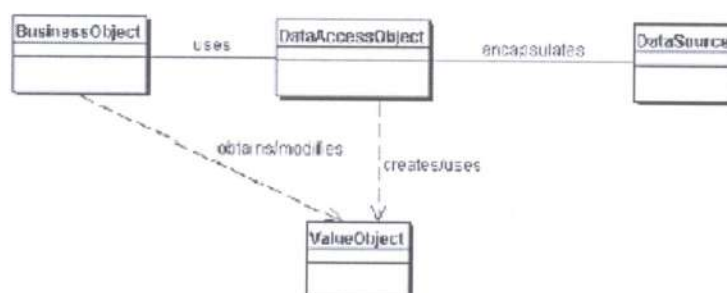


Figura 23 – Diagrama de classes do padrão Data Access Object

Fonte: ALUR et al (1999).

Diagrama de sequência é mostrado na figura 24.

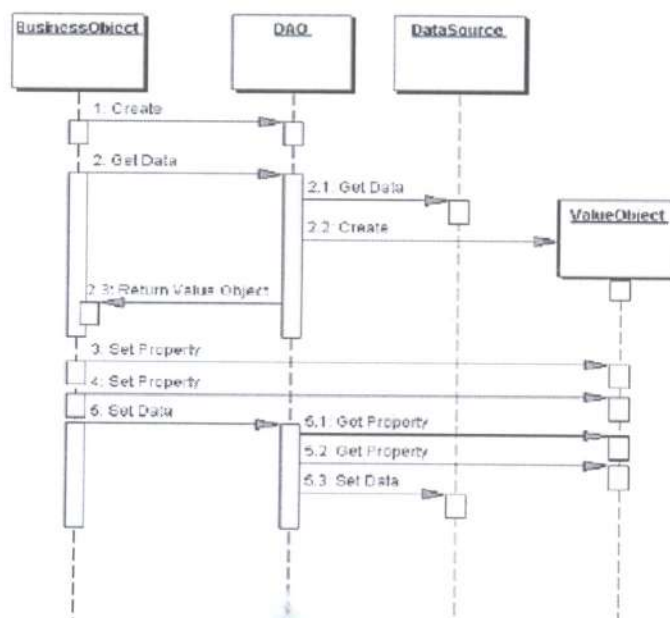


Figura 24 – Diagrama de classes do padrão Data Access Object

Fonte: ALUR et al (1999).

Conseqüências:

- Faz com que o acesso a dados se torne transparente para os objetos de negócios uma vez que os detalhes de acesso estão encapsulados nas classes *DAO*.
- Facilita migração para diferentes implementações de bancos de dados porque basta que apenas camada seja substituída.
- Reduz complexidade de código nos objetos de negócios.
- Centraliza todo o acesso a dados dentro de uma camada única.

2.2.4.2 Service Activator

Em algumas situações, clientes não necessitam esperar a resposta após a invocação de um método de um *EJB*. Em outras palavras, pode haver necessidade de uma comunicação assíncrona, algo que componentes *EJB* não implementam até a sua versão 2.0. O padrão *Service Locator*, segundo (Alur et al, 1999), deve ser utilizado para receber mensagens e requisições assíncronas dos clientes. Ao receber a mensagem, *Service Locator* localiza e invoca o método do componente de negócios para processar requisição assincronamente.

Diagrama de classes é mostrado na figura 25.

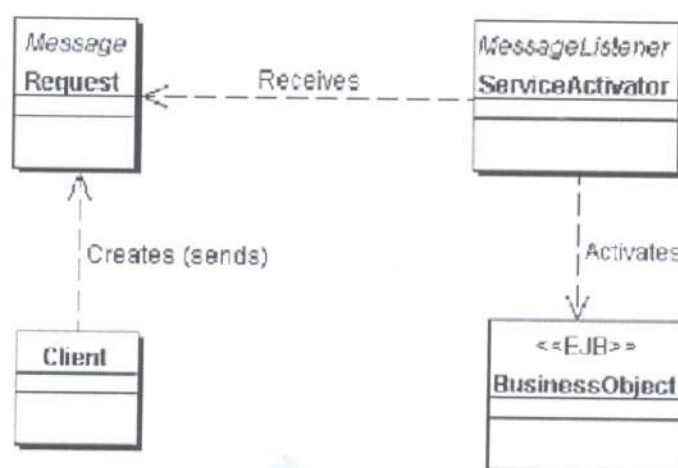


Figura 25 – Diagrama de classes do padrão Service Activator

Fonte: ALUR et al (1999).

Diagrama de seqüência é mostrado na figura 26.

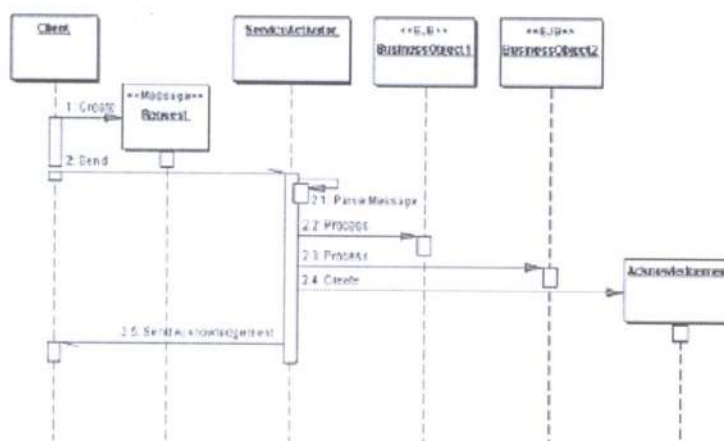


Figura 26 – Diagrama de classes do padrão Service Activator

Fonte: ALUR et al (1999).

Conseqüências:

- Provê processamento assíncrono para qualquer EJB.
- Pode ser executado como um processo independente, porém precisa ser monitorado para que sua disponibilidade seja garantida.

2.3 JAVA SERVER FACES (JSF)

Java Server Faces é um *framework* que implementa o *Design Pattern* MVC (*Model View Controller*) e que vem sendo adotado como o *framework* MVC oficial da tecnologia JEE. Para que possamos entender o seu funcionamento, é necessário antes definir arquitetura MVC. Aplicações corporativas possibilitam que os usuários manipulem diversos tipos de informação (dados) como, por exemplo, carrinhos de compras, aquisição de passagens aéreas ou qualquer outro tipo de dados dependendo do domínio que aplicação atinge. A esse tipo de dados chamamos de *model* (modelo). Numa aplicação web, HTML é utilizado para prover visualização dessas informações. JSF conecta essa *view* (visualização) ao modelo. Além disso, ele opera também como *controller* (controlador) que reage com o

usuário processando suas ações e eventos, ligando-os ao código que atualiza modelo ou visualização (HALL, 2004).

Dessa forma, *JSF* implementa arquitetura clássica MVC:

- **Conversão de dados.** Usuário entra com os dados na forma de texto, porém a classe de negócio precisa dessa informação no formato de número, data ou algum outro tipo.
- **Validação e tratamento de erros.** JSF facilita validação de informações entradas pelo usuário, campos obrigatórios e exibe mensagens de erros apropriadas, livrando dessa forma o programador de tarefas tediosas.
- **Internacionalização.** Framework trata por exemplo da codificação correta de caracteres nas páginas HTML.
- **Componentes customizados.** Desenvolvedores de componentes tem a possibilidade de programar componentes complexos de visualização que depois podem ser reutilizados pelos designers das páginas (HALL, 2004).

Figura 27 mostra uma visão de alto nível do framework JSF.

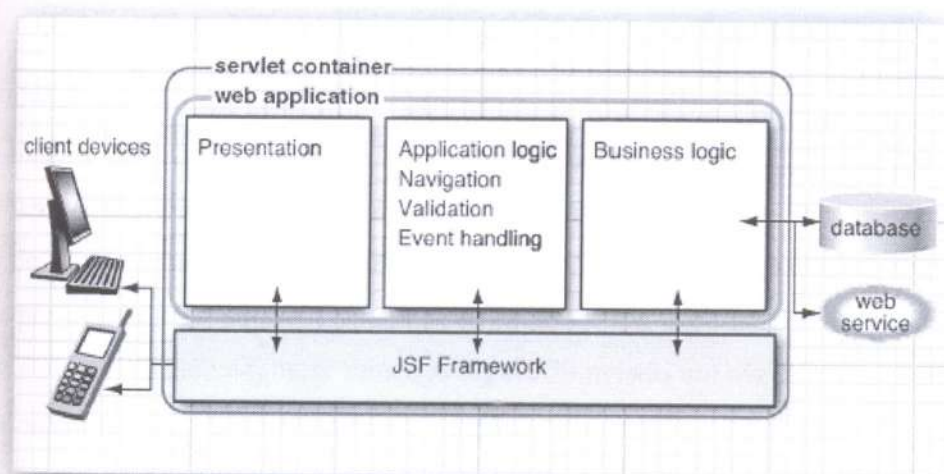


Figura 27 – Visão de alto nível do *framework JSF*

Fonte: HALL (2004, p.29).

E a figura 28 mostra o ciclo de vida de JSF.

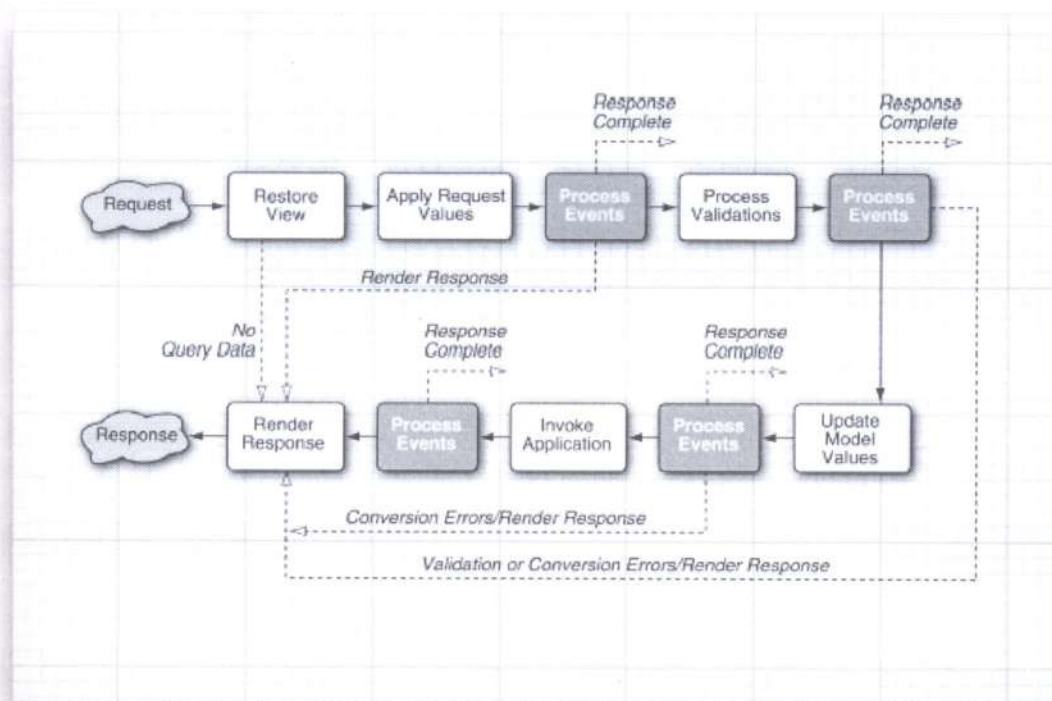


Figura 28 – Ciclo de vida de JSF

Fonte: HALL (2004, p.35).

2.4 DEPENDENCY INJECTION

O termo *Dependency Injection* é usado para classificar um tipo específico de inversão de controle onde a responsabilidade de configurar dependências entre classes é assumida por um terceiro objeto. Desta forma, atinge-se um acoplamento mais fraco entre as classes e por consequência maior reuso. A Figura 29 mostra um diagrama de classes com a classe *ConferenceFacade*, representando o *Facade* dos serviços da Conferência e a classe *MessageDAOImpl* e sua interface, que implementam o DAO responsável por efetuar as operações de persistência na base de dados (BARRETO, 2006).

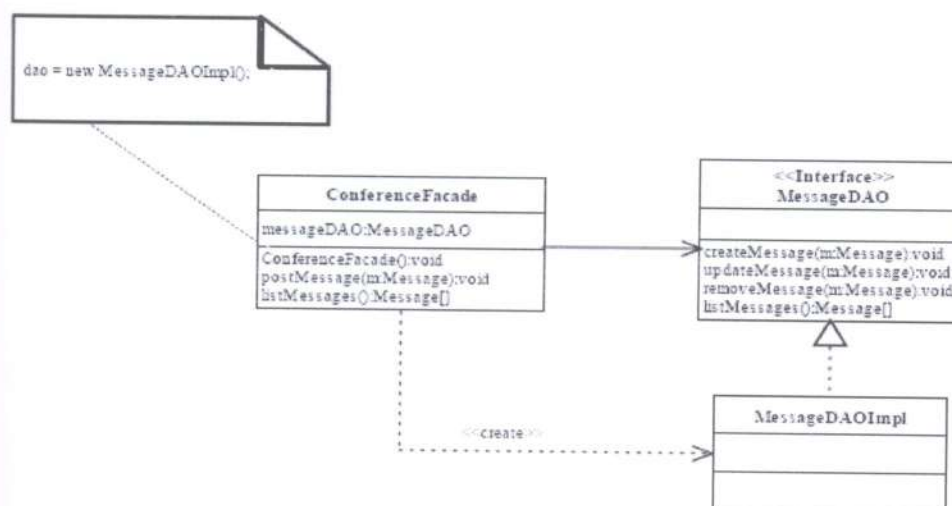


Figura 29 – Dependências Configuradas sem Dependency Injection

Fonte: BARRETO (2006, p.82)

Apesar da classe *ConferenceFacade* referenciar a interface *MessageDAO*, ela é dependente da implementação da interface pois precisa instanciá-la para poder usá-la. Através de *Dependency Injection*, um terceiro objeto denominado Montador (Assembler) é inserido. Este objeto é responsável por instanciar e configurar as dependências das classes relacionadas. A Figura 30 exemplifica o mesmo modelo da Figura 29, mas desta vez usando *Dependency Injection* (BARRETO, 2006).

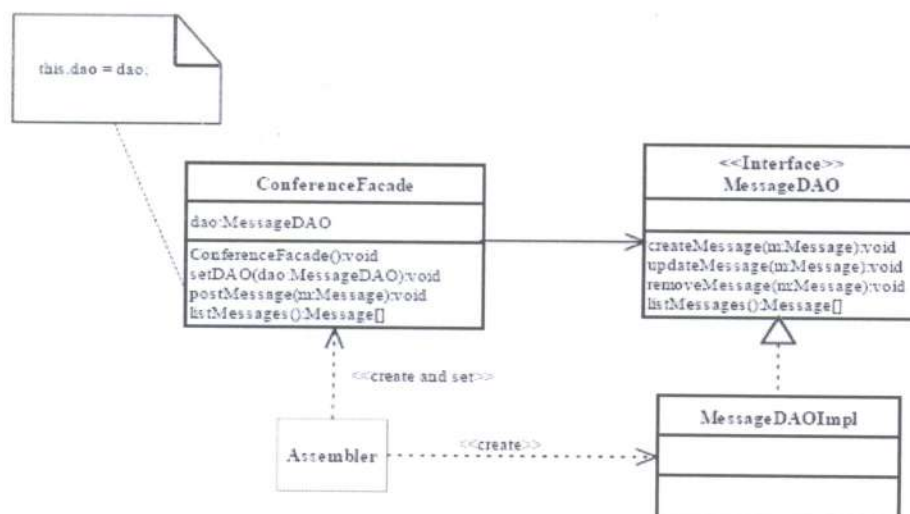


Figura 30 – Dependências Configuradas com Dependency Injection

Fonte: BARRETO (2006, p.82)

2.5 JPA (JAVA PERSISTENCE API)

A maior novidade na nova plataforma Java para desenvolvimento de aplicações corporativas, *Java Enterprise Edition 5 (JEE5)*, está na sua facilidade de uso. A maior mudança na tecnologia *EJB (Enterprise Java Beans)* foi a introdução de *JPA (Java Persistence API)*. *JPA* lida com a maneira como os dados relacionais são mapeados para objetos Java (entidades persistentes) e como esses objetos são guardados no banco de dados para que possam ser acessados posteriormente. De uma forma mais resumida, podemos dizer que *JPA* padroniza mapeamento objeto-relacional (MOR) na plataforma *JEE5*. (BISWAS e ORT, 2006)

Esta nova versão de *EJB 3.0* é mais fácil de se aprender do que a versão anterior 2.1, o que resulta numa maior produtividade na hora de desenvolver uma aplicação baseada nessa plataforma. Um dos recursos que possibilitou essas facilidades foi a introdução do conceito de *annotations* (anotações) que substituíram o uso excessivo de arquivos XML utilizados na versão anterior. Uma classe persistente agora pode ter a sua configuração contida no corpo da própria classe. (BISWAS e ORT, 2006)

As anotações são precedidas com sinal '@' e não afetam a lógica de programação contida na classe. Um exemplo de anotação seria '@Entity'. Esta anotação é utilizada

quando se quer definir uma classe como sendo uma classe persistente, ou seja, uma classe que representa uma entidade do banco de dados. Exemplo desse código é mostrado a seguir:

```
@Entity
@Table (name = "cliente")
public class Cliente {
    //código de implementação da classe
}
```

Neste exemplo define-se uma classe 'Cliente' que é associada a uma tabela 'cliente' no banco de dados. Além da anotação '@Entity', existem várias outras anotações que são utilizadas para definir todas as propriedades das tabelas assim como a forma como as mesmas se relacionam entre si.

Exemplos de outras anotações utilizadas pela JPA:

- @Id = define a chave primária da classe
- @Column = mapeia um atributo de classe a um campo no banco de dados
- @OneToMany = define um relacionamento do tipo Um-para-muitos
- @ManyToOne = define um relacionamento do tipo Muitos-para-um

3 IMPLEMENTAÇÃO DOS PADRÕES DE PROJETO

Conforme foi destacado na introdução desta monografia, o presente trabalho pretende mostrar a aplicação de alguns padrões de projeto através de um software de gerenciamento de tarefas. Os padrões aplicados foram divididos em grupos, de acordo com a classificação apresentada no referencial teórico deste trabalho: padrões de projeto GoF, padrões de projeto da camada de visualização, padrões de projeto da camada de negócio e padrões de projeto da camada de persistência.

Como se trata de um sistema que lida diretamente com o banco de dados, o qual não é objetivo do presente estudo, foi criado um banco de dados para ser utilizado com a aplicação. Apresenta-se a seguir o diagrama entidade relacionamento desse banco de dados para um melhor entendimento dos padrões mostrados mais à frente.

Nas seções seguintes são abordados os padrões de projeto utilizados, sendo mostrados exemplos de código fonte na linguagem de programação Java de como esses padrões foram utilizados na prática.

Considerando o tamanho deste projeto, que é pequeno se comparado às aplicações corporativas, alguns destes padrões de projeto podem parecer inadequados para esta situação. No entanto, o objetivo desta monografia é criar um software modelo que possa ser usado por outras pessoas como sendo um ponto de partida para a construção de sistemas de maior porte.

Cada padrão aplicado conterà uma explicação justificando a sua utilização e os benefícios que o mesmo traz para o desenvolvimento de sistemas corporativos. Além disso, serão apresentadas as partes de código fonte relevantes à sua implementação e, quando necessário, os diagramas mostrando a modelagem do projeto.

3.1 DIAGRAMA ENTIDADE RELACIONAMENTO DO BANCO DE DADOS

Apesar de não ser o foco de estudo desta monografia, é mostrado aqui o diagrama entidade relacionamento (Figura 31) para que os padrões de projetos demonstrados possam ser analisados e compreendidos.

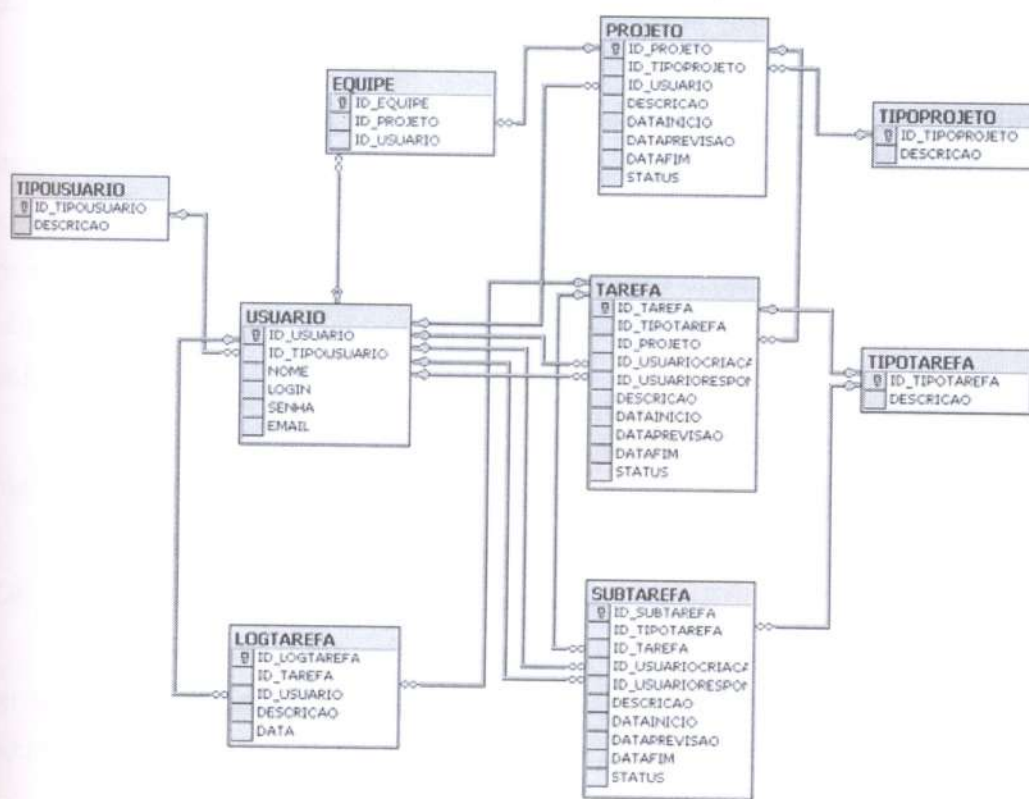


Figura 31 - Diagrama entidade relacionamento do projeto

3.2 IMPLEMENTAÇÃO DE PADRÕES DE PROJETO GoF

Os padrões de projeto GoF foram os primeiros a serem documentados. Embora eles não sejam padrões especificamente corporativos, a sua utilização é de grande importância em todo o tipo de projeto, uma vez que os mesmos formam uma base na qual muitos outros padrões são apoiados.

No projeto que foi desenvolvido como parte desta monografia, alguns desses padrões importantes foram utilizados, e a seguir será mostrado como os mesmos foram implementados.

3.2.1 Implementação do padrão de projeto *Singleton*

O padrão de projeto *Singleton* nos garante que dentro da aplicação existirá apenas uma única instância de um determinado objeto. Dentro da biblioteca *Java Persistence API*, a qual foi utilizada neste projeto na camada de persistência, a classe *EntityManager* é usada para gerenciar o acesso a banco de dados. Por questões de desempenho, é aconselhável que haja apenas uma única instância desta classe dentro do projeto. Para este fim foi utilizado o padrão de projeto *Singleton*.

A listagem 1 mostra uma parte do código da classe *Database.java* que implementa o padrão *Singleton*.

Listagem 1 – Implementação do padrão *Singleton*

```
private static EntityManagerFactory emf;  
public static EntityManager getEntityManager( )  
{  
    if ( emf == null )  
        emf = Persistence.createEntityManagerFactory("JProject-warPU");  
    return emf.createEntityManager();  
}
```

Com esta implementação, fica assegurado de que não será criada uma nova instância da classe *EntityManager*, caso a mesma já exista.

3.2.2 Implementação do padrão de projeto *Factory Method*

O padrão de projeto *Factory Method* define uma interface para a criação de um objeto, porém deixa para as subclasses decidirem qual classe devem instanciar. O código abaixo representa a classe *DAOFactory*, que é responsável por criar objetos DAO que podem ser usados com mais de um mecanismo de mapeamento, neste caso banco de dados relacional e arquivos XML. A fábrica entrega objetos diferentes de acordo com os parâmetros passados.

Listagem 2 – Classe DAOFactory

```
public class DAOFactory
{
    public GenericDAO getDAO (String classe, String tipo)
    {
        return Class.forName(classe + tipo).newInstance();
    }
}
```

Foi definido na especificação do sistema que, dependendo da tecnologia usada para armazenar os dados, seria usada a seguinte nomenclatura: [Nome da Classe][Tecnologia usada]. As classes resultantes seriam, por exemplo, UsuarioXML e UsuarioBD. Para criar um DAO referente à classe Usuario e com a utilização da tecnologia XML, pode ser usado o código da listagem 3.

Listagem 3 – Criação de objeto para uso com XML

```
GenericDAO d = DAOFactory.getDAO("Usuario", "XML");
d.create(obj);
```

A última linha do código mostrado persiste o objeto criado dentro de um arquivo XML. Se for usado um banco de dados relacional, o código para a criação do objeto permaneceria o mesmo e apenas a chamada da fábrica seria alterada, conforme a listagem 4.

Listagem 4 – Criação de objeto para uso com banco de dados

```
GenericDAO d = DAOFactory.getDAO("Usuario", "BD");
d.create(obj);
```

Com a utilização deste padrão conseguimos abstrair a criação de objetos DAO, permitindo que a tecnologia de armazenamento usada possa ser alterada de forma flexível, sem a necessidade de alteração do código fonte.

3.2.3 Implementação do padrão de projeto *Command*

O padrão de projeto *Command* tem o propósito de encapsular uma requisição como um objeto, permitindo que os clientes parametrizem diferentes requisições, filas ou fazer o registro de log de requisições e dar suporte a operações que podem ser desfeitas. Dentro do sistema utilizamos este padrão, por exemplo, para encapsular todas as operações feitas relacionadas à classe 'Tarefa'. Uma classe específica 'TarefaProcessador' desempenha este papel. Uma outra classe, a qual chamamos de 'Executor', é quem recebe todas as solicitações de execução de processos e, através dos parâmetros recebidos, encaminha essas requisições às classes responsáveis. A classe 'TarefaMovimento' é responsável por encapsular o tipo de operação que deve ser executada. Uma instância dessa classe é passada para a classe 'Executor'.

A listagem 5 mostra a estrutura da classe 'TarefaMovimento'.

Listagem 5 – Classe TarefaMovimento

```
public class TarefaMovimento implements Movimento
{
    private Tarefa tarefa;
    private String comando;
    ...
}
```

A listagem 6 mostra a estrutura da classe 'TarefaProcessador':

Listagem 6 – Classe TarefaProcessador

```

public class TarefaProcessador implements Processador
{
    public Object execute(Movimento mov)
    {
        TarefaMovimento tm = (TarefaMovimento) mov;
        if (tm.getComando().equals(ListaComando.ATRIBUI_TF)
        {
            atribuirTarefa(tm);
        }
        else if (tm.getComando().equals(ListaComando.CANCELA_TF)
        {
            cancelaTarefa(tm);
        }
    }
}

```

Uma classe 'ListaComando' é usada para definir todas as operações e as classes responsáveis por executar essas operações, conforme mostrado na listagem 7.

Listagem 7 – Classe ListaComando

```

public class ListaComando
{
    public static String ATRIBUI_TF = "negocio.TarefaProcessador";
    public static String CANCELA_TF = "negocio.TarefaProcessador";
    ...
}

```

A listagem 8 mostra a classe 'Executor':

Listagem 8 – Classe Executor

```
public class Executor
{
    public Object executarComando(Movimento mov)
    {
        Processador p = (Processador) Class.forName(mov.getComando()).newInstance();
        return p.execute(mov);
    }
}
```

Dentro do sistema, em lugares onde é preciso executar alguns processos específicos da classe ‘Tarefa’, basta instanciar um ‘Executor’ e definir os parâmetros adequados. Por exemplo, para chamarmos o método ‘atribuiTarefa’, podemos usar o código mostrado na listagem 9.

Listagem 9 – Chamada do método atribuiTarefa

```
Executor executor = new Executor( );
Movimento tm = new TarefaMovimento();
tm.setTarefa(tarefa);
tm.setComando(ListaComando. ATRIBUI_TAREFA);
executor.executarComando(tm);
```

Se quisermos tratar do controle de transações no nosso sistema de uma forma transparente, basta implementar a classe ‘Executor’ como um *EJB*. Dessa forma, qualquer processo iniciado a partir da classe ‘Executor’ (pelo fato de ele ser um *EJB*) se tornará automaticamente um processo transacional.

3.3 IMPLEMENTAÇÃO DE PADRÕES DE PROJETO DA CAMADA DE APRESENTAÇÃO

A camada de apresentação de um sistema é basicamente responsável por criar a interface gráfica para usuário, controlar os eventos disparados pelo usuário e outros. Uma separação clara desta camada facilita a manutenção futura do software e permite o acoplamento fraco do sistema, ou seja, dada a necessidade, torna-se mais fácil aplicar outras tecnologias que venham a surgir, uma vez que a mesma não lida diretamente, por exemplo, com acesso a banco de dados. A seguir serão apresentadas as implementações de alguns padrões de projeto que atuam na camada em questão.

3.3.1 Implementação do padrão de projeto *Front Controller*

Este padrão de projeto sugere que todas as requisições vindas dos clientes devem passar por um mecanismo de controle, para que o cliente não consiga acessar as páginas diretamente, antes que algumas validações e verificações sejam feitas. O *framework JavaServer Faces* possui recursos avançados que implementam este padrão. Dentro do arquivo `web.xml` é possível especificar qual a classe `servlet` será responsável por essa tarefa. A listagem 10 mostra a parte relevante do arquivo `web.xml`.

Listagem 10 – Arquivo `web.xml`

```
<servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>FacesController</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.jsf</url-pattern>
</servlet-mapping>
```

De acordo com as configurações definidas neste arquivo, a nossa aplicação se comportará da seguinte forma: uma classe 'FacesController' será responsável por receber todas as requisições vindas dos clientes e efetuar assim todo o pré-processamento necessários. Isso pode incluir algumas regras do sistema como, por exemplo, verificar se o usuário está logado, validar algumas permissões de segurança e outros. `<url-pattern>*.jsf</url-pattern>` define que todas páginas que terminam com .jsf, que neste caso são todas as páginas do sistema, serão tratadas por este controlador.

A aplicação deste padrão nos permite ter um controle centralizado de todas as requisições, melhorar a segurança do nosso sistema, implementar de forma mais fácil auditoria e monitoramento, etc.

3.3.2 Implementação do padrão de projeto *Composite View*

Este padrão vem para solucionar o problema de páginas web que são compostas por outras páginas. Em sistemas web corporativos, é muito comum que algumas partes da página sejam comuns a várias outras páginas como, por exemplo, cabeçalho, rodapé e menu da página.

Em páginas *JSP* isso pode ser alcançado através das diretivas de página:

```
<%@ include file="menu.jsp"%>
```

No entanto, se a camada de visualização está sendo gerada com a tecnologia *Servlet*, o padrão pode ser aplicado da seguinte forma:

```
public class clienteListServlet extends HttpServlet
{
    includePage("menu.jsp");
}
```

3.4 IMPLEMENTAÇÃO DE PADRÕES DE PROJETO DA CAMADA DE NEGÓCIO

Nesta camada acontece o processamento das regras de negócio do sistema corporativo. Diversos padrões de projeto atuam e podem ser aplicados nesta camada, sendo que um dos principais objetivos sempre é encapsular a execução das regras de negócio, prover o suporte para lidar com as transações, reforçar a reusabilidade dos componentes do sistema, etc.

3.4.1 Implementação do padrão de projeto *Business Delegate*

Este padrão reduz o acoplamento entre clientes da camada de visualização e serviços de negócio. Ele esconde os detalhes de implementação, tais como a procura e acesso à arquitetura *EJB*. A listagem 11 mostra a implementação do padrão.

Listagem 11 – Implementação do padrão *Business Delegate*

```
class SessionFacade implements SessionBean
{
    //métodos ejb
    //métodos de negócio
    void servico1()
    {
        //Implementação do serviço 1
    }
    void servico2()
    {
        //Implementação do serviço 2
    }
}
class BusinessDelegate
{
    SessionFacadeHome home;
    BusinessDelegate()
```

```

{
    Context ctx = new InitialContext();
    Object ref = ctx.lookup("SessionFacade");
    home = (SessionFacadeHome)PortableRemoteObject.narrow(ref,
        SessionFacadeHome.class);
}

void servico1()
{
    SessionFacade sf = home.create();
    sf.servico1();
}

void servico2()
{
    SessionFacade sf = home.create();
    sf.servico2();
}
}

class ClienteComBusinessDelegate
{
    BusinessDelegate bd;
    ClienteComBusinessDelegate()
    {
        bd = new BusinessDelegate();
    }
    void metodo()
    {
        bd.servico1();
    }
}

```

Usando o *Business Delegate*, o código cliente só enxerga invocações locais, e toda a complexidade das chamadas remotas fica oculta dentro dele. Além de facilitar o desenvolvimento da aplicação cliente, o uso do *Business Delegate* desacopla a camada de apresentação e a tecnologia da camada de negócio. Caso haja a necessidade de migrar de

EJB para qualquer outra tecnologia de componentes de negócios, a aplicação cliente permaneceria intacta, ficando as conseqüências limitadas apenas nos *Business Delegates*.

3.5 IMPLEMENTAÇÃO DE PADRÕES DE PROJETO DA CAMADA DE PERSISTÊNCIA

Esta camada é responsável por abstrair o acesso a um banco de dados, ou seja, dentro de um sistema corporativo, esta é a única camada que tem a permissão de acessar de forma direta o banco de dados. Neste projeto foi utilizada *Java Persistence API (JPA)* para esse fim.

3.5.1 Implementação da *JPA (Java Persistence API)*

JPA é um dos recursos mais empolgantes na versão 5 da plataforma *JEE* como um todo. Isso se deve ao fato de que o uso desta API abstrai totalmente o acesso a banco de dados, assim como dispensa a utilização de arquivos XML. A ligação das classes com as entidades do banco de dados é feita diretamente no código fonte da classe com a utilização de anotações. A listagem 12 mostra a classe persistente Tarefa.

Listagem 12 – Classe Tarefa

```
public class Tarefa implements Serializable
{
    @Id
    @Column(name = "ID_TAREFA", nullable = false)
    private Integer idTarefa;
    @Column(name = "ID_TIPOTAREFA", nullable = false)
    private int idTipotarefa;
    @Column(name = "DESCRICAO", nullable = false)
    private String descricao;
    @Column(name = "DATAINICIO", nullable = false)
    @Temporal(TemporalType.TIMESTAMP)
```

```

private Date datainicio;
@Column(name = "DATAPREVISAO", nullable = false)
@Temporal(TemporalType.TIMESTAMP)
private Date dataprevisao;
@Column(name = "DATAFIM")
@Temporal(TemporalType.TIMESTAMP)
private Date datafim;
@Column(name = "STATUS", nullable = false)
private char status;
@JoinColumn(name = "ID_PROJETO", referencedColumnName = "ID_PROJETO")
@ManyToOne
private Projeto idProjeto;
@JoinColumn(name = "ID_TAREFA", referencedColumnName = "ID_TIPOTAREFA",
insertable = false, updatable = false)
@OneToOne
private Tipotarefa tipotarefa;
@JoinColumn(name = "ID_USUARIOCRIACAO", referencedColumnName =
"ID_USUARIO")
@ManyToOne
private Usuario idUsuariocriacao;
@JoinColumn(name = "ID_USUARIORESPONSABEL", referencedColumnName =
"ID_USUARIO")
@ManyToOne
private Usuario idUsuarioresponsavel;
@OneToMany(cascade = CascadeType.ALL, mappedBy = "idTarefa")
private Collection<Subtarefa> subtarefaCollection;
@OneToMany(cascade = CascadeType.ALL, mappedBy = "idTarefa")
private Collection<Logtarefa> logtarefaCollection;

```

Uma vez criadas e configuradas as classes de persistência, o seu uso se torna muito fácil. Por exemplo, se for necessário obter todos os registros da tabela TIPOTAREFA, o código mostrado na listagem 13 pode ser utilizado.

Listagem 13 – Obter todos os registros da tabela TIPOTAREFA

```
GenericDB db = new GenericDB( );  
Collection tipotarefas = new ArrayList<Tipotarefa>( );  
tipotarefas = db.findAll("Tipotarefa");
```

Fica evidente através deste exemplo que o processo de desenvolvimento de software ganha muito em termos de produtividade, uma vez que dispensa quase por completo, exceto em algumas situações específicas, o uso da linguagem SQL. As instruções SQL para manipulação de banco de dados ainda são utilizadas, porém as mesmas são geradas em tempo de execução através da *JPA*.

3.5.2 Implementação do padrão de projeto DAO (*Data Access Object*)

Este padrão de projeto tem como o seu principal objetivo encapsular toda a lógica envolvida no acesso às fontes de dados. Como neste projeto foi utilizada a tecnologia *JPA* para abstrair o acesso a dados, a implementação do padrão *DAO* fica mais resumida.

Primeiro definimos uma interface que define os métodos básicos de manipulação de dados e que todas as classes *DAO* devem implementar. Essa interface chamamos de *GenericDAO* e o seu código é mostrado na listagem 14.

Listagem 14 – Interface *GenericDAO*

```
public interface GenericDAO  
{  
    public void incluir (Object o);  
    public void atualizar (Object o);  
    public void excluir (Object o);  
    public List listarTodos ( );  
    public Object localizar (int id);  
}
```

Conforme mostrado na seção 3.2.2, dentro do sistema existem classes que manipulam os bancos de dados relacionais e outras que manipulam os arquivos XML, sendo que as duas manipulam a mesma classe de domínio. No entanto, todas elas devem implementar a interface GenericDAO, que é quem define o comportamento básico para manipulação de dados. A declaração da classe UsuarioBD é feita da seguinte forma:

```
public class UsuarioBD implements GenericDAO{...}
```

A declaração da classe UsuarioXML implementa a mesma interface:

```
public class UsuarioXML implements GenericDAO{...}
```

Com a utilização deste padrão de projeto no nosso sistema conseguimos abstrair totalmente a lógica que está envolvida na persistência dos dados. Além disso, caso um dia se tornasse necessário implementar uma nova tecnologia chamada de, por exemplo, XYZ, bastaria criar as classes que lidassem com a lógica envolvida nessa tecnologia de persistência, implementando a nossa interface GenericDAO. Para o restante do sistema, não haveria nenhuma alteração no código fonte.

4 CONCLUSÃO

Os padrões de projetos corporativos aplicados ao processo de desenvolvimento de software se mostraram eficazes e eficientes na solução de problemas. Ficou claro, no entanto, que nem todos os padrões de projetos podem e devem ser aplicados a quaisquer sistemas. É preciso estudar e conhecer detalhadamente cada um deles para poder avaliar se dentro de um determinado projeto, considerando em primeiro lugar o seu tamanho e a sua complexidade, o seu uso é justificável.

Alguns dos padrões de projeto aplicados não foram fáceis de implementar, como foi o caso do padrão Command, e em aplicações de menor porte seria difícil justificar o seu uso. No entanto, em sistemas corporativos onde o tratamento de transações é crucial, a maneira como este padrão foi implementado em conjunto com a tecnologia EJB aumenta muito a manutenibilidade do sistema e a produtividade da equipe. Além disso, com este padrão foi possível atingir o baixo acoplamento do sistema, uma vez que a execução das transações fica contida em um ponto central.

Por outro lado, a tecnologia JPA é de implementação bastante fácil e a forma como ela abstrai todo o acesso aos dados melhora muito a legibilidade do código e a produtividade dos desenvolvedores. O uso da linguagem SQL para gerar as instruções de banco de dados é substituído, quase que na sua totalidade, por uma programação orientada a objetos, de fácil manipulação e manutenção.

O framework *JavaServer Faces* também se mostrou muito eficiente. Com ele foi possível implementar vários padrões de projeto na camada de apresentação de maneira bastante abstraída e encapsulada, porque o próprio framework já possui esses padrões embutidos. Um exemplo disso foi a implementação do padrão de projeto *Front Controller*. Ao invés de criar toda a estrutura de controle das requisições, apenas foi aproveitado e estendido o recurso já existente.

Os padrões GoF também se mostraram muito importantes e, apesar de não tratarem de requisitos especificamente corporativos, ficou claro que eles formam a base e estão presentes em quase todos os outros padrões de projeto mais avançados.

Utilizar *design patterns* é crucial em projetos de software para que estes possam atingir um nível de profissionalismo em que a fácil manutenção e a produtividade despontam como algumas das características mais importantes.

REFERÊNCIAS

ALUR, Deepak; CRUPI, John; MALKS, Dan. **Core J2EE Patterns**: Best Practices and Design Strategies. Sun Microsystems.

BARRETO, Celso Gomes. **Agregando Frameworks de Infra-Estrutura em uma Arquitetura Baseada em Componentes**: Um Estudo de Caso no Ambiente AulaNet. Dissertação (Mestrado em ciências de computação) – Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2006.

BISWAS, Rahul; ORT Ed. **The Java Persistence API - A Simpler Programming Model for Entity Persistence**. Disponível em: <<http://java.sun.com/developer/technicalArticles/J2EE/jpa/>>. Acesso em: 04 jun. 2007.

COOPER, James W. **The Design Patterns**: Java Companion. Addison-Wesley, 1998.

GAMMA, E. et al. **Design Patterns**: Elements of Reusable Object-Oriented Software. Holland: M.C. Eeachers, 1998.

HALL, Prentice. **Core Java Server Faces**. Laxxuss, 2004.

JENDROCK, Eric et al. **The Java EE5 Tutorial**: For Sun Java System Application Server Platform Edition 9. Santa Clara – California: Sun Microsystems Inc, 2006.

MARINESCU, Floyd. **EJB Design Patterns**: Advanced Patterns, Processes, and Idioms. New York: John Wiley & Sons, Inc., 2002.

ROMAN, Ed; SRIGANES, Rima Patel; BROSE Gerald. **Mastering Enterprise JavaBeans**. Indianapolis: Wiley Publishing Inc., 2002.