Design Patterns Casual Class

R. Alexandre Dumas, 1268 cj. 134 – Chac. Sto. Antonio – São Paulo – SP
55 11 5183 2999 – www.globalcode.com.br

Design Patterns

Segundo [Gamma, et all] *"Um design pattern é uma descrição de comunicação de objetos e classes que são customizados para resolver um problema genérico de design em um contexto particular."*

Design Pattern = Padrão de Modelagem

Podemos entender que um pattern é uma solução para problemas de modelagem que ocorrem com freqüência em situações específicas.

Documentamos um Design Pattern através de características essenciais e tipicamente descrevem:

- **Nome do Pattern**
- **Problema**: descreve a situação enfrentada pelos desenvolvedores;
- **Solução**: forma de modelagem independente de cenário específico;
- **Conseqüências**: ganhos na implementação e resultados esperados;

Classificamos os Design Patterns em três categorias:

- **Comportamental**: voltados para a modelagem da comunicação entre os objetos e também o fluxo em cenários complexos;
- **Criação**: patterns que criam objetos para você sem que você tenha que instanciar a classe diretamente, isto provê para o seu programa maior flexibilidade em qual objeto deve ser criado em cada caso;
- **Estrutura**: ajudam a você a compor grupo de objetos em largas estruturas;

"Cada pattern descreve um problema que ocorre com freqüência em nosso ambiente, e então descreve o núcleo da solução para tal problema, de uma maneira que possamos utilizar esta solução milhares de vezes" [Christopher Alexander]

Tabela de Patterns GoF (Gang of Four)

⊗ Comportamental ∅ Criação ⊕ Estrutural

Categoria	Pattern	Descrição
∅	Abstract Factory	Encapsula e centraliza regras de criação de objetos de uma determinada família
⊕	Adapter / Wrapper	Converte uma interface de uma classe em uma interface esperada por algum cliente
⊕	Bridge	Desvincula abstração de implementação
∅	Builder	Separa processos complexos de construção do objeto da sua representação de forma que o mesmo processo possa ser utilizado para diferentes representações
⊗	Chain of Responsability	Permite que mais de um objeto recepcione um mesmo request
⊗	Command	Encapsula uma ação que é acionada por um cliente sem que ele conheça detalhes da ação
⊕	Composite	Compõe múltiplos objetos em forma de árvore
⊕	Decorator	Anexa responsabilidades adicionais para um objeto dinamicamente
⊕	Facade	Define uma interface de alto nível que torna simples o uso de sub-partes do sistema
∅	Factory Method	Centraliza responsabilidades de criação de objetos em uma classe
⊕	Flyweight	Compartilha objetos para suportar grande quantidade de pequenos objetos com alta granularidade
⊗	Interpreter	Utilizado para modelar interpretador de linguagens
]⊗	Iterator	Provê uma maneira simples de interagir com um conjunto de objetos agregados
⊗	Mediator	Define um objeto que encapsula como um conjunto de objetos vai interagir
⊗	Memento	Sem violar o encapsulamento, captura e exporta o estado interno do objeto para que este seja restaurado posteriormente

⊗	Observer	Define uma dependência de um-para-muitos entre objetos e quando o estado do objeto observado é alterado ele notifica todos os dependentes (observadores) automaticamente
∅	Prototype	Visa aumentar performance na criação de objetos utilizando clone de protótipos de objetos criados previamente
⊕	Proxy	Cria nova semântica de acesso a um objeto através de métodos proxy
∅	Singleton	Garante que só haverá um objeto de uam determinada classe
⊗	State	Modela uma solução para melhor gerenciamento de estado de objetos
⊗	Strategy	Permite a divisão de um longo algoritmo em pequenas peças
⊗	Template Method	Quando um conjunto de classes deve conter um comportamento semelhante
⊗	Visitor	Representa em um conjunto de objetos as operações que determinado método deve executar

⊗ Comportamental ∅ Criação ⊕ Estrutural

Tabela de Patterns J2EE

Design Patterns J2EE é um conjunto de patterns derivado dos patterns GoF especializados em soluções no ambiente J2EE. A aplicação prática / física dos Design Patterns é chamada de idioma. Um idioma é vinculado à uma tecnologia e específico para uma linguagem de programação.

Os patterns J2EE garantem:

- Modularidade;
- Proteção e exposição;
- Extensibilidade de componentes;
- Regras e responsabilidades;
- Contratos;
- Comportamento dinâmico;
- Performance;

Categoria	Pattern	Breve Descrição
∅	Business Delegate	Reduz acoplamento entre a camada Web e a camada EJB
∅	Composite Entity	Modela uma rede de entidades de negócio relacionadas
⊗	Composite View	Gerencia layout de conteúdo composto por múltiplas visualizações
⊕	Data Access Object (DAO)	Abstrai e encapsula mecanismos de acesso a dados
⊕	Fast Lane Reader	Aumenta performance em dados tabulados
⊗	Front Controller	Centraliza requisições de usuários
⊗	Intercepting Filter	Pré e pós-processamento de requisições
⊗∅⊕	Model-View-Controller	Separa representação do dado, do comportamento da aplicação e da apresentação
∅	Service Locator	Simplifica o acesso do cliente a serviços de negócio enterprise
∅	Session Facade	Coordena operações entre múltiplos objetos de negócio em um workflow
∅	Transfer Object	Efetua transferência de dados entre camadas
∅	Value List Handler	Interage com uma lista virtual de forma eficiente
⊗	View Helper	Simplifica o acesso ao estado de um modelo e à lógica de acesso a dados

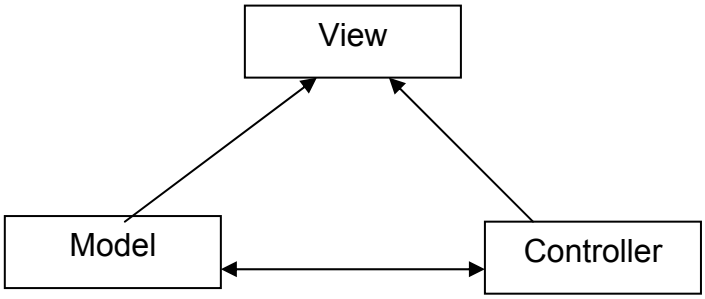
⊗ Camada Apresentação ∅ Camada de Negócio ⊕ Camada de Integração

Vantagens de uso de Design Patterns

- É uma maneira de compartilhar arquitetura entre projetos;
- É uma maneira de compartilhar conhecimento entre desenvolvedores;
- É uma padronização;
- É uma documentação implícita;
- Facilita o aprendizado;
- Facilita a manutenção;
- Impõe implicitamente melhores práticas em projeto P.O.O.;
- Reduz o custo total de propriedade do software;
- Facilita a comunicação entre os desenvolvedores;

Mais sobre Design Patterns

No começo da década de 1980 linguagens orientadas a objetos estavam apenas começando a ser utilizadas por uma minoria de programadores. A mais comum era Smalltalk e a linguagem C++ estava apenas no início. A idéia de Design Patterns já existia mesmo em linguagens estruturadas, porém tal conceito era mais conhecido como framework. Um framework muito conhecido é o Model-View-Controller para Smalltalk [Krasner and Pope, 1988], que divide o problema de interface com usuário em três partes:



Cada parte é um objeto ou um conjunto de objetos que atendem a determinadas responsabilidades e tal separação e muito rica em reusabilidade de lógica de negócio em várias formas de visualização. A forma de conectar cada um dos objetos e estabelecer comunicação e troca de dados é um ótimo exemplo de aplicação e ou criação de design patterns. Para definir as diferenças de fato entre framework e design pattern sugiremos a seguinte frase:

"Design patterns é mais focado em reuso de temas de arquitetura recorrentes, enquanto frameworks focam detalhes de design e implementação". [Coplien & Schmidt, 1995]

Singleton

- *Problema:* você precisa garantir que somente haverá um único objeto de uma classe;
- *Solução:* criar um objeto estático / de classe utilizando o modificador static;
- *Aplicação prática:* quando você deseja garantir que um objeto seja um ponto de entrada único em todo seu aplicativo, como por exemplo, um ponto central de comunicação como o banco de dados, um spool de impressão, etc.
- *Código Exemplo*

```
public class Aplicacao {
  private static Aplicacao instance = new Aplicacao();
  //Construtor private para garantir o single
  private Aplicacao() {}
  public static Aplicacao getInstance() {
    return instance;
  }
}
```

Factory

- *Problema:* você não pode antecipar a classe do objeto que deve ser criado. Você vai receber argumento variável que indicará o tipo do objeto a ser criado;

- *Solução:* criaremos uma fábrica de objetos que conterá toda a lógica de criação otimizada de objetos, o usuário final terá contato direto com a fábrica de objeto e nenhum relacionamento direto e forte com objetos específicos fabricados;

- *Aplicação prática:* é um pattern muito utilizado na API EJB onde a home interface de um EJB representa uma fábrica abstrata de objetos, tal fábrica é criada em tempo de deployment (instalação do EJB no container) e otimiza o uso de recursos do servidor criando pooling (cache de objetos de um mesmo tipo) de objetos para evitar a construção / destrução excessiva de objetos.

- *Código Exemplo*

```
package br.com.globalcode.designpattern.customerfactory;

public abstract class Customer {
  private String nome;
  private double rendaMensal;

  public Customer(String nome, double rendaMensal) {
    setNome(nome);
    setRendaMensal(rendaMensal);
  }

  public String getNome() {
    return nome;
  }
  public void setNome(String nome) {
    this.nome = nome;
  }
  public void setRendaMensal(double rendaMensal) {
    this.rendaMensal = rendaMensal;
  }
  public double getRendaMensal() {
```

```
    return rendaMensal;
  }
  public abstract boolean analiseCredito();
  public abstract String getCategoria();

}
```

```
package br.com.globalcode.designpattern.customerfactory;

public class DefaultCustomer extends Customer {

  public DefaultCustomer(String n, double d) {
    super(n, d);
  }
  public boolean analiseCredito() {
    //Verificações normais de crédito
    return true;
  }
  public String getCategoria() {
    return "Cliente Padrão";
  }
}
```

```
package br.com.globalcode.designpattern.customerfactory;

public class RiskCustomer extends Customer {

  public RiskCustomer(String n, double d) {
    super(n ,d);
  }
  public boolean analiseCredito() {
    //Verificações específicas para cliente do grupo de risco
devido a baixa renda
    return false;
  }
  public String getCategoria() {
```

```
    return "Cliente Risco";
  }
}
```

```
package br.com.globalcode.designpattern.customerfactory;

public class SecureCustomer extends Customer{

  public SecureCustomer(String n, double d) {
    super(n, d);
  }
  public boolean analiseCredito() {
    //Não faz nenhuma verificação, crédito pré-aprovado
    return true;
  }
  public String getCategoria() {
    return "Cliente com alta credibilidade";
  }

}
```

```
package br.com.globalcode.designpattern.customerfactory;

public class CustomerFactory {
  private static double RENDA_MEDIA = 5000.0;
  private static double RENDA_BAIXA = 1000.0;

  private static CustomerFactory instance = new CustomerFactory();

  private CustomerFactory() {}
  public static CustomerFactory getInstance() {
    return instance;
  }

  public Customer newCustomer(String name, double rendaMensal) {
    Customer r = null;
```

```
    if(rendaMensal <= RENDA_BAIXA) {
      r = new RiskCustomer(name, rendaMensal);
    }
    else if(rendaMensal > RENDA_BAIXA && rendaMensal<=RENDA_MEDIA)
{
      r = new DefaultCustomer(name, rendaMensal);
    }
    else {
      r = new SecureCustomer(name ,rendaMensal);
    }
    return r;
  }
}
```

```
package br.com.globalcode.designpattern.customerfactory;

public class Crediario {
  public static void main(String args[]) {
    Customer clientes[] = {
    CustomerFactory.getInstance().newCustomer("Arnold Layne",
240.0),
      CustomerFactory.getInstance().newCustomer("Mick Jagger",
4500.0),
      CustomerFactory.getInstance().newCustomer("Paul Mcartney",
12000.0) };
    Crediario.analisarCredito(clientes);
  }
  public static void analisarCredito(Customer clientes[]) {
    for(int x=0;x < clientes.length;x++) {
      System.out.println("Cliente: " + clientes[x].getNome() + " -
" + clientes[x].getCategoria());
      System.out.println("Crédito: " +
(clientes[x].analiseCredito() ? "aprovado" : "negado"));
    }
  }
}
```

- *UML*

  Dependendo dos dados recebidos o CustomerFactory cria um tipo específico de Customer

  - Risk Customer
  - DefaultCustome
  -SecureCustomer

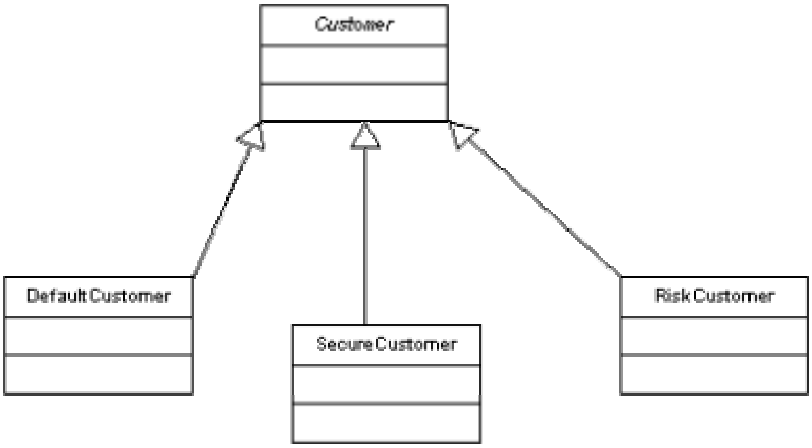
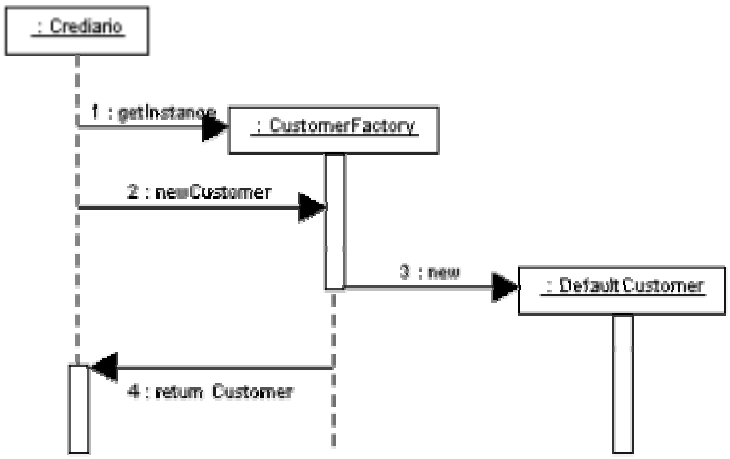


Diagrama de seqüência



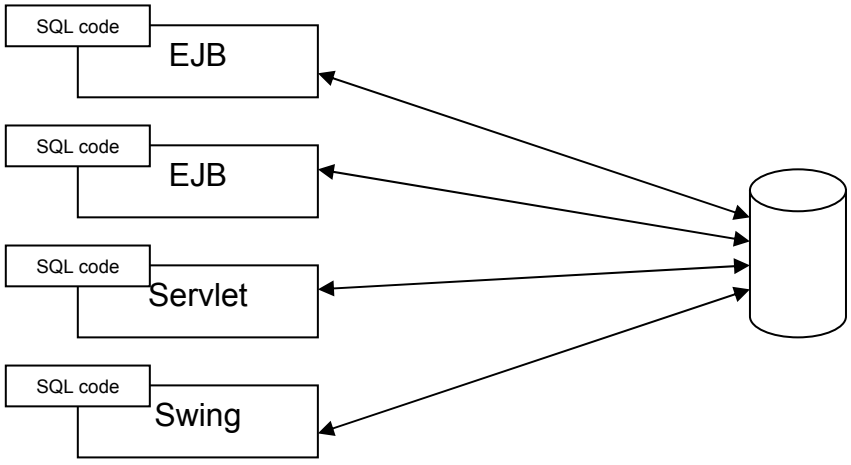
Data Access Object

- *Problema:* código de acesso a dados da aplicação espalhados em várias classes dificultando o entendimento, manutenção e mudanças no modelo de dados ou troca de fornecedor de dados;

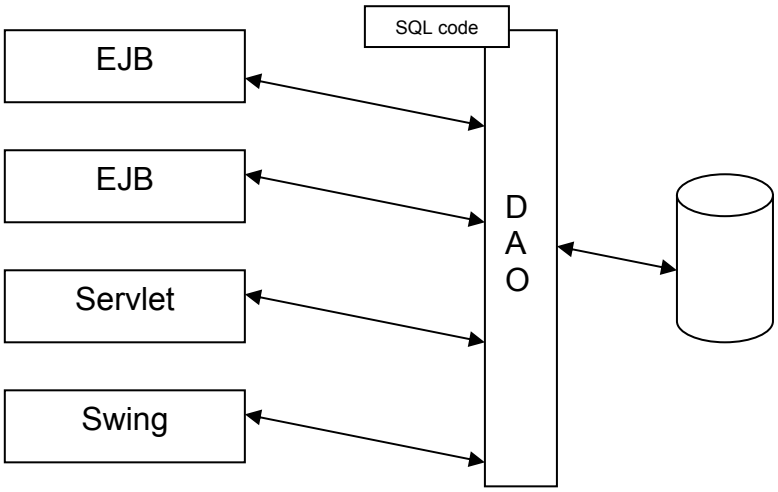
- *Solução:* criar um ponto de central em um pequeno conjunto de classes que são responsáveis por acessar os dados da aplicação;

- *Aplicações práticas:* quando desenvolvemos uma solução com o formato de produto e esta solução será instalada em diversos ambientes devemos flexibilizar nossa aplicação de forma que ela possa ser configurada para trabalhar com múltiplos sistemas de gerenciamento de banco de dados como MySQL, MS-SQL, Sybase, Oracle e outros mais. O Design Pattern DAO prevê uma maneira eficiente de resolvermos este problema;

- *Errado*



- *Correto*



- *Código Exemplo*

```
package br.com.globalcode.designpattern.dao;

public class Projeto {
  private int id;
  private String nome;
  public Projeto(int id, String nome) {
    this.id = id;
    this.nome = nome;
  }
  public int getId() {
    return this.id;
  }
  public String getNome() {
    return this.nome;
  }

  public void setNome(String nome) {
    this.nome = nome;
  }
}
```

```
package br.com.globalcode.designpattern.dao;

import java.sql.*;
import javax.sql.*;
import javax.naming.*;
import java.util.Vector;

public class BaseDAO {

  public BaseDAO() {
  }
  protected Connection getConnection() throws SQLException {
    Connection conn = null;
    DataSource ds = null;
    try {
      Context initCtx = new InitialContext();
      ds = (DataSource)
initCtx.lookup("java:comp/env/CasualClass");
      conn = ds.getConnection();
    }
    catch(NamingException e) {
      throw new SQLException(e.getMessage());
    }
    return conn;
  }
  public Projeto[] getProjetos() throws SQLException {
    return this.getProjetos("");
  }
  public Projeto[] getProjetos(String nomeOuParteDoNome) throws
SQLException {
    Connection connection = this.getConnection();
    Statement st = connection.createStatement();
    String sql = null;
    if(nomeOuParteDoNome.equals("")) {
      sql = "select projeto_id, projeto_nome from projetos";
    }
    else {
```

```
      sql = "select projeto_id, projeto_nome from projetos where
projeto_nome like '%" + nomeOuParteDoNome + "%'";
    }
    ResultSet rs = st.executeQuery(sql);
    Vector temp = new Vector();
    while(rs.next()) {
      temp.add(new Projeto(rs.getInt("projeto_id"),
rs.getString("projeto_nome")));
    }
    rs.close();
    st.close();
    connection.close();
    Projeto[] retorno = null;
    if(temp.size() > 0) {
      retorno = new Projeto[temp.size()];
      temp.copyInto(retorno);
    }
    return retorno;
  }
}
```

Front Controller

- *Problema:* o sistema requer um ponto centralizado de gerenciamento de requisições de usuários. Quando não trabalhamos com centralização de requisições enfrentamos os seguintes problemas:
  - *Para cada tipo de visualização de dados temos um controlador induzindo a duplicata de código;*
  - *Navegação nas views é de responsabilidade das próprias views;*
  - *Dificuldade de manutenção e compreensão da solução;*

- *Solução:* utilizar um ponto centralizador de requisições;

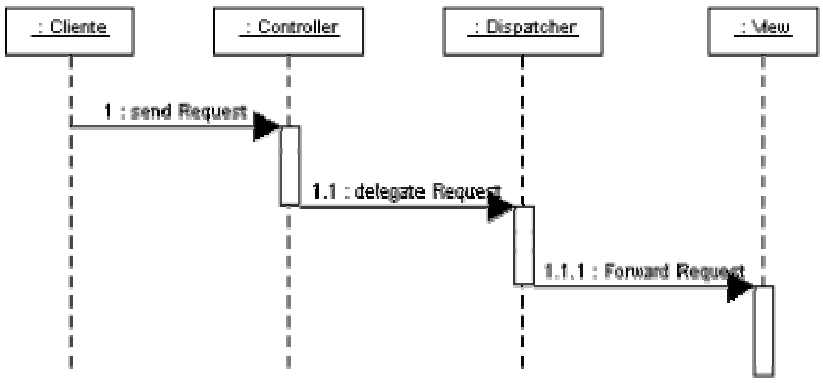
- *Aplicações práticas:* quando temos diversos formulários HTML podemos criar um servlet que recebe um argumento, por exemplo, action que define que tipo de comando o cliente deseja solicitar permitindo assim que você insira com facilidade código de verificação de segurança, logs, altere e filtre conteúdo, etc.

- *Código Exemplo*

```
public class FrontController extends HttpServlet {

  protected void processRequest(HttpServletRequest
    request, HttpServletResponse response)
    throws ServletException, java.io.IOException {
    String page;

    try {
      //Classe que vai analisar o request
      RequestHelper helper = new
         RequestHelper(request);
      //Obtem um objeto de uma sub-classe de command
      //que contém a ação a ser executada
      Command cmdHelper= helper.getCommand();

      //Executa operação e recebe como resposta a página
      //que o client deve ser "redirecionado"
      page = cmdHelper.execute(request, response);

    }
```

```
      catch (Exception e) {
        //...
      }
      //Faz o "redirecionamento" do client para a página
      //de resposta / view
      dispatch(request, response, page);
    }

    protected void doGet(HttpServletRequest request,
      HttpServletResponse response)
      throws ServletException, java.io.IOException {
        processRequest(request, response);
    }
    protected void doPost(HttpServletRequest request,
      HttpServletResponse response)
      throws ServletException, java.io.IOException {
          processRequest(request, response);
    }
    protected void dispatch(HttpServletRequest request,
      HttpServletResponse response,
      String page)
    throws  javax.servlet.ServletException,
      java.io.IOException {
      RequestDispatcher dispatcher =
        getServletContext().getRequestDispatcher(page);
      dispatcher.forward(request, response);
    }
}
```

- *UML*



Intercepting Filter

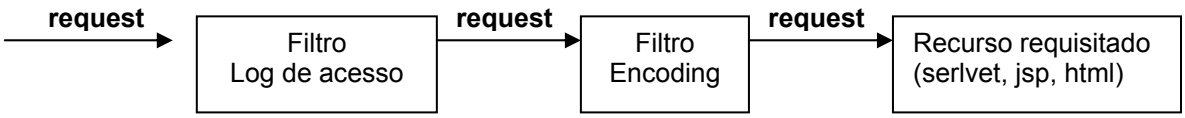
- *Problema:* A maioria das aplicações tem requerimentos, como segurança e log, que serão aplicados a todas as requests. A adição deste serviço em cada parte da aplicação demanda uma grande quantidade de tempo, está muito venerável a erros (é fácil esquecermos uma de implementar segurança ou log em uma página) e além de tudo, está solução é difícil de ser mantida.

- *Solução:* Este pattern propõe que "centralizemos" este trabalho, criando um filtro que intercepta o recebimento das requisições (Request) e do envio da resposta (Response). Desta maneira o filtro poderá redirecionar ou ainda processar dados da request, além de poder processar dados da response, antes que os dados sejam realmente enviados para o cliente

- *Aplicação prática:*

    1. Loggin
    2. Segurança
    3. Filtro de conteúdo
    4. Debug
    5. Processamento de xml e xsl
    6. Determinação do enconding adequado

    Também podemos utilizar mais que um filtro, formando o que chamamos chain filter (cadeia de filtros). Veja um exemplo na figura abaixo:



Neste exemplo temos dois filtros, o primeiro é responsável pelo log de acesso a recursos configurados no filtro. Enquanto o segundo é responsável por indicar o encoding adequado na response, depois que os dois filtros são executados o recurso requisitado é chamado.

- *Benefícios:*
    1. Centralização de lógica utilizada por diversos componentes.

2. Fica mais fácil de adicionarmos ou removermos serviços sem que isto afete o código existente.

- *Código Exemplo*

Neste exemplo o usuário quer inserir um novo projeto, ou seja, solicita a página inserirProjeto.jsp.
No servidor existem filtros configurados para interceptar a request e loggar as informações deste acesso (LogAccessFilter) e verificar se o usuário tem permissão para acessar esta página (SecurityCheckerFilter), somente se estas duas operações forem realizadas com sucesso é que a request será redirecionada para a página inserirProjetos.jsp.

```
package br.com.globalcode.designpatterns.interceptingfilter;

import javax.servlet.ServletException;
import javax.servlet.ServletContext;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.http.HttpSession;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import br.com.globalcode.util.Log;

public class LogAccessFilter implements Filter {

 private FilterConfig config = null;
 private Log logger;


 public void init(FilterConfig config) throws ServletException {
     this.config = config;
     String logFile = config.getInitParameter("logfile");
     System.out.println("logfile: " + logFile);
     logger = new Log(logFile, "LogAccessFilter");
```

```
    }

    public void destroy() {
        config = null;
        logger.close();
    }

    public  void doFilter(ServletRequest request, ServletResponse
response, FilterChain chain) throws IOException, ServletException {
        System.out.println("doFilter ");
        HttpServletRequest req = (HttpServletRequest)request;
        String host = req.getRemoteHost();
        String requestedUri = req.getRequestURI();
   logger.logIt("host: " + host + " requested uri : " + requestedUri);
        // próximo filtro ou então uri solicitada
        chain.doFilter(req,response);
    }
```

Deplyment Descriptor: web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>

<filter>
 <filter-name>LogAccessFilter</filter-name>
 <filter-class>

     br.com.globalcode.designpatterns.interceptingfilter.LogAccessFil
     ter
 </filter-class>
 <init-param>
   <param-name>logfile</param-name>
   <param-value>c://logAccessFilter.log</param-value>
 </init-param>
```

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

</web-app>
```

Classe para log em arquivo : Log.java

```
package br.com.globalcode.util;

import java.io.*;
import java.util.Date;

public class Log {

  String nomeArquivo;
  BufferedWriter bufWriter;
  public String header = " | ";

  public Log(String nomeArquivo, String nomeClasse){
    try{
      this.nomeArquivo = nomeArquivo;
      System.out.println("LOG --> " + nomeArquivo);
      this.bufWriter = new BufferedWriter (new
                              FileWriter(this.nomeArquivo,true));
      this.header += nomeClasse + " | ";
    }
    catch(IOException e){
      System.out.println("Nao foi possivel criar/abrir o arg. log ");
      e.printStackTrace();
    }
  }

  public void logIt(String message){
```

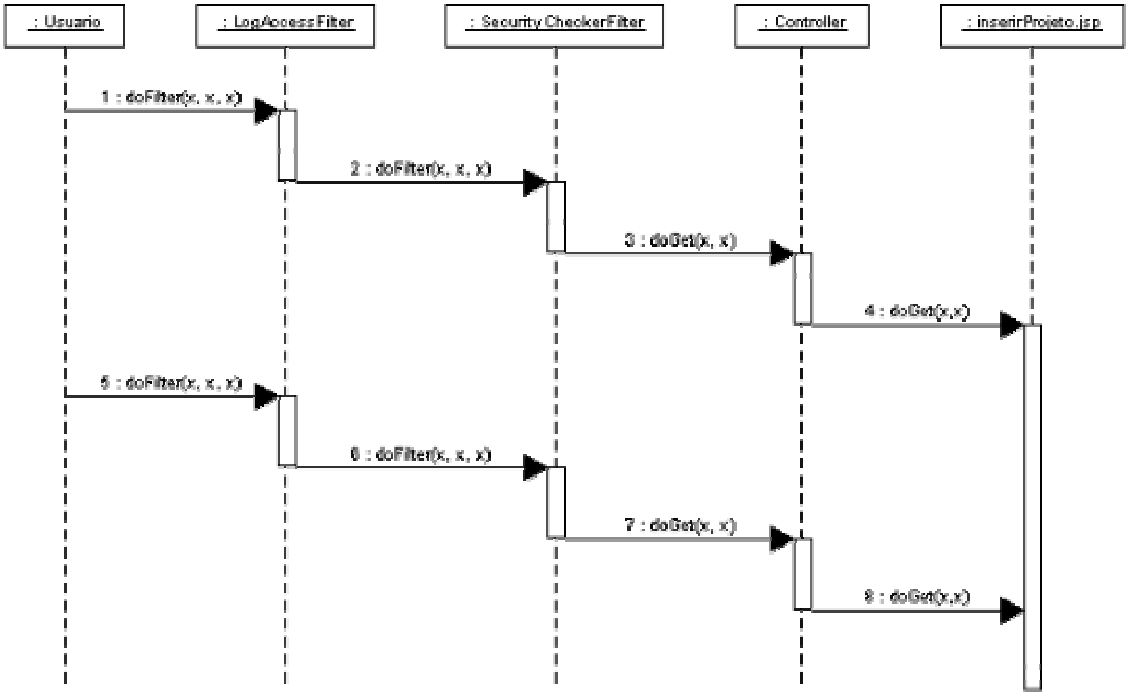
```
      try{
        bufWriter.write(new Date() +" | "  + header + message);
        bufWriter.newLine();
        bufWriter.flush();
      }
      catch(IOException e){
        System.out.println("Nao foi possivel escrever no arquivo de
       log");
        e.printStackTrace();
      }
    }
    //BOA PRATICA:Feche sempre qualquer "canal de comunicação"
       (streams)

    public void close(){
      try{
        bufWriter.close();
      }
      catch(IOException e){
        System.out.println("Nao foi possivel fechar o arquivo de log");
        e.printStackTrace();
      }
     }

    // BOA PRATICA : Utilize o método main para testar o seu
       componente.
    public static void main(String args[]){
      Log meuLog = new Log("meuLog.log","TESTE");
      meuLog.logIt("Dentro do método main!");
      meuLog.close();
    }
  }
```

Jsp para testar o filtro: inserirProjeto.jsp

```
<HTML>
<HEAD>Inserir Projeto</HEAD>
```

```
<BODY>
<FORM name = 'postform' method='POST' action = 'inserirProjeto.jsp'>

<TABLE border = "1">
<TR>
  <TD>Nome do Projeto: </TD>
   <TD><INPUT TYPE = 'text' name = 'txtNomeProjeto'></TD>
</TR>
<TR>
  <TD>Cliente:</TD>
   <TD><INPUT TYPE = 'text' name = 'txtCliente'></TD>
</TR>
<TR>
  <TD>Data estimada de entrega: </TD>
  <TD><INPUT TYPE = 'text' name = 'txtDataEstimadaEntrega'></TD>
</TR>

<TR colspatn = "2">
   <TD><INPUT TYPE = 'submit' name = 'submit' ></TD>
</TR>
</TABLE>
</FORM>
</BODY></HTML>
```

- *UML*


: Usuario
: LogAccessFilter
: SecurityCheckerFilter
: Controller
: inserirProjeto.jsp
1 : doFilter(x, x, x)
2 : doFilter(x, x, x)
3 : doGet(x, x)
4 : doGet(x,x)
5 : doFilter(x, x, x)
6 : doFilter(x, x, x)
7 : doGet(x, x)
8 : doGet(x,x)

Service Locator

- *Problema:* Em aplicações enterprise que utilizam EJB, pool de conexões e outros componentes distribuídos utilizamos JNDI (Java Naming & Directory Interface) para fazermos lookup e obtermos uma referência para estes objetos. Quando esta operação é repetida muitas vezes em diferentes lugares do código, a manutenção é dificultada e a performance pode ser afetada pela criação desnecessária de JNDI inital contexts e número de lookups.

- *Solução:* centralizar e fazer cache em um pequeno conjunto de classes a responsabilidade de localizar e fornecer referências aos principais recursos utilizados na aplicação facilitando assim a manutenção e reduzindo a duplicidade de código;

- *Aplicações práticas:* para que sua classe utilize recursos e componentes J2EE hospedados em um container você deve escrever código de pesquisa no catálogo JNDI além de fabricar instancias quando o caso de EJB. Na prática utilizamos um Service Locator ainda que inconscientemente em classe com nomes típicos como: Acme, Útil e outros;

- *Benefícios:*
  1. Encapsula a complexidade do lookup;
  2. Acesso uniforme aos serviços
  3. Facilita a adição de novos componentes
  4. Aumenta a performance da rede
  5. Aumenta a performance do cliente através de cache

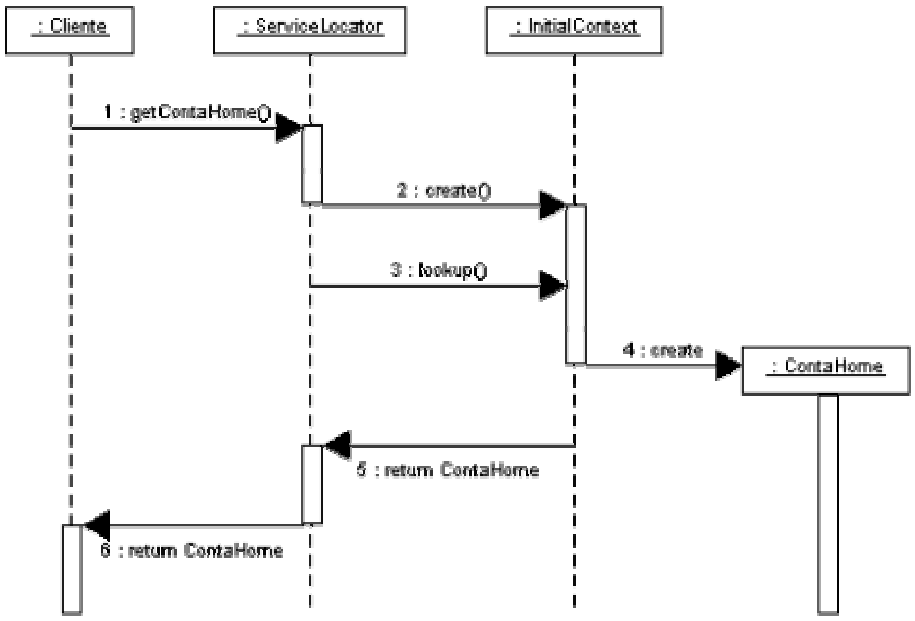
- *Código Exemplo*

Neste exemplo a classe ServiceLocator encapsula através do método getInitialContext() o processo de lookup. Repare que não lançamos uma exception relacionada ao processo de lookup e sim uma exception customizada (CustomException).

```
public class ServiceLocator {
public static final String JNDI_CONTA= "globalcode/ejb/Conta";
private static ContaHome home;

private static Context getInitialContext() throws NamingException{
   return new InitialContext();
}
```

```
public static ContaHome getContaHome() throws CustomException {
try{
   Object ref=getInitialContext().lookup(JNDI_CONTA);
    home = (ContaHome) PortableRemoteObject.narrow(ref, ContaHome.class);
    return home;
}

catch(NamingException ne){
  throw new CustomException(ne);
}
```

- *UML*



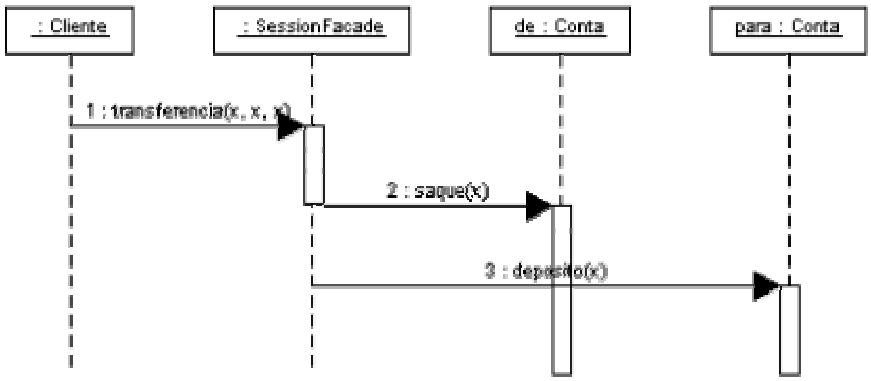
Session Facade

- *Problema:* em aplicações com múltiplas camadas no J2EE enfrentamos os seguintes problemas:
  - *Alto acoplamento entre cliente e objetos de negócio;*
  - *Muitas chamadas a métodos entre o cliente e servidor, resultando em muito tráfego na rede;*
  - *Alta exposição de métodos de negócio induzindo ao uso incorreto de objetos;*

- *Solução:* desenvolver uma classe que represente um processo no servidor e expor para o cliente somente objetos de alto nível que acionam tais processos;

- *Aplicações práticas:* em sistemas J2EE que utilizam a camada Web e EJB para reduzir o acoplamento entre a camada Web e a camada EJB criamos um EJB SessionFacade com um processo transacional que é acionado pela camada Web. Tal processo transacional efetua a chamada a, por exemplo, 5 métodos em outros 5 EJB's. Temos um grande vantagem no sentido de acoplamento pois a camada Web no lugar de ficar vinculada a 5 EJB's vincula-se apenas a um EJB;

- *UML*



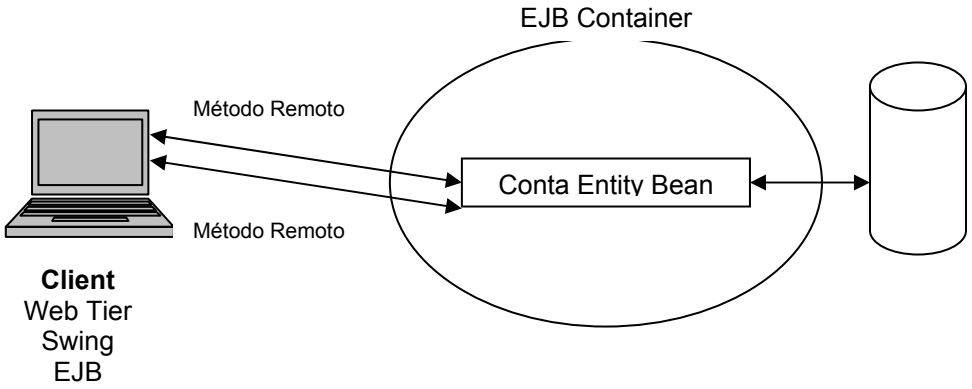
Value Object

- *Problema:* quando trabalhamos com objeto remoto cada método acionado representa uma potencial viagem na rede tornando o custo de acesso ao objeto remoto muito alto;

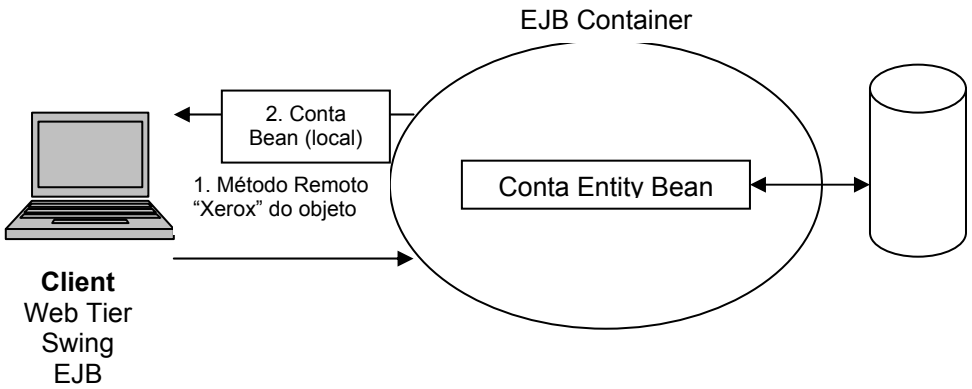
- *Solução:* transferir em uma única chamada uma cópia do objeto remoto para o cliente;

- *Aplicação prática:* sempre que utilizamos entity beans enfrentamos este problema, por exemplo, para uma user interface Swing apresentar os dados de um Pedido é melhor obter um Xerox do Entity Bean a ser apresentado do servidor a efetuar chamadas a getters do entity bean;

- *Errado*



- *Correto*



- *Exemplo*

```
package ejb.entity;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;
import java.util.Date;

public interface Conta extends EJBObject
{
  public void setNome(String nome) throws RemoteException;
  public String getNome() throws RemoteException;
  public void setNumero(String mne) throws RemoteException;
  public String getNumero() throws RemoteException;
  public void setAgencia(String mne) throws RemoteException;
  public String getAgencia() throws RemoteException;
  public double getSaldo() throws RemoteException;
  public ContaValue getLocalConta() throws RemoteException;
}
```

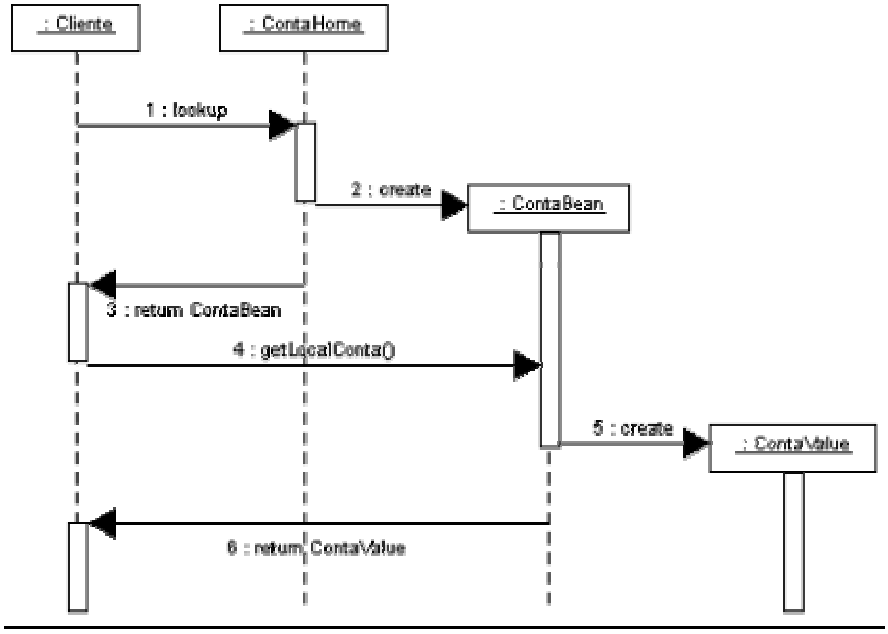
```
package ejb.entity;

import javax.ejb.EntityBean;
import javax.ejb.EntityContext;
import javax.naming.InitialContext;

public class ContaBean extends ContaValue implements EntityBean {
  EntityContext ctx;
  public String ejbCreate (String nome,  String numero, String
                                              agencia, double saldo)
  {
     this.numero = numero;
     this.nome = nome;
     this.agencia = agencia;
     this.saldo = saldo;
     return null;
  }
```

```
  public void ejbPostCreate(String numero) { }
  public void setEntityContext(EntityContext ctx) {
     this.ctx = ctx;
  }
  public void unsetEntityContext() { ctx = null; }
  public void ejbActivate() { }
  public void ejbPassivate() { }
  public void ejbLoad() { }
  public void ejbStore() { }
  public void ejbRemove() { }

  public ContaValue getLocalConta() {
    ContaValue value = new ContaValue(nome,numero,agencia,saldo);
    return value;
  }
}
```

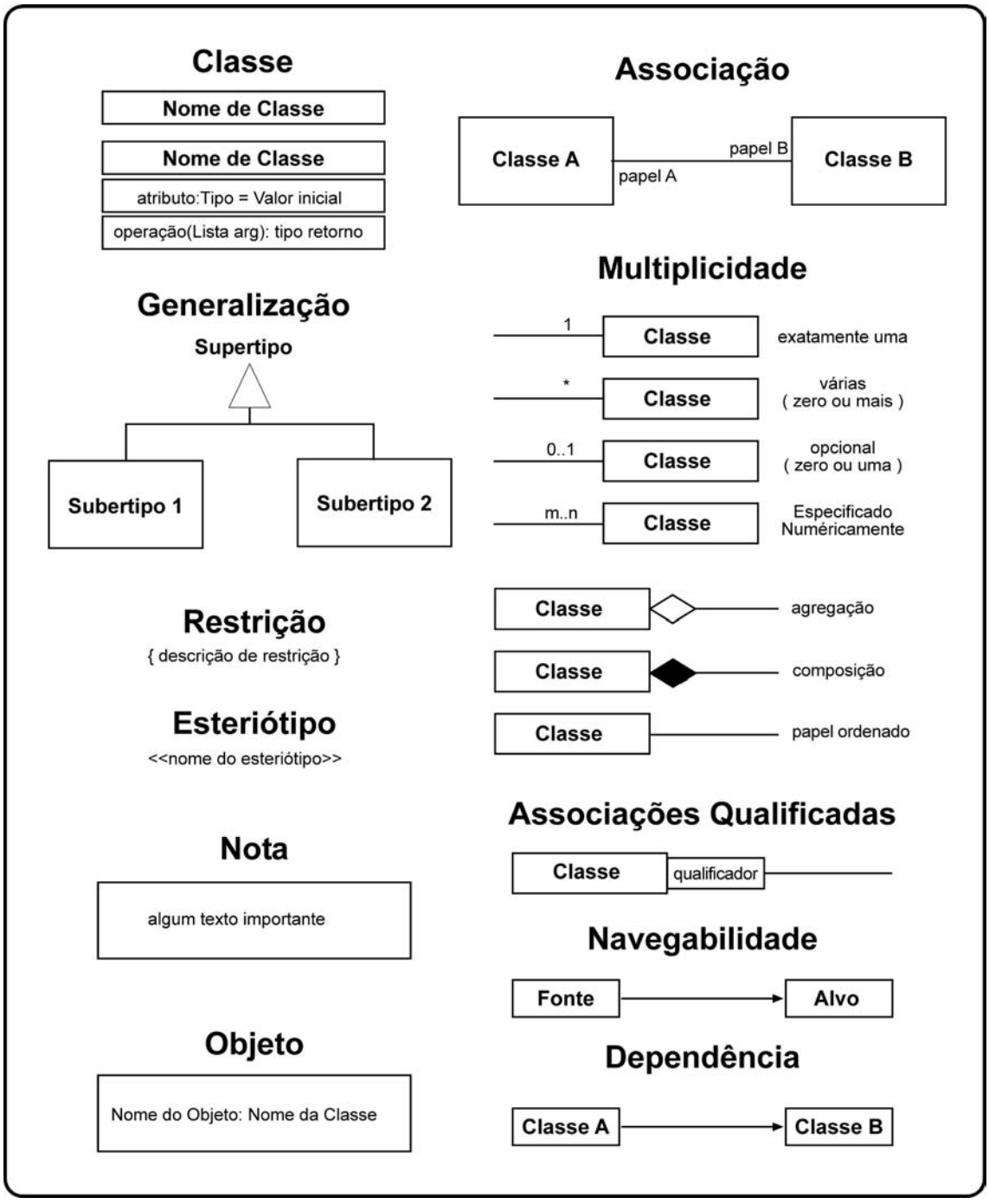
```
package ejb.entity;

import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import javax.ejb.FinderException;
import java.rmi.RemoteException;
import java.util.Collection;

public interface ContaHome extends EJBHome
{
   public Conta create(String nome,  String numero, String agencia,
double saldo) throws RemoteException, CreateException;

   public Conta findByPrimaryKey (String numero) throws
RemoteException, FinderException;

   public Collection findAll() throws RemoteException,
FinderException;
}
```

```
package ejb.entity;
import java.io.Serializable;
public class ContaValue implements Serializable {
    public String nome;
    public String numero;
    public String agencia;
    public double saldo;

    public ContaValue(){}
    public ContaValue(String nome, String numero, String agencia,
                                                    double saldo) {
      this.nome=nome;
      this.numero=numero;
      this.agencia=agencia;
      this.saldo=saldo;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
    public String getNome() {
        return nome;
    }
    public void setNumero(String numero) {
        this.nome = numero;
    }
    public String getNumero() {
        return numero;
    }
    public void setAgencia(String agencia) {
        this.agencia = agencia;
    }
    public String getAgencia() {
        return this.agencia;
    }
    public double getSaldo() {
        return saldo;
    }
}
```

- *UML*


: Cliente
: ContaHome
1 : lookup
2 : create
: ContaBean
3 : return ContaBean
4 : getLocalConta()
5 : create
: ContaValue
6 : return ContaValue

Apêndice I – Resumo notação UML



Apêndice II – Abreviações e terminologias

Abreviação	Descrição
API	Application programming interface
CORBA	Common Object Request Broker Architecture
DB	Database
DBMS	Database Management System
DNS	Domain Name Service
DOM	Document Object Model
EIS	Enterprise Information System
EJB	Enterprise JavaBeans
ERP	Enterprise Resource Planning
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol over SSL
IDL	Interface Definition Language
J2EE	Java 2 Platform, Enterprise Edition
J2SE	Java 2 Platform, Standard Edition
JAF	JavaBeans Activation Framework
JAR	Java Archive
JDBC	Java Database Connectivity
JMS	Java Message Service
JNDI	Java Naming and Directory Interface
JNI	Java Native Interface
JSP	JavaServer Pages
JTA	Java Transaction API
JTS	Java Transaction Service
JVM	Java Virtual Machine
MTBF	Mean Time Between Failures
NDS	Novell Directory Services
ODBC	Open Database Connectivity

OMG	Object Management Group
ORB	Object Request Broker
OS	Operating System
RMI-IIOP	Remote Method Invocation over Internet Inter-ORB Protocol
RPC	Remote Procedure Call
SAX	Simple API for XML
SSL	Secure Socket Layer
TCP/IP	Transmission Control Protocol / Internet Protocol
UML	Unified Modeling Language
URL	Unifrom Resource Locator
WAP	Wireless Application Protocol
XML	eXtensible Markup Language