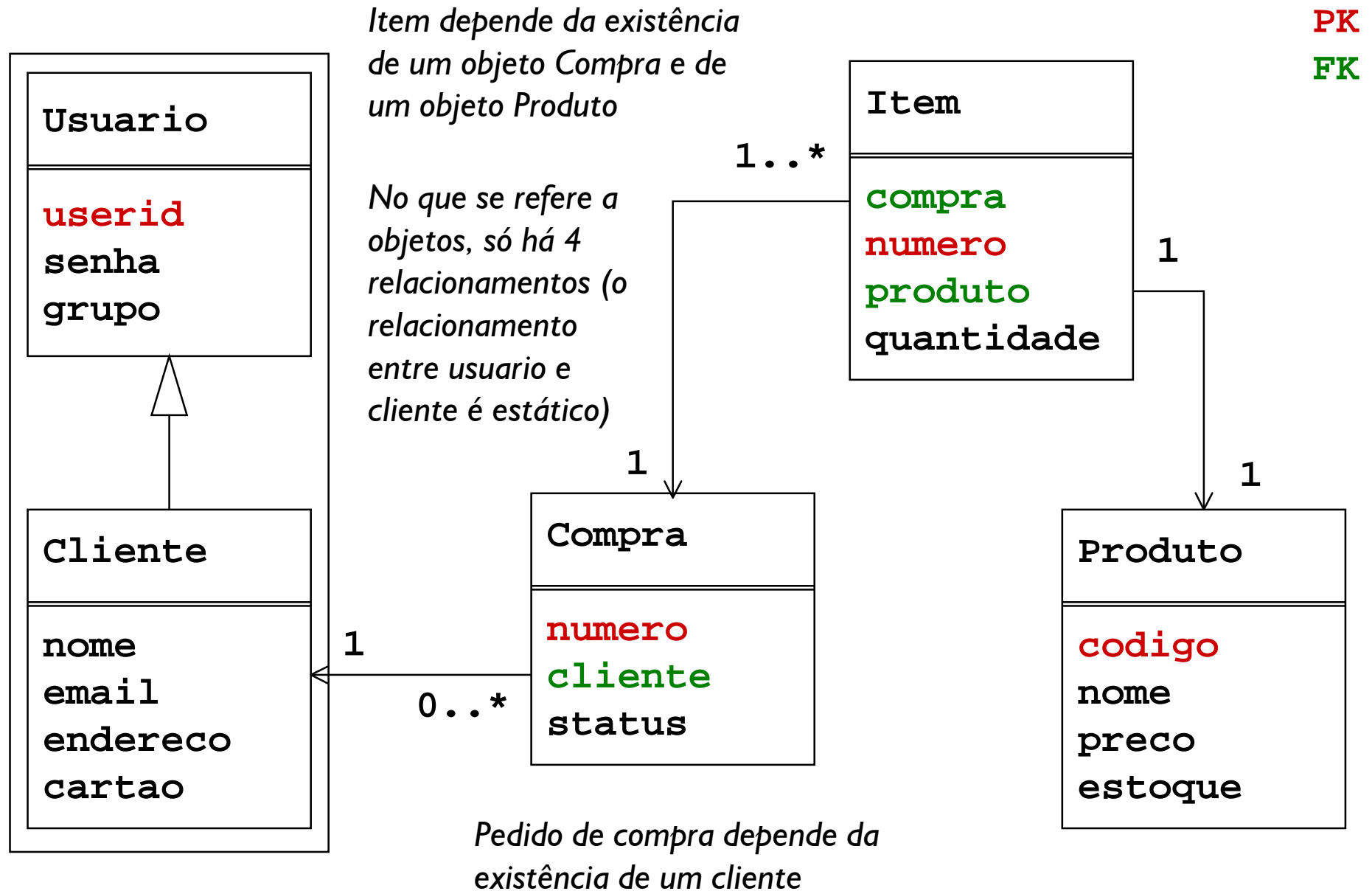


J530 - Enterprise JavaBeans

Laboratório EJB e J2EE
Uma aplicação
completa

- *O objetivo deste módulo é construir e implantar uma aplicação J2EE completa*
- *Inicialmente, será mostrada como exemplo uma aplicação consistindo de vários beans, JSPs e servlets usando serviços de transações e segurança*
 - *A aplicação utiliza alguns padrões de projeto J2EE*
 - *Todos os passos de compilação e montagem estão embutidos no build.xml*
- *As aplicações utilizarão o banco HSQLDB, que é o banco nativo do JBoss*

Aplicação exemplo: relacionamentos



Aplicação exemplo: tabelas no banco (I)


```
create table produtos (  
    id integer primary key,  
    nome varchar(64),  
    preco numeric(10,2),  
    qte integer  
);  
  
create table usuarios (  
    id varchar(8) primary key,  
    senha varchar(16),  
    grupo varchar(16)  
);  
  
create table clientes (  
    id varchar(8) primary key,  
    nome varchar(64),  
    email varchar(64),  
    endereco varchar(256),  
    cartao varchar(16),  
    constraint fk_usuario foreign key (id)  
        references usuarios(id)  
);
```

Aplicação exemplo: tabelas no banco (2)

```
create table compras (  
    id integer primary key,  
    cliente varchar(8),  
    status varchar(16),  
    constraint fk_cliente foreign key (cliente)  
        references clientes(id)  
);
```

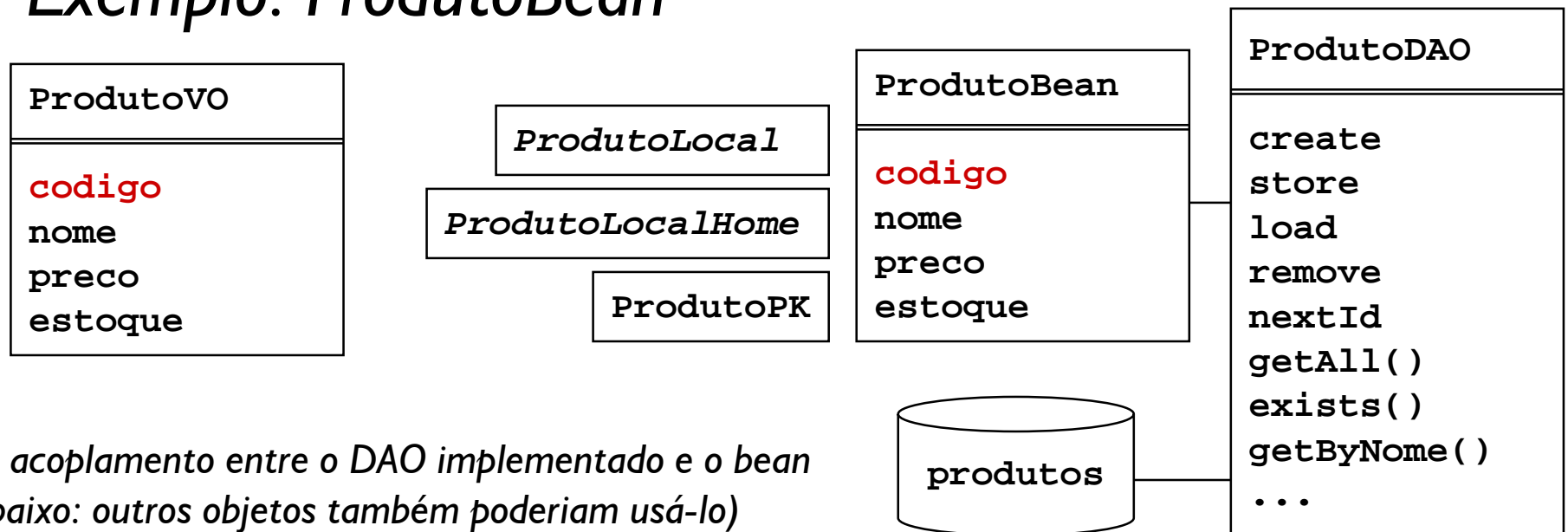
```
create table itens (  
    id integer primary key,  
    compra integer,  
    produto integer,  
    qte integer,  
    constraint fk_compra foreign key (compra)  
        references compras(id),  
    constraint fk_produto foreign key (produto)  
        references produtos(id)  
);
```

*Poderia ser melhor implementado
com PK composta de compra + id*



Aplicação exemplo: entity beans

- Cada entity bean (BMP) possui
 - Um par de *interfaces locais* e uma classe *PK*
 - Um *Value Object*: objeto serializável que pode armazenar seu estado (também chamado de *Data* ou *Transfer object*)
 - Um *DAO*: objeto que encapsula a lógica de persistência e é chamado a partir dos métodos callback do bean
- Exemplo: *ProdutoBean*

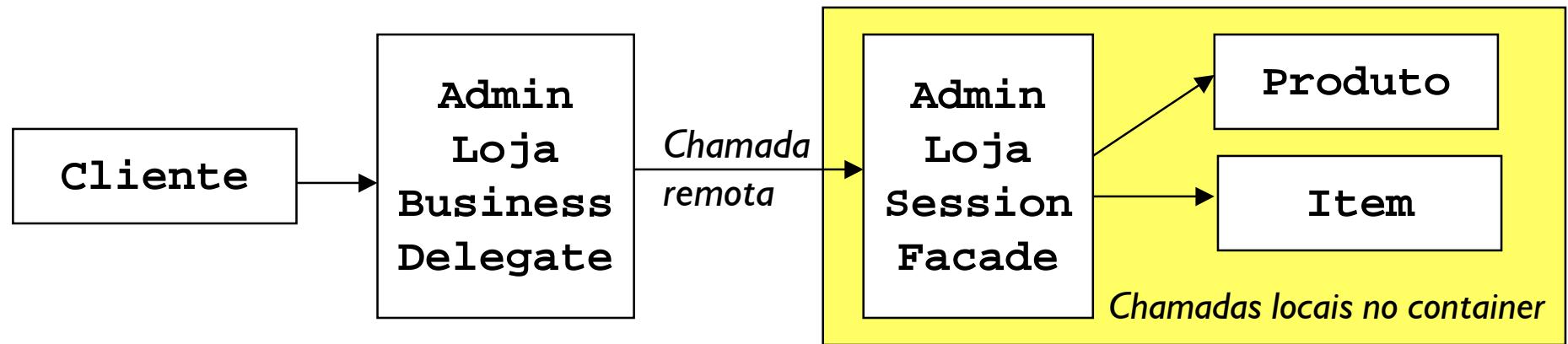


Aplicação exemplo: session beans

- *Session beans implementam todas as operações da aplicação. Três são fachadas:*
 - **AdminLojaSessionFacade**: operações de administração da loja: criar produtos, aumentar estoque, etc. (Stateless)
 - **LoginSessionFacade**: operações de autenticação, criação e alteração de clientes cadastrados. (Stateless)
 - **LojaSessionFacade**: representa a sessão de um cliente fazendo compras na loja (Stateful)
- *Parte de LojaSessionFacade é um carrinho de compras, que é implementando em outro bean*
 - **CarrinhoSession**: contém operações do carrinho de compras (Stateful)

Aplicação exemplo: fachadas

- Os *session beans* isolam o cliente da lógica de negócios (*entity beans*) implementando todas as operações
 - São acessados remotamente pelos clientes, portanto, têm um par de interfaces remotas
- Cada bean que implementa operações de fachada possui um *Business Delegate*, que isola o cliente da tecnologia usada na implementação da aplicação
 - Cliente não recebe exceções EJB ou sequer precisa saber da API *javax.ejb. Business Delegate* captura essas exceções e as inclui em exceções da aplicação (*LojaException*)



Aplicação exemplo: message-driven beans

- A aplicação possui três MDBs
 - **ControleEstoqueMDB**: recebe pedidos para aumentar o estoque. Os pedidos chegam em uma fila usada pelo `AdminLojaSessionFacade` no método `solicitarCompra()`
 - **ProcessarCompraMDB**: recebe dados de uma compra e os utiliza para processá-la (verificar cartão de crédito, etc.) depois, altera o status da compra (para concluído ou suspenso) e passa as informações para o `MailerMDB`
 - **MailerMDB**: usa a API `JavaMail` para enviar um e-mail ao cliente informando se seu pedido foi processado ou não.

Aplicação exemplo: Service Locator

- *A aplicação isola os beans e clientes (delegates) de precisarem usar a API JNDI oferecendo métodos utilitários que devolvem beans, filas, conexões de bancos de dados e variáveis de ambiente*
 - *Foi implementado um único **ServiceLocator** genérico (não um por bean - que é uma outra estratégia comum)*
 - *No **ServiceLocator** genérico, os métodos requerem que se passe o caminho JNDI dos objetos a serem localizados*

■ Métodos

<code>java.sql.Connection</code>	<code>findJDBCConnection(nome)</code>
<code>javax.jms.Connection</code>	<code>findDefaultJMSConnection()</code>
<code>javax.jms.Queue</code>	<code>findJMSQueue(nome, alternativo)</code>
<code>javax.ejb.EJBHome</code>	<code>findRemoteEJB(nome, classe)</code>
<code>javax.ejb.EJBLocalHome</code>	<code>findLocalEJB(nome)</code>
<code>java.lang.Object</code>	<code>findEnvEntry(nome)</code>

Aplicação exemplo: camada Web

- *A camada Web da aplicação exemplo usa três servlets controladores e comandos*
 - *FrontController & Command pattern*
- *Também usa os Value Objects da aplicação EJB como View Helpers (para preencher os JSPs) e uma pequena tag library*
- *Os servlets são clientes e interagem com os Business Delegates*
- *Não foi usada autenticação Web*
 - *Usamos autenticação EJB através do LoginBusinessDelegate*

Como implantar e executar

- 1. Verifique a configuração do seu ambiente inspecionando o arquivo *build.properties*
- 2. Crie primeiro as tabelas usando
 - > *ant create-table*
- 3. Depois monte a aplicação com
 - > *ant deploy*
- 4. Para executar os clientes EJB, use
 - > *ant run.all.test.clients*
- 5. Para acessar a aplicação via Web, use o browser
 - *http://localhost:8080/loja/*
 - Cadastre primeiro um usuário do tipo admin para criar alguns produtos.
 - Inicie o James ou outro servidor de e-mail local.
- Gere documentação (no diretório docs) usando
 - > *ant generate-docs*

- *Neste exercício você vai implementar uma parte da aplicação demonstrada em sala de aula.*
 - *Use e configure seus build.xml para que você possa compilar e verificar os arquivos gerados continuamente, perdendo menos tempo com depurações complicadas.*
 - *Use os arquivos e estrutura de pacotes fornecida!*
- *O exercício consiste de duas partes:*
 - *1. Criação de uma aplicação EJB usando um Entity Bean (BMP), um Session Bean e um Message-driven bean*
 - *2. Integração da aplicação acima para a Web usando JSP, servlets e custom tags. O código Web já está pronto mas será necessário editar os deployment descriptors e construir um EAR*

Parte I - Exercício I

- a) *Crie um Entity Bean com interfaces locais que esteja sincronizado com os dados da tabela de produtos do banco.*
 - *Não coloque código JDBC no bean. Use o DAO fornecido com métodos create/load/store/remove e chame os métodos do DAO de dentro do bean.*
 - *Use também ProdutoVO (value object) para transferir dados. O DAO preenche esse bean e o EJB pode recebê-lo como parâmetro ou retorná-lo em seus métodos.*
- (b) *Configure o seu ejb-jar e use o verifier para validar o seu EJB-JAR (use a tarefa do Ant).*
- (c) *Se tudo estiver OK, faça deploy e observe eventuais erros.*
- (d) *Escreva um cliente simples para testar o seu EJB*
 - *Para isto, registre (no ejb-jar.xml) e use também um par de interfaces remotas (somente com esta finalidade)*

Parte I - Exercício 2

- *a) Implemente as operações abaixo (definidas na classe AdminLojaBusinessDelegate) em um session bean. Faça com que o BD localize e chame os métodos do session bean*
 - `criarProduto(String nome, BigDecimal preco): String`
 - `removerProduto(String codigo)`
 - `ProdutoVO[] listarProdutos()`
 - `solicitarCompra(String codigo, int quantidade)`
 - `esvaziarEstoque(String codigo)`
- *b) Acesse o EJB de produto através de uma interface local.*
 - *Implemente os métodos acima usando Produto*
 - *Não se esqueça de configurar no ejb-jar.xml as referências para o produto a partir do outro bean.*
 - *Mapeie referências locais a nomes JNDI reais no arquivo jboss.xml.*
 - *Use, se desejar, o service locator fornecido.*
- *c) Use o cliente fornecido para testar a aplicação*

Parte I - Exercício 3

- *a) Crie um MDB (ControleEstoqueMDB) que receba pedidos de aumento de estoque assíncronamente através da escuta de uma fila JMS (use queue/testQueue).*
 - *A mensagem enviada deve conter o código do produto e a quantidade desejada. O MDB deve, então, localizar o bean (Produto) e fazer a alteração.*
 - *Configure o bean no ejb-jar.xml, verifique e faça o deployment.*
 - *Não se esqueça de registrar os nomes JNDI globais de todos os beans, filas e bancos de dados no jboss.xml.*
- *b) Altere o método solicitarCompra para que ele em vez de alterar o produto diretamente, envie para a fila onde está cadastrado o MDB para que ele leia e processe o pedido.*
- *c) Rode o cliente e teste a aplicação.*

- *Objetivo: adaptar a aplicação anterior para que funcione também via cliente Web.*
 - *a) Use os três custom tags fornecidos (que já são automaticamente compilados e empacotados pelo Ant) no classpath do WAR, os JSP que estão prontos, o servlet e os comandos (falta incluir chamadas no lugar dos comentários).*
 - *b) Preencha os comandos (pacote web.command) com chamadas ao BusinessDelegate para que a aplicação funcione na Web,*
 - *d) Configure o web.xml com referências para os EJBs*
 - *d) Use o build.xml fornecido para empacotar o WAR e em um EAR e faça o deployment.*

Exercícios extras (para turmas avançadas)

- 1. Converta a aplicação da parte I em CMP
- 2. (na aplicação exercício) Implemente autenticação e autorização na aplicação e defina privilégios de administrador aos métodos que alteram os dados da aplicação.
- 3. (na aplicação exemplo) Use controle de transações para garantir que pedidos rejeitados (status: "Suspendido") sejam desfeitos
 - Pode não ser possível resolver o problema apenas demarcando transações, já que parte do processo envolve chamadas assíncronas (MDB)
- 4. Converta a aplicação exemplo em CMP com CMR

Curso J530: Enterprise JavaBeans

Revisão 2.0 - Junho de 2003

© 2001-2003, Helder da Rocha
(helder@acm.org)

 argonavis.com.br