

J931

Padrões
de Projeto J2EE

Padrões da Camada
de Integração

4

Helder da Rocha (helder@acm.org)  argonavis.com.br

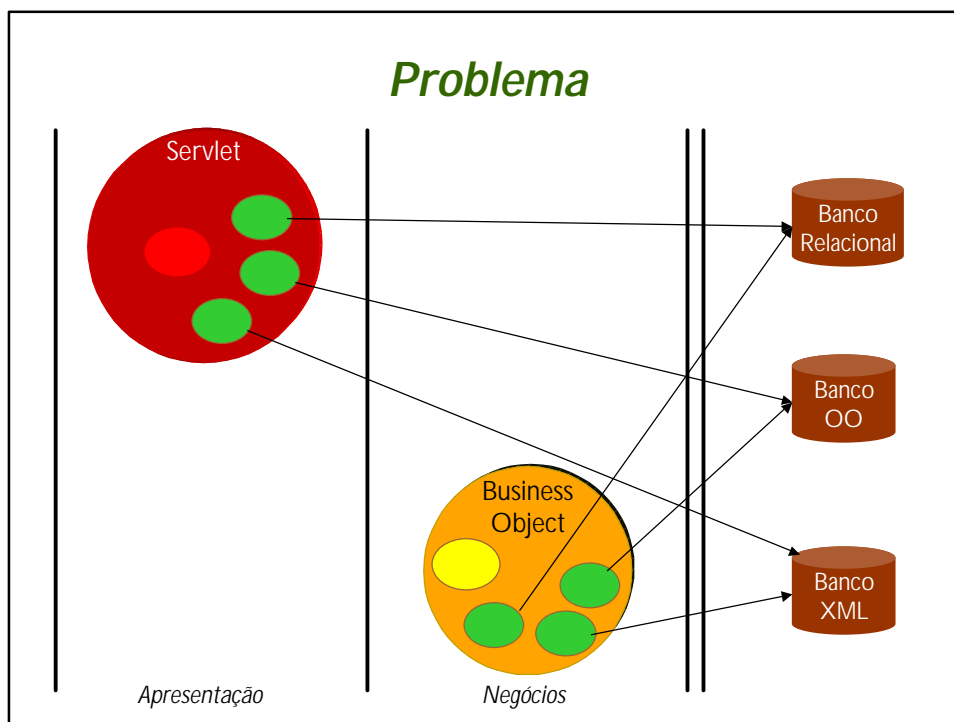
Introdução

- A camada de integração encapsula a lógica relacionada com a *integração do sistema com a camada de informação distribuída (EIS)*
- É acoplada com a camada de negócios sempre que esta camada precisar de dados ou serviços que residem na camada de recursos (dados)
- Os componentes nesta camada podem usar tecnologias de acesso aos serviços específicos que isolam (*JDBC, JMS, middleware proprietário, etc.*)

18

Data Access Object (DAO)

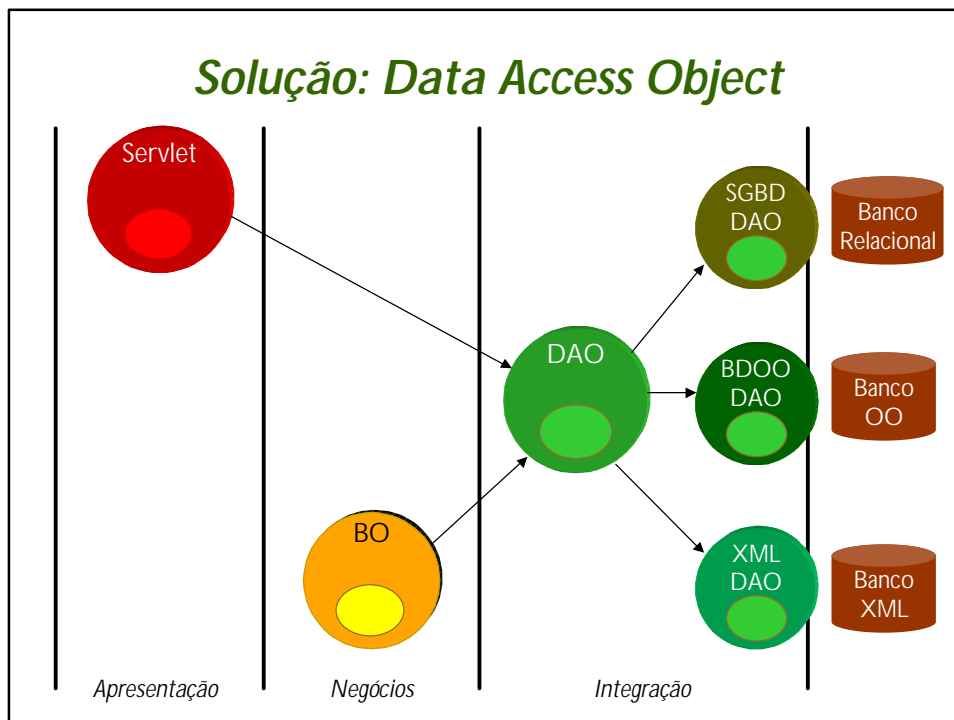
Objetivo: Abstrair e encapsular todo o acesso a uma fonte de dados. O DAO gerencia a conexão com a fonte de dados para obter e armazenar os dados.



Descrição do problema

- *Forma de acesso aos dados varia consideravelmente dependendo da fonte de dados usado*
 - *Banco de dados relacional*
 - *Arquivos (XML, CSV, texto, formatos proprietários)*
 - *LDAP*
- *Persistência de objetos depende de integração com fonte de dados (ex: business objects)*
 - *Colocar código de persistência (ex: JDBC) diretamente no código do objeto que o utiliza ou do cliente amarra o código desnecessariamente à forma de implementação*
 - *Ex: difícil passar a persistir objetos em XML, LDAP, etc.*

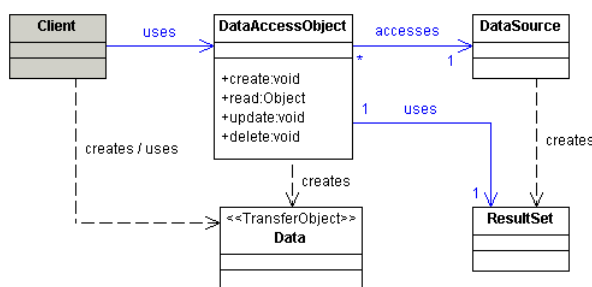
Solução: Data Access Object



Descrição da solução

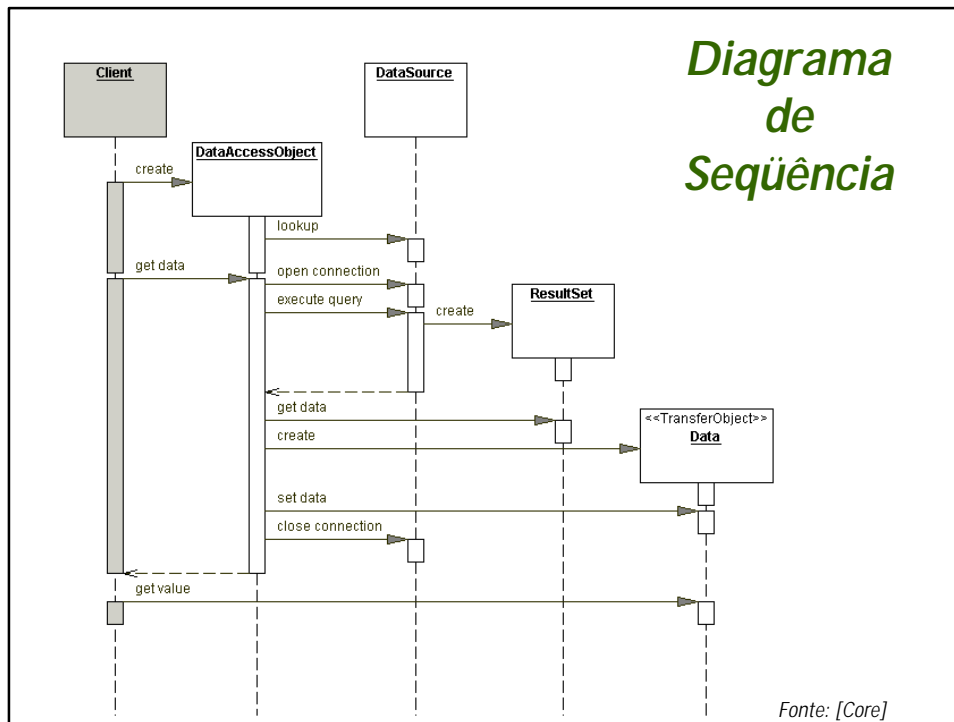
- **Data Access Object (DAO)** oferece uma interface comum de acesso a dados e esconde as características de uma implementação específica
 - Uma API: métodos genéricos para ler e gravar informação
 - Métodos genéricos para concentrar operações mais comuns (simplificar a interface de acesso)
- DAO define uma interface que pode ser implementada para cada nova fonte de dados usada, viabilizando a substituição de uma implementação por outra
- DAOs não mantêm estado nem cache de dados

Estrutura UML



- **Client**: objeto que requer acesso a dados: Business Object, Session Façade, Application Service, Value List Handler, ...
- **DataAccessObject**: esconde detalhes da fonte de dados
- **DataSource**: implementação da fonte de dados
- **Data**: objeto de transferência usado para retornar dados ao cliente. Poderia também ser usado para receber dados.
- **ResultSet**: resultados de uma pesquisa no banco

Fonte: [SIC]



Melhores estratégias de implementação

- *Custom DAO Strategy*
 - *Estratégia básica. Oferece métodos para criar, apagar, atualizar e pesquisar dados em um banco.*
 - *Pode usar **Transfer Object** para trocar dados com clientes*
- *DAO Factory Method Strategy*
 - *Utiliza Factory Methods em uma classe para recuperar todos os DAOs da aplicação*
- *DAO Abstract Factory Strategy*
 - *Permite criar diversas implementações de fábricas diferentes que criam DAOs para diferentes fontes de dados*

Conseqüências

- *Transparência quanto à fonte de dados*
- *Facilita migração para outras implementações*
 - *Basta implementar um DAO com mesma interface*
- *Reduz complexidade do código nos objetos de negócio (ex: Entity Beans BMP)*
- *Centraliza todo acesso aos dados em camada separada*
 - *Qualquer componente pode usar os dados (servlets, EJBs)*
- *Camada adicional*
 - *Pode ter pequeno impacto na performance*
- *Requer design de hierarquia de classes (Factory)*

Exemplos de DAO

- *DAO para cada Business Object*
- *DAO para serviços arbitrários*

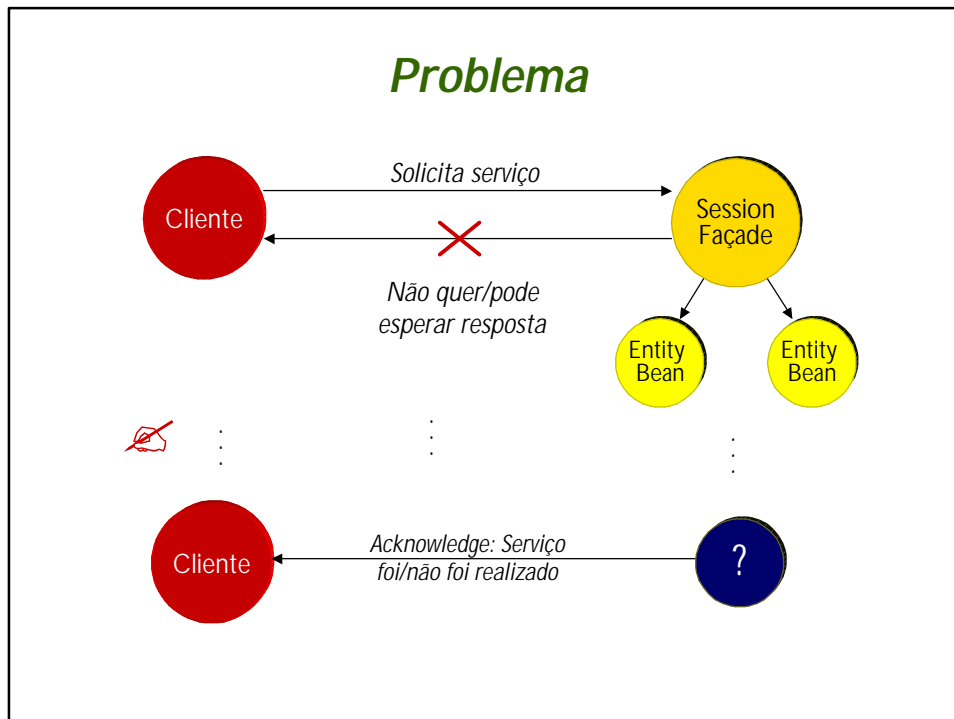
Exercícios

- 1. Analise a implementação das estratégias de Data Access Object
- 2. Analise o código do DAO existente (XML) e implemente um DAO e código para armazenar as mensagens no banco de dados:
 - a) Implemente a interface MessageBeanDAO
 - b) Implemente um mecanismo de seleção do meio de persistência escolhido através do web.xml e um Factory Method através do qual a aplicação possa selecionar o DAO desejado

19

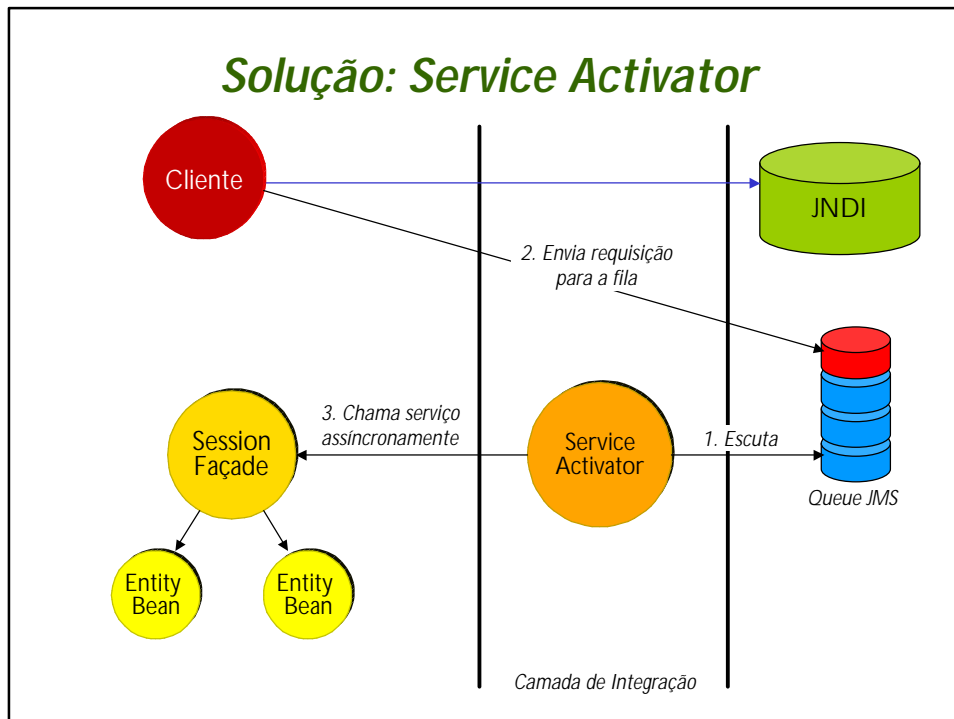
Service Activator

Objetivo: Receber requisições e mensagens assíncronas do cliente. Ao receber uma mensagem, o Service Activator localiza e chama os métodos de negócio necessários nos componentes para atender à requisição, de forma assíncrona.



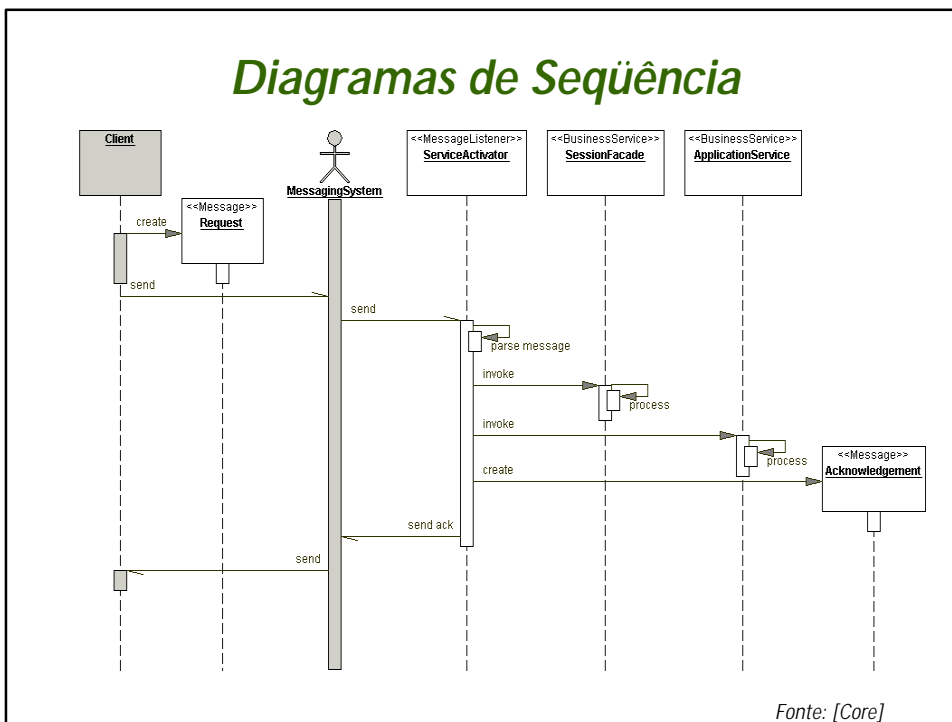
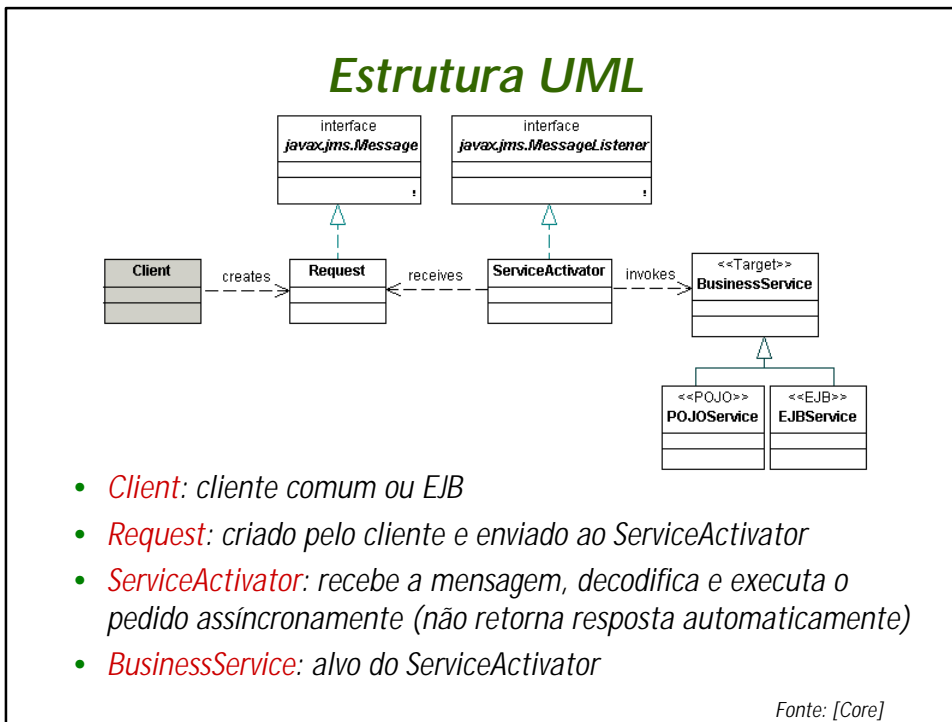
Descrição do problema

- Algumas aplicações precisam operar de forma **assíncrona** com um componente
 - Não pode esperar que uma requisição seja respondida ou não tem interesse na resposta (pelo menos não imediatamente)
- Acesso de componentes da camada de negócios da forma convencional (usando sua interface remota) é sempre realizado de forma síncrona
 - É preciso implementar um ponto de acesso no qual clientes possam **deixar requisições** para serem encaminhadas a EJBs logo que possível
 - EJBs devem poder "acordar" quando receberem a notificação e executar o serviço solicitado



Descrição da solução

- O Service Activator é um *listener JMS*
 - Pode ser implementado como um *Message-driven bean* ou como aplicação standalone
- Clientes que precisarem de chamar um serviço *assíncronamente* podem criar e enviar uma mensagem ao Service Activator
 - A mensagem é *enviada a uma fila JMS* usada pelo Activator
 - O Service Activator extrai as informações do serviço e localiza o componente que poderá executá-lo. Em seguida, chama os métodos necessários para atender à requisição do cliente.
 - Se o cliente desejar, o Service Activator pode enviar uma mensagem confirmando a realização do serviço com sucesso.



Estratégias de Implementação

- *POJO Service Activator Strategy*
 - *Standalone JMS Listener. Solução quando não se usa EJB.*
- *MDB Service Activator Strategy*
 - *Service Activator é um Message-Driven Bean*
- *Service Activator Aggregator Strategy*
 - *Permite dividir a tarefa em sub-tarefas que são executadas por processadores assíncronos separados*
- *Estratégias para envio de resposta ao cliente*
 - *Database Response Strategy: data polling pelo cliente*
 - *Email Response Strategy: e-mail enviado ao cliente*
 - *JMS Message Response Strategy: mensagem enviada*

Conseqüências

- *Integra JMS em aplicações corporativas*
 - *Em qualquer plataforma em que houver uma implementação JMS Service Activator pode ser implantado*
- *Permite processamento assíncrono para componentes da camada de negócios*
 - *Session e Entity Beans não podem ser chamados de forma assíncrona. Service Activator se coloca entre eles e o cliente para viabilizar isto*
- *Pode ser um listener JMS standalone*
 - *Não precisa ser implementado com MDB*
 - *Funciona em implementações antigas de EJB*
 - *Pode ser executado como um processo standalone*

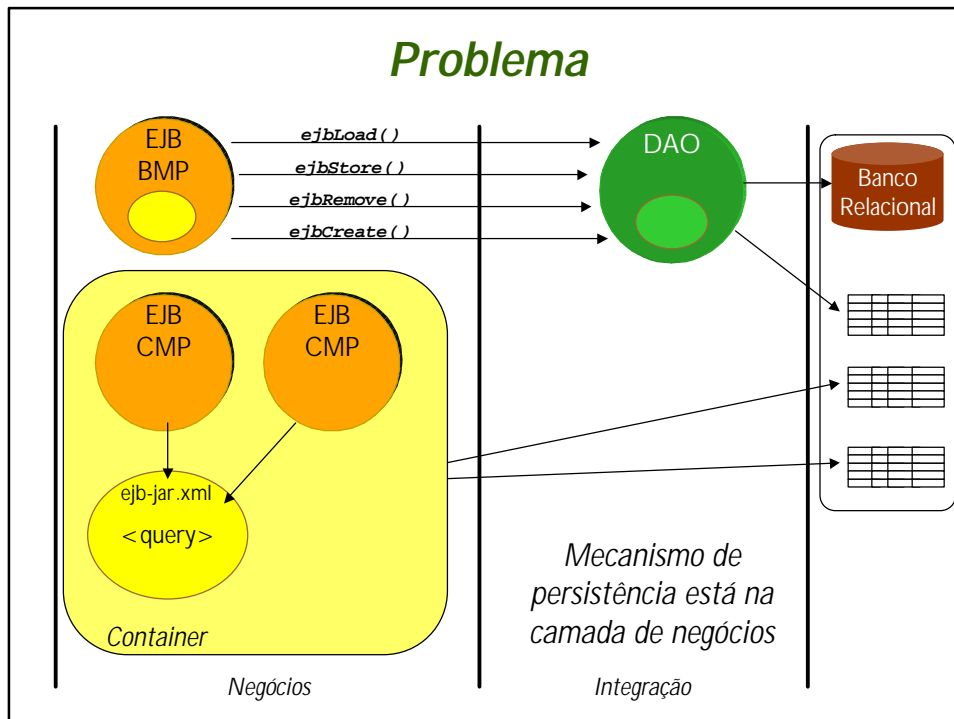
Exercícios

- 1. *Analise a implementação das estratégias de Service Activator*
- 2. *Implemente um Service Activator para chamar o serviço de "aumentar estoque de produtos"*
 - a) *Implemente-o como um MDB e configure-o*
 - b) *No cliente, crie uma mensagem que informe, através de um cabeçalho, a quantidade de produtos a ser adquirido e o código do produto.*
 - c) *No método onMessage(), utilize o código para localizar um bean e faça a alteração.*

20

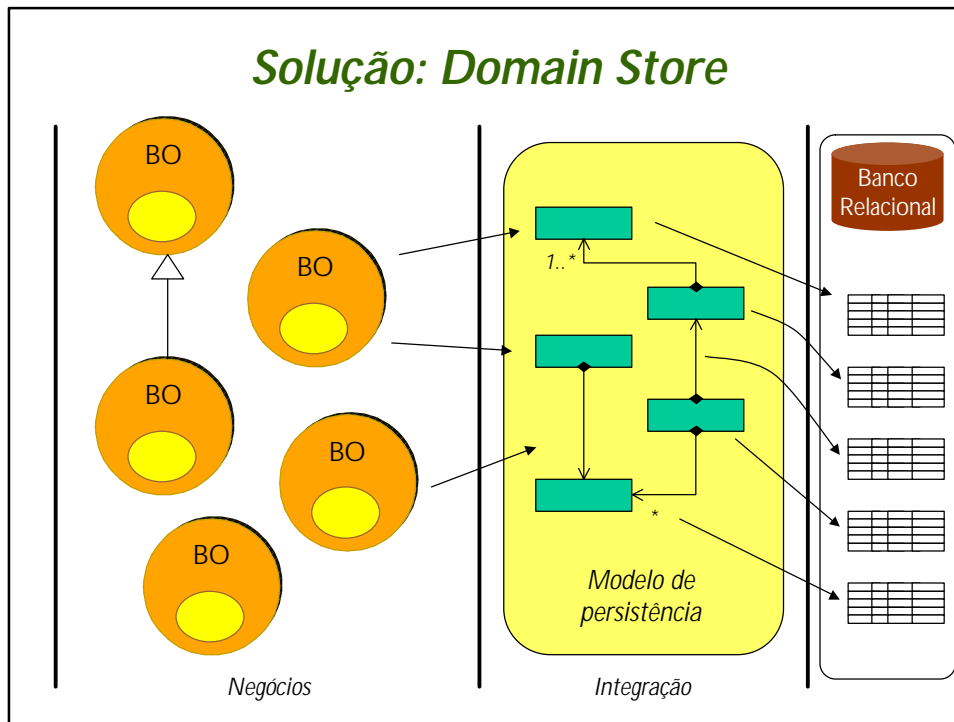
Domain Store

Objetivo: Oferecer um mecanismo transparente de persistência para objetos de negócio. Domain Store é usado quando se deseja objetos de negócio persistentes mas sem que o modelo de persistência faça parte do modelo de objetos.



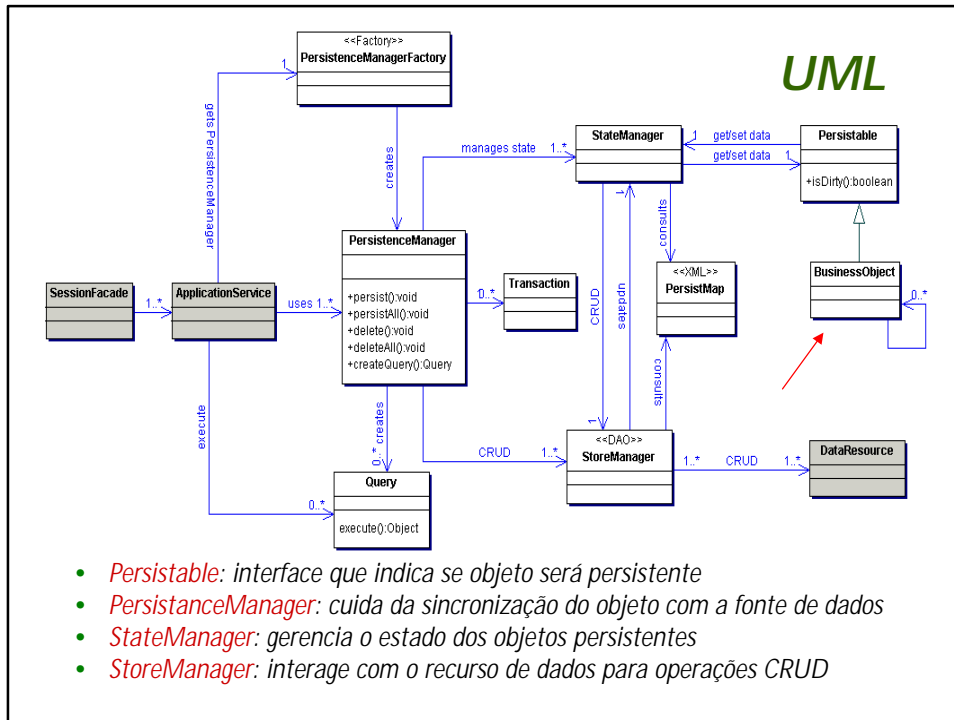
Problema e Motivação

- *Deseja-se separar o mecanismo de persistência do modelo de objetos*
 - *Evita-se incluir detalhes relacionados a persistência nos Business Objects*
- *Deseja-se ter objetos persistentes mas não usar Entity Beans*
- *O sistema não possui um EJB Container*
- *O modelo de objetos usa herança e outros relacionamentos complexos que dificultam a implementação com CMP ou BMP*

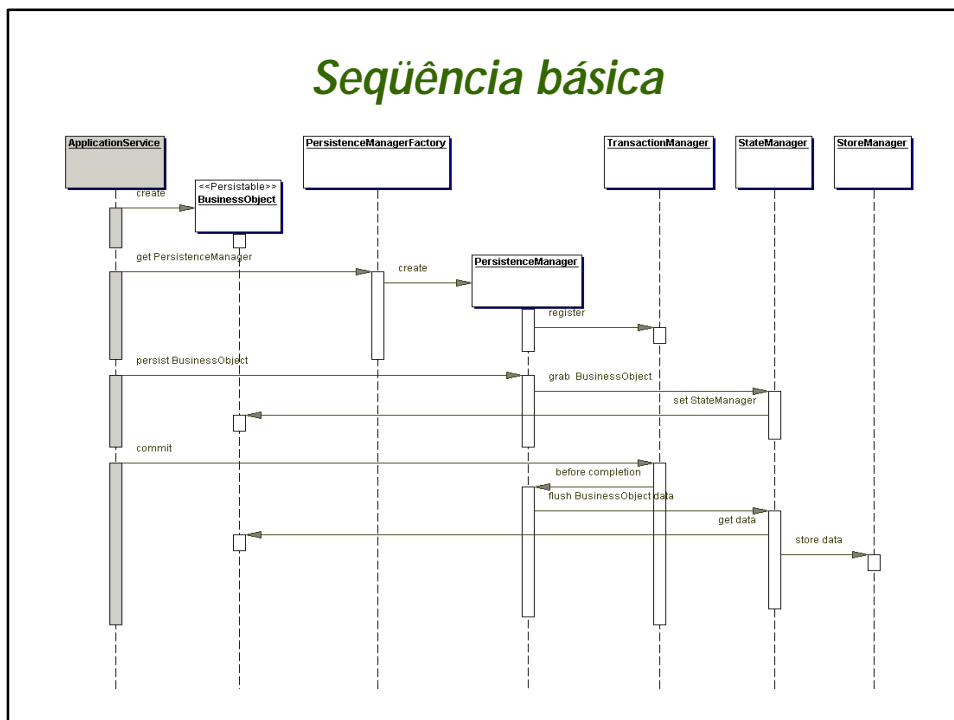


Solução

- Use um *Domain Store* para persistir transparentemente um modelo de objetos
- O mecanismo de persistência de *Domain Store* é separado do modelo de objetos
 - Esses mecanismos diferem dos incluídos em J2EE (CMP e BMP) que incluem suporte à persistência no modelo de objetos
 - Podem ser soluções padronizadas (ex: JDO) ou proprietárias: OJB, Hibernate, etc.



- **Persistable**: interface que indica se objeto será persistente
- **PersistenceManager**: cuida da sincronização do objeto com a fonte de dados
- **StateManager**: gerencia o estado dos objetos persistentes
- **StoreManager**: interage com o recurso de dados para operações CRUD



Seqüência

- 1. *ApplicationService* cria *BusinessObject*
- 2. *ApplicationService* obtém *PersistenceManager* do *PersistenceManagerFactory*
- 3. *PersistenceManager* se registra com *TransactionManager*
- 4. *ApplicationService* pede a *PersistenceManager* para persistir *BusinessObject*
- 5. *PersistenceManager* cria *StateManager* e pede para obter *BusinessObject*
- 6. *StateManager* informa a *BusinessObject* que ele é seu *StateManager*
- 7. *ApplicationService* diz ao *PersistenceManager* para cometer a transação
- 8. *TransactionManager* diz ao *PersistenceManager* para incluir todos os *StateManagers* na transação
- 9. *PersistenceManager* manda *StateManager* liberar seus dados
- 10. *StateManager* obtém dados do *BusinessObject*
- 11. *StateManager* manda *StoreManager* guardar dados
- 12. Transação é cometida

Estratégias

- *Custom Persistence Strategy*
 - *Requer criação de uma solução de persistência proprietária ou da utilização de um produto (OJB, Hibernate ou similar)*
- *Java Data Objects Strategy*
 - *Usa JDO, que faz geração automática de diversas classes e permite configurar persistência através de arquivos XML*

Conseqüências

- *Pode aumentar a complexidade: construir um sistema de persistência é difícil*
 - *Pode-se usar soluções prontas ou JDO*
- *Melhora compreensão de frameworks de persistência*
 - *Evita que se misture com lógica de negócios*
- *Um sistema de persistência completo pode ser excessivo para um pequeno projeto*
- *Separa lógica de objetos de negócio da lógica de persistência*
 - *Facilita testes do modelo de objetos persistentes*

21

Web Service Broker

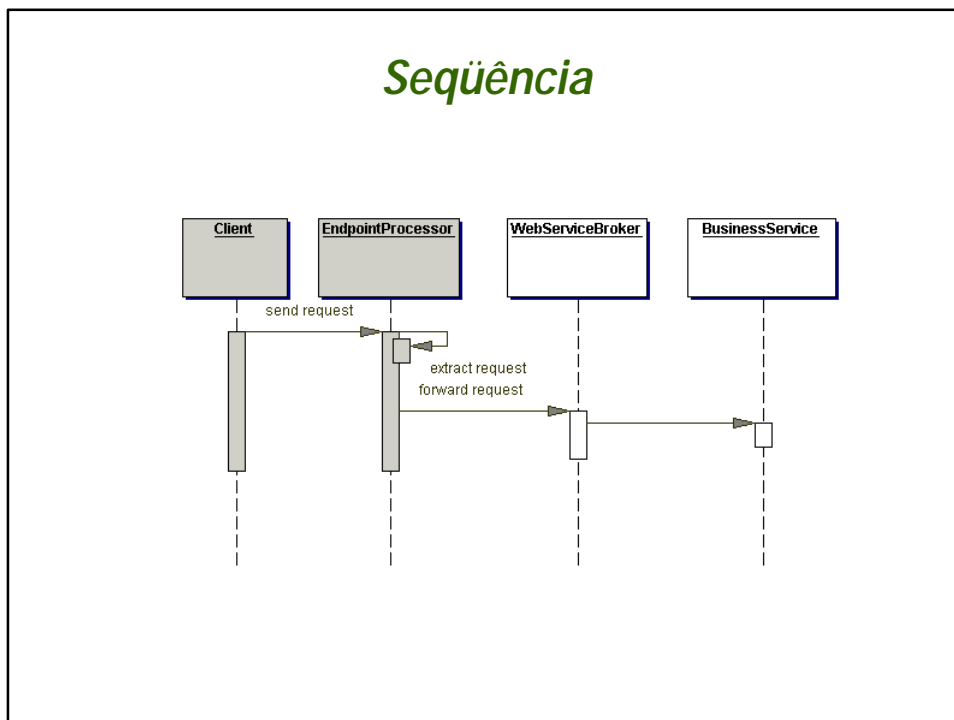
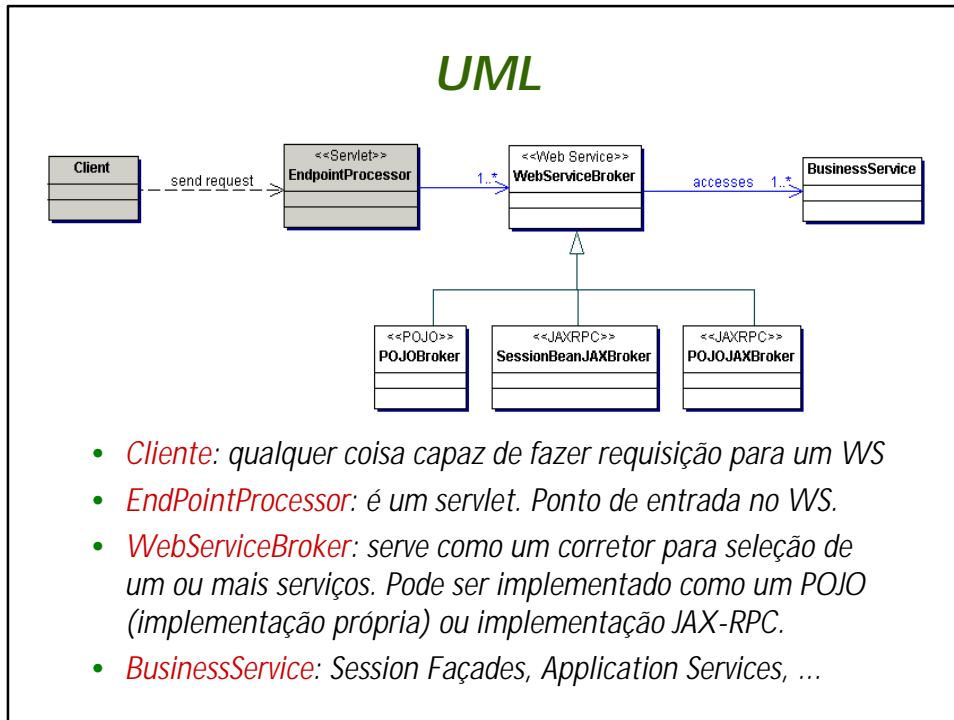
Objetivo: Expor um ou mais serviços usando XML e protocolos Web através de uma interface de baixa granularidade. Web Service Broker é uma fachada para Web Services

Problema

- *Deseja-se oferecer acesso a um ou mais serviços usando protocolos Web e XML*
 - *Reusar e expor serviços existentes para os clientes*
 - *Monitorar e limitar o uso dos serviços*
- *Deseja-se integrar com aplicações heterogêneas e expor os serviços através de protocolos abertos*
- *Deseja-se oferecer uma ponte entre requerimentos de negócio e serviços existentes.*

Solução

- *Use um **Web Service Broker** para disponibilizar e rotear para um ou mais serviços, usando XML e protocolos Web*
 - *Pode ser implementado através da exposição de um documento WSDL*
 - *Pode-se usar um Session Bean service endpoint (J2EE 1.4) ou um objeto JAX-RPC (ambos expostos através de WSDL)*



Estratégias

- *Custom XML Messaging Strategy*
 - *Consiste no envio de mensagens (SOAP) para o Web Service Broker (JAXM ou similar)*
- *Java Binder Strategy*
 - *Usa JAXB ou similar para processar e gerar Transfer Objects a partir de documentos XML*
- *JAX-RPC Strategy*
 - *Solução mais simples utilizando a API JAX-RPC*

Conseqüências

- *Introduz uma camada entre cliente e os serviços*
 - *Limita os serviços acessíveis aos clientes*
- *Session Façades remotas existentes precisam ser refatoradas para suportar acesso local*
- *Performance de rede pode sofrer devido aos protocolos Web*

Fontes

[SJC] *SJC Sun Java Center J2EE Patterns Catalog.*

<http://developer.java.sun.com/developer/restricted/patterns/J2EEMPatternsAtAGlance.html>. *Versão desatualizada dos padrões J2EE do SJC, porém tem exemplos adicionais.*

[Blueprints] *J2EE Blueprints patterns Catalog.*

<http://java.sun.com/blueprints/patterns/catalog.htm>. *Contém padrões extras usados na aplicação Pet Store.*

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

<http://java.sun.com/blueprints/corej2eepatterns/index.html>. *Descrição detalhada dos padrões SJC. O site é mais atualizado que o livro.*

Curso J931: J2EE Design Patterns

Versão 1.1

www.argonavis.com.br

© 2003, Helder da Rocha
(helder@acm.org)