



## Fundamentos de Taglibs, MVC, Struts e Cactus

*Helder da Rocha*  
[www.argonavis.com.br](http://www.argonavis.com.br)

- *Este capítulo tem como objetivo apenas apresentar os fundamentos básicos de taglibs, frameworks MVC e testes J2EE*
  - *Uma abordagem mais profunda foge do escopo deste curso, que não trata apenas de aplicações Web.*
  - *Para maior aprofundamento, consulte os exemplos de aplicações incluídas no CD (cap08) e as referências no final desta apresentação.*
- *Esta apresentação está dividida em três partes. As duas últimas poderão ser omitidas ou reduzidas caso o instrutor ache necessário (são tópicos opcionais neste curso)*

- *1. Custom tags*
  - *O que são taglibs e custom tags?*
  - *Como usar e criar custom tags*
  - *Tipos de tags e principais componentes*
  - *Fundamentos de JSTL*
- *2. Design de aplicações Web*
  - *Aplicações JSP procedurais: Model 1*
  - *Arquitetura MVC e JSP Model 2*
  - *Usando Struts para implementar JSP Model 2*
- *3. Testes em aplicações Web*
  - *Como instalar o Cactus*
  - *Exemplos de testes de servlets e taglibs com Cactus*

# I. Custom tags

- JSP com JavaBeans fornecem um meio de diminuir código Java da página, mas não totalmente
  - Designers de página ainda têm que usar elementos de script para loops e lógica condicional (getProperty e setProperty não bastam)
  - Nem sempre os JavaBeans são suficientes para encapsular toda a lógica da aplicação
- A especificação prevê a criação de elementos XML personalizados (custom tags) para resolver essas limitações
  - Organizados em bibliotecas (taglibs)
  - Cada biblioteca tem seu próprio namespace
- Taglibs são declaradas no início de cada página ...
- ... e usadas em qualquer lugar

`<%@taglib uri="http://abc.com/ex" prefix="exemplo"%>`

namespace

`<exemplo:dataHoje />`

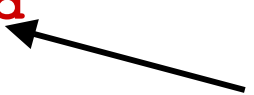
→  
produz

Tuesday, May 5, 2002 13:13:13 GMT-03

# Como usar custom tags

- A URL usada para identificar o prefixo de um custom tag não precisa ser real (e apontar para um local)
  - Serve apenas como **identificador**
  - Ligação entre a especificação da biblioteca (arquivo TLD) e o identificador é feito no arquivo **web.xml**

```
<web-app>
  ...
  <taglib>
    <taglib-uri>http://abc.com/ex</taglib-uri>
    <taglib-location>
      /WEB-INF/mytaglib.tld
    </taglib-location>
  </taglib>
</web-app>
```



Este é o deployment  
descriptor do Taglib.

Localização real!

# Exemplo de arquivo TLD

```
<?xml version="1.0" ?>
<!DOCTYPE taglib PUBLIC
"-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
"http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
```

```
<taglib>
```

```
  <tlib-version>1.0</tlib-version>
```

```
  <jsp-version>1.2</jsp-version>
```

```
  <short-name>exemplo</short-name>
```

```
  <uri>http://abc.com/ex</uri>
```

Sugestão de prefixo  
(autor de página pode  
escolher outro na hora)

URI identifica o prefixo.  
(autor de página tem que  
usar exatamente esta URI)

```
  <tag>
```

```
    <name>DataHoje</name>
```

```
    <tag-class>exemplos.DateTag</tag-class>
```

```
    <description>Data de hoje</description>
```

```
  </tag>
```

```
</taglib>
```

# Implementação

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class DateTag extends TagSupport {
    /**
     * Chamado quando o tag terminar.
     */
    public int doEndTag() throws JspException {
        try {
            Writer out = pageContext.getOut();
            java.util.Date = new java.util.Date();
            out.println(hoje.toString());
        } catch (java.io.IOException e) {
            throw new JspException (e);
        }
        return Tag.EVAL_PAGE;
    }
}
```

Para tags que não precisam processar o corpo use a interface **Tag** ou sua implementação **TagSupport**

Use **doStartTag()** para processamento antes do tag, se necessário

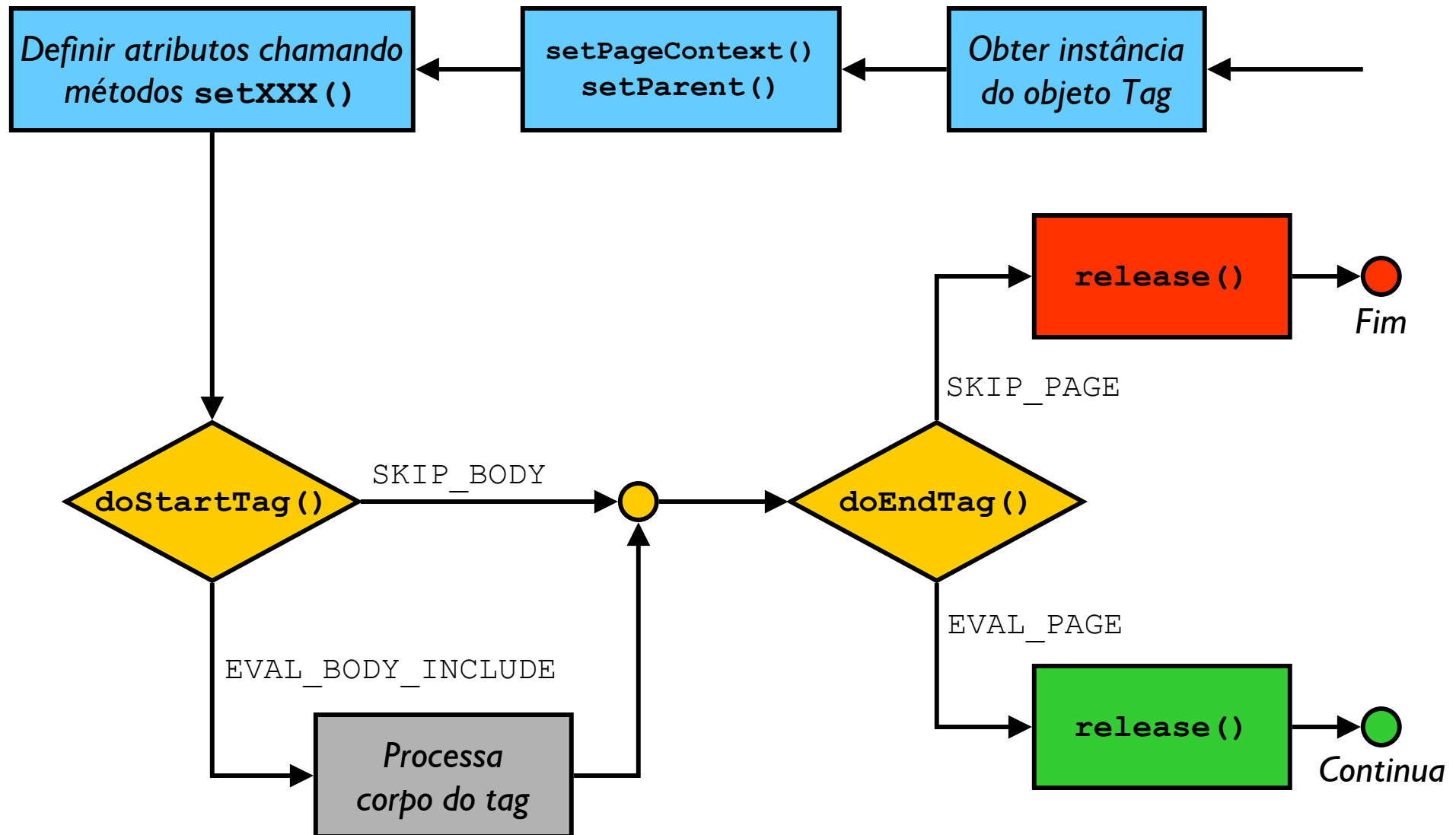
Para este método, pode ser **EVAL\_PAGE** ou **SKIP\_PAGE**

# Tipos de tags

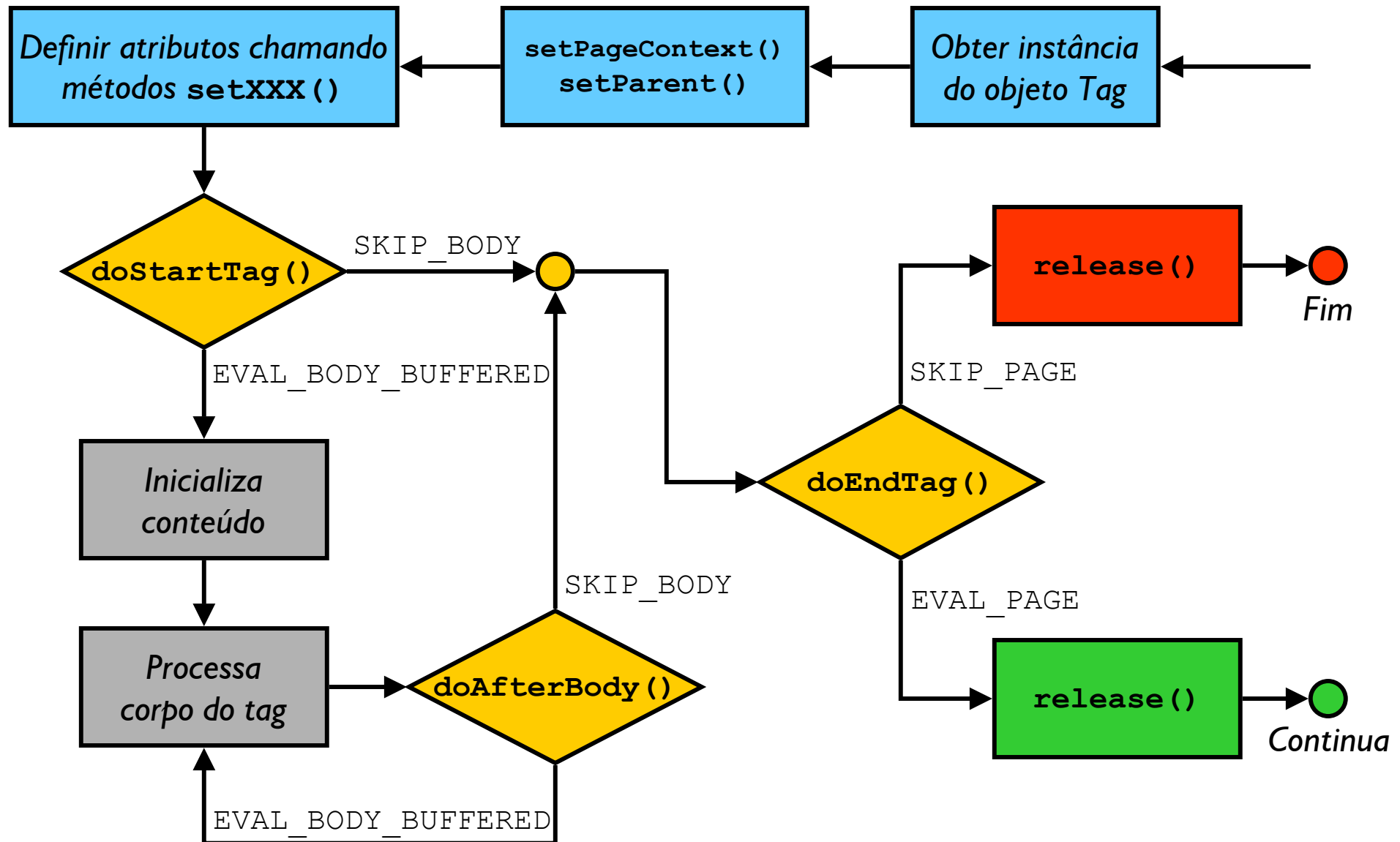
- Há vários tipos de custom tags. Cada estratégia utiliza diferentes classes base e métodos
- 1. Diferenciados por herança:
  - **Tags simples**: implementam a interface **Tag** (**TagSupport** é uma implementação neutra).
  - **Tags com corpo** que requer processamento: implementam **BodyTag** (**BodyTagSupport** é implementação neutra)
- Diferenciados por outras características
  - Tags que possuem **atributos**
  - Tags que **definem variáveis de scripting** fora do seu escopo (requerem classe extra com "Tag Extra Info")
  - Tags que **interagem** com outros tags



# Ciclo de vida de objetos Tag



# Ciclo de vida de objetos BodyTag



- Para definir atributos em um tag é preciso

1. Definir método

*setXXX()* com o nome do atributo

```
<xyz:upperCase text="abcd" />
```

```
public class UpperCaseTag {  
    public String text;  
    public void setText(String text) {  
        this.text = text;  
    } (...)
```

2. Declarar atributo no descritor (TLD)

```
<tag> <name>upperCase</name> (...)  
    <attribute>  
        <name>text</name>  
        <required>true</required>  
        <rtexprvalue>false</rtexprvalue>  
    </attribute> (...) </tag>
```

- Os atributos devem setar campos de dados no tag
  - Valores são manipulados dentro dos métodos *doXXX()*:

```
public int doStartTag() throws JspException {  
    Writer out = pageContext.getOut();  
    out.println(text.toUpperCase()); (...)
```

# Obtenção do conteúdo do Body

- O objeto **out**, do JSP, referencia a instância *BodyContent* de um tag enquanto processa o corpo
  - *BodyContent* é subclasse de *JspWriter*
  - Tag decide se objeto *BodyContent* deve ser jogado fora ou impresso (na forma atual ou modificada)
- Exemplo

```
public int doAfterBody() throws JspException {  
    BodyContent body = getBodyContent();  
    String conteudo = body.getString();  
    body.clearBody();  
    (...)  
    getPreviousOut().print(novoTexto);  
}
```

← Guarda conteúdo do tag

← Apaga conteúdo

← Imprime texto na página (e não no body do Tag)

# Exemplos de Custom Tags

- *Veja exemplos/cap08/taglibs/*
  - *Vários diferentes exemplos de custom tags (do livro [6])*
  - *Código fonte em taglib/src/taglibdemo/\*.java*
  - *Páginas exemplo em src/\*Test.jsp (6 exemplos)*
  - 1. Configure build.properties, depois, monte o WAR com:*  
*> ant build*
  - 2. Copie o WAR para o diretório webapps do Tomcat*  
*> ant deploy*
  - 3. Execute os tags, acessando as páginas via browser:*  
*http://localhost:porta/mut/*
- *Veja também exemplos/cap08/mvc/hellojsp\_2/*
  - *Aplicação MVC que usa custom tags (veja como executar na próxima seção)*

# JSP Standard Tag Library

- *Esforço de padronização do JCP: JSR-152*
  - *Baseado no Jakarta Taglibs (porém bem menor)*
- *Oferece dois recursos*
  - *Conjunto padrão de tags básicos (Core, XML, banco de dados e internacionalização)*
  - *Linguagem de expressões do JSP 1.3*
- *Oferece mais controle ao autor de páginas sem necessariamente aumentar a complexidade*
  - *Controle sobre dados sem precisar escrever scripts*
  - *Estimula a separação da apresentação e lógica*
  - *Estimula o investimento em soluções MVC*

# Como usar JSTL

- 1. Fazer o download da última versão do site da Sun
- 2. Copiar os JARs das bibliotecas desejadas para o diretório WEB-INF/lib/ da sua aplicação Web e os arquivos TLD para o diretório WEB-INF/
- 3. Declarar cada taglib e associá-la com seu TLD no deployment descriptor web.xml.
- 4. Incluir em cada página que usa os tags:  

```
<%@ taglib uri="uri_da_taglib"  
        prefix="prefixo" %>
```
- 5. Usar os tags da biblioteca com o prefixo definido no passo anterior  

```
<prefixo:nomeTag atributo="..."> ...  
</prefixo:nomeTag>
```

# Quatro bibliotecas de tags

- *Core library: tags para condicionais, iterações, urls, ...*  
`<%@ taglib uri="http://java.sun.com/jstl/ea/core" prefix="c" />`
  - *Exemplo: <c:if test="..." ... >...</c:if>*
- *XML library: tags para processamento XML*  
`<%@ taglib uri="http://java.sun.com/jstl/ea/xml" prefix="x" />`
  - *Exemplo: <x:parse>...</x:parse>*
- *Internationalization library*  
`<%@ taglib uri="http://java.sun.com/jstl/ea/fmt" prefix="fmt" />`
  - *Exemplo: <fmt:message key="..." />*
- *SQL library*  
`<%@ taglib uri="http://java.sun.com/jstl/ea/sql" prefix="sql" />`
  - *Exemplo: <sql:update>...</sql:update>*



# Linguagem de expressões

- *Permite embutir em atributos expressões dentro de delimitadores `${...}`*
  - *Em vez de `request.getAttribute("nome")`  
`${nome}`*
  - *Em vez de `bean.getPessoa().getNome()`  
`${bean.pessoa.nome}`*
- *Suporta operadores aritméticos, relacionais e binários*
- *Converte tipos automaticamente*  
`<tag item="${request.valorNumerico}" />`
- *Valores default*  
`<tag value="${abc.def}" default="todos" />`

# Principais ações

- *Suporte à impressão da linguagem expressões*

- `<c:out value="${pessoa.nome}" />`

- *Expressões condicionais*

- `<c:if test="${pessoa.idade >= 18}">`

- `<a href="adultos.html">Entrar</a>`

- `</c:if>`

- `<c:choose>`

- `<c:when test="${dia.hora == 13}">`

- `<c:out value="${mensagemEspecial}" />`

- `</c:when>`

- `<c:otherwise>`

- `<c:out value="${mensagemPadrao}" />`

- `</c:otherwise>`

- `</c:choose>`

- *Iteração*

- `<c:forEach items="${pessoas}" var="p" varStatus="s">`

- `<c:out value="${s.count}"/>. <c:out value="${p}"/>`

- `</c:forEach>`

# Internacionalização, XML e SQL

- *Ler propriedade de ResourceBundle*
  - `<fmt:message key="chave.do.bundle" />`
- *Operações diretas em banco de dados*
  - `<sql:query dataSource="${dsn}">`  
`SELECT...</sql:query>`
  - `<sql:transaction>`, `<sql:update>`, etc.
- *Operações com XML*
  - *Uso de expressões XPath em tags JSTL para XML*
  - *Ações XML: <x:out>, <x:set>, <x:if>, <x:choose>, <x:forEach> (atributo select contém expr. XPath)*
  - `<x:parse>` *Processa XML usando DOM ou filtro SAX*
  - `<x:transform>` *Realiza transformação XSLT.*

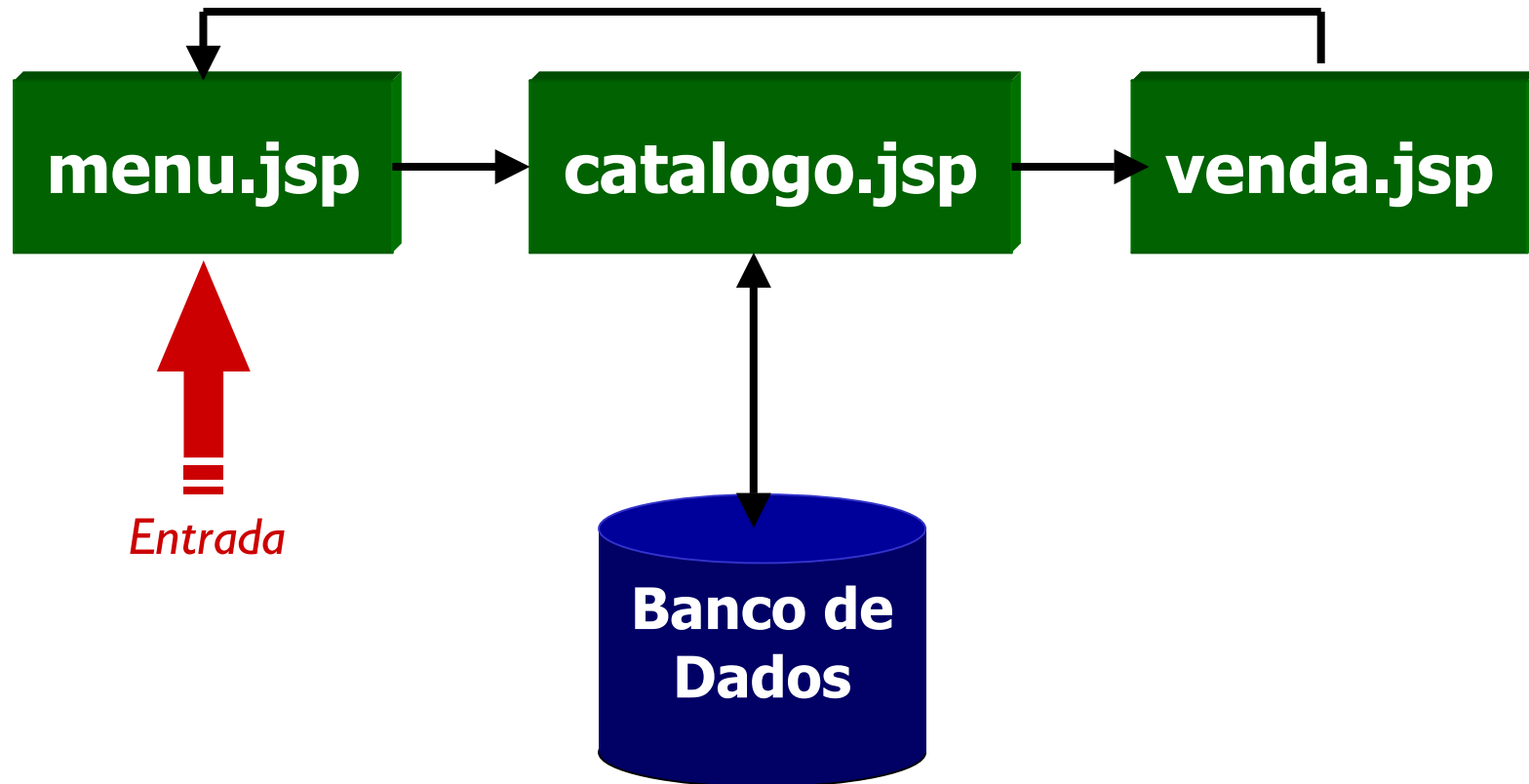
## 2. Design de aplicações JSP

- *Design centrado em páginas*
  - *Aplicação JSP consiste de seqüência de páginas (com ou sem beans de dados) que contém código ou links para chamar outras páginas*
- *Design centrado em servlet (FrontController\* ou MVC)*
  - *Aplicação JSP consiste de páginas, beans e servlets que controlam todo o fluxo de informações e navegação*
  - *Este modelo favorece uma melhor organização em camadas da aplicação, facilitando a manutenção e promovendo o reuso de componentes.*
  - *Um único servlet pode servir de fachada*
  - *Permite ampla utilização de J2EE design patterns*

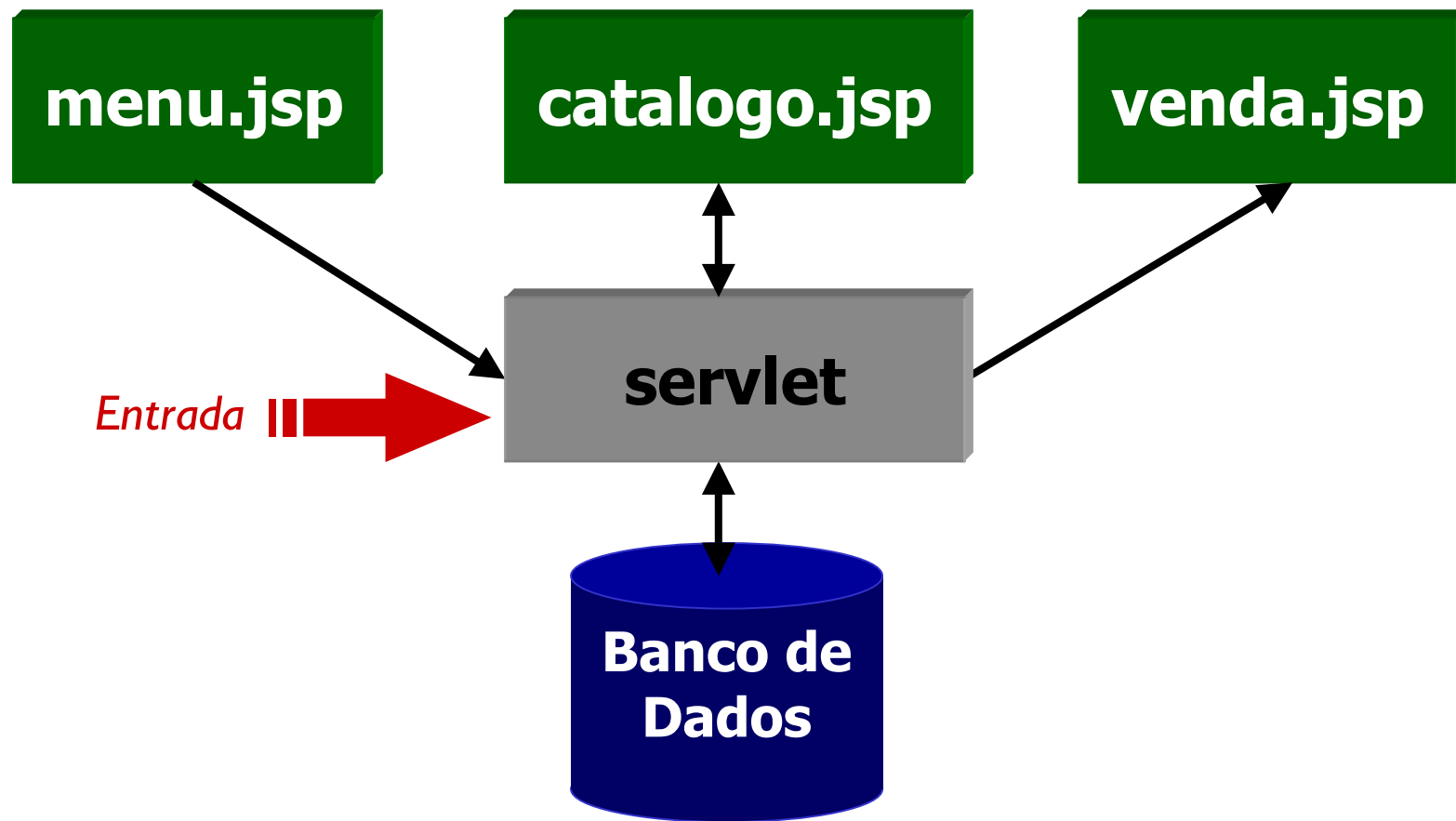
---

\* *FrontController é um J2EE design pattern. Vários outros design patterns serão identificados durante esta seção. Para mais informações, veja Sun Blueprints [7]*

# Layout centrado em páginas (JSP Model 1)

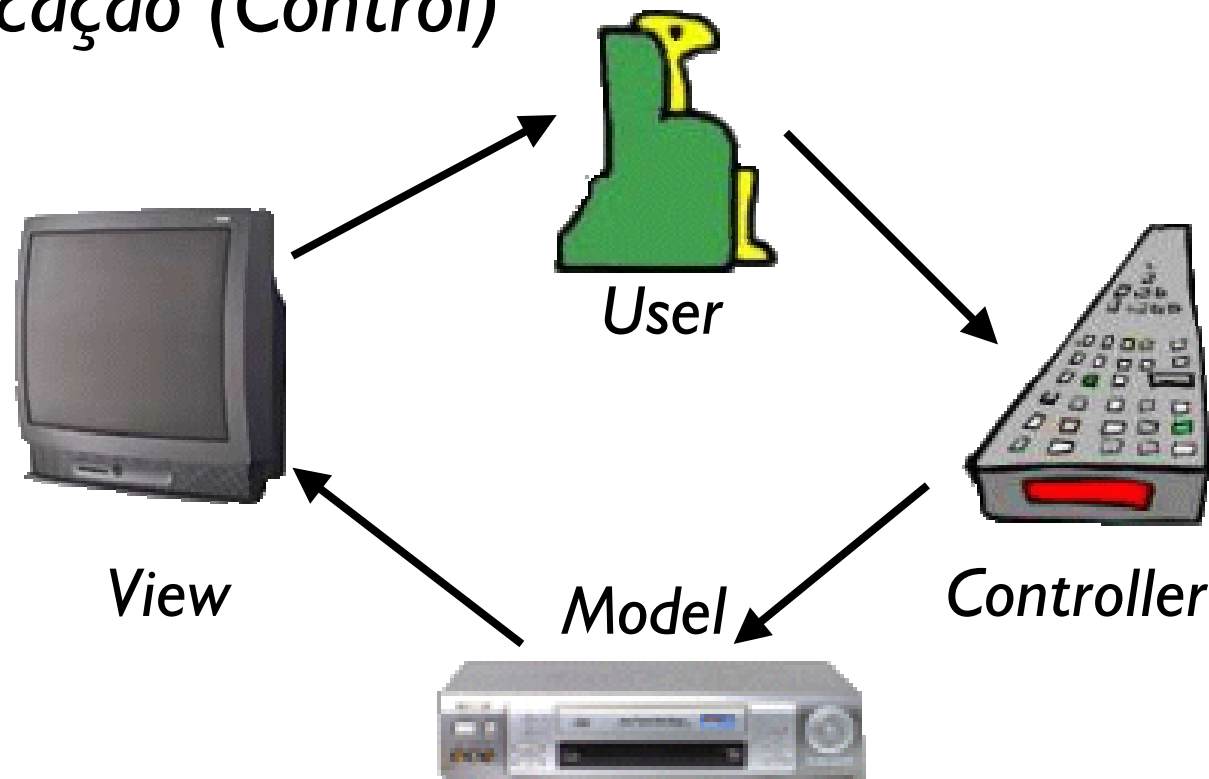


# Layout centrado em servlet (JSP Model 2)



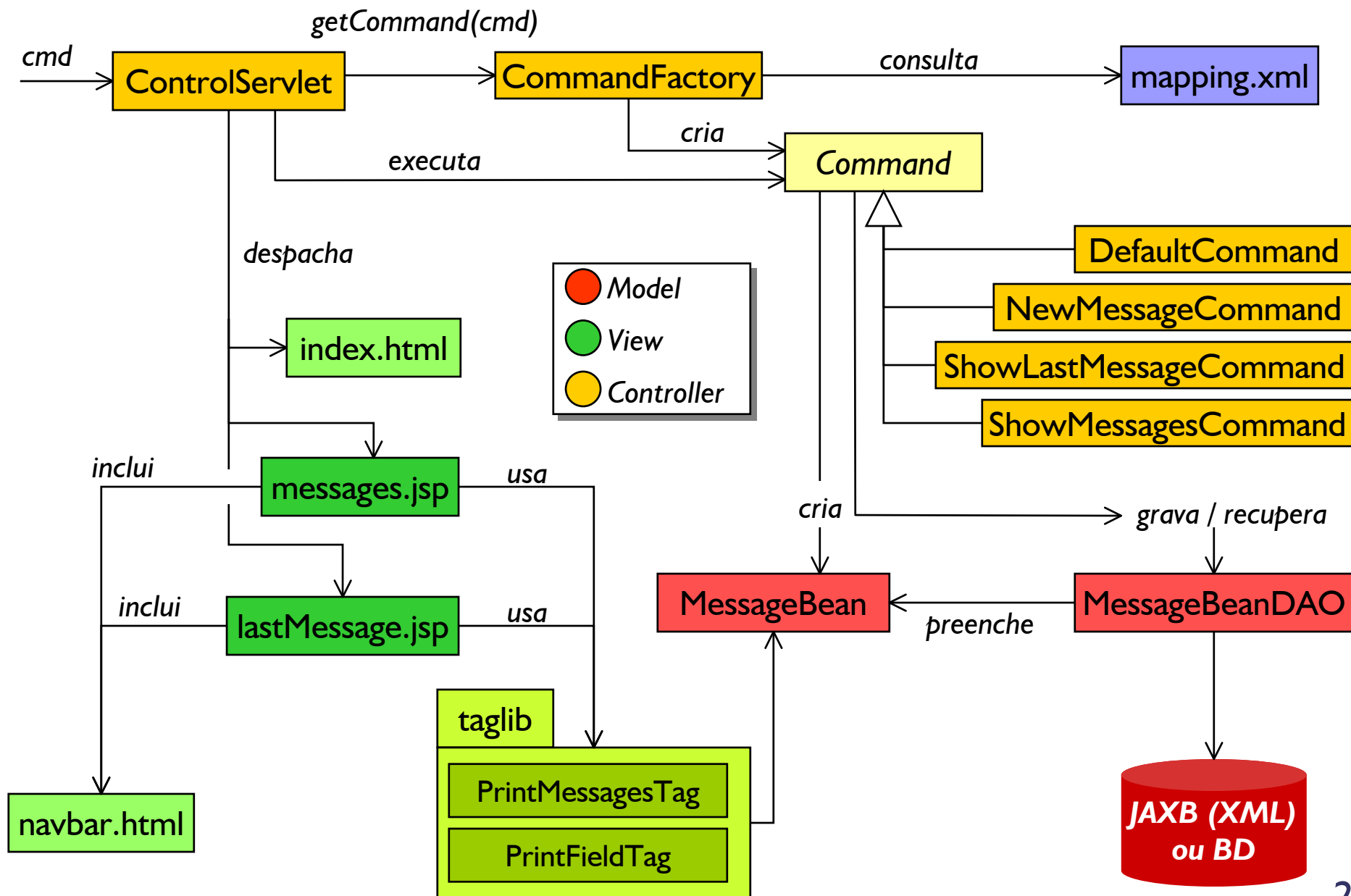
# O que é MVC

- Padrão de arquitetura: **M**odel **V**iew **C**ontroller
- Técnica para separar dados ou lógica de negócios (Model) da interface do usuário (View) e do fluxo da aplicação (Control)



# Exemplo de implementação

exemplos/cap08/mvc/hellojsp\_2





# Mapeamentos de comandos ou ações

- No exemplo `hellojsp_2`, o **mapeamento** está armazenado em um arquivo XML (`webinf/mapping.xml`)

```
<command-mapping> (...)  
  <command>  
    <name>default</name>  
    <class>hello.jsp.DefaultCommand</class>  
    <success-url>/index.html</success-url>  
    <failure-url>/index.html</failure-url>  
  </command>  
  <command>  
    <name>newMessage</name>  
    <class>hello.jsp.NewMessageCommand</class>  
    <success-url>/lastMessage.jsp</success-url>  
    <failure-url>/index.html</failure-url>  
  </command>  
  <command>  
    <name>showAllMessages</name>  
    <class>hello.jsp.ShowMessagesCommand</class>  
    <success-url>/messages.jsp</success-url>  
    <failure-url>/index.html</failure-url>  
  </command>  
</command-mapping>
```

# Comandos ou ações (Service to Worker)

- Comandos implementam a interface **Command** e seu método `Object execute(HttpServletRequest request, HttpServletResponse response, MessageBeanDAO dao);`
- Criados por **CommandFactory** na inicialização e executados por `ControlServlet` que os obtém via **getCommand(nome)**
- Retornam página de sucesso ou falha (veja `mapping.xml`)
- Exemplo: `ShowMessagesCommand`:

```
public class ShowMessagesCommand implements Command {  
  
    public Object execute(...) throws CommandException {  
        try {  
            MessageBean[] beanArray = dao.retrieveAll();  
            request.setAttribute("messages", beanArray);  
            return successUrl;  
        } catch (PersistenceException e) {  
            throw new CommandException(e);  
        }  
    }  
}
```

# Data Access Objects (DAO)

- *Isolam a camada de persistência*
  - *Implementamos persistência JAXB, mas outra pode ser utilizada (SGBDR) sem precisar mexer nos comandos.*
- *Interface da DAO:*

```
public interface MessageBeanDAO {  
    public Object getLocator();  
  
    public void persist(MessageBean messageBean)  
                throws PersistenceException;  
  
    public MessageBean retrieve(int key)  
                throws PersistenceException;  
  
    public MessageBean[] retrieveAll()  
                throws PersistenceException;  
  
    public MessageBean retrieveLast()  
                throws PersistenceException;  
}
```

# Controlador (FrontController)

- Na nossa aplicação, o controlador é um **servlet** que recebe os nomes de comandos, executa os objetos que os implementam e repassam o controle para a página JSP ou HTML retornada.

```
public void service( ..., ... ) ... {  
    Command command = null;  
    String commandName = request.getParameter("cmd");  
  
    if (commandName == null) {  
        command = commands.getCommand("default");  
    } else {  
        command = commands.getCommand(commandName);  
    }  
  
    Object result = command.execute(request, response, dao);  
    if (result instanceof String) {  
        RequestDispatcher dispatcher =  
            request.getRequestDispatcher((String)result);  
        dispatcher.forward(request, response);  
    }  
    ...  
}
```

*Método de CommandFactory*

*Execução do comando retorna uma URI*

*Repassa a requisição para página retornada*

# ValueBean ViewHelper (Model)

- *Este bean é gerado em tempo de compilação a partir de um DTD (usando ferramentas do JAXB)*

```
public class MessageBean
    extends MarshallableRootElement
    implements RootElement {

    private String _Time;
    private String _Host;
    private String _Message;

    public String getTime() {...}
    public void setTime(String _Time) {...}

    public String getHost() {...}
    public void setHost(String _Host) {...}

    public String getMessage() {...}
    public void setMessage(String _Message) {...}

    ...
}
```

← interfaces JAXB permitem que este bean seja gravado em XML (implementa métodos `marshal()` e `unmarshal()` do JAXB)

# Página JSP (View) com custom tags

## ■ Página messages.jsp (mostra várias mensagens)

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<%@ taglib uri="/hellotags" prefix="hello" %>
<html>
<head><title>Show All Messages</title></head>
<body>
<jsp:include page="navbar.html" />
<h1>Messages sent so far</h1>
<table border="1">
<tr><th>Time Sent</th><th>Host</th><th>Message</th></tr>

<hello:printMessages array="messages">
  <tr>
    <td><hello:printField property="time" /></td>
    <td><hello:printField property="host" /></td>
    <td><hello:printField property="message" /></td>
  </tr>
</hello:printMessages>

</table>
</body>
</html>
```

# Para executar o exemplos

- 1. Mude para `exemplos/cap08/mvc/hellojsp_2`
- 2. Configure `build.properties`, depois rode  
> `ant DEPLOY`
- 3. Inicie o servidor (Tomcat ou JBoss)
- 4. Rode os testes do Cactus  
> `ant RUN-TESTS`
- 5. Rode a aplicação, acessando a URI  
`http://localhost:porta/hellojsp/`
- 6. Digite mensagens e veja resultados. Arquivos são gerados em `/tmp/mensagens` (ou `c:\tmp\mensagens`)

- *Framework para facilitar a implementação da arquitetura MVC em aplicações JSP*
- *Oferece*
  - *Um servlet controlador configurável através de documentos XML externos, que despacham requisições a classes Action (comandos) criadas pelo desenvolvedor*
  - *Uma vasta coleção de bibliotecas de tags JSP (taglibs)*
  - *Classes utilitárias que oferecem suporte a tratamento de XML, preenchimento de JavaBeans e gerenciamento externo do conteúdo de interfaces do usuário*
- *Onde obter: [jakarta.apache.org/struts](http://jakarta.apache.org/struts)*



# Componentes MVC no Struts

- **Model (M)**
  - Geralmente um objeto Java (JavaBean)
- **View (V)**
  - Geralmente uma página HTML ou JSP
- **Controller (C)**
  - `org.apache.struts.action.ActionServlet` ou subclasse
- **Classes ajudantes**
  - *FormBeans*: encapsula dados de forms HTML (M)
  - *ActionErrors*: encapsulam dados de erros (M)
  - *Custom tags*: encapsulam lógica para apresentação (V)
  - *Actions*: implementam lógica dos comandos (C)
  - *ActionForward*: encapsulam lógica de redirecionamento (C)

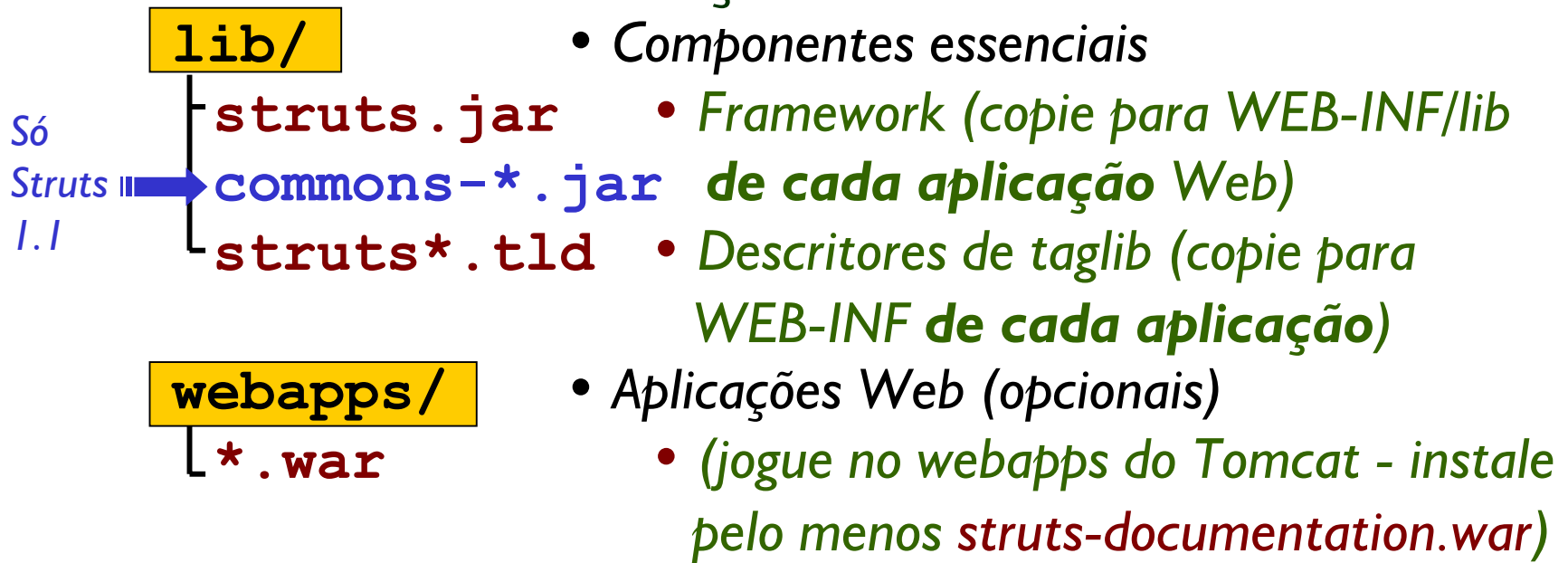
# Componentes da distribuição

## ■ Requisitos

- J2SDK 1.4 ou J2SDK 1.3 + JAXP
- Servlet container, servlet.jar e Jakarta Commons (Struts 1.1)

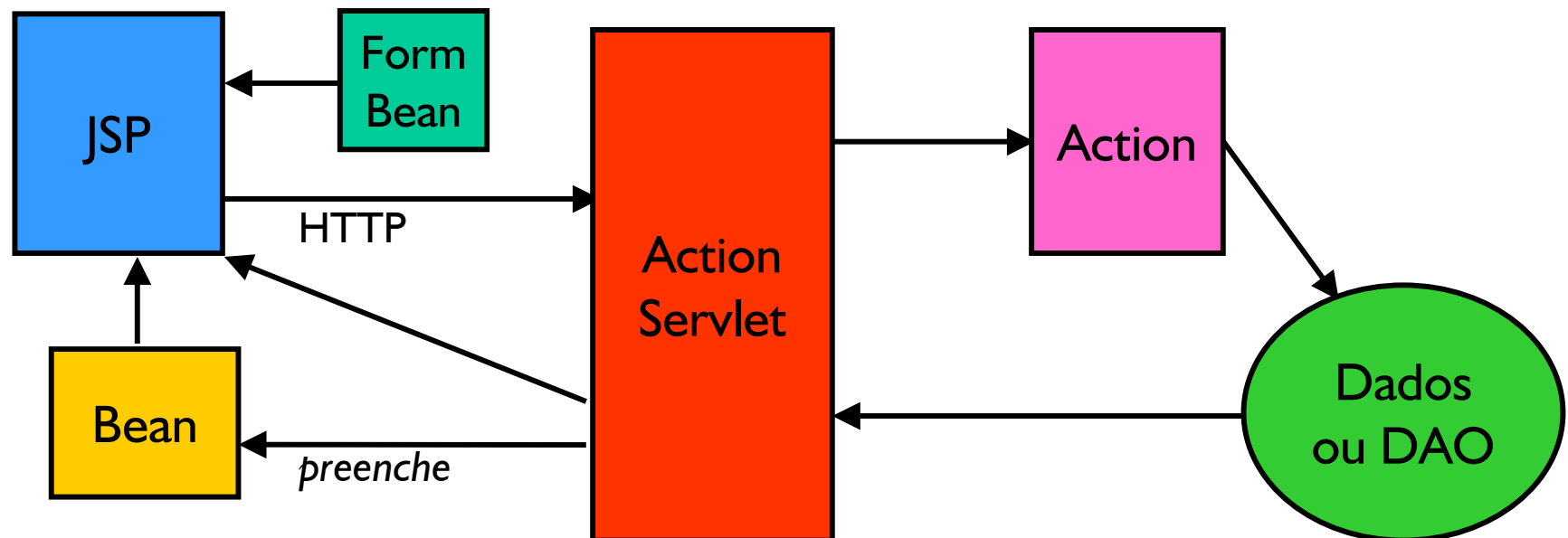
## ■ Distribuição binária (pré-compilada)

- Abra o ZIP da distribuição. Conteúdo essencial:



# Como funciona?

- **Principais componentes**
  - **ActionServlet**: despachante de ações
  - **Action**: classe estendida por cada ação (comando) a ser implementada (usa Command design pattern)
  - **struts-config.xml**: arquivo onde se define mapeamentos entre ações, páginas, beans e dados



# Como instalar

- 1. Copiar os arquivos necessários para sua aplicação
  - Copie *lib/struts.jar* e *lib/commons-\*.jar* para seu *WEB-INF/lib* (não coloque no *common/lib* do Tomcat ou no *jre/lib/ext* do JDK ou o struts não achará suas classes!)
  - Copie os TLDs das bibliotecas de tags que deseja utilizar para o *WEB-INF* de sua aplicação (copie todos)
- 2. Para usar o servlet controlador (MVC)
  - Defina-o como um `<servlet>` no seu *web.xml*
  - Crie um arquivo *WEB-INF/struts.config.xml* com mapeamentos de ações e outras as configurações
- 3. Para usar cada conjunto de taglibs
  - Defina, no seu *web.xml*, cada taglib a ser instalada
  - Carregue a taglib em cada página JSP que usá-la

# Configuração do controlador no web.xml

- Acrescente no seu web.xml

```
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>
    org.apache.struts.action.ActionServlet
  </servlet-class>
  <init-param>
    <param-name>config</param-name>
    <param-value>
      /WEB-INF/struts-config.xml
    </param-value>
  </init-param>
  ... outros init-param ...
</servlet>
```

- Acrescente também os <servlet-mapping> necessários
- Crie e configure as opções de [struts-config.xml](#)
- Veja nos docs: [/userGuide/building\\_controller.html](#)
  - use os arquivos de struts-example.war para começar

# Configuração das Taglibs

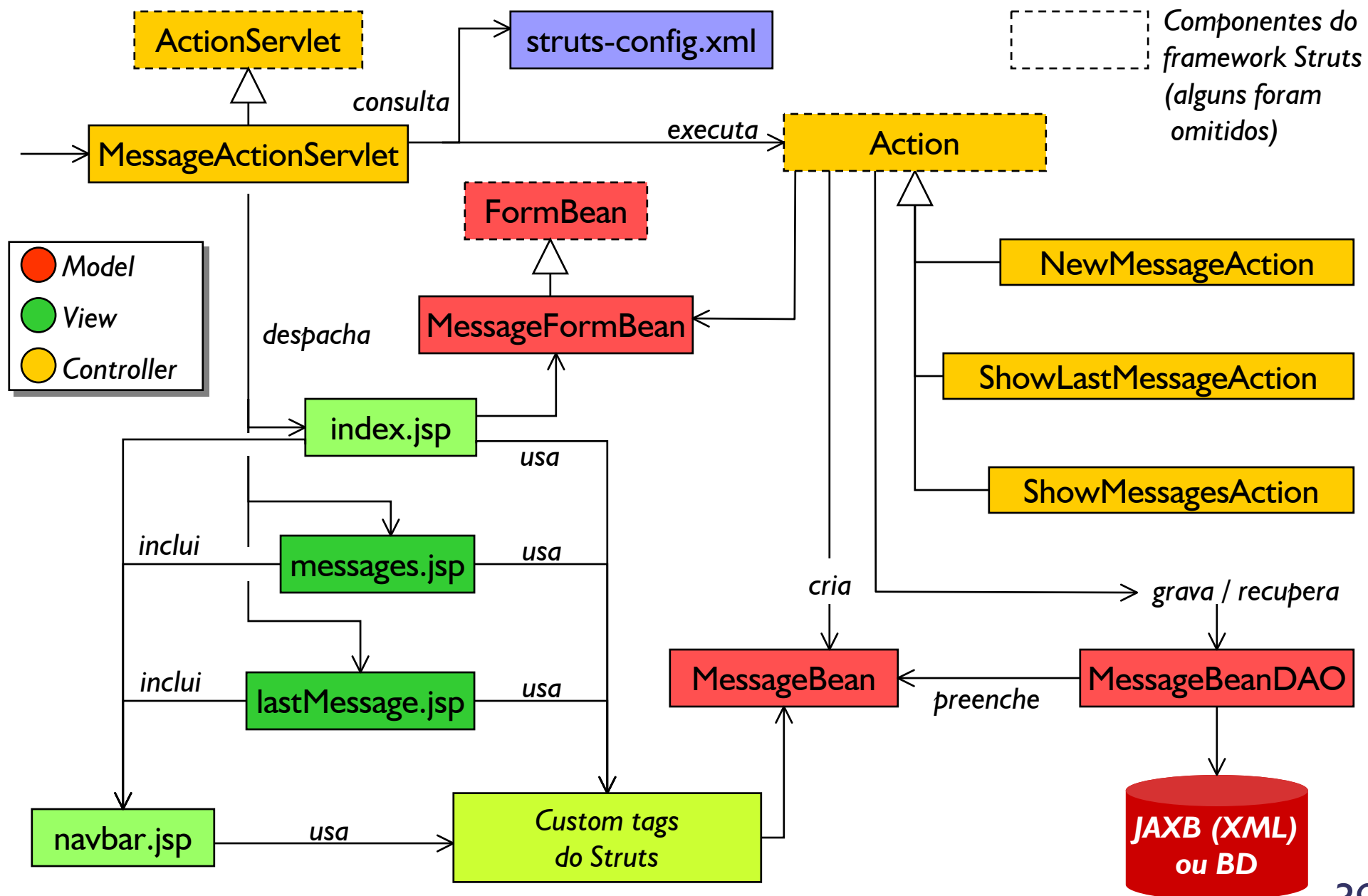
- Acrescente em **web.xml**
- Veja detalhes na aplicação *struts-example.war* ou nos docs:  
*[/userGuide/building\\_controller.html#dd\\_config\\_taglib](#)*

```
<taglib>
  <taglib-uri>/WEB-INF/struts-bean.tld</taglib-uri>
  <taglib-location>/WEB-INF/struts-bean.tld
</taglib-location>
</taglib>
<taglib>
  <taglib-uri>/WEB-INF/struts-form.tld</taglib-uri>
  <taglib-location>/WEB-INF/struts-form.tld
</taglib-location>
... outros taglibs ...
</taglib>
```

- Acrescente em cada página JSP

```
<@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
...
```

# Implementação de hellojsp com Struts



# Mapeamentos (ActionMappings)

## ■ Veja webinf/struts-config.xml

```
<struts-config>
  <form-beans>
    <form-bean name="newMessageForm" type="hello.jsp.NewMessageForm" />
  </form-beans>
  <global-forwards>
    <forward name="default" path="/index.jsp" />
  </global-forwards>
```

```
<action-mappings>
  <action path="/newMessage" type="hello.jsp.NewMessageAction"
    validate="true"
    input="/index.jsp" name="newMessageForm" scope="request">
    <forward name="success" path="/showLastMessage.do" />
  </action>
  <action path="/showLastMessage"
    type="hello.jsp.ShowLastMessageAction" scope="request">
    <forward name="success" path="/lastMessage.jsp" />
  </action>
  <action path="/showAllMessages"
    type="hello.jsp.ShowMessagesAction" scope="request">
    <forward name="success" path="/messages.jsp" />
  </action>
</action-mappings>
```

```
  <message-resources parameter="hello.jsp.ApplicationResources" />
</struts-config>
```



# FormBeans

- *Form beans permitem simplificar a leitura e validação de dados de formulários*
  - *Devem ser usados em conjunto com custom tags da biblioteca `<html:* />`*

```
<html:form action="/newMessage" name="newMessageForm"
            type="hello.jsp.NewMessageForm">
  <p>Message: <html:text property="message" />
    <html:submit>Submit</html:submit>
  </p>
</html:form>
```

Configuração em  
struts-config.xml

```
public class NewMessageForm extends ActionForm {
  private String message = null;
  public String getMessage() { return message; }
  public void setMessage(String message) {
    this.message = message;
  }
  public void reset(...) {
    message = null;
  }
  public ActionErrors validate(...) {...}
}
```

- *ActionErrors* encapsulam erros de operação, validação, exceções, etc.
  - *Facilitam a formatação e reuso de mensagens de erro.*
- *Exemplo: Método validate() do form bean:*

```
public ActionErrors validate(ActionMapping mapping,
                           HttpServletRequest request) {
    ActionErrors errors = new ActionErrors();
    if ( (message == null) || (message.trim().length() == 0) ) {
        errors.add("message",
                  new ActionError("empty.message.error"));
    }
    return errors;
}
```

- *Como imprimir:*

`<html:errors />`

Nome de campo  
<input> ao qual o  
erro se aplica.

Este valor corresponde a uma  
chave no ResourceBundle

- *Informações localizadas podem ser facilmente extraídas de Resource Bundles através de*  
`<bean:message key="chave" />`
  - *Locale default é usado automaticamente (pode ser reconfigurado)*
- *Exemplo de ResourceBundle*

```
empty.message.error=<tr><td>Mensagem não pode ser  
vazia ou conter apenas espaços em branco.</td></tr>  
new.message.input.text=Digite a sua mensagem  
message.submit.button=Enviar Mensagem
```

*hello/jsp/ApplicationResources\_pt.properties*

- *Configuração em struts-config.xml*  
`<message-resources  
parameter="hello.jsp.ApplicationResources" />`
- *Exemplo de uso:*  
`<p><bean:message key="new.message.input.text" />`

# Action (Controller / Service To Worker)

- *Controlador processa comandos chamando o método execute de um objeto Action*

```
public class ShowMessagesAction extends Action {  
  
    private String successTarget = "success";  
    private String failureTarget = "default";  
  
    public ActionForward execute(ActionMapping mapping,  
                                ActionForm form,  
                                HttpServletRequest request,  
                                HttpServletResponse response)  
                                throws IOException, ServletException {  
        try {  
            MessageBeanDAO dao =  
                (MessageBeanDAO) request.getAttribute("dao");  
            MessageBean[] beanArray = dao.retrieveAll();  
            request.setAttribute("messages", beanArray);  
            return (mapping.findForward(successTarget));  
        } catch (PersistenceException e) {  
            throw new ServletException(e);  
        }  
    }  
} ...
```

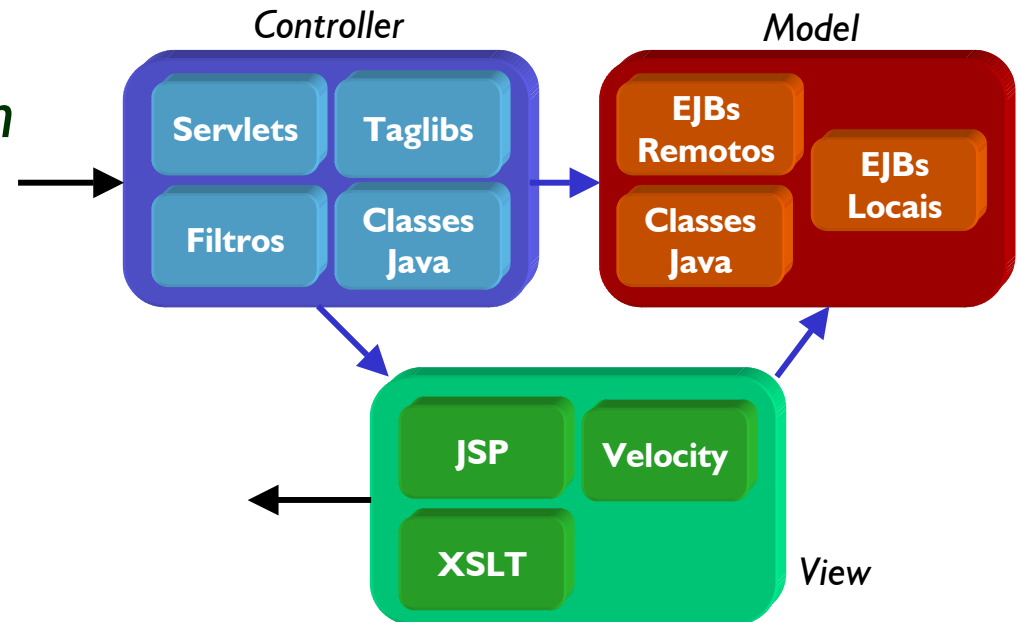
# Como rodar o exemplo

- 1. Mude para `exemplos/cap08/mvc/hellojsp_3`
- 2. Configure `build.properties`, depois rode  
    > `ant DEPLOY`
- 3. Inicie o servidor (Tomcat ou JBoss)
- 4. Rode os testes do Cactus  
    > `ant RUN-TESTS`
- 5. Rode a aplicação, acessando a URI  
    `http://localhost:porta/hellojsp-struts/`
- 6. Digite mensagens e veja resultados. Arquivos são gerados em `/tmp/mensagens` (ou `c:\tmp\mensagens`)

- *É um framework que oferece facilidades para testar componentes J2EE*
  - *Componentes Web (Camada de **C**ontrole)*
  - *Camada EJB (**M**odel) e cliente (**V**iew): indiretamente*
- *Produto Open Source do projeto Jakarta*
  - *Metas de curto prazo: testar componentes acima + EJB*
  - *Metas de longo prazo: oferecer facilidades para testar todos os componentes J2EE; ser o framework de referência para testes in-container.*
- *Cactus estende o JUnit framework*
  - *Execução dos testes é realizada de forma idêntica*
  - *TestCases são construídos sobre uma subclasse de `junit.framework.TestCase`*

# Para que serve?

- Para testar aplicações que utilizam componentes J2EE
- Arquitetura MVC
  - Servlets, filtros e custom tags (**C**ontroladores)
  - JSPs (camada de apresentação: **V**iew, através de controladores)
  - EJB (**M**odelo de dados/lógica de negócios)
- Cactus testa a integração desses componentes com seus containers
  - **não usa stubs** - usa o próprio container como servidor e usa JUnit como cliente
  - comunicação é intermediada por um **proxy**

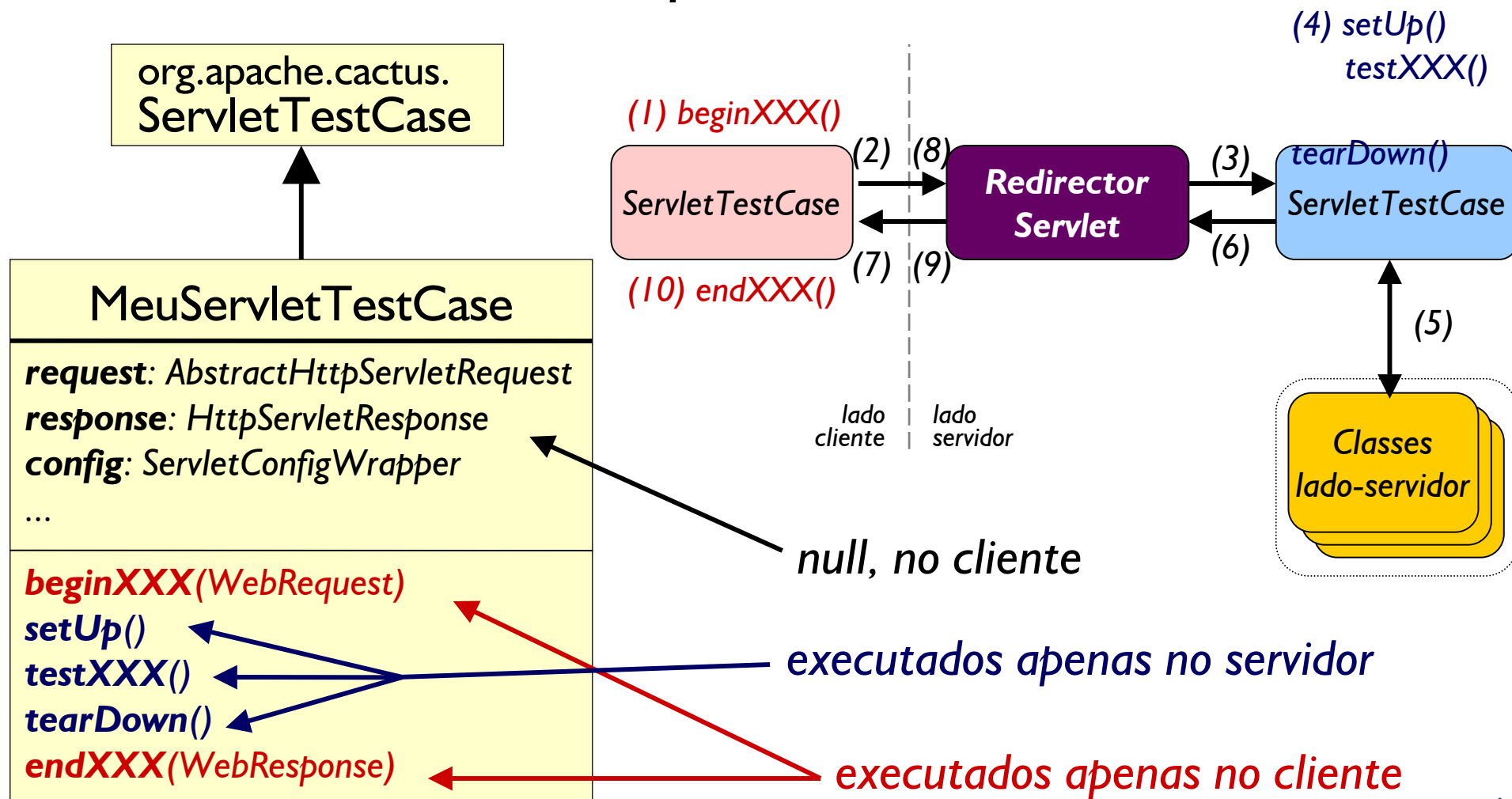


# Como funciona?

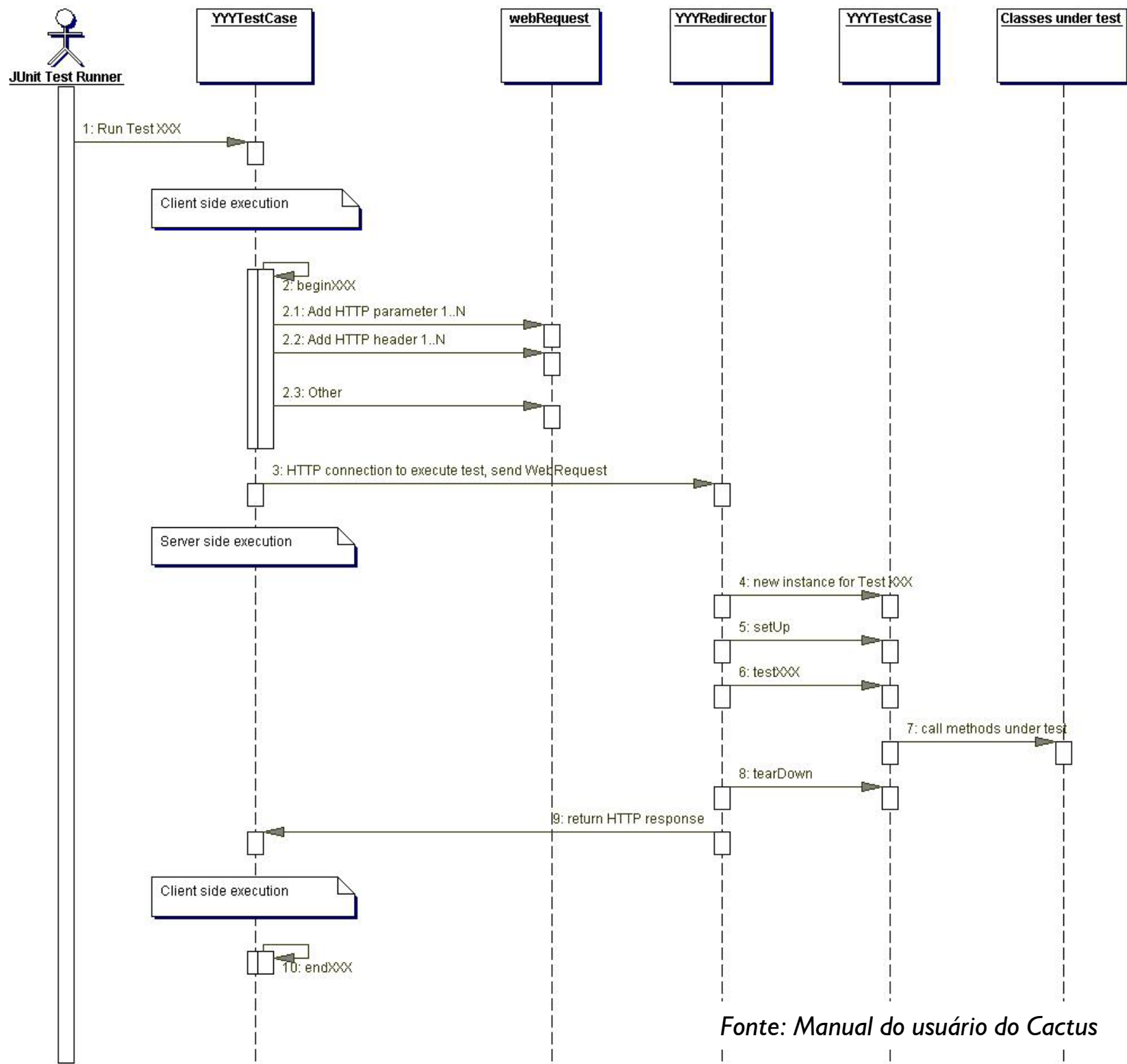
- *Cactus utiliza os test cases simultaneamente no cliente e no servidor: **duas cópias***
  - *Uma cópia é instanciada pelo servlet container*
  - *Outra cópia é instanciada pelo JUnit*
- *Comunicação com o servlet container é feita através de um proxy (XXXRedirector)*
  - *JUnit envia requisições via HTTP para proxy*
  - *Proxy devolve resultado via HTTP e JUnit os mostra*
- *Há, atualmente (Cactus 1.3) três tipos de proxies:*
  - ***ServletRedirector**: para testar servlets*
  - ***JSPRedirector**: para testar JSP custom tags*
  - ***FilterRedirector**: para testar filtros de servlets*



- Parte da mesma classe (*ServletTestCase*) é executada no cliente, parte no servidor



# Diagrama UML



Fonte: Manual do usuário do Cactus

# ServletTestCase (ou similar)

- Para cada método `XXX()` a ser testado, pode haver:
  - Um `beginXXX()`, para inicializar a requisição do cliente
    - encapsulada em um objeto `WebRequest` a ser enviado ao servidor
  - Um `testXXX()`, para testar o funcionamento do método no servidor (deve haver ao menos um)
  - Um `endXXX()`, para verificar a resposta do servidor
    - devolvida em um objeto `WebResponse` retornada pelo servidor
- Além desses três métodos, cada `TestCase` pode conter
  - `setUp()`, opcional, para inicializar objetos no servidor
  - `tearDown()`, opcional, para liberar recursos no servidor
- Os métodos do lado do servidor têm acesso aos mesmos objetos implícitos disponíveis em um servlet ou página JSP: `request`, `response`, etc.

# Cactus: exemplo

- Veja **cactusdemo.zip** (distribuído com esta palestra)
  - Usa duas classes: um servlet (**MapperServlet**) e uma classe (**SessionMapper**) que guarda cada parâmetro como atributo da sessão e em um *HashMap* - veja fontes em **src/xptoolkit/cactus**
- Para rodar, configure o seu ambiente:
  - **build.properties** - localização dos JARs usados pelo servidor Web (CLASSPATH do servidor)
  - **runtests.bat** (para Windows) e **runtests.sh** (para Unix) - localização dos JARs usados pelo JUnit (CLASSPATH do cliente)
  - **lib/client.properties** (se desejar rodar cliente e servidor em máquinas separadas, troque as ocorrências de localhost pelo nome do servidor)
- Para montar, execute:
  - 1. **ant test-deploy** instala cactus-tests.war no tomcat
  - 2. o servidor (Tomcat 4.0 startup)
  - 3. **runtests.bat** roda os testes no JUnit

veja demonstração

cactusdemo

# CactusDemo: servlet

- O objetivo deste servlet é
  - 1) gravar qualquer parâmetro que receber na sessão (objeto session)
  - 2) devolver uma página contendo os pares nome/valor em uma tabela
  - 3) imprimir resposta em caixa-alta se `<init-param> ALL_CAPS` definido no `web.xml` contiver o valor `true`

```
public void doGet(...) throws IOException {  
    SessionMapper.mapRequestToSession(request);  
    writer.println("<html><body><table border='1'>");  
    // (... loop for each parameter ...)  
    if (useAllCaps()) {  
        key = key.toUpperCase();  
        val = val.toUpperCase();  
    }  
    str = "<tr><td><b>" + key + "</b></td><td>" + val + "</td></tr>";  
    writer.println(str);  
    // (...)  
    writer.println("</table></body></html>");  
}
```

(1) Grava request em session

(3) Retorna true se `<init-param> "ALL_CAPS"` contiver "true"

(2) Trecho de MapperServlet.java

- Escreveremos os testes para avaliar esses objetivos

# CactusDemo: testes

MapperServletTest.java

```
public class MapperServletTest extends ServletTestCase { (...)
    private MapperServlet servlet;
    public void beginDoGet(WebRequest cSideReq) {
        cSideReq.addParameter("user", "Jabberwock");
    }
    public void setUp() throws ServletException {
        this.config.setInitParameter("ALL_CAPS", "true");
        servlet = new MapperServlet();
        servlet.init(this.config);
    }
    public void testDoGet() throws IOException {
        servlet.doGet(this.request, this.response);
        String value = (String) session.getAttribute("user");
        assertEquals("Jabberwock", value);
    }
    public void tearDown() { /* ... */ }
    public void endDoGet(WebResponse cSideResponse) {
        String str = cSideResponse.getText();
        assertTrue(str.indexOf("USER</b></td><td>JABBERWOCK") > -1);
    }
}
```

*Simula DD  
<init-param>*

*Simula servlet  
container*

*Verifica se parâmetro foi  
mapeado à sessão*

*Verifica se parâmetro aparece na tabela HTML*

# Exemplo: funcionamento

## Cliente (JUnit)

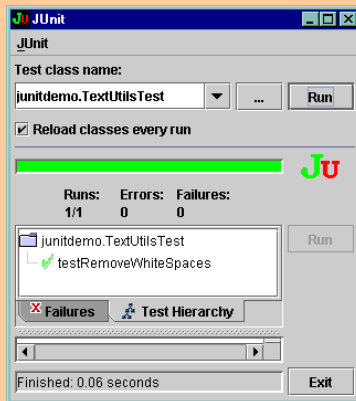
**beginDoGet** (WebRequest req)

- Grava parâmetro:

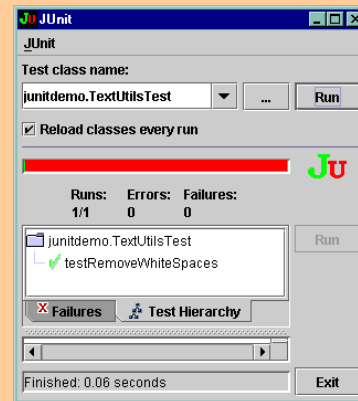
nome = **user**

value = **Jabberwock**

**SUCCESS!!**



**FAIL!**



**endDoGet** (WebResponse res)

- Verifica se resposta contém

**USER**</b></td><td>**JABBERWOCK**

## Servidor (Tomcat)

**ReqInfo**

2 conexões HTTP:

- Uma p/ rodar os testes e obter saída do servlet
- Outra para esperar resultados de testes (info sobre exceções)

**TestInfo**

**Output**

**setUp()**

- Define init-params no objeto config
- Roda init(config)

**testDoGet()**

- Roda doGet()
- Verifica se parâmetro (no response) foi mapeado à sessão

**tearDown()**

falha local

falha remota

&

- Onde encontrar
  - `http://httpunit.sourceforge.net`
- Framework para testes funcionais de interface (teste tipo "caixa-preta")
  - Verifica a resposta de uma aplicação Web ou página HTML
  - É teste funcional caixa-preta (não é "unit")
  - Oferece métodos para "navegar" na resposta
    - links, tabelas, imagens
    - objetos DOM (Node, Element, Attribute)
- Pode ser combinado com Cactus no `endXXX()`
  - Argumento `com.meterware.httpunit.WebResponse`
- Acompanha **ServletUnit**
  - stub que simula o servlet container



# Resumo da API do HttpUnit

- **WebConversation**

- *Representa uma sessão de cliente Web (usa cookies)*

- ```
WebConversation wc = new WebConversation();
```

- ```
WebResponse resp = wc.getResponse("http://xyz.com/t.html");
```

- **WebRequest**

- *Representa uma requisição*

- **WebResponse**

- *Representa uma resposta. A partir deste objeto pode-se obter objetos WebLink, WebTable e WebForm*

- **WebLink**

- *Possui métodos para extrair dados de links de hipertexto*

- **WebTable**

- *Possui métodos para navegar na estrutura de tabelas*

- **WebForm**

- *Possui métodos para analisar a estrutura de formulários*

# HttpUnit com Cactus

Veja MapperServletTest2.java

- Troque o **WebResponse** em cada **endXXX()** por **com.meterware.httpunit.WebResponse**

```
public void endDoGet(com.meterware.httpunit.WebResponse resp)
    throws org.xml.sax.SAXException {
    WebTable[] tables = resp.getTables();
    assertNotNull(tables);
    assertEquals(tables.length, 1); // só há uma tabela
    WebTable table = tables[0];
    int rows = table.getRowCount();
    boolean keyDefined = false;
    for (int i = 0; i < rows; i++) {
        String key    = table.getCellAsText(i, 0); // col 1
        String value = table.getCellAsText(i, 1); // col 2
        if (key.equals("USER")) {
            keyDefined = true;
            assertEquals("JABBERWOCK", value);
        }
    }
    if (!keyDefined) {
        fail("No key named USER was found!");
    }
}
```

# Outros testes com Cactus

- Testes em taglibs (JspRedirector)
  - Veja exemplos em `cactusdemo/taglib/src`
- Testes em filtros (FilterRedirector)
  - Usa proxy FilterRedirector
  - Teste básico é verificar se método `doFilter()` foi chamado
  - Veja exemplos em `cactusdemo/src/xptoolkit/AuthFilter`
- Testes indiretos em páginas JSP (camada **V**iew)
  - Ideal é JSP não ter código Java
  - Principais testes são sobre a interface: `HttpUnit!`
- Testes indiretos em EJB (camada **M**odel)
  - Indireto, através dos redirectors + `JUnitEE`
  - Redirectors permitem testar EJBs com interface local ou remota chamados por código no servidor

veja

hellojsp\_2

# Testes em aplicações Web: conclusões

- Aplicações Web são difíceis de testar porque dependem da comunicação com servlet containers
  - *Stubs, proxies e APIs, que estendem ou cooperam com o JUnit, tornam o trabalho mais fácil*
  - *Neste bloco, conhecemos três soluções que facilitam testes de unidade, de integração e de caixa-preta em aplicações Web*
- **Stubs** como **ServletUnit** permitem testar as **unidades** de código mesmo que um servidor não esteja presente
- **Proxies** como os "redirectors" do **Cactus** permitem testar a **integração** da aplicação com o container
- Uma **API**, como a fornecida pelo **HttpUnit** ajuda a testar o **funcionamento** da aplicação do ponto de vista do usuário

# Exemplos no CD

- Os exemplos abaixo usam Cactus
  - *cap06/helloservlet* (use *ant RUN-TESTS*)
  - *cap08/mvc/hellojsp\_2*
  - *cap08/mvc/hellojsp\_3*
  - *cap08/test/cactusdemo* (use *runtests.bat* ou *.sh*)
  - *cap08/test/strutsdemo*
- Para os dois últimos, é preciso configurar seu ambiente. Os dois primeiros já incluem todo o ambiente necessário.

- 1. Escreva um custom tag que receba o nome de uma cor HTML (red, blue, etc.) e um número como atributos, e um texto como corpo. Sua execução deve imprimir o texto recebido na cor selecionada e no tamanho em pontos recebido.
  - Use CSS: `<span style="color: red; font-size: 10pt">texto</span>`
- 2. Escreva um DAO (Data Access Object) que implemente a interface MessageBeanDAO do exemplo hellojsp\_2 para acesso a um banco de dados.
  - Crie a tabela para armazenar a mensagem. Colunas "data", "host" e "mensagem" (se possível, faça isto usando o build.xml do Ant)
  - Implemente o DBMessageBeanDAO usando o Cloudscape datasource (coloque-o no lugar das chamadas ao JDOMessageBeanDAO)
  - Defina os parâmetros necessários (dsn) no web.xml
- 3. Escreva testes JUnit (não Cactus) para a MessageBeanDAO.

- [1] Richard Hightower e Nicholas Lesiecki. *Java Tools for eXtreme Programming*. Wiley, 2002. Explora as ferramentas Ant, JUnit, Cactus, JUnitPerf, JMeter, HttpUnit usando estudo de caso com processo XP.
- [2] <http://www.computer-programmer.org/articles/struts/>
- [3] *Manual do Struts* Copie o arquivo struts-documentation.war (se quiser, altere o nome do WAR antes) para o diretório webapps/ to Tomcat 4.0 ou deploy do JBoss.
- [4] *Apache Cactus User's Manual*. Contém tutorial para instalação passo-a-passo.
- [5] Jim Farley, *Java Enterprise in a Nutshell*. O'Reilly, 2002. Contém tutorial curso e objetivo sobre JSP e taglibs.
- [6] Fields/Kolb. *Web Development with JavaServer Pages*. Manning, 2000. Contém dois capítulos dedicados a taglibs básicos e avançados.
- [7] *Sun J2EE Blueprints*.

*helder@ibpinet.net*

***www.argonavis.com.br***