DANILO HENRIQUE MACHADO

UTILIZAÇÃO DE PADRÕES DE ARQUITETURA (PATTERNS) EM APLICAÇÔES J2EE

Monografia apresentada à Escola Politécnica de São Paulo para a obtenção do diploma do curso de MBA em Engenharia de Software

Área de concentração:
Patterns J2EE

Orientador:
Prof. Reginaldo Arakaki

São Paulo
2002

AGRADECIMENTOS

Agradeço ao meu orientador Prof. Reginaldo Arakaki pelos conselhos e diretrizes; à Profª Selma Regina pela oportunidade de participar do curso de Engenharia de Software e a minha família pela motivação e força.

Agradeço também meus colegas de trabalho pelos esclarecimentos tecnológicos e a todas as pessoas que diretamente ou indiretamente me ajudaram a superar os desafios.

RESUMO

Este trabalho apresenta as vantagens e desvantagens em utilizar Patterns na tecnologia J2EE. O projeto explica os fundamentos da tecnologia J2EE e de Pattern, além de apresentar cinco Patterns e analisá-los segundo a ISO 9126, abordando um conhecimento técnico do assunto. Os resultados obtidos têm fundamentos empíricos. Basearam-se na experiência com a linguagem e implementação dos Patterns apresentados para verificar sua utilização. O texto tem como objetivo auxiliar o desenvolvedor da tecnologia J2EE na difícil tarefa de compreender os Patterns, e utilizá-los em determinadas situações.

ABSTRACT

This lecture presents the benefits and disadvantages of using the tool Patterns from the J2EE environment. The project explains the basis of the J2EE Technology and Patterns, besides presents five Patterns and analyzes according to ISO 9126, approaching a technical knowledge of the subject. The results obtained have empirical foundations. Based on experience with the Language and implementation of the Patterns presented to verify its application. The text has the scope to help the developer of the J2EE technology at the hard work to understand the Patterns, and know how to use them in different situations.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS

J2EE	– Java 2 Enterprise Edition
EJB	– Enterprise Java Beans
CMP	– Container Manager Persisntece
BMP	– Bean Manager Persistence
XML	– Extensible Markup Language
HTML	– Hiper Text Markup Language
JDBC	– Java Database Connectivity
JMS	– Java Message Service
JSP	– Java Server Pages
JNDI	– JavaTM Naming and Directory Interface
ENC	– Enviroment Naming Context
RMI	– Java Remote Method Invocation
API	– Application Programam Interface
JRMP	– Java Remote Method Protocol
IIOP	– Inter-ORB Protocol

SUMÁRIO

1 OBJETIVO DO TRABALHO

Este trabalho tem como objetivo identificar e apresentar as principais vantagens e desvantagens na utilização de design patterns em aplicações J2EE.

Para entendimento dessa obra, foi elaborada uma pequena introdução sobre design patterns bem como da plataforma J2EE. Essa introdução será feita no capítulo 3 e é essencial para a compreensão dos capítulos subsequentes. Caso, seja necessário um entendimento maior, deve-se utilizar os links da bibliografia.

No capítulo 4, alguns patterns serão apresentados e analisados de acordo com os elementos da ISO 9126: Negócios, Desenvolvimento, Infra-estrutura de hardware e software, Manutenção e Segurança. Abaixo está uma descrição de cada um desses elementos.

Desenvolvimento: este elemento analisa o impacto do pattern no desenvolvimento do projeto. O importante é saber se o pattern auxiliou o programador a fazer um código de fácil leitura, com pouco acoplamento e em menos tempo, ou se o pattern prejudicou o projeto, inserindo complexidade e dificultando o entendimento do código.

Manutenção: este elemento está intimamente ligado ao desenvolvimento e em alguns patterns aparecem juntos. Analisa se o pattern contribuiu de alguma forma para que a manutenção seja mais rápida, fácil e segura.

Segurança: este elemento não faz referência a segurança da aplicação contra possíveis invasões de hackers ou autenticação do usuário, mas atua no sentido de realizar operações corretamente, dar consistência nos dados armazenados, além de gerenciar corretamente as transações.

Infra-estrutura: qualquer pattern que de alguma forma contribui para aumentar o desempenho de uma aplicação ou diminui o tráfego da rede, foi categorizado como infra-estrutura. Esse elemento está relacionado diretamente ao custo da aplicação. Quanto mais operações por minuto uma aplicação consegue processar, menos máquinas serão necessárias, diminuindo o custo em infra-estrutura.

Não é escopo dessa monografia defender as vantagens de se utilizar a plataforma J2EE da Sun Microsystem, nem indicar servidores de aplicação de qualquer fabricante. O capítulo 5, onde serão apresentados os resultados obtidos, utilizou-se o servidor de aplicação JBoss, somente por ser gratuito. O objetivo principal dessa monografia é apresentar as vantagens de se usar os designs patterns em tal plataforma.

2 MOTIVAÇÂO

Desde a criação da linguagem Java em 1994, até os dias atuais, a área de tecnologia evoluiu muito e o mercado mudou bastante. Nesses anos que passaram, a linguagem tida como promissora, mostrou suas qualidades entre elas a mais forte talvez seja a portabilidade. A própria propaganda da Sun Microsystem era "Write Once Run Anywhere", que significava que uma aplicação escrita para funcionar em uma plataforma poderia ser portada para outra plataforma sem problemas.

Pela facilidade da linguagem e por sua vasta utilização em sistema de Internet, o número de programadores cresceu para 2.5 milhões em apenas 5 anos. Esses números impressionam principalmente pela rapidez que a linguagem tomou um mercado tão competitivo.

Atualmente quase todo serviço está disponível na Internet, desde um simples site de consulta de informações, até sistemas distribuídos envolvendo diferentes plataformas e inúmeras transações. Essas aplicações que possuem requisitos específicos, de um modo geral, são mais difíceis de serem construídas, gastando-se mais tempo e esforço e requerem desenvolvedores altamente qualificados, entre outros fatores que dificultam a realização do projeto.

É exatamente para auxiliar e facilitar a criação desse tipo de aplicação que a SUN Microsystem criou uma plataforma chamada J2EE. Com isso, algumas operações que antes deveriam ser implementadas pelo desenvolvedor, a API já dá suporte, como por exemplo: gerenciamento

de transações, persistência de dados, objetos distribuídos, entre outros.

Porém, para aproveitar todos os recursos dessa nova tecnologia e ainda respeitar a necessidade de se criar uma aplicação escalável, diminuir o acoplamento entre as camadas, além de aumentar a performance, é necessário conhecer os patterns J2EE.

Esta monografia visa mostrar a real necessidade que existe em se compreender os patterns J2EE para criar uma aplicação com as qualidades desejadas pelo mercado.

3 FUNDAMENTOS

3.1.1 Introdução J2EE

A Sun Microsystem define a plataforma J2EE como uma arquitetura de componentes para desenvolvimento e instalação de componentes baseados em aplicações distribuídas. Aplicações escritas usando essa arquitetura são escaláveis, transacionais e possuem segurança para múltiplos usuários. Essas aplicações podem ser escritas uma vez e sem modificações no código, serem portadas para qualquer plataforma que suporte a especificação J2EE.

Atualmente existe uma necessidade de se desenvolver aplicações em ambientes distribuídos no menor tempo possível, diminuindo custos, reutilizando componentes, aumentando a segurança e a confiança no sistema. Nesse sentido a plataforma Java 2 Enterprise Edition (J2EE), oferece uma arquitetura baseada em componentes para o desenvolvimento e a instalação (deploy) dessas aplicações. Oferece ainda uma integração com XML, modelo de segurança unificado, além do controle de transações.

Com base em Alur;Crupi;Malks(2001) as vantagens seriam:

- Empresas podem desenvolver produtos e rodar em qualquer sistema que suporte a plataforma J2EE. "Sem grandes esforços", esses produtos estarão disponíveis para diferentes sistemas. (Isso só é possível se as aplicações utilizarem apenas as funcionalidades padrão da plataforma J2EE, sem depender de tecnologias específicas de determinados vendedores).

- Corporações tirando vantagem da portabilidade tornam-se independentes de vendedores e podem escolher produtos que implementem a plataforma J2EE que mais lhe agrade, diminuindo custos com plataformas diferentes.
- Programadores podem aumentar a produtividade se preocupando nas regras de negócio do sistema, sem implementar a parte de infra-estrutura. O servidor de aplicação gerencia serviços complexos como: multithread, transação, recursos alocados, sincronismo e gerenciamento do ciclo de vida dos componentes.
- Programadores java podem aprender rapidamente a desenvolver aplicações utilizando a plataforma J2EE.
- Empresas podem proteger seu investimento adotando a plataforma J2EE, uma vez que é uma plataforma padrão que diversas indústrias implementam, não sendo uma arquitetura proprietária fechada.
- Aplicações distribuídas podem ser desenvolvidas em menor tempo

3.1.2 Arquitetura

A aplicação é divida em componentes de acordo com a função. Esses componentes podem ser instalados em máquinas diferentes dependendo da camada que cada componente pertence. Na figura 1, temos um exemplo de uma aplicação J2EE. Como pode ser visto a aplicação é dividida em quatro camadas: cliente (apresentação cliente), servidor Web (apresentação), negócios (lógica de negócios) e informação do sistema.

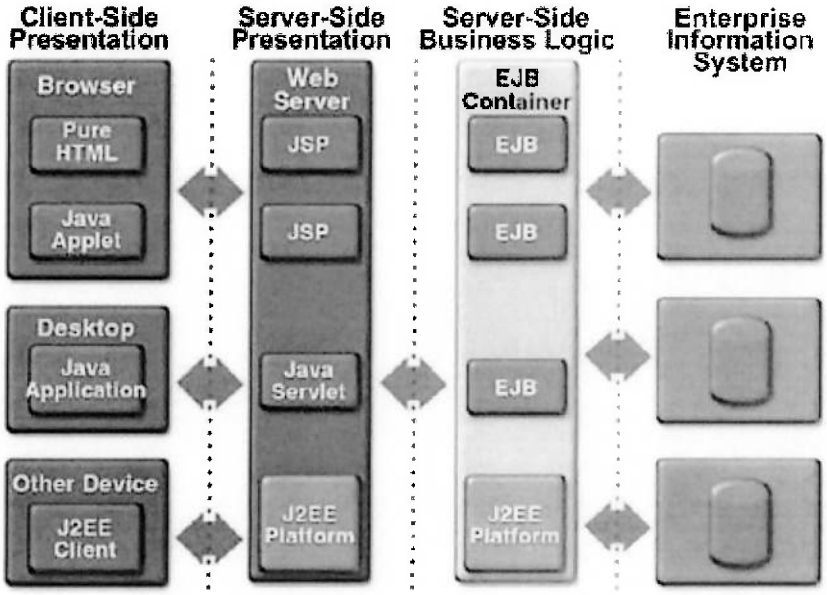


Figura 1 (Sun Microsystems –http://java.sun.com/j2ee)

Camada Cliente: É a camada que interage com o usuário. Recebe os dados da camada servidor web e as mostra na tela para o usuário. A plataforma J2EE suporta diferentes tipos de cliente, incluindo cliente HTML, applets ou mesmo aplicações java.

Camada Servidor Web: É responsável por receber a requisição do cliente, chamar a camada de negócio e gerar a resposta apropriada para o cliente. Na plataforma J2EE, servlets e JSP implementam essa camada

Camada de negócios: É o centro das regras de negócios da aplicação. Possui as interfaces necessárias para realizar todas as funções desejadas pelo cliente. Os componentes de negócios são tipicamente desenvolvidos como componentes EJB com suporte do EJB Container (tópico 3.1.2).

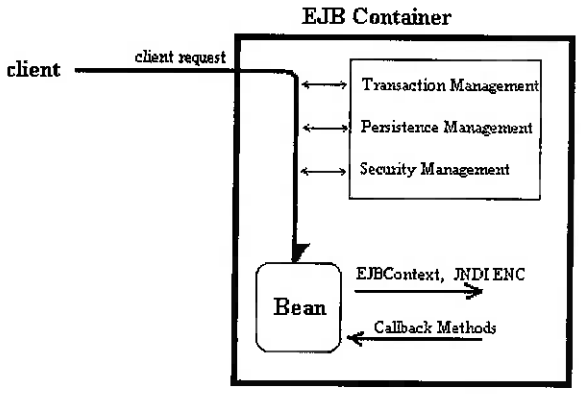
Informação do sistema: Esta camada é responsável pelas informações do sistema, incluindo o sistema de banco de dados, processamento de

transações e sistema legados. Também é responsável pela integração de aplicações J2EE com aplicações não J2EE(sistemas legados).

3.1.3 EJB Container

Enterprise beans é um dos componentes da plataforma J2EE. São escritos em java e rodam em um ambiente especial chamado EJB Container. O container gerencia todos os aspectos de um enterprise bean em tempo de execução como: acesso remoto a outro bean, segurança, persistência, transação, concorrência e acesso ao pool de recursos.

O container é mais um conceito do que uma construção física. O container isola o enterprise bean do acesso direto de aplicações clientes, funcionando como um intermediador entre a aplicação cliente e os beans. Quando uma aplicação cliente invoca um método remoto, o container intercepta a invocação para assegurar a persistência, transação e segurança. Sendo assim o programador deve-se concentrar apenas nas regras de negócios e deixar o restante com o EJB Container. A figura 2 mostra o funcionamento do container.



EJB Containers manage enterprise beans at runtime

Figura 2(Sun Microsystems – http://java.sun.com/ejb)

Para reduzir a utilização de processador e memória pela aplicação, o container cria um pool de recursos e gerencia o ciclo de vida de todos os beans cuidadosamente (mais de um bean pode ser gerenciado simultaneamente). Quando um bean não está sendo utilizado, o container o coloca no pool para ser usado por outro cliente, e somente o carrega novamente quando for necessário. Enquanto um bean está no pool, a referência remota no cliente permanece intacta. Quando o cliente invoca algum método, o container recarrega o bean para responder ao request. Toda essa operação no container é desconhecida pela aplicação cliente.

Além disso, o container gerencia conexões a banco de dados via JDBC e o acesso a outros enterprise beans. Um enterprise bean pode interagir com o container de três maneiras diferentes: métodos callback, interface EJBContext, ou pela interface de nome e diretórios Java™ Naming and Directory Interface (JNDI).

Métodos CallBack

Todo bean implementa um sub-tipo de uma interface EnterpriseBean que define diversos métodos chamados callback. Cada método callback é invocado em diferentes momentos no ciclo de vida de um bean, como: ativa-lo, gravar os dados, removê-lo da memória, entre outros. Deve-se conhecer o ciclo de vida do bean para saber quando cada método é chamado e que tipo de operação deve ser executada. Isso auxiliará para implementar uma aplicação correta e eficiente.

EJBContext

Todo bean obtém um objeto do tipo EJBContext, que é a referência direta para o container. Esta interface possui métodos para obter

informações de uma requisição, como a identidade do cliente, o estado de uma transação ou obter referências remotas para si mesmo.

Interface de Nomes e diretórios(JNDI)

JNDI é a extensão padrão da plataforma java para acessar nomes de sistemas como LDAP, NetWare, sistemas de arquivos, etc. Todo bean automaticamente tem acesso a um sistema especial de nomes chamado Contexto de Nomes de Ambiente(Enviroment Naming Context - ENC). ENC é gerenciado pelo container e acessa os beans usando JNDI. Com isso, o bean pode ter acesso a recursos como conexões JDBC, a outros beans e a propriedades específicas de um bean.

Os componentes enterprise beans interagem com o EJB Container por um modelo bem definido. Os beans podem ser classificados como: EntityBean, SessionBean e MessageDrivenBean (EJB 2.0).

3.2 Enterprise Beans

Para criar um componente EJB, o desenvolvedor deve implementar duas interfaces (HomeInterface e RemoteInterface) além da classe principal do Bean. Essas interfaces expõem as habilidades do bean para o cliente e todos os métodos que estiverem nessas interfaces estarão acessíveis para o cliente. Na HomeInterface devem ficar os métodos referentes ao ciclo de vida do bean como: create() e remove(). A RemoteInterface deve possuir os métodos de negócios como: calcularFatura(), verificarEstoque(), etc.

3.2.1 Session Bean

Session Bean representa o processo do modelo de negócios. Eles podem ser comparados a verbos porque normalmente executam uma ação qualquer que pode ser: chamar um outro bean, finalizar uma compra, calcular um desconto, etc. Session bean não é persistente, ou seja, seus dados não são gravados em Banco de Dados. Após uma requisição a um Session Bean, o cliente não tem mais nada que o associe à aquele bean que foi chamado.

Existem dois tipos de Session: Stateful e Stateless. Quando se deseja que o Session guarde algum valor ele deve ser do tipo Stateful. Isto faz com que um único cliente acesse um único bean e durante a requisição do cliente, os dados do bean são mantidos. Os dados serão apagados somente quando o cliente remover o bean. Essa abordagem deve ser implementada com cuidados extras já que afeta diretamente na performance do sistema.
O Session Stateless não mantém informação, podendo ser compartilhado entre diversos clientes e isto aumenta a performance da aplicação

3.2.2 Entity Bean

Entity Bean representa o objeto de negócio do sistema. Ele é persistente e normalmente cada entidade possui uma tabela relacionada no Banco de Dados. Cada linha corresponde a uma entidade e cada campo da tabela corresponde a um atributo do Bean. Existem dois modos de persistência: BMP (Bean Manager Persistence) e CMP (Container Bean Persistence). Utilizando BMP, o desenvolvedor deve

3.2.1 Session Bean

Session Bean representa o processo do modelo de negócios. Eles podem ser comparados a verbos porque normalmente executam uma ação qualquer que pode ser: chamar um outro bean, finalizar uma compra, calcular um desconto, etc. Session bean não é persistente, ou seja, seus dados não são gravados em Banco de Dados. Após uma requisição a um Session Bean, o cliente não tem mais nada que o associe a aquele bean que foi chamado.

Existem dois tipos de Session: Stateful e Stateless. Quando se deseja que o Session guarde algum valor ele deve ser do tipo Stateful. Isto faz com que um único cliente acesse um único bean e durante a requisição do cliente, os dados do bean são mantidos. Os dados serão apagados somente quando o cliente remover o bean. Essa abordagem deve ser implementada com cuidados extras já que afeta diretamente na performance do sistema.
O Session Stateless não mantém informação, podendo ser compartilhado entre diversos clientes e isto aumenta a performance da aplicação.

3.2.2 Entity Bean

Entity Bean representa o objeto de negócio do sistema. Ele é persistente e normalmente cada entidade possui uma tabela relacionada no Banco de Dados. Cada linha corresponde a uma entidade e cada campo da tabela corresponde a um atributo do Bean. Existem dois modos de persistência: BMP (Bean Manager Persistence) e CMP (Container Bean Persistence). Utilizando BMP, o desenvolvedor deve

se preocupar em escrever todo o código necessário para acessar o banco de dados. Isso traz dois inconvenientes: aumenta a possibilidade de erros e tempo de trabalho. Já no CMP, o EJB Container gera automaticamente o código de acesso ao Banco de Dados, aumentando a produtividade. Mas o programador deve estar atento ao número de requisições que o EJBContainer faz ao Banco de Dados, pois é comum realizar requisições que não são necessárias.

Vários clientes podem acessar um Entity bean simultaneamente, porém todas as operações de escrita devem ter o suporte de transação garantindo a integridade dos dados. Para isso, basta especificar no "deployment descriptor" que aquele método exige uma transação. O EJBContainer gerencia totalmente a transação de um método, não sendo necessário código adicional a ser implementado.

Similar a um Banco de Dados, cada entidade deve possuir um identificador único, chamado de chave primária. Por exemplo, em uma entidade de pessoas a chave primária poderia ser o seu RG. Com isso, fica fácil para uma entidade ser encontrada, além de garantir a unicidade.

Um Entity Bean pode se relacionar com outro bean, como por exemplo, a entidade de pessoas e a entidade de cargos. Implementa-se essa relação diferentemente para CMP e BMP. Usando BMP, o desenvolvedor deve implementar nos métodos de acesso ao banco essa relação. Já no CMP, o EJB Cointainer se encarrega de fazer essa relação, sendo necessário apenas acrescentar os campos e tabelas relacionados no deployment descriptor(xml)

3.2.3 Objetos Distribuídos

As interfaces remote e home são interfaces do tipo Java RMI. A interface java.rmi.Remote é usada por objetos distribuídos para representar o bean em uma máquina diferente. O enterprise bean é um objeto distribuído que significa que a classe é instanciada e fica residente no EJBContainer, sendo acessível para aplicações em outras máquinas.

Para que uma instância de objeto esteja disponível em outra máquina, é necessário encapsular a instância em um objeto especial chamado skeleton. Este tem a conexão para outro objeto especial chamado stub que implementa a interface remota. Ele mantém a conexão via socket para o skeleton e toda vez que um método de negócios é invocado na interface do sub, ele manda uma mensagem para o skeleton informando que um método foi chamado. Quando o skeleton recebe essa mensagem, identifica os métodos invocados e os argumentos, acessando o método correspondente da instância atual. A instância executa o método e retorna o resultado para o skeleton que envia para o stub. Esta troca de mensagens é mostrada na figura3:

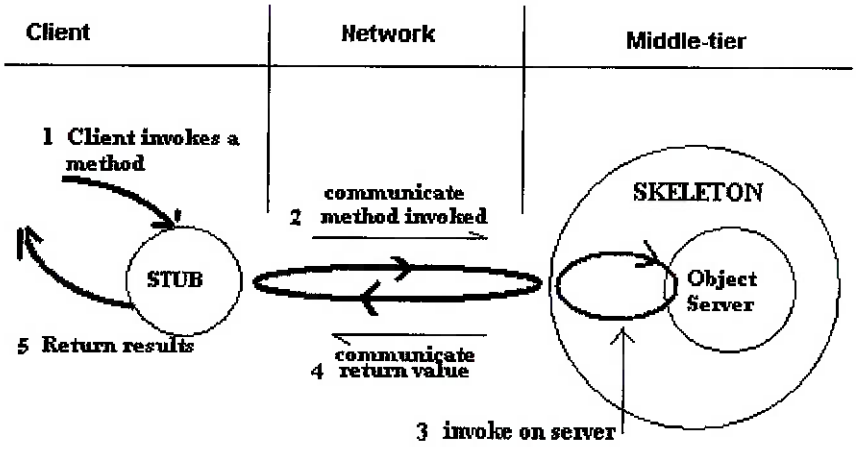


Figura 3(Sun Microsystems – http://java.sun.com/)

Para a aplicação cliente, o stub aparentemente executa o método invocado localmente. Na verdade, o stub é apenas um objeto da rede que envia uma requisição através da rede e então o skeleton repassa para a instância atual. A instância invocada é que tem a implementação do método e que efetivamente gera o resultado.

Na plataforma J2EE, o skeleton da interface remote e home são implementados pelo container e não pelo bean. Isso garante que todo método invocado pela aplicação cliente é primeiro tratado pelo container e somente então repassado a instância do bean.

Protocolos de objetos distribuídos definem o formato das mensagens de redes enviadas entre as máquinas. Esses protocolos podem se tornar bem complicados para o desenvolvedor, porém ele não precisa se preocupar, pois o EJBContainer gerencia isto automaticamente. A maioria dos servidores EJB suporta Java Remote Method Protocol (JRMP) e/ou CORBA's Internet Inter-ORB Protocol (IIOP).

O programador somente desenvolve a classe do bean e sua interface remota, os detalhes de comunicação da rede são escondidos. Para o programador não importa se o servidor EJB usa JRMP ou IIOP, ou os dois ao mesmo tempo. A especificação requer que se utilize uma versão da API Java RMI quando se estiver trabalhando com um bean remotamente. Java RMI é uma API para acessar objetos distribuídos.

3.4 DESIGN PATTERN

3.4.1 Introdução

Em 1970, Cristopher Alexander escreveu uma série de livros documentando pattern em engenharia civil e arquitetura. Segundo ele, pattern descreve um problema que ocorre várias vezes em um ambiente, bem como a solução para esse problema, de modo que se possa resolvê-lo sempre da mesma maneira. Embora Alexander estivesse descrevendo pattern em prédios e construções, a comunidade de software adotou a idéia de pattern baseado em seu trabalho.

Pattern é a ligação entre problemas e soluções. Simples, porém isso nos possibilita documentar um problema recorrente e conhecido e sua solução em um contexto particular.

Sendo assim, outras pessoas que tenham um mesmo problema podem utilizar a solução proposta, além de facilitar as discussões entre pessoas da área, já que os patterns funcionam como uma linguagem universal e conhecida. Um elemento importante na definição de pattern é o termo recorrente, uma vez que seu objetivo é encorajar o reuso de soluções. Algumas outras características são:

- Patterns são escritos em um formato estrutural
- Patterns previne a reinvenção da roda
- Existem patterns em diferentes níveis de abstração
- Patterns são artefatos de reuso
- Patterns fazem a ligação entre design e boas práticas

- Diferentes patterns podem ser utilizados juntos para resolver um problema
- Continuamente patterns são aperfeiçoados

3.4.2 Elementos de Designs Pattern

Nessa monografia, o pattern foi dividido em 6 elementos essenciais: nome, contexto, problema, motivação, solução e conseqüências. Abaixo está o significado de cada um deles.

Nome: O nome de um pattern é usado para descrever o problema e/ou a solução em uma ou duas palavras. Dar nomes a um pattern aumenta o nível de abstração e possibilita ter um vocabulário conhecido de pattern, tornando possível discussões entre pessoas da área.

Contexto: Esse elemento descreve em que contexto esse pattern deve ser utilizado. Pode-se dizer que o contexto é a pre-condição para que o desenvolvedor venha a ter determinados problemas (próximo elemento).

Problema: Descrição detalhada dos problemas encontrados em um determinado contexto, sem a utilização do pattern. Nesta monografia, para cada pattern, são apresentados os problemas mais comuns em uma aplicação J2EE antes de aplicá-los.

Solução: Contém a descrição dos passos para implementar a solução apresentada, e em alguns patterns contém diagramas de classes e/ou de seqüência.

Conseqüências: Apresentar quais são as conseqüências de usar o pattern para aquele contexto. Para todos os patterns, no capítulo 4, são apresentadas as vantagens e as desvantagens de utilizá-lo com base em Alur,Crupi, Malks(2001).

3.4.3 Anti-pattern

Se um pattern representa a melhor solução, o anti-pattern representa uma lição aprendida. Inicialmente proposta por Andrew Koenig, em Novembro de 1995, tem como definição aquilo que descreve uma solução ruim para resolver uma situação difícil. É importante conhecer os anti-pattern para identificá-los e evitá-los.

Atualmente existem livros especializados no assunto, é o caso do livro intitulado Anti-Patterns de Brown, Malveau, McCormick e Mowbray (http://www.anti-pattern.com). Os exemplos desse livro são parecidos com algumas armadilhas apresentadas em "Pitfalls of Object-Oriented Development" de Bruce Webster e "Lições aprendidas" de Tom Love.

4 ANÁLISE DOS ELEMENTOS DE ARQUITETURA

4.1 Value Object

4.1.1 Contexto

Aplicações clientes precisam trocar dados com enterprise beans

4.1.2 Problema

Aplicações J2EE implementam componentes de negócios como session beans ou entity beans. Alguns métodos desses componentes retornam dados para o cliente. Geralmente o cliente invoca os métodos dos objetos de negócios várias vezes até obter todos os valores necessários. Toda vez que um método de um objeto de negócio é chamado, sendo um Session Bean ou um Entity Bean, uma requisição remota será feita. Dessa forma, em uma aplicação EJB o alto número de invocações remotas pode causar um aumento no tráfego da rede.

4.1.3 Motivação

- Todos os acessos a um enterprise bean são feitos via interface remota. Toda requisição feita é potencialmente uma requisição de método remoto com aumento no tráfego da rede.
- Freqüentemente uma aplicação tem mais operações de leitura do que de escrita.
- Geralmente o cliente requisita ao enterprise bean o valor de mais de um atributo. Sendo assim, o cliente deve fazer diversas chamadas remotas até conseguir os dados necessários.
- O número de invocações remotas feitas pelo cliente pode influenciar na performance da rede.

4.1.4 Solução

Deve-se usar o pattern "Value Object" para encapsular os dados de negócio. Um simples método é usado para enviar e receber o objeto ao cliente. Quando o cliente invoca o bean, cria um "objeto de valor" e atualiza os atributos desse objeto com os valores disponíveis. Esse novo objeto criado é passado por valor a aplicação cliente, que tem acesso a um objeto cópia. Métodos locais do novo objeto deixam os valores de seus atributos disponíveis a aplicação cliente. Esses métodos são chamados de "getters" e para cada atributo a ser lido existe um método get correspondente (ver código 1). Porém, qualquer mudança feita nos atributos do objeto não influenciará no objeto original que está no enterprise bean.

Na figura 4 é apresentado o diagrama de classes. O objeto de valor (Value Object) é criado quando necessário pelo enterprise bean e retornado ao cliente. Note que o bean possui os seguintes métodos: getData() para criar e retornar uma instância de ValueObject ao cliente, setData(ValueObject) para receber uma instância de ValueObject como parâmetro e atualizar os valores do bean.

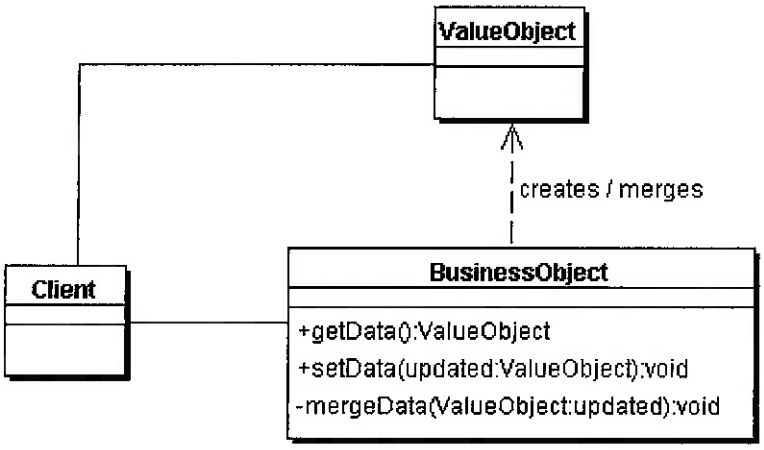


Figura 4(Sun Microsystem - http://java.sun.com.br/blueprints)

Na figura 5 é apresentado o diagrama de seqüência desse pattern para ficar mais claro o processo que existe entre, a aplicação cliente fazer a requisição e o objeto de negócios criar e retornar uma instância de Value Object.

Fica nítido no diagrama que as alterações feitas no objeto que o cliente recebeu(Client Copy) não afetarão os valores dos objetos que estão no servidor(Server Copy).

Entre o cliente e o objeto de negócio pode existir a interface do Session Facade, que não foi inserida para não acrescentar complexidade ao diagrama.

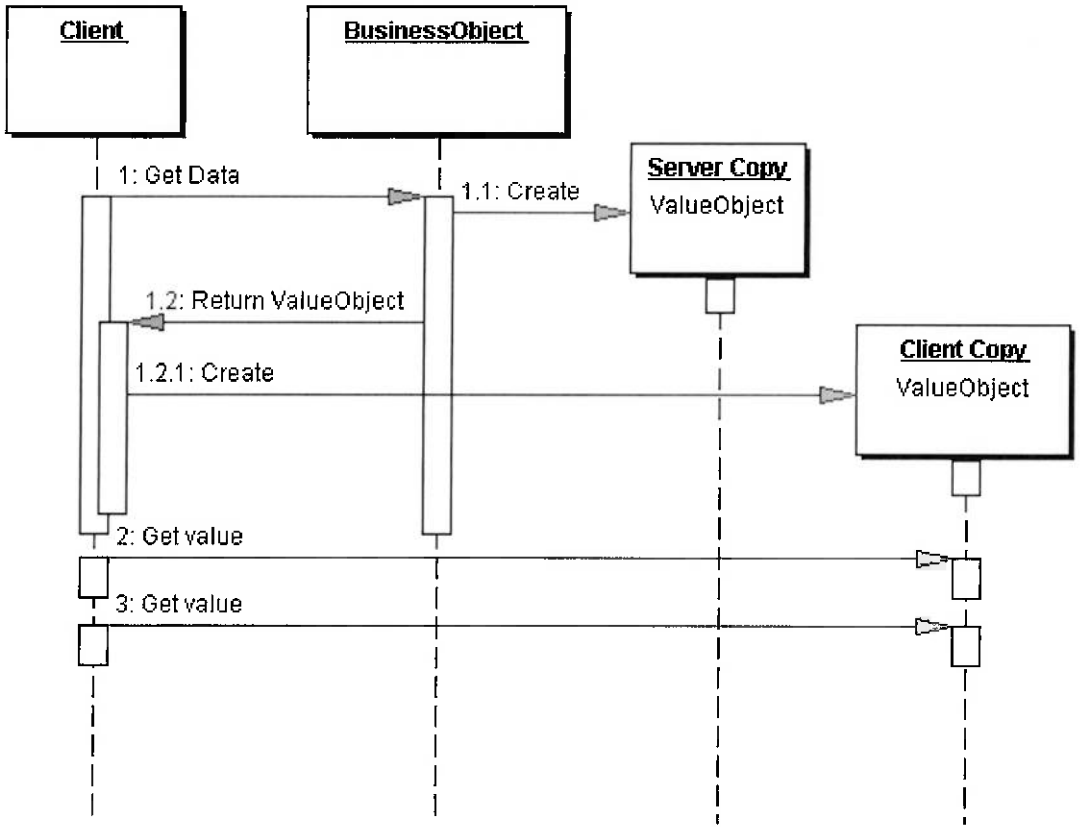


Figura 5(Sun Microsystem - http://java.sun.com.br/blueprints)

Elementos da figura 5

Aplicação Cliente

Representa o cliente do enterprise bean que pode ser um usuário final ou um outro entity bean.

Business Object

Representa um Session Bean ou um Entity Bean, e é responsável por criar o objeto de valor e retorná-lo ao cliente.

Value Object

Representa um objeto Java serializável arbitrário.

4.1.5 Updatable Value Objects

Nessa estratégia, o objeto de valor possui duas funções: a primeira e mais fácil de entender, é levar informações dos beans para a aplicação cliente poder fazer a consulta aos dados dos beans. Normalmente, isso é feito através de um método, na figura 4 o método é getData(), que retorna ao cliente um objeto de valor. Porém, como já foi mencionado, esse objeto é uma cópia do bean e qualquer mudança nesse objeto não refletirá em mudanças dos valores dos beans.

Para que os valores dos atributos dos beans sejam atualizados, métodos "setters" são implementados, um para cada atributo. Sendo assim, voltamos ao problema original que era de fazer várias chamadas remotas para realizar uma operação. A segunda função do objeto de valor nessa estratégia é exatamente evitar que isso ocorra. A aplicação cliente modifica os valores dessa cópia e depois envia esse objeto

através de apenas um método setData(objeto de valor) que está no Bean. No código 1 temos um código de um objeto de valor.

Código1

```
public class MonografiaVO implements java.io.serializable
{
	private Integer idMonografia;
	private String nomeMonografia;
	private String autorMonografia;

	public void setNome(String nome)
	{
		if ( nome != null )
		{
			this.nomeMonografia = nome;
		}
	}
	public void setAutor(String autor)
	{
		if ( autor != null )
		{
			this.autorMonografia = autor;
		}
	}
	public String getNome()
	{
		return this.nomeMonografia;
	}
	public String getAutor()
	{
		return this.autorMonografia;
	}
}
```

É interessante para o desenvolvedor deixar no objeto de valor as validações de seus atributos, com isso ganha-se em performance, pois todos os objetos de negócios que chegarem ao bean já possuem valores corretos. Outro ponto importante é implementar algum mecanismo para verificar se os valores recebidos estão diferentes dos atuais, para não fazer acesso ao banco de dados sem necessidade.

Quando o bean recebe o objeto de valor, os novos valores devem ser lidos para atualizar seus atributos. No código 2, é apresentado o método setData() do bean.

Código 2

```
public void setData(MonografiaVO monVO)
{
        set.Nome(monVO.getNome);
        set.Autor(monVO.getAutor);
}
```

Essa estratégia adiciona um problema quando um bean possui mais de uma aplicação cliente. O cliente A faz uma requisição ao bean B e recebe o objeto de valor, antes do cliente A chamar o método setData(objeto de valor), e o cliente C também faz uma requisição para o bean B. Enquanto o cliente C está processando e atualizando o objeto de valor(cópia), o cliente A invoca o método setData passando o objeto de valor com as devidas modificações. Após algum tempo, o cliente C também chama o método setData.

Essa operação não vai gerar exceção para nenhum dos clientes, mas haverá uma inconsistência no banco de dados. O Bean B possui apenas

os valores que o último cliente enviou. O bean deve implementar algum sincronismo para que esse tipo de erro não ocorra.

Na figura 6 é apresentado o diagrama de sequência utilizando essa estratégia. A diferença básica entre esse diagrama e o da figura 5 é a presença dos métodos setData() para o cliente poder atualizar o valores do objeto de negócio.

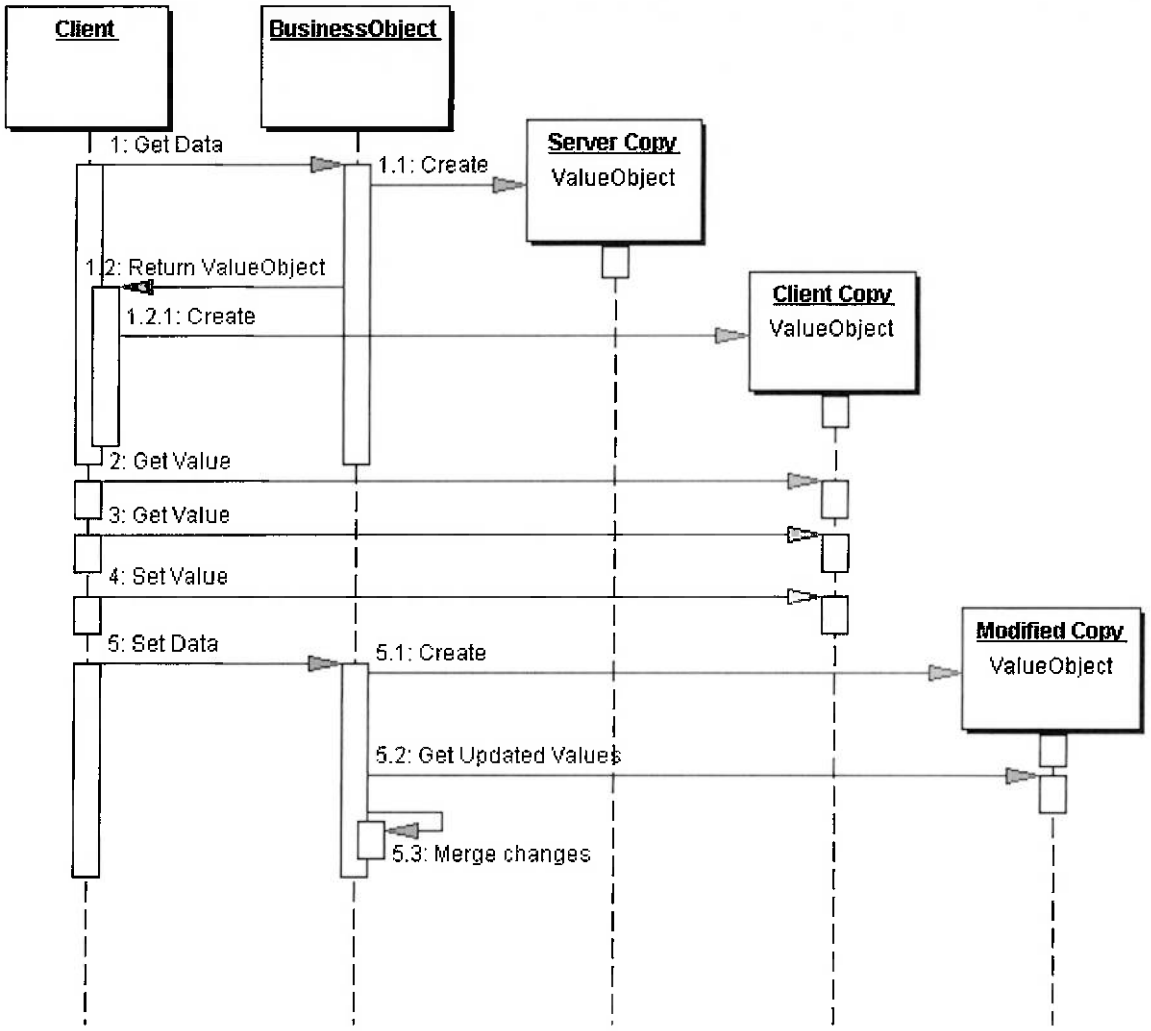


Figura 6(Sun Microsystem - http://java.sun.com.br/blueprints)

4.1.6 Análise segundo a ISO 9126

Desenvolvimento/Manutenção

Simplificar entity beans e interfaces remotas: O entity bean provê um método getData() para enviar o objeto de valor com todos os seus atributos. Isso elimina a necessidade do bean ter a implementação de vários métodos get para cada atributo, bem como sua declaração na interface remota.
Igualmente, o entity bean provê um método setData() para atualizar os valores dos atributos do entity bean em um único método, isso elimina a necessidade do bean ter a implementação de vários métodos set para cada atributo, bem como sua declaração na interface remota.

Desenvolvimento/Manutenção

Reduzir a duplicação de código: Usando a estratégia "Entity Inherit Value Object", onde o bean é filho (extends) do objeto de valor, permiti reduzir ou eliminar a duplicação de código entre a entidade e o objeto de valor. O tempo de codificação e manutenção não influência muito, pois o ganho em tempo na eliminação de código é mínimo.

Infra-estrutura de software e hardware

Reduzir o tráfego na rede: Em lugar de várias chamadas remotas através da rede para obter os valores dos atributos de um bean, será feito apenas uma chamada remota getData(). Analogamente, será necessário chamar apenas um método (setData) para atualizar os valores de um bean, contribuindo para diminuir o tráfego da rede e o número de requisições que a aplicação recebe. Com isso, melhora-se o tráfego na rede e a performance da aplicação.

Desenvolvimento/Segurança

Aumentar a complexidade: Quando se disponibiliza um método setData() e existe mais de uma aplicação cliente para utilizá-lo, valores não consistentes podem ser inseridos no banco de dados.

Uma vez que as modificações foram feitas no objeto de valor (cópia) no lado do cliente, o método setData() é invocado e o objeto de valor é passado como parâmetro. O entity bean recebe o objeto modificado e atualiza seus atributos. Contudo, outros clientes podem ter requisitado o mesmo objeto de valor anteriormente que não reflete mais o objeto de valor que o entity bean possui.

Nesse caso é possível que informações que deveriam ser gravadas no banco de dados sejam perdidas ou inconsistentes, afetando a segurança e a confiabilidade da aplicação. Controle de versão é uma maneira de evitá-los. Um dos atributos do entity bean poderia ser o número da versão ou a data da última modificação. O número da versão ou a data da última atualização é copiado para o objeto de valor na sua criação. Com isso o entity bean consegue saber se o cliente possui uma cópia do objeto desatualizada e retorna uma mensagem de erro.

Para tal complexidade, o tempo gasto na programação/manutenção deve aumentar. Essa estratégia só deve ser utilizada quando existe a necessidade de atualizações nas informações dos beans por diversos clientes. Em alguns casos onde a aplicação cliente somente lê as informações do "Value Object" ou só existe uma aplicação cliente, esse controle não precisa ser implementado.

4.2 SESSION FACADE

4.2.1 Contexto

Enterprise Bean encapsula os objetos de negócios diminuindo a complexidade dos clientes invocarem os métodos dos beans.

4.2.2 Problema

É comum em aplicações J2EE termos as seguintes situações:

- Existência de uma forte dependência entre a aplicação cliente e os objetos de negócios(beans)
- A aplicação cliente faz inúmeras requisições ao servidor prejudicando a performance
- Ausência de uma estratégia de acesso aos métodos dos objetos de negócios, aumentando o risco de falhas.

Na arquitetura J2EE existe uma camada chamada de lógica de negócios (business logic). A função dessa camada é receber a requisição das aplicações clientes, processá-la, fazendo acesso ao banco de dados ou qualquer outro tipo de operação e retornar o valor esperado para aplicação cliente. Os objetos que fazem isso são chamados de objetos de negócios e normalmente são beans.

A aplicação cliente pode interagir diretamente com os métodos dos objetos de negócios, desde que a assinatura do método esteja na interface remota (EJBRemote). O problema aparece exatamente quando isso ocorre, pois a aplicação cliente deve entender e conhecer

as informações dos objetos de negócios. Cria-se então, uma dependência entre a aplicação cliente e os objetos de negócios. O cliente passa a ter a responsabilidade de conhecer a relação entre os objetos de negócios, além de gerenciar transações.

De acordo com o aumento dos requisitos do cliente, o grau de complexidade também aumenta, já que novos objetos de negócios deverão ser conhecidos e acessados. Futuras manutenções a serem feitas nesse código correm o risco de demorar mais tempo e passíveis de erros.

Quando uma aplicação cliente acessa um método de um objeto de negócio, é feita uma requisição remota ao servidor. Quanto mais requisições forem feitas, maior será o tráfego da rede. Se um cliente fizer 5 acessos remotos para realizar apenas 1 tarefa, 4 chamadas remotas foram feitas sem necessidade.

Como exemplo podemos citar uma aplicação para autenticar um usuário a fazer um cadastro de uma monografia. A aplicação cliente precisa acessar os métodos dos objetos de negócios:

UsuarioEJB método procurarUsuário(nomeusuario), para saber se o usuário existe
UsuarioEJB método verificarSenha(senha), para validar a senha
UsuarioEJB método pegaGrupo(nomeusuario), para saber a que grupo ele pertence
GrupoEJB método validaGrupo(), para saber se o usuário daquele grupo pode entrar nessa parte do sistema
PermissaoEJB método validaOperação, para saber se o usuário daquele grupo pode fazer essa operação

4.2.3 Motivação

- Disponibilizar uma interface simples para as aplicações clientes, escondendo toda a interação entre os objetos de negócios.
- Diminuir o número de objetos de negócios que são expostos aos clientes através da rede, diminuindo o acoplamento e melhorando o tráfego da rede.
- Prover uma interface uniforme, deixando independente a implementação do serviço.

4.2.4 Solução

Esse pattern provê uma interface com apenas os métodos necessários para a aplicação cliente. Essa abstração faz com que o cliente não tenha que conhecer as complexas interações entre os objetos de negócios. O Session Facade conhece todos os objetos participantes e possui a lógica para obter os dados necessários a aplicação cliente. A aplicação cliente não pode mais acessar diretamente os métodos dos beans, somente através do Session Facade, uma vez que a interface remota (EJBRemote) dos objetos passa a ser local(EJBLocal).

Para ficar mais claro podemos utilizar o exemplo 1 onde era necessário chamar 5 métodos de duas entidades para autenticar um usuário para realizar um cadastro. Utilizando esse pattern, a aplicação cliente acessaria apenas um método autenticar() do Session Facade. Seria função do Session Facade fazer o lookup nos beans necessários e requisitar aos métodos desses beans as operações para realizar a autenticação. Do ponto de vista da aplicação cliente, não existe outro bean a não ser o Session Facade.

O Session Facade também fica responsável pelas transações da aplicação. Sendo um ponto central de chamadas para outros beans (lookup), essa interface torna-se um ótimo lugar para gerenciá-las. Novamente a aplicação cliente não sabe se durante a operação ocorreu um erro e foi necessário voltar as operações até aquele momento(rollback). Quando o Session Facade gera um exceção (EJBException) o container a interpreta e realiza o rollback.

É interessante o desenvolvedor identificar as entidades envolvidas em um fluxo de trabalho e criar um Session Facade para cada fluxo. Para diminuir o acoplamento entre as entidades e ganhar flexibilidade, o Session Facade não deve acessar métodos de objetos negócios que não seja do seu fluxo de trabalho. Para isso, ele deve acessar os métodos do Session Facade que gerencia aquele fluxo.

Na figura 7 é apresentado o diagrama de seqüência desse pattern. Note que o cliente só tem acesso aos métodos do Session Facade e esse é responsável por acessar os objetos de negócios necessários para realizar a operação desejada. O Session Facade pode se utilizar de outro pattern, o "Data Access Object" para acesso ao banco de dados. Esse pattern também será apresentado.

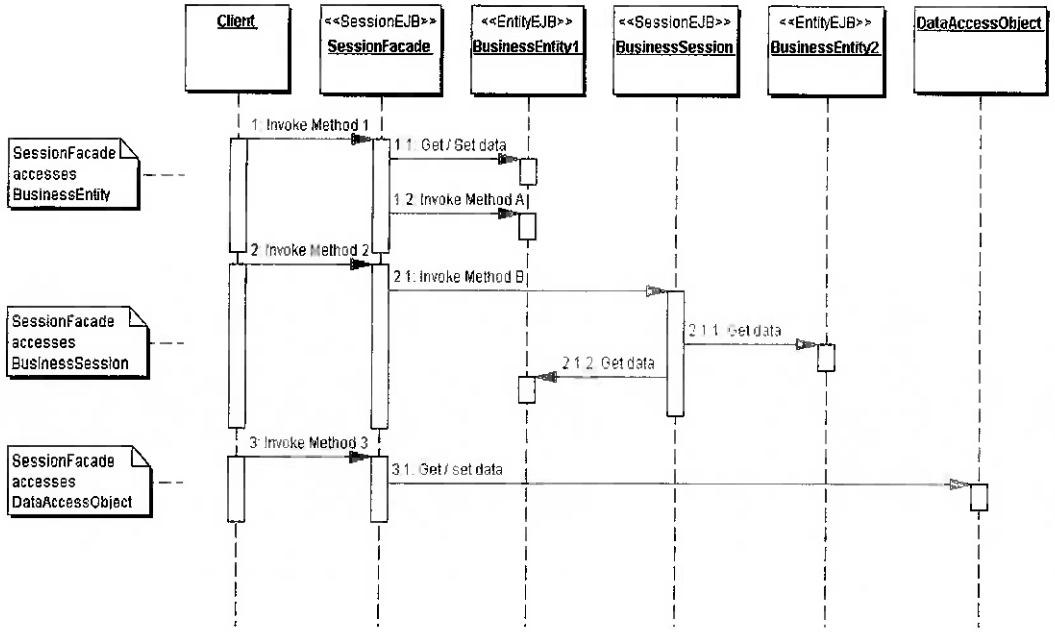


Figura 7(Sun Microsystem - http://java.sun.com.br/blueprints)

Elementos da figura 7

Cliente

Representa a aplicação cliente do Session Facade. Ele pode ser um cliente remoto ou um cliente local, sendo um outro Session Facade.

SessionFacade

O Session Facade é implementado como um Session Bean. Ele faz a ligação entre os clientes e os objetos de negócios.

Business Object

Os objetos de negócios são as entidades da aplicação, podendo ser Entity Bean, Session Bean e Message Driven Bean. Eles realizam determinadas operações ou acessam informações que serão repassadas ao Session Facade.

4.2.5 Análise segundo a ISO 9126

Desenvolvimento/Manutenção

Introduzir camada de controle de negócios: Session Facade representa uma camada de controle entre o cliente e a camada de negócios. Em pequenas aplicações, pode-se pensar que esse pattern não apresenta grandes vantagens, porém sua utilização é necessária em todas as aplicações que um cliente remoto precisa interagir com os beans.

As interações entre os componentes de negócio podem ser muito complexas. Esse pattern abstrai essa complexidade e disponibiliza ao cliente uma interface fácil de entender e usar, ajudando no desenvolvimento e na manutenção do bean. O Session Facade provê uma camada de acesso uniforme para todos os tipos de clientes e consegue proteger e esconder os componentes de negócios.

Com essa camada adicional, o tempo de desenvolvimento aumenta, contudo esse aumento é muito pequeno e recompensado por uma série de motivos que serão apresentados a seguir.

Redução de acoplamento: O uso de uma camada entre a aplicação cliente e os componentes de negócios fazem com que reduza o grau de acoplamento entre eles. Com isso, o código da aplicação cliente reduz e fica mais simples já que ele delega ao Session Facade as interações entre os objetos de negócios. É mais vantajoso usar o Session Facade para gerenciar e abstrair as interações entre os objetos de negócio a deixá-los dependentes.

Como ganho principal tem-se: aumento na flexibilidade, facilidade nas mudanças e centralização das interações. Manutenção nos objetos de negócios participantes acarreta em modificações no Session Facade, porém essas modificações são facilmente gerenciáveis, pois elas não se propagam aos clientes diminuindo o tempo da manutenção.

Criar apenas uma classe como Facade: Não se deve criar apenas uma classe pai como SessionFacade para fazer a ligação de todos os clientes com todos os beans. Com isso, além de criar uma classe confusa, todos os desenvolvedores deveriam compartilhá-la, diminuindo a produtividade, aumentando o tempo de desenvolvimento e manutenção, além do risco de perda de código caso uma ferramenta de controle de versões não fosse usada. A solução correta não seria resolvida comprando tal ferramenta, mas criando um Session Facade para cada fluxo de trabalho ou até mesmo para cada bean quando necessário.

Não obedecer aos limites do caso de uso: Normalmente um session facade deve interagir apenas com os beans relacionados ao caso de uso correspondente. Caso seja necessário acessar dados de outros beans, deve-se obtê-los através do facade desse bean. Isso evita o acoplamento de Session Facades a Beans que não estão diretamente ligados. Como citado anteriormente, quanto menos acoplamento um sistema tiver mais fácil será sua manutenção.

Duplicação de código: Com o crescimento do projeto, é comum que alguns métodos possuam códigos duplicados em diversos Session Facades. Uma maneira de resolver o problema, aumentando a produtividade, seria criar camadas de serviços (session bean ou classes em java) que encapsulem as regras de negócios comum as entidades.

Infra-estrutura de software e hardware

Aumento de Performance: O uso deste pattern também influencia na performance uma vez que elimina as interações diretas entre os clientes e os objetos de negócios. As interações entre os objetos de negócios possivelmente estão em uma rede de alta velocidade ou até no mesmo container, sendo assim as interações entre eles são mais rápidas do que as interações entre os clientes e os objetos de negócios.

Segurança/Manutenção

Controle de acesso e transação: Session Facade representa um ponto central e ideal para gerenciar políticas de segurança e controle de transações, pois todas as requisições de aplicações clientes passam por ela. Outra vantagem está na segurança, uma vez que, por desatenção do programador ou por existir controle de transações ao longo dos Entities e Sessions, criar uma nova transação quando uma já está aberta, ocasionando um "dead lock". Somente o método inicial do Session Facade deve ser configurado como "RequiresNew" para criar uma transação e todos os outros métodos devem ter "Supports" para apenas suportarem a transação já existente.

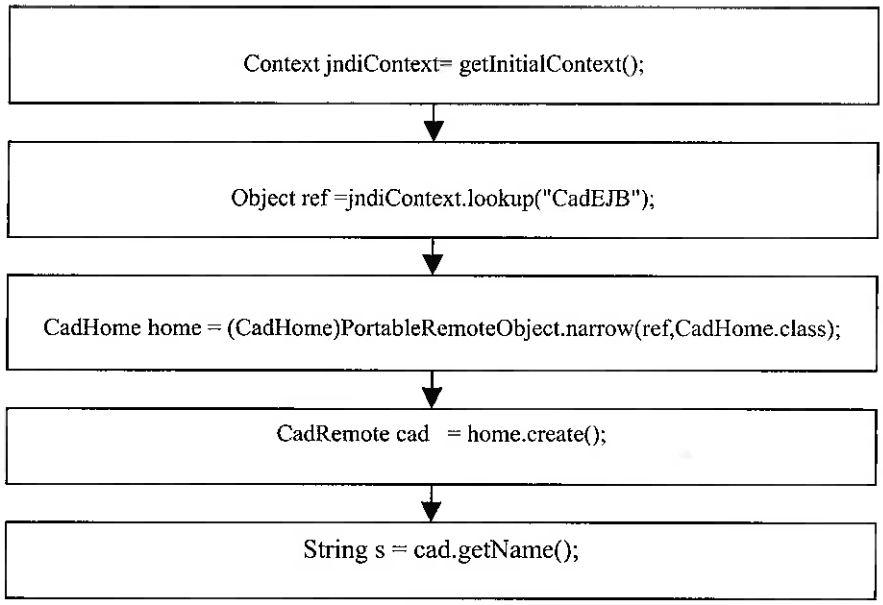
4.3 SERVICE LOCATOR

4.3.1 Contexto

Os clientes precisam fazer lookup e create para acessar os objetos de negócios, deixando suas interfaces mais complexas.

4.3.2 Problema

As aplicações clientes J2EE interagem com os componentes EJB para utilizar os serviços de negócios (métodos dos beans) e acesso ao banco de dados. Mas para isso, o cliente deve utilizar o serviço de lookup para conseguir a interface EJBHome do bean, podendo assim procurar por uma instância específica do bean (método EJBFind), removê-la (método EJBRemove) ou ainda criar uma nova instância desse bean (método EJBCreate). Criando uma nova instância, o cliente teria acesso a interface EJBObject, e somente agora o cliente pode acessar seus métodos de negócios. Esse processo é descrito em detalhes na figura 8.



Sendo este um processo comum, vários clientes utilizaram o serviço de lookup e o código ficará duplicado por diversas classes. Além disso, pode ser utilizado um cache para guardar a referência para as interfaces EJBHome, para não realizar lookup desnecessários, diminuindo o processamento da aplicação.

4.3.3 Motivação

- Clientes EJB precisam utilizar a API JNDI para fazer o lookup dos beans
- Processo de lookup e criação dos beans pode ser complexo e repetido em múltiplos clientes em uma aplicação.
- Esse processo pode utilizar muito recurso e impactar na performance da aplicação.
- Aplicações podem precisar restabelecer a conexão a um bean previamente acessado.
- Processo de criação do contexto inicial através do método getInitialContext() é proprietário bem como dependente do tipo de objeto que se deseja iniciar. Por exemplo, o contexto para JMS é diferente do contexto para entity bean.

4.3.4 Solução

Utilizar o pattern Service Locator para abstrair o uso do JNDI, escondendo a complexidade de criar o contexto inicial(initial context), o lookup do EJBHome e a recriação do EJBObject. Múltiplos clientes podem utilizar o Service Locator para reduzir a complexidade do código e aumentar a performance da aplicação provendo um serviço de cache. Este pattern centralizando as operações de lookup e create,

disponibiliza uma interface uniforme a todos os clientes além de facilitar possíveis modificações.

Na figura 9 é apresentado o diagrama de seqüência desse pattern. Note que o cliente deixa o Service Locator se encarregar de criar um Initial Context e o lookup.

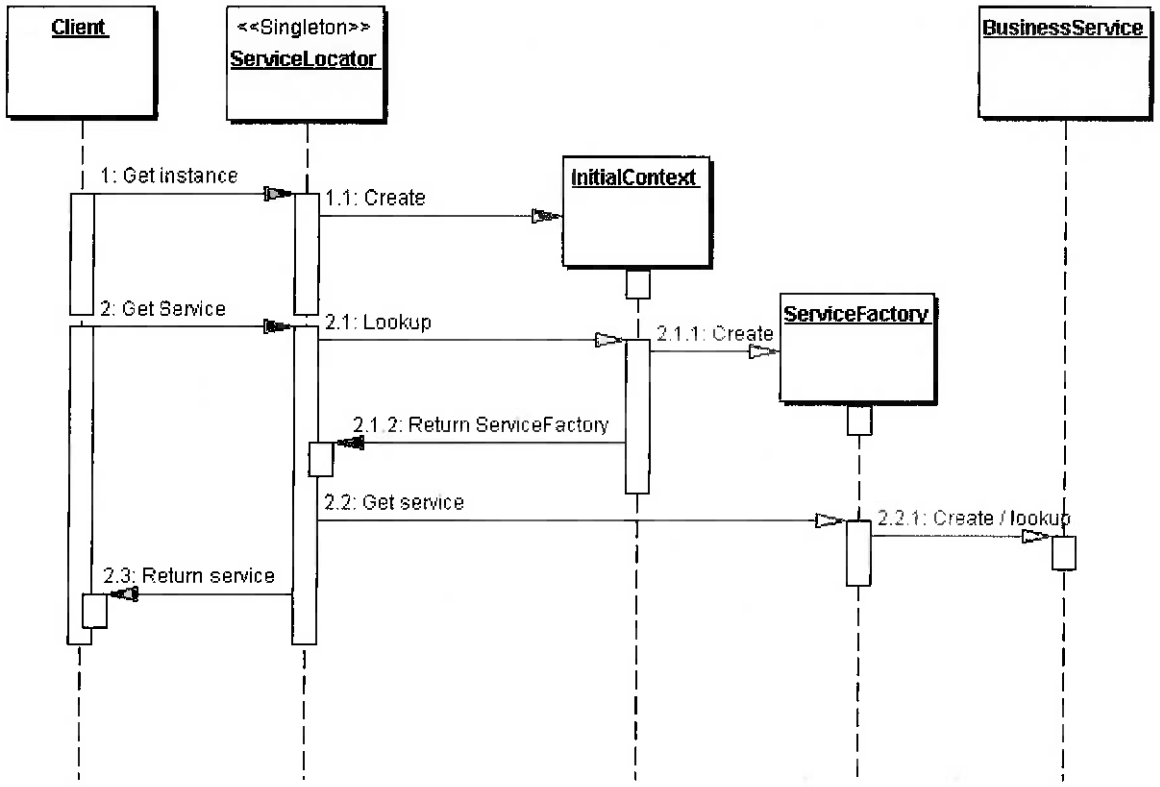


Figura 9 (direitos reservados a Sun Microsystem)

Elementos da figura 9

Cliente

Aplicação cliente invoca o Service Locator para localizar o bean que provê métodos a este cliente. Um bean pode utilizar o Service Locator para localizar outro bean.

Service Locator

O Service locator é a interface que abstrai toda a complexidade do processo de lookup e do create, provendo uma interface simples e uniforme. Isso reduz a complexidade do cliente.

InitialContext

O objeto Initial context é ponto de início para o processo de lookup e criação. Esse objeto pode variar dependendo do tipo do objeto de negócios requerido. Um Service Locator que provê serviço para vários tipos de objetos de negócios (enterprise beans e componentes JMS) utiliza múltiplos tipos de objetos de contexto.

ServiceFactory

O ServiceFactory representa um objeto que gerencia o ciclo de vida dos objetos Business Service. O objeto Service Factory para um enterprise bean é um objeto EJBHome. A partir dela, o ServiceFactory tem acesso aos métodos create, remove e find.

BusinessService

O BusinesService é o bean que o cliente deseja acessar. Esse objeto é criado, pesquisado (lookup) ou removido pelo ServiceFactory.

4.3.5 Análise segundo a ISO 9126

Desenvolvimento

Abstrair a complexidade: Este pattern encapsula a complexidade da aplicação cliente para fazer o "lookup" e instanciar um bean. Todos os clientes que necessitam fazer essas operações deverão utilizar a interface do pattern. Com isso, a maior parte do código fica na

interface, deixando a lógica dos clientes mais simples e diminuindo o tempo de codificação. Outra vantagem é reduzir a duplicação de código, uma vez que, o processo de lookup e create são iguais para todos os beans.

Manutenção

Acesso uniforme dos clientes: Como todos os "lookups" dos clientes serão feitos através desse bean, isso assegura que todos os clientes estão criando os beans da mesma forma e pela mesma interface. Isso é uma grande vantagem quando for necessário fazer uma modificação, pois apenas um bean deve ser alterado. Outra vantagem está na possibilidade de gravar em um log específico para lookup e create, caso seja necessário depurar erros de uma aplicação.

Desacoplamento: Como a lógica de negócios do lookup está na interface do pattern e não no cliente, ocorre uma diminuição no acoplamento entre o bean e o cliente, facilitando possíveis alterações na aplicação.

Infra-estrutura

Aumentar a performance do cliente: A interface do Service Locator pode guardar o contexto inicial(initial context) e a referência para a interface EJBHome, eliminando desnecessários processos de lookup, aumentando a performance do sistema.

4.4 BUSINESS DELEGATE

4.4.1 Contexto

Em aplicações multi-camadas e distribuídas é necessário fazer invocações de métodos remotos para enviar e receber dados através das camadas. Os clientes ficam expostos a complexidade de lidar com os componentes distribuídos.

4.4.2 Problema

A aplicação cliente faz requisições diretamente a métodos dos objetos de negócios criando um acoplamento entre eles. Qualquer mudança nos objetos de negócios resulta em alterações nos clientes.

Um outro fator de preocupação é a aplicação cliente fazer diversas requisições através da rede para acessar os objetos de negócios, sem utilizar cache. Isso pode prejudicar o tráfego da rede.

4.4.3 Motivação

- Camada cliente tem que acessar os serviços dos objetos de negócios
- Diferentes tipos de clientes têm que realizar esse acesso.
- É desejável minimizar o acoplamento entre os clientes e os objetos de negócios, escondendo detalhes de implementação do serviço, como lookup e acesso.
- Reduzir o tráfego da rede entre o cliente e os objetos de negócios

4.4.4 Solução

Utilizar o Business Delegate para reduzir o acoplamento entre a camada cliente e a camada dos objetos de negócios. A interface esconde os detalhes de implementação dos serviços de negócios, sendo uma abstração do lado do cliente, assim como o Session Facade é uma abstração no lado do servidor. Essa abstração tenta resguardar a aplicação cliente de possíveis modificações nos métodos dos objetos de negócios.

Alguns desenvolvedores acreditam que utilizando este pattern acrescenta uma complexidade desnecessária. Porém, utilizar este pattern, além do benefício principal de esconder os detalhes de implementação dos serviços de negócios, esta interface pode gerenciar as exceções EJB geradas e convertê-las em exceções conhecidas pela aplicação. Essa interface também pode implementar um sistema transparente de recuperação a falhas, não expondo um erro à aplicação cliente diretamente, sem antes tentar resolvê-la.

Dependendo da estratégia utilizada pode ser construído um mecanismo de cache de resultados e de referências, podendo ter algum ganho em performance evitando acessos desnecessários a métodos remotos.

Quando esse pattern é utilizado com o Session Facade, normalmente existe uma relação de um para um entre seus métodos. Para cada método existente na interface do Business delegate, deve existir um correspondente no Session Facade. Para cada novo método criado no Facade, a interface do Delegate deve ser modificada. Quando ocorrerem alterações nos sistemas, estas alterações devem ser feitas no lado do servidor evitando uma propagação para o cliente.

O Business Delegate pode utilizar um outro pattern chamando Service Locator para fazer abstrair o processo de lookup e create dos beans. Na figura 10 é apresentado o diagrama de seqüência desse pattern. Note que o cliente só faz acesso aos métodos do Business Delegate e este se encarrega de fazer o lookup e create dos beans.

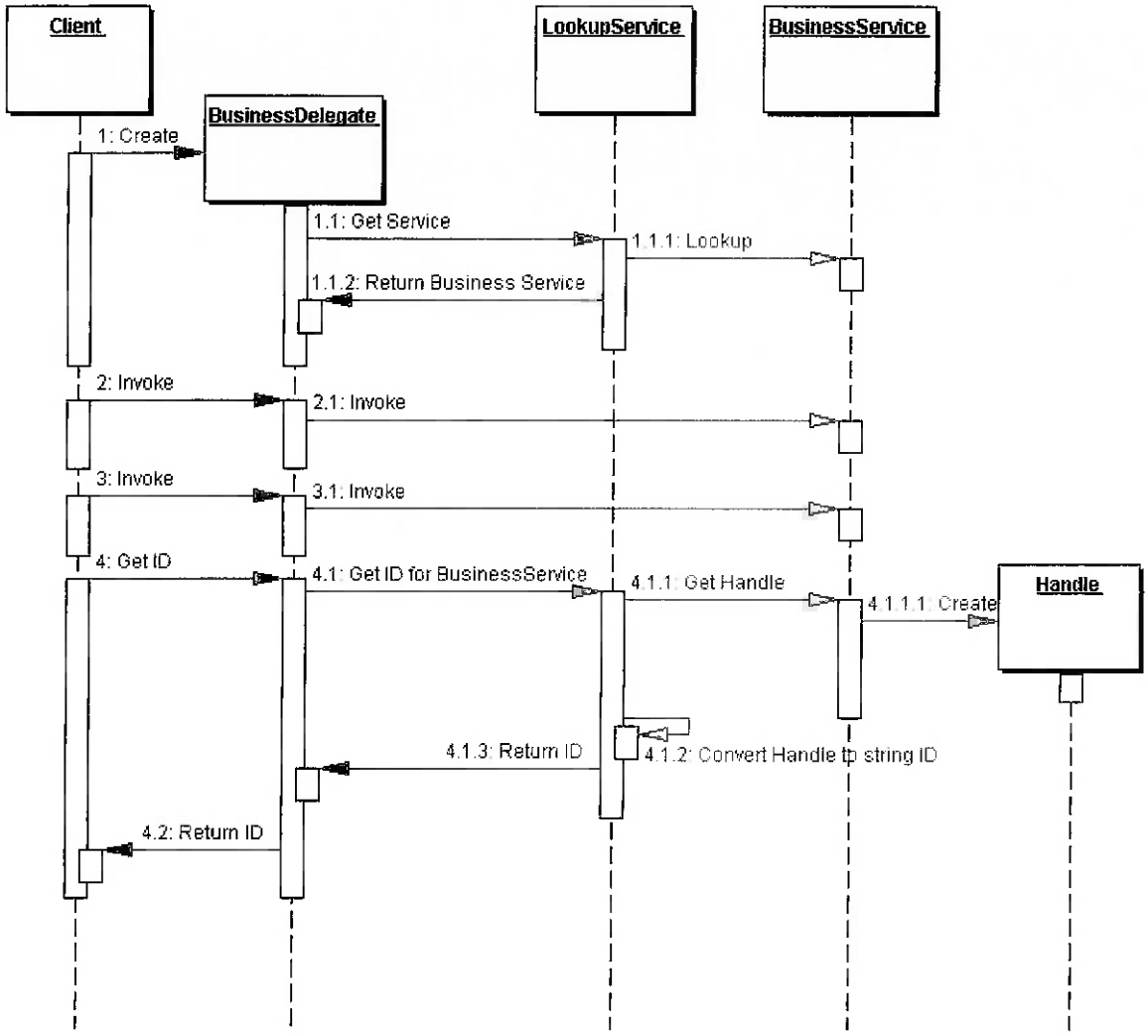


Figura 10(Sun Microsystem - http://java.sun.com.br/blueprints)

Elementos da figura 10

Cliente

É uma aplicação cliente que utiliza o business delegate para acessar um método do objeto de negócio.

Business Delegate

Essa interface é visível para o cliente e faz a ligação entre ele e os beans que possui os serviços de negócios.

LookupService

Esse pattern utiliza o LookupService para localizar um bean necessário. Ele encapsula os detalhes da implementação do lookup e retorna para o cliente a interface home(EJBHome) do bean desejado.

BusinessService

O Business Service é um componente de negócio, que pode ser implementado como um enterprise bean. Esse componente provê algum tipo de serviço para o cliente.

4.4.5 Análise segundo a ISO 9126

Desenvolvimento

Camada adicional: Para muitos, a interface do Business Delegate é uma camada adicional desnecessária entre a camada de apresentação e a camada de negócios, aumentando a complexidade e o tempo de programação. Porém as vantagens que esse pattern apresenta serão apresentados abaixo:

Manutenção

Reduzir acoplamento: Esse pattern reduz o acoplamento entre a camada de apresentação e a camada de negócios, escondendo sua

implementação. Cada método existente na interface do pattern possui um correspondente na camada de negócios. Quando ocorrer alguma alteração em métodos da camada de negócios, deve-se alterar o método da interface para evitar propagar essa mudança para o cliente. Utilizando esse pattern, a manutenção deve ser mais fácil e rápida. Contudo deve-se levar em consideração o tempo adicional para sua criação.

Segurança

Exceções conhecidas: As aplicações clientes acessam os métodos dos beans da camada de negócios por esse pattern e fica como função transformar as exceções EJB em exceções conhecidas para a aplicação cliente. Deixa de modo mais claro as exceções para a aplicação, evitando possíveis enganos de tratamento de erros genéricos e duplicados pelo cliente.

Infra-estrutura

Performance: O Business Delegate pode fazer cache de serviços para a camada de apresentação de requisições comuns. Dessa forma, há uma melhoria no desempenho da aplicação quando as informações necessárias estão no cache. Não havendo a necessidade de processar novamente o "request", a aplicação consegue responder mais rapidamente a aplicação cliente e consequentemente, processar mais transações por minuto.

Esconder métodos remotos: Como foi citado anteriormente, as aplicações clientes acessam os métodos da camada de negócios de forma transparente, podendo dar ao desenvolvedor dessas aplicações a

impressão errada que são apenas invocações a métodos locais. Esse pensamento pode acarretar em inúmeras chamadas para métodos remotos para realizar uma simples operação. Como conseqüência, haverá um aumento no tráfego da rede.

Desenvolvimento/Manutenção

Recuperação de erros: A interface desse pattern, sendo uma ligação entre os clientes e os métodos da camada de negócios, é um bom lugar para implementar recuperação automática de erros. Caso a recuperação ocorra normalmente, o cliente recebe a informação de modo transparente como se nada tive ocorrido. Porém, se a recuperação falhar, uma notificação de erro deve ser enviada ao cliente.
Esse tipo de abordagem é interessante para não expor todos os erros aos clientes. Implementar essa recuperação de erros requer um esforço adicional no desenvolvimento, aumentando o tempo total do projeto. Além disso, deve-se considerar essa recuperação, como sendo mais um método a ser testado e para ser feito a manutenção.

4.5 DATA ACCESS OBJECT - DAO

4.5.1 Contexto

O acesso a dados depende do tipo de dados e de sua implementação. O acesso ao banco de dados difere para cada tipo podendo ser relacional, orientado a objeto, arquivo, etc.

4.5.2 Problema

Em aplicações J2EE é comum que a aplicação utilize um repositório de dados em algum momento. Existem diversos tipos de repositórios de dados que utilizam diferentes APIs . Algumas aplicações podem utilizar um repositório em outro sistema, por exemplo, em um mainframe, ou até mesmo a tecnologia LDAP(Lightweight Directory Access Protocol) entre outros.

Tipicamente, uma aplicação utiliza o entity bean para acessar o banco de dados. Como foi visto na introdução, existem dois tipos de implementação de bean: BMP e CMP. No BMP existe claramente todo o tipo de comando para manipular o acesso aos dados, o programador é responsável pela criação desse código. Já no CMP, o programador não escreve um linha de código de acesso a dados, o EJBContainer é responsável por isso, sendo assim esse pattern só é válido quando está utilizando BMP.

Aplicações podem utilizar a API JDBC para acessar os dados residentes em um banco de dados relacional. Essa API provê ao programador acesso e manipulação dos dados através de comandos

SQL. Porém, cada empresa de banco de dados possui suas particularidades em determinados comandos.

A mudança é ainda maior quando se deseja utilizar diferentes tipos de repositórios de dados. Os mecanismos de acesso e suporte a API são bem diferentes entre banco de dados relacional e orientado a objetos. Quando a aplicação precisa acessar dados de um sistema legado ou de um mainframe, possivelmente utiliza-se uma API proprietária, com comandos diferentes dos usuais.

Esta diversidade de tipos de repositórios de dados que podem ser utilizados, cria uma dependência entre o código da aplicação e o código de acesso ao banco de dados. Essa dependência pode dificultar a manutenção caso seja necessário mudar o banco de dados que a aplicação utiliza.

4.5.3 Motivação

- Componentes como entity bean e session bean precisam acessar e guardar informações em um repositório de dados, que pode ser, relacional, orientado a objetos, sistema legado, LDAP entre outros.
- A API de acesso ao repositório pode não ser padrão ou ser proprietária
- A portabilidade dos componentes é diretamente afetada quando mecanismos específicos da API são introduzidos em seu código
- Componentes devem ser transparentes para o tipo de repositório atual e prover facilidade para eventuais modificações.

4.5.4 Solução

Usar o Data Access Object(DAO) para abstrair e encapsular todo o acesso ao repositório de dados. O pattern implementa os mecanismos de acesso requerido para obter e guardar os dados no banco de dados. O banco de dados poder ser relacional, orientado a objetos, LDAP, etc.

Os componentes de negócios utilizam uma interface simples e não tem conhecimento de qual banco de dados está utilizando. No código 3 foi implementado um bean onde todo o acesso ao banco de dados está no código da classe.

Código 3

```
public class Monografia implements javax.ejb.EntityBean
{
    public Integer ejbCreate(nome, autor,  topico, versao) throws
                                                    CreateException
    {
        this.id = this.getId();
        this.nome = nome;
        this.autor=autor;
        this.topico=topico;
        this.versao=versao;
        Connection con = null;
        PreparedStatement ps = null;

        try
        {
            con = this.getConnection();
```

```
                ps = com.preparedStatement("insert into
                monografia(nome, autor,  topico, versao) values
                (?,?,?,?,?)");
                ps.setInt(1,id);
                ps.setString(2,nome);
                ps.setString(3,autor);
                ps.setString(4,topico);
                ps.setString(5,versao);

                if(ps.executeUpdate() != -1)
                {
                        throw new CreateException("Erro ao criar a
                        monografia de " + nome);
                }
                return id;
        }
        catch(SQLException e)
        {
                ;
        }
    }
}
```

Utilizando o DAO, não existe o código de acesso ao banco de dados, só o código para acessar os métodos do DAO que fará o acesso ao banco. Esses métodos normalmente são para o bean criar, remover e atualizar os valores do banco. O código 4 apresenta somente a utilização de um método de criação de uma nova instância.

Código 4

```
public class Monografia implements javax.ejb.EntityBean
{
```

```
    public Integer ejbCreate(nome, autor,  topico, versao)
                                                throws CreateException
    {
        this.id = this.getId();
        this.nome = nome;
        this.autor=autor;
        this.topico=topico;
        this.versao=versao;

        //
      DAOFactory oracleFactory = DAOFactory.getFactory(oracle);
    MonografiaDAO moDAO = oracleFactory.getMonografiaDAO();

        Integer id = null;
        id = moDAO.insert(nome,autor,topico,versao);
        if(id == null)
        {
            throw new CreateException("Erro ao criar a
                                        monografia de " + nome);
        }
        return id;
    }
}
```

Na figura 11 é apresentado o diagrama de seqüência desse pattern. Note que o objeto de negócios não acessa diretamente o repositório(datasource). Esse pattern ainda utiliza-se de outro pattern o Value Object para retornar os valores ao bean.

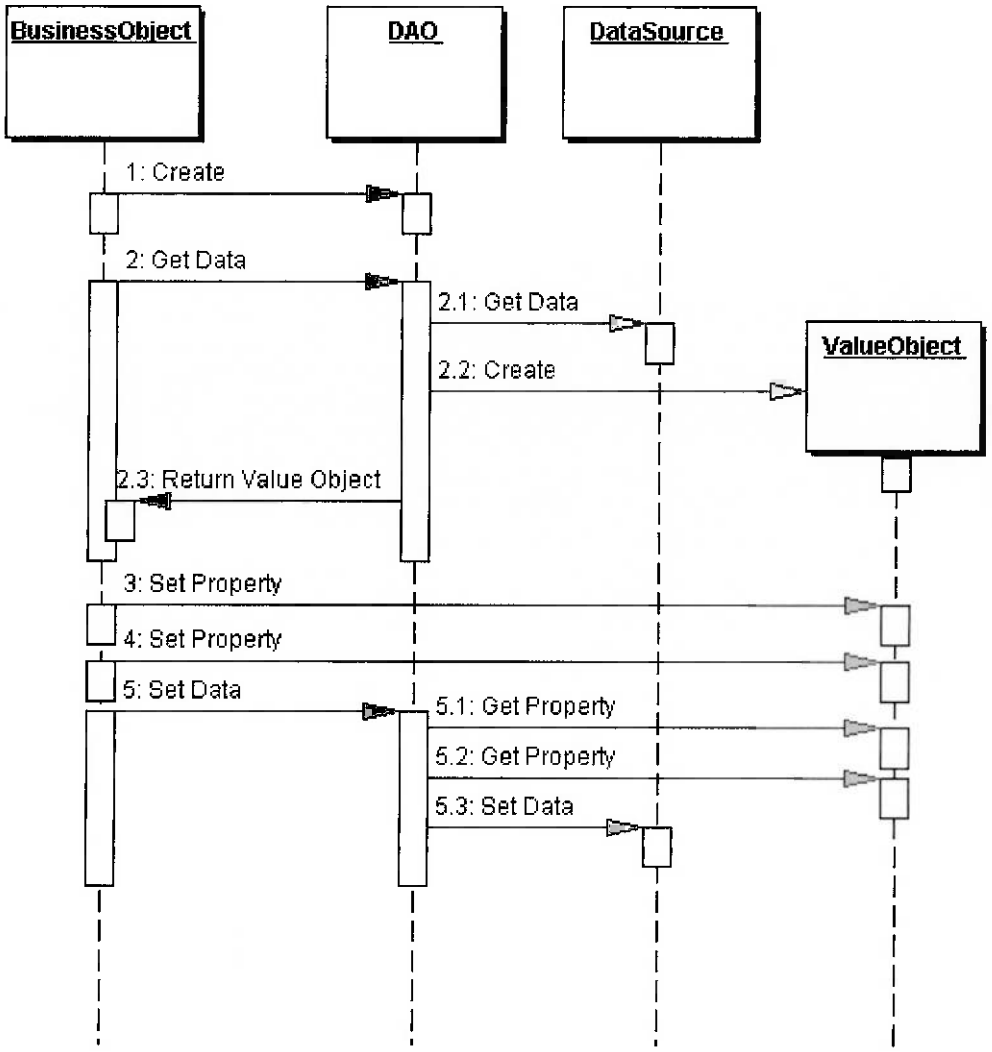


Figura 11(Sun Microsystem - http://java.sun.com.br/blueprints)

Elementos da figura 11

BusinessObject

O Business Object nesse caso é o cliente que requisita os dados ao DAO. O business object pode ser implementado como um entity ou session bean, ou qualquer outro objeto java.

DataAccessObject

Esse é o objeto principal desse pattern, e abstrai a relação que existe entre o BusinessObject e o datasource. Disponibiliza métodos de

acesso ao banco de dados como insert, delete e update. O business object invoca esse métodos sem ter o conhecimento de onde esses dados estão.

Datasource

Esse objeto representa o repositório de dados podendo ser um banco de dados relacional, orientado a objetos, um repositório XML, LDAP, etc.

ValueObject

O Value Object foi o primeiro pattern abordado nesta monografia, ele representa um objeto "serializável" possuindo as informações necessárias ao business object.

4.5.5 Análise segundo a ISO 9126

Manutenção

Transparência no acesso: Esse pattern é uma nova camada que é inserida entre os objetos de negócios e o Banco de Dados. Os objetos da camada de negócios podem usar o banco de dados de forma transparente através da interface do pattern, sem a necessidade de conhecer detalhes do acesso aos dados.

Isso faz com que não exista uma dependência entre os objetos de negócios e a implementação do banco de dados. Essa independência é realmente importante quando se deseja utilizar um novo banco de

dados, havendo mudanças somente na interface DAO, não propagando para os objetos de negócios.

Simplificar código: Na camada deste pattern fica todo o código de acesso ao banco de dados. Para os objetos de negócios basta acessar seus métodos de forma transparente para obter os valores desejados. Isso faz com que o código dos objetos de negócios seja simplificado, facilitando o entendimento do código. Este fator é importante em manutenções futuras, quanto mais rápido se entende o código, mais rápido é feita a manutenção e com menos risco de errar.

Desenvolvimento

Não utilizar CMP: O EJB Container gerencia todo o acesso aos dados quando se utiliza CMP - "container managed persistence", não sendo adequado aplicar esse pattern nesse escopo. O DAO é interessante ser utilizado quando o modelo de persistência é BMP - "bean managed persistence", onde todo o código de acesso aos dados é escrito pelo desenvolvedor.

5 RESULTADOS OBTIDOS

O capítulo 4 mostrou as vantagens e desvantagens em se utilizar os patterns Value Object, Session Facade, Service Locator, Business Object e Data Access Objects, com base no livro Core J2EE Patterns de Alur,Crupi, Malks(2001).

O capítulo 5 tem como objetivo verificar essas afirmações. Para isso, o autor se baseou na experiência pessoal de quase um ano utilizando a tecnologia J2EE e na implementação de todos os patterns.

Serão apresentados trechos de códigos somente para facilitar a visualização das vantagens e desvantagens dos patterns na arquitetura J2EE apresentadas no capítulo 4. As regras de negócios criadas para os beans são fictícias e não são exemplos para a criação de um sistema.

É interessante perceber as diferenças entre os códigos que utilizam patterns e os códigos sem sua utilização. Assim é mais fácil compreender as vantagens e desvantagens dos patterns.

5.1 Value Object

Esse pattern apresenta como vantagem principal a redução significativa de chamadas remotas que uma aplicação cliente faz a um servidor de aplicação. Para verificar, será apresentado uma aplicação cliente sem utilizar Value Object (código 5) e o mesmo cliente utilizando-o (código 6) .

Para esse exemplo, foi criado o bean Monografia que possui 7 atributos e a classe cliente que vai acessá-lo. Sem a utilização desse pattern, seria necessário acessar os 7 métodos "getters" dessa classe para ler todos os atributos. Cada chamada a esses métodos é uma requisição remota feita ao servidor, aumentando o tráfego na rede e o número de requisições que o servidor recebe. O código 5 demonstra isso:

Código5

```
Context jndiContext = getInitialContext();
Object ref = jndiContext.lookup("jndi/MonografiaEJB");
MonografiaHome home = (MonografiaHome)
        PortableRemoteObject.narrow(ref,MonografiaHome.class);
monografia  = Home.findByPrimaryKey(id);

//Chamadas remotas
monografia.getAutor();
monografia.getTema();
monografia.getAno();
monografia.getOrientador();
monografia.getDisciplina();
monografia.getFaculdade();
monografia.getTopicos();
```

Utilizando esse pattern o cliente passa a utilizar apenas um método. O retorno desse método é um objeto serializável com todos os atributos da classe Monografia. O objeto Value Object disponibiliza métodos getters para acessar seus atributos, porém esse objeto já está na máquina cliente e as requisições serão locais. Sendo assim, o cliente faz apenas uma requisição remota ao servidor para obter todas os atributos de uma classe, melhorando a performance do sistema. O código 6 apresenta a nova implementação do cliente .

Código 6

```
Context jndiContext = getInitialContext();
Object ref = jndiContext.lookup("jndi/MonografiaEJB");
MonografiaHome home = (MonografiaHome)
PortableRemoteObject.narrow(ref,MonografiaHome.class);
Monografia mon  = Home.findByPrimaryKey(id);
//chamada remota
MonografiaVO moVO = mon.getData();
//chamadas locais no objeto serializável
moVO.getAutor();
moVO.getTema();
moVO.getAno();
moVO.getOrientador();
moVO.getDisciplina();
moVO.getFaculdade();
moVO.getTopicos();
```

O código 7 apresenta o bean que cria o Value Object e repassa ao cliente pelo método getData(). Esse método é bem simples, criando uma instância de MonografiaVO com os atributos da classe Monografia.

Código 7

```
public class Monografia implements javax.ejb.EntityBean
{
        ;

        public MonografiaVO getData()
        {
                return new MonografiaVO (this.id, this.autor, this.tema,
                this.ano, this.orientador, this.disciplina, this.faculdade,
                this. topico)
        }
}
```

É recomendável utilizar esse pattern até mesmo quando a aplicação cliente é um outro bean que esteja no mesmo container. Não ocorrerá ganho de performance se a versão do EJB for a 2.0 utilizando interfaces locais. O maior ganho será na organização do código. Já no EJB 1.1, como toda requisição a um bean é remota, mesmo estando no mesmo container, a utilização desse pattern torna-se essencial.

Os códigos 5 e 6 apresentaram aplicações clientes que precisam obter um objeto cópia do bean para ler todos os seus atributos de uma só vez. Quando uma aplicação cliente precisar ler e atualizar esses valores, é necessário inserir os novos valores dos atributos do objeto cópia (MonografiaVO). Após isso, o cliente deve invocar o método setData de um Session Facade que requisita localmente o método setData do bean passando o objeto atualizado como parâmetro. Esse método é apresentado no código 8.

Código 8

```
public class Monografia implements javax.ejb.EntityBean
{
      public void setData(MonografiaVO moVO)
      {
            this.id,= moVO.getId();
            this.autor = moVO.getAutor();
            this.tema = moVO.getTema();
            this.ano = moVO.getAno();
            this.orientador = moVO.getOrientador();
            this.disciplina = moVO.getDisciplina();
            this.faculdade = moVO.getFaculdade();
            this. topico = moVO.getTopico();
      }
}
```

Esse pattern apresenta problemas quando mais de uma aplicação cliente precisa atualizar os valores de um bean. Mecanismos de controle devem ser implementados e o tempo para codificá-los deve ser levado em consideração.

5.2 Session Facade

Talvez esse pattern seja o mais importante e o mais utilizado nas aplicações J2EE. Como benefícios principais está o desacoplamento entre as aplicações clientes e os beans, criação de um ponto central para controle de transação, bem como diminuição no número de acesso remoto aos métodos dos beans, melhorando a performance do sistema.

O código 9 é a implementação de um cliente que faz acesso diretamente aos beans para realizar a tarefa de finalizar a compra de um usuário. É fácil notar que quando não se utiliza esse pattern, o cliente precisa saber a regra de negócios da aplicação, realizando o processo de lookup e create de todos os beans envolvidos.

Código 9

```
public static void main(String[] args)
{
        Context jndiContext = getInitialContext();
        Object ref = jndiContext.lookup("jndi/EstoqueEJB");
        EstoqueHome home = (EstoqueHome)
        PortableRemoteObject.narrow(ref,EstoqueHome.class);
        Estoque est  = home.findByPrimaryKey(idProduto);
        //retirar um produto do estoque
        est.retiraUmProduto(idProduto);
        if(est.verificaQuantidadeMinima(idProduto))
        {
                //se existem poucas quantidades de um produto, o sistema
                de pedir mais produtos para o fornecedor
                Context jndiContext = getInitialContext();
```

```
            Object ref = jndiContext.lookup("jndi/ForncedorEJB");

            ForncedorHome home = (ForncedorHome)
            PortableRemoteObject.narrow(ref,ForncedorHome.class);

            Fornecedor forn = home.create(idProduto);
            forn.emitirPedidoForncedor();
        }
        //Emitir o pedido
        Context jndiContext = getInitialContext();
        Object ref = jndiContext.lookup("jndi/PedidoEJB");
        PedidoHome home = (PedidoHome)
        PortableRemoteObject.narrow(ref,PedidoHome.class);
        Pedido pedido  = home.create(idProduto,idUsuario,data);
        pedido.emitirPedido();
}
```

Os beans devem ter seus métodos expostos na interface remota para que qualquer cliente possa acessá-los. Isso é apresentado no código 10.

Código 10

```
public interface Pedido extends EJBObject
{
  public void emitirPedido() throws RemoteException
}

public interface Forncedor extends EJBObject
{
  public void emitirPedidoForncedor() throws RemoteException
}
```

```
public Estoque extends EJBObject
{
        public boolean verificarQuantidadeMinima(Integer
        idProduto) throws RemoteException
}
```

No código 11 é apresentado o mesmo cliente, porém acessando a interface de um Session Facade para realizar a mesma tarefa de finalizar o pedido de um usuário. A aplicação cliente precisa fazer apenas um lookup, o do Session Facade e invocar apenas um método remoto. Esse foi um exemplo simples, mas o número de chamadas remotas e lookup pode aumentar conforme a complexidade do sistema, prejudicando a performance do sistema.

Código 11

```
Context jndiContext = getInitialContext();
Object ref = jndiContext.lookup("jndi/SessionCompraEJB");
SessionHome home = (SessionCompraHome)
    PortableRemoteObject.narrow(ref,SessionCompraHome.class);
SessionCompra sessCompra  = home.create();
//apenas uma chamada remota
sessCompra.finalizarPedido(idProduto,idUsuario,data);
```

É importante notar as diferenças nas interfaces usadas nos códigos 9 e 11. No código 9, as interfaces estendiam de EJBObject, e no código 11 estendem de EJBLocalObject, sendo acessível apenas para o Session Facade. Isso facilita a visualização do código e a manutenção já que os clientes passam a usar apenas uma interface remota que encapsula todo o processo.

Pode-se imaginar que se deseja realizar uma tarefa simples e habitual de mudar o nome de um bean ou acrescentar outro bean na regra de negócio. Utilizando o Session Facade, a maioria das mudanças não teria impacto sobre os clientes. Isso é importante porque não se sabe quantos clientes uma aplicação pode ter, nem onde esses clientes estão. Sem utilizar esse pattern, qualquer alteração nas regras de negócios terá impacto a todos os clientes que estão utilizando aquele bean.

Como ponto central dos processos, a interface do Session facade torna-se o lugar ideal para gerenciar as transações de forma segura. Com isso, os clientes que antes eram obrigados a fazer esse gerenciamento de forma não confiável passam a utilizar esse recurso através do Session Facade.

Pode-se utilizar a facilidade do XDoclet inserindo comentários nos métodos dos beans que precisam ter suporte a transação. Durante a compilação, o XDoclet lê esses comentários e cria o código necessário no ejb-jar.xml. Um exemplo disso está no comentário 1, que determina que toda vez que o método for chamado, uma nova transação será criada

Comentário 1

```
/**
* @ejb:create-method     view-type="local"
* @ejb:transaction       type = "RequiresNew"
*/
```

É importante para o desenvolvedor manter um Session Facade para cada caso de uso, ou conjunto de casos de uso similares. Caso

contrário o código do Session Facade ficará confuso, já que todos os métodos disponíveis para as aplicações clientes estarão em apenas uma classe. Outra desvantagem é que existe a possibilidade de mais de uma pessoa atualizar esse código, causando erros ou perda de produtividade.

5.3 Service Locator

O processo de criação do Initial Context é proprietário, quando um cliente deseja acessar um bean em um servidor de aplicação como o Jboss ou weblogic, os valores dos parâmetros do Initial Context mudam.

Uma das funções do Service Locator é encapsular esse processo. Alguns servidores de aplicação como o JBoss e o Weblogic da BEA, possuem um arquivo com os parâmetros essenciais para criar o contexto inicial. O código 11 apresenta esse arquivo.

Código 11
jndi.properties (JBoss)

```
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
java.naming.provider.url=localhost
```

weblogic.properties (Weblogic)

```
java.naming.factory.initial=weblogic.jndi.WLInitialContextFactory
java.naming.provider.url= t3://hostname:port
```

Com esse arquivo o programador consegue um desacoplamento entre os beans que precisam criar o contexto inicial da sua implementação proprietária. É interessante perceber que o Service Locator vai além da criação do contexto inicial e seu uso apenas para esse objetivo não é vantajoso.

Esse pattern pode guardar o contexto inicial, não realizando o processo de criação desnecessariamente. Ele pode estar no lado dos clientes que vão acessar os beans ou no lado do server. Implementando o Service Locator como Singleton, existirá apenas uma instância da classe e a variável InitialContext, um só valor. O construtor da classe é privado, não sendo acessível para outra classe criar uma nova instância. Para isso, o cliente deve invocar o método getInstance(). O código 12 apresenta parte dessa classe.

Código 12

```
public class ServiceLocator
{
   private static ServiceLocator serv;
   InitialContext context = null;

   private ServiceLocator()  throws ServiceLocatorException
   {
       try{
              context = new InitialContext();
       }
       catch(NamingException ne){
            throw new Exception(ne);
       }
   }

  public static ServiceLocator getInstance() throws Exception
  {
     if (serv == null){
      serv = new ServiceLocator();
     }
     return serv;
  }
}
```

O Service Locator pode ainda fazer cache das interfaces EJBHome e EJBObject. O mais viável é guardar a interface EJBHome, pois ele não é a referência para um bean específico, mas para um grupo de bean. O EJBHome é a referência para o bean chamado MonografiaEJB, enquanto a interface EJBObject é a referência para a instância do bean MonografiaEJB que possui o id =10 por exemplo.

O código 13 apresenta um método getHome que verifica se já existe uma interface EJBHome no Map, e caso não exista, cria uma nova e a guarda no Map.

Código 13

```
public EJBHome getHome(String name, Class clazz)  throws
ServiceLocatorException
{
   try{
      if (map.containsKey(name)){
            EJBHome home = (EJBHome) map.get(name);
      }else{
            Object objref = context.lookup(name);
            EJBHome home =
           (EJBHome)PortableRemoteObject.narrow(objref, clazz);
            map.put(name,home);
       }
      return home
   }
   catch(NamingException ex){
    throw new Exception();
   }
}
```

Com o cache do Initial Context e das interfaces EJBHome, a aplicação deixa de realizar dois processos "pesados", melhorando a performance do sistema. No capítulo 4, figura 8, foi apresentado esse fluxo e a implementação seria similar com o código 14 abaixo.

Código 14

```
Context jndiContext = getInitialContext();
Object ref = jndiContext.lookup("jndi/MonografiaEJB");
MonografiaHome home = (MonografiaHome)
PortableRemoteObject.narrow(ref,MonografiaHome.class);
Monografia mon  = Home.findByPrimaryKey(id);
String name = Mon.getNameAutor();
```

No código 15 é apresentado a implementação do cliente utilizando esse pattern. Note que a simplificação no código é mínima, a grande vantagem está no cache do Initial Context e da interface home do bean.

Código 15

```
ServiceLocator serviceLocator = ServiceLocator.getInstance();
ProjectHome projectHome = (ProjectHome)serviceLocator.getHome
(Monografia,Monografia.class);
Monografia mon  = Home.findByPrimaryKey(id);
String name = Mon.getNameAutor();
```

No Jboss existe uma outra maneira mais fácil de implementar o cache das interfaces EJBHome. Todas as interfaces que o Session Facade(Stateless) for utilizar deve ter seu lookup no método setEntityContext(). Esse é um método CallBack que é invocado toda vez que o Session vai para o pool. Enquanto esse bean estiver no pool, esse método não será mais invocado. Em testes realizados após a

primeira chamada para esse Session, não foi necessário realizar o lookup dos beans. Essa estratégia foi implementada somente no jboss. O código 16 mostra esse método.

```
public void setSessionContext(SessionContext ctx)
{
	try
	{
		monHome = (MonLocalHome)ic.lookup(MonografiaEJB);
		userHome = (UserLocalHome)ic.lookup(UserEJB);
	}
	catch(Exception e)
	{
	}
}
```

5.4 Business Delegate

Esse pattern tem como vantagem principal diminuir o acoplamento entre a aplicação cliente e os objetos de negócios, além da "tradução" das exceções geradas pelos objetos de negócios. Qualquer exceção que um bean gerar, o cliente receberá um RemoteException. Pode-se imaginar que em determinadas situações, precisa-se conhecer qual exceção ocorreu para executar determinada tarefa, sem que o cliente perceba que isso esteja acontecendo.

Em uma aplicação bancária que possui diversos tipos de aplicações clientes como: usuário de internet, caixa do banco, máquina de auto-atendimento, quando ocorrer uma exceção, possivelmente exista um tratamento diferente para cada aplicação. Enquanto um deve mostrar uma tela de erro e mandar um e-mail para central reportando o erro, no caixa do banco apenas uma mensagem deve ser apresentada.

Esse pattern pode ser utilizado junto com o Service Locator no lado do cliente, para encapsular o processo de Initial Context e Lookup dos beans e o Session Facade no lado do servidor para encapsular as requisições e o fluxo de trabalho. O código 17 apresenta o código de um Business Delegate.

Código 17

```
public class ResourceDelegate
{
  private SessionFacade sess;
  public BusinessDelegate() throws Exception
  {
```

```
    try{
       SessionFacadeHome home = (SessionFacadeHome)
       ServiceLocator.getInstance().getHome("BankFacade",
                                                homeClazz);
       sess = home.create();
    }
    catch(RemoteException ex){;}
}

public void CheckAmount()
{
    try{
       sess.checkAmount();
   }
  catch(Exception e) {
       String exc = TranslateException(ex);
       if (exc.equals('Falta dinheiro'))
       {
            //verificar qual aplicação chamou esse método
            if ( ctx.getCallerPrincipal().getName().equals('caixa') )
            {
                 showFormtogetMoney();
                 getMoreMoney();
            }

            if(ctx.getCallerPrincipal().getName().equals('auto-
                                                atendimento') ){
                 showErrorToUser('Falta Dinheiro');
                 sendMail('Falta Dinheiro');
            }
       }
   }
}
```

A função de encapsular a tradução de exceções pode não ser necessária em algumas aplicações. Sendo assim, a utilização desse pattern somente com essa finalidade não é aconselhável. Esse encapsulamento pode ser útil quando a aplicação possui mais de um tipo de cliente e cada um deles teria que implementar uma rotina para tradução de exceções. Para simplificar o código do cliente e diminuir a duplicação de código, é mais interessante criar uma interface(Business Delegate) que encapsule essa tradução e sirva para todos os clientes.

O Business Delegate pode ainda fazer cache de algumas informações para que a aplicação cliente não precise fazer requisições desnecessárias aos objetos de negócios. Com isso, o Business Delegate funcionaria como um proxy, talvez essa seja a função mais utilizada pelos desenvolvedores no Business Delegate. A melhoria na performance do sistema dependerá da quantidade de requisições que serão economizadas.

Para informações que não são alteradas freqüentemente, essa abordagem não tem problema. Para as que sofrem alterações rapidamente, mecanismos para atualizar o cache deverão ser implementados. Pode ser utilizando JMS, enviando uma mensagem para o cliente avisando que o cache está inválido ou com um a thread configurada para a cada t segundos atualizar as informações. É importante ter conhecimento disto, pois é adicionada uma complexidade a mais no lado do cliente que precisa atualizar as informações e utilizando JMS, adiciona-se complexidade no lado do servidor também.

Durante a fase de codificação, o Business Delegate pode ser utilizado para testar a aplicação cliente sem a necessidade da regra de negócio

no lado do servidor esteja implementado. Imagine que duas equipes estejam envolvidas em um projeto, uma cuidando dos clientes e a outra dos beans. A equipe cliente poderia configurar os métodos do Business Delegate para retornar os dados necessários para que a equipe possa testar a aplicação cliente sem que os beans estejam implementados.

5.5 Data Access Object

Para verificar a utilização desse pattern, foi feita a mudança de banco de dados de uma aplicação para identificar quais alterações seriam necessárias. A aplicação estava rondando no servidor de aplicação JBOSS, que é gratuito. O banco de dados era o HiperSonic, também gratuito e que é instalado durante a instalação do JBOSS. A mudança seria alterar o banco dados padrão da aplicação de HiperSonic para Oracle.

No código 04 está a implementação de um bean utilizando HiperSonic:

```
public class Monografia implements javax.ejb.EntityBean
{
      public Integer ejbCreate(nome, autor,  topico, versao) throws
                                                        CreateException
      {
            this.id = this.getId();
            this.nome = nome;
            this.autor=autor;
            this.topico=topico;
            this.versao=versao;

            Connection con = null;
            PreparedStatement ps = null;

            try
            {
                  con = this.getConnection();
                  id   = seq.proximoValor();
```

```
                ps = com.preparedStatement("insert into
        monografia(nome, autor,  topico, versao) values (?,?,?,?,?)");
                ps.setInt(1,id);
                ps.setString(2,nome);
                ps.setString(3,autor);
                ps.setString(4,topico);
                ps.setString(5,versao);

                if(ps.executeUpdate() != -1)
                {
                        throw new CreateException("Erro ao criar a
                                                monografia de " + nome);
                }
                return id;
            }
        }
}
```

Esse seria o mesmo código para utilizar Oracle ou qualquer outro banco de dados relacional. Caso a aplicação utilize funções proprietárias, será necessário fazer a adaptação para funções correspondentes do novo banco de dados. Utilizando comandos SQL padrões, fica garantida uma transição tranqüila entre bancos relacionais.

Nessa aplicação foi criado um bean para fornecer as chaves primárias. Esse bean executa duas funções no banco de dados: a primeira selecionar o valor do atributo que guarda o índice das chaves primárias e a segunda, atualizar o valor lido. Funções proprietárias

como "sequence" do Oracle, desempenham a mesma função, mas ficando dependente do banco de dados.

Na configuração do JBoss deve-se criar um arquivo que contenha os dados do novo servidor de banco de dados como: nome de usuário, senha, nome do banco, etc. No JBoss, esse arquivo tem o nome de oracle-service.xml No weblogic da BEA, essa configuração é feita via console de administração, mas também é bem simples. Além disso, deve-se copiar o driver jdbc do novo banco para o diretório de lib do servidor de aplicação.

Percebe-se que a utilização do pattern Data Access Object nesse caso, onde se utilizou um banco de dados relacional e utilizando comandos padrões SQL, não foi essencial e não trouxe grandes benefícios. Porém quando a mudança está no tipo de banco de dados e não no proprietário, esse pattern torna-se necessário.

Utilizando este pattern não existe uma dependência direta entre os entity beans e o Banco de Dados. O DAO encapsula todo o acesso ao banco, diminuindo o acoplamento e facilitando a manutenção quando se desejar mudar o banco de dados utilizado pela aplicação.

Para executar essa mudança poderia existir uma equipe que seria responsável em alterar todos os Data Access Object. Esta equipe não iria interferir na equipe de desenvolvimento por alterar beans que a equipe de desenvolvimento não precisa modificar. A equipe de mudança só precisa se preocupar em manter as interfaces locais com as mesmas assinaturas, mas implementando em outro tipo de repositório de dados. Assim que a equipe de manutenção fizer todas as alterações e disponibilizar os novos DAO, a equipe de desenvolvimento nem perceberá que a aplicação está utilizando outro repositório de dados.

6 CONCLUSÃO

Com base nos resultados obtidos, pode-se perceber que alguns patterns são vantajosos em determinadas situações, enquanto outros não. É errado pensar que para construir uma aplicação de qualidade, escalável, sem acoplamento, precisa-se utilizar todos os patterns de uma só vez.

Essa monografia apresentou apenas cinco patterns, porém existem outros que também podem ser utilizados. O objetivo desse trabalho foi mostrar que para implementar uma aplicação J2EE, deve-se antes conhecer e compreender os patterns, para obter vantagens como: desacoplamento, cache de objetos e dados, aumento na performance e facilidade na manutenção.

A utilização dos patterns J2EE por pessoas que não compreendem sua finalidade poderá afetar o projeto com um todo. Camadas (layers) adicionais desnecessárias, aumento na complexidade do código e no número de erros são algumas dessas conseqüências que poderão interferir no custo final e no prazo do projeto.

A tabela 1, apresenta de forma resumida e com base nos resultados obtidos, vantagens e desvantagens em se utilizar os cinco patterns abordados nessa monografia, além de sugerir quando cada um deles deve ser utilizado.

Tabela 1

	Vantagens	Desvantagens	Quando Usar
Value Object	- Diminuir o número de chamadas remotas que o cliente realiza para obter os atributos necessários dos beans. - Aumento na performance da aplicação	- Acrescentar complexidade quando mais de uma aplicação cliente puder atualizar um bean. Algum tipo de controle de versão deve ser implementado.	- Em qualquer tipo de aplicação que um cliente precisar obter os atributos de um bean. O cliente pode ser um bean ou uma aplicação cliente. - É recomendável utilizar entre beans EJB1.1, pois toda chamada, mesmo dentro do EJBContainer é remota.
Session Facade	- Diminuir o acoplamento entre a aplicação cliente e os beans. - Isenta a aplicação cliente de conhecer as regras de negócios. - Ótimo lugar para fazer o controle de acesso e de transação	- Quando se utiliza apenas um Session Facade para todos os casos de usos. Para cada caso de uso, ou para casos de uso similares, deve existir um Session Facade	- Esse pattern talvez seja uma unanimidade, e deve ser usado toda vez que existir uma aplicação cliente que acesse métodos dos beans. Pode ser usado tanto em pequenas aplicações como

	- Aumento da performance		em sistemas complexos.
Service Locator	- Cache das interfaces remotas. -Diminuir duplicidade de código. - Aumento na performance	- Utilizar esse pattern apenas para abstrair o processo de Initial context, já que quase todos os servidores de aplicação possuem um arquivo com essa configuração.	- Pode ser utilizado tanto na aplicação cliente como nos beans.
Business Delegate	- Diminuir o acoplamento entre os clientes e os beans. - Cache de informações. - Aumento na performance - Pode ser utilizado para traduzir as exceções geradas.	- Dar a idéia errada que os métodos invocados são locais e não remotos. - Incluir complexidade caso as informações guardadas precisem ser atualizas com freqüência.	- Quando se deseja fazer cache de informação do lado do cliente. - Quando existe a necessidade de tradução das exceções geradas para realizar alguma operação.
Data Access Object	- Tirar o acoplamento que existe entre os	- Inclusão de uma camada desnecessária	- Quando não há certeza de qual tipo de

	beans e a implementação do banco de dados	quando não existe a possibilidade de trocar o tipo de repositório de dados. - A mudança entre banco de dados relacional, também não necessita utilizar esse pattern. - Não pode ser utilizado com CMP.	repositório de dados será utilizado, ou quando já se sabe que o mesmo será modificado. - Não é possível utilizar com CMP

Além da utilização de patterns, outro ponto que deve ser considerado pelo desenvolvedor antes de implementar um projeto J2EE é conhecer os artifícios que a própria linguagem oferece, para otimizações e possuir interfaces de programação modernas. Configurar corretamente o servidor de aplicação assim como utilizar adequadamente os métodos CallBacks, são práticas que aumentam a performance de uma aplicação.

Com o intuito de aumentar a produtividade, deve ser utilizado ambientes de desenvolvimento (IDE) modernos que permitem realizar operações como "refactor". Com isso, alterações nos atributos e classes são feitos com mais segurança e agilidade. Atualmente existem ferramentas como o xDoclet que geram automaticamente os arquivos xml (descriptors). Esses arquivos tomam um tempo razoável do

programador para a sua criação e com o seu uso o tempo diminui bastante.

Para compilar o código, o programador pode utilizar ferramentas como o ant que facilitam e organizam esse processo. Antes de projetar uma aplicação J2EE, deve-se tomar cuidado em conhecer e analisar os patterns existentes, além de possuir ferramentas e ambientes de desenvolvimento adequadas.

Em um mercado tão competitivo como o de software, aumentar a produtividade e performance da aplicação pode significar uma economia de tempo, máquina e dinheiro. Investir no conhecimento das pessoas que trabalham na sua empresa para conhecerem e utilizarem corretamente os patterns é uma das formas de trazer esses benefícios.

7 BIBLIOGRAFIA

Core J2EE Patterns - Best Practices and Design Strategies
Deepak Alur – John Crupi – Dan Malks

Design Pattern – Advanced Patterns, Processes and Idioms
Floyd Marinescu

Enterpise JavaBeans
Richard Monson-Haefel

Sites

http://java.sun.com/blueprints/corej2eepatterns/Patterns/index.html
http://developer.java.sun.com/developer/technicalArticles/J2EE/corepatterns/
http://www.javaworld.com/javaworld/jw-06-2002/jw-0607-j2eepattern-p2.html
http://www.enteract.com/~bradapp/docs/patterns-intro.html
http://hillside.net/
http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/DEA2eTOC.html