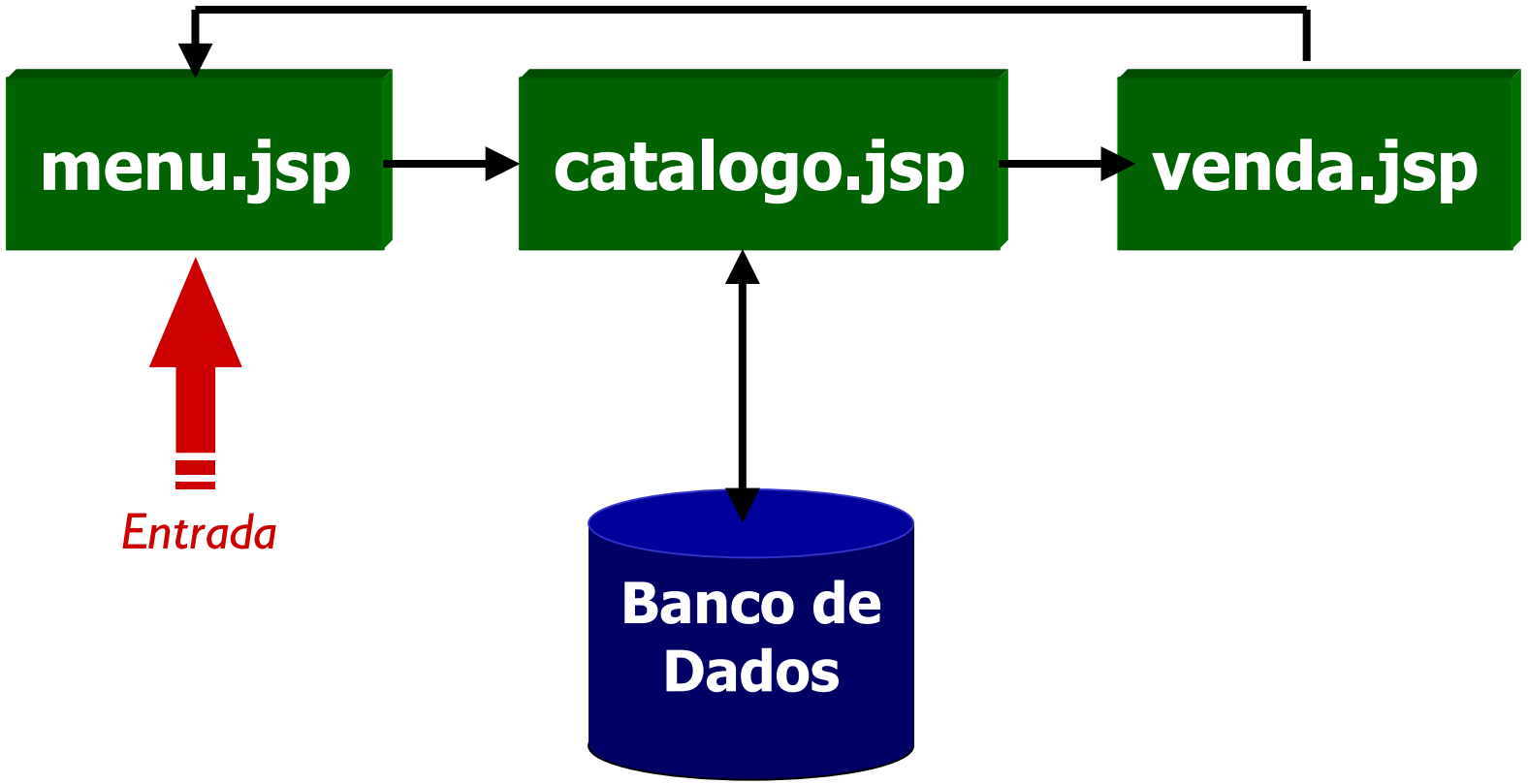
12 Model View Controller

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*

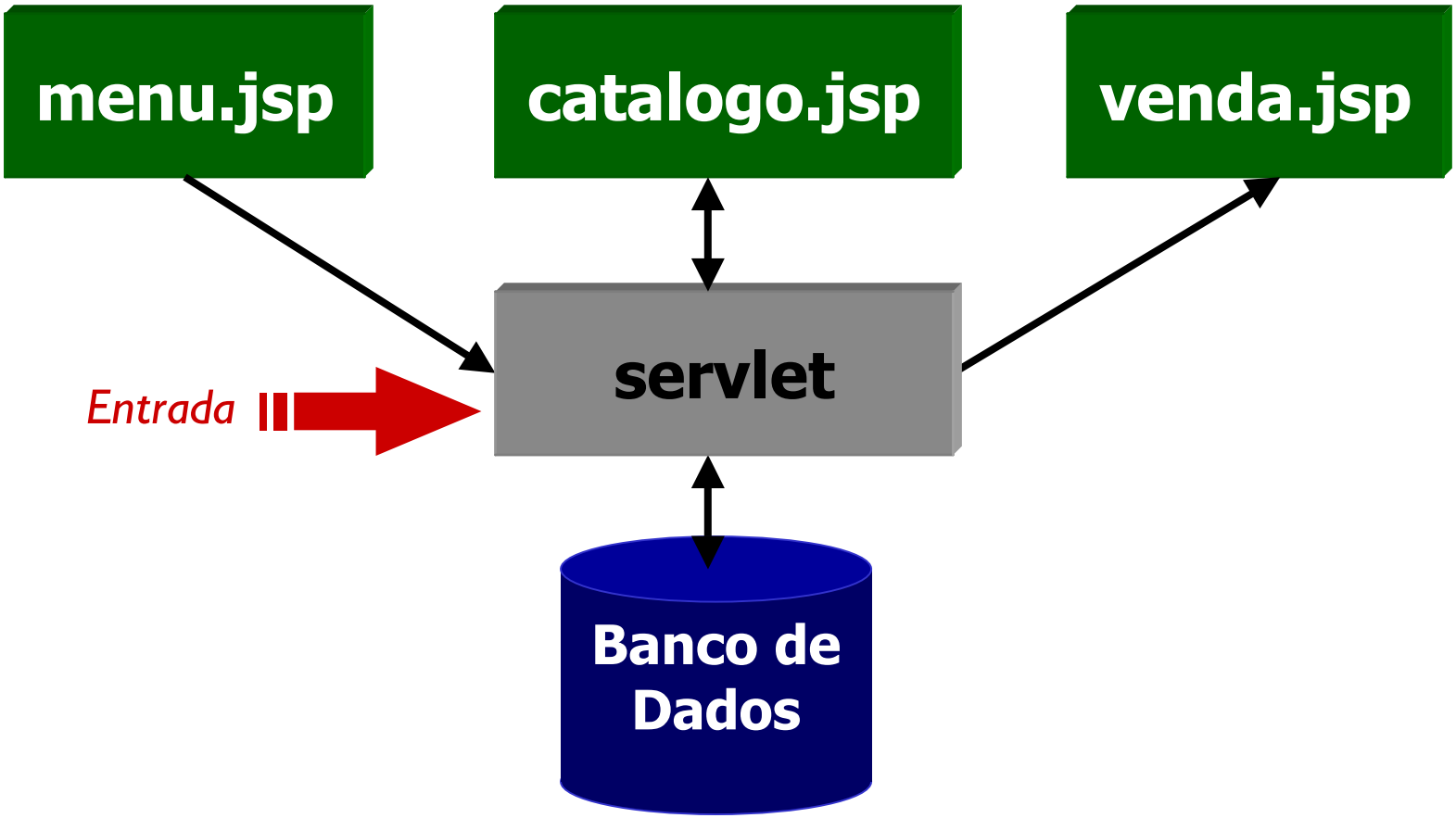
** FrontController é um J2EE design pattern. Vários outros design patterns serão identificados durante esta seção. Para mais informações, veja Sun Blueprints [7]*

Layout centrado em páginas (JSP Model 1)



Fonte: [6]

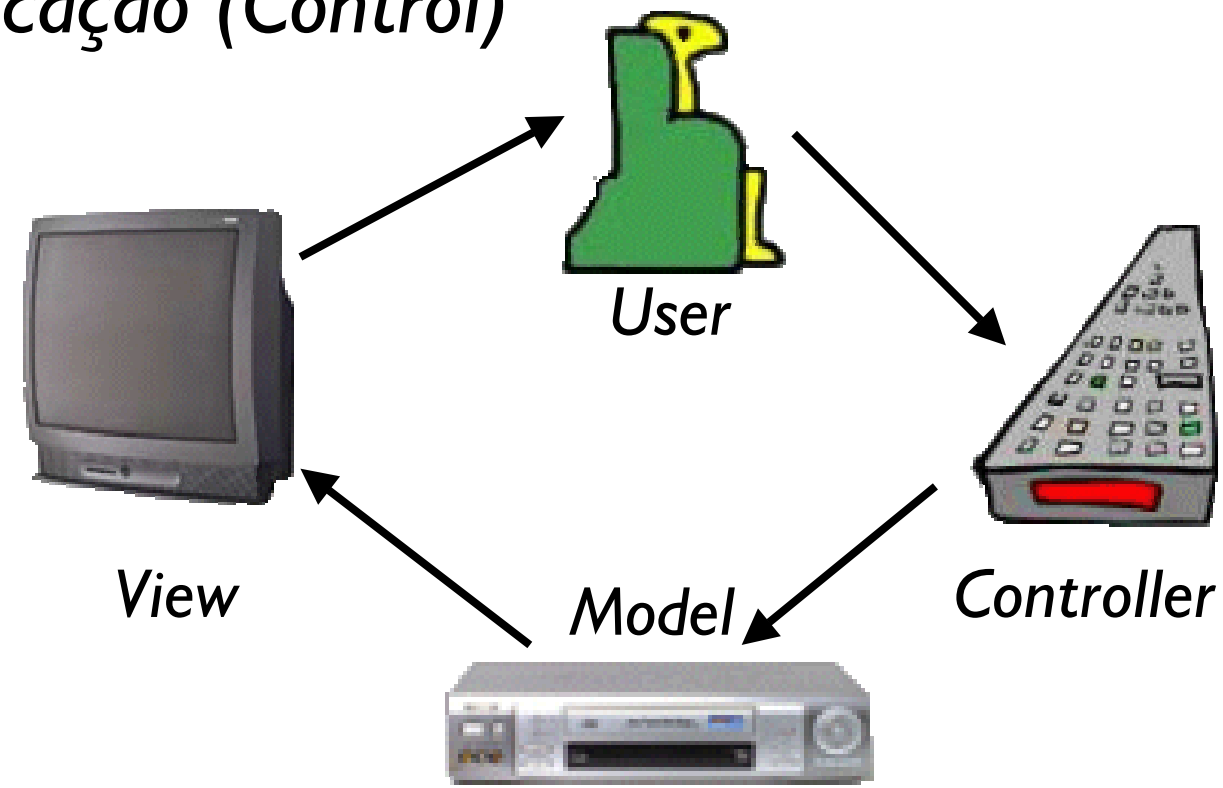
Layout centrado em servlet (JSP Model 2)



Fonte: [6]

O que é MVC

- *Padrão de arquitetura: Model View Controller*
- *Técnica para separar dados ou lógica de negócios (Model) da interface do usuário (View) e do fluxo da aplicação (Control)*



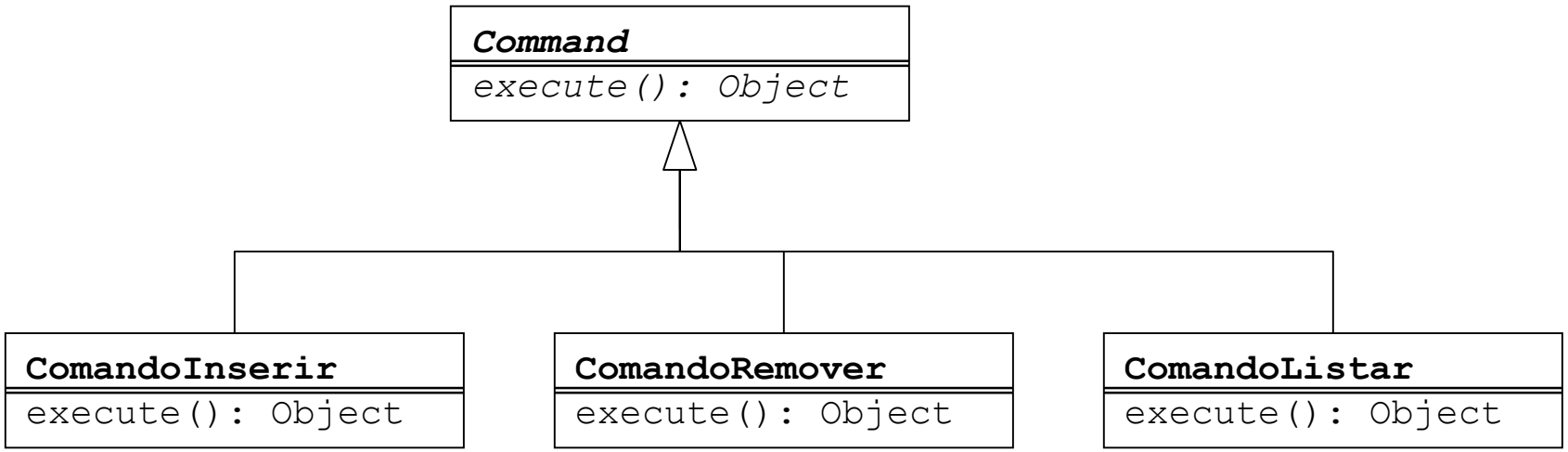
Fonte: *http://www.computer-programmer.org/articles/struts/*

Como implementar?

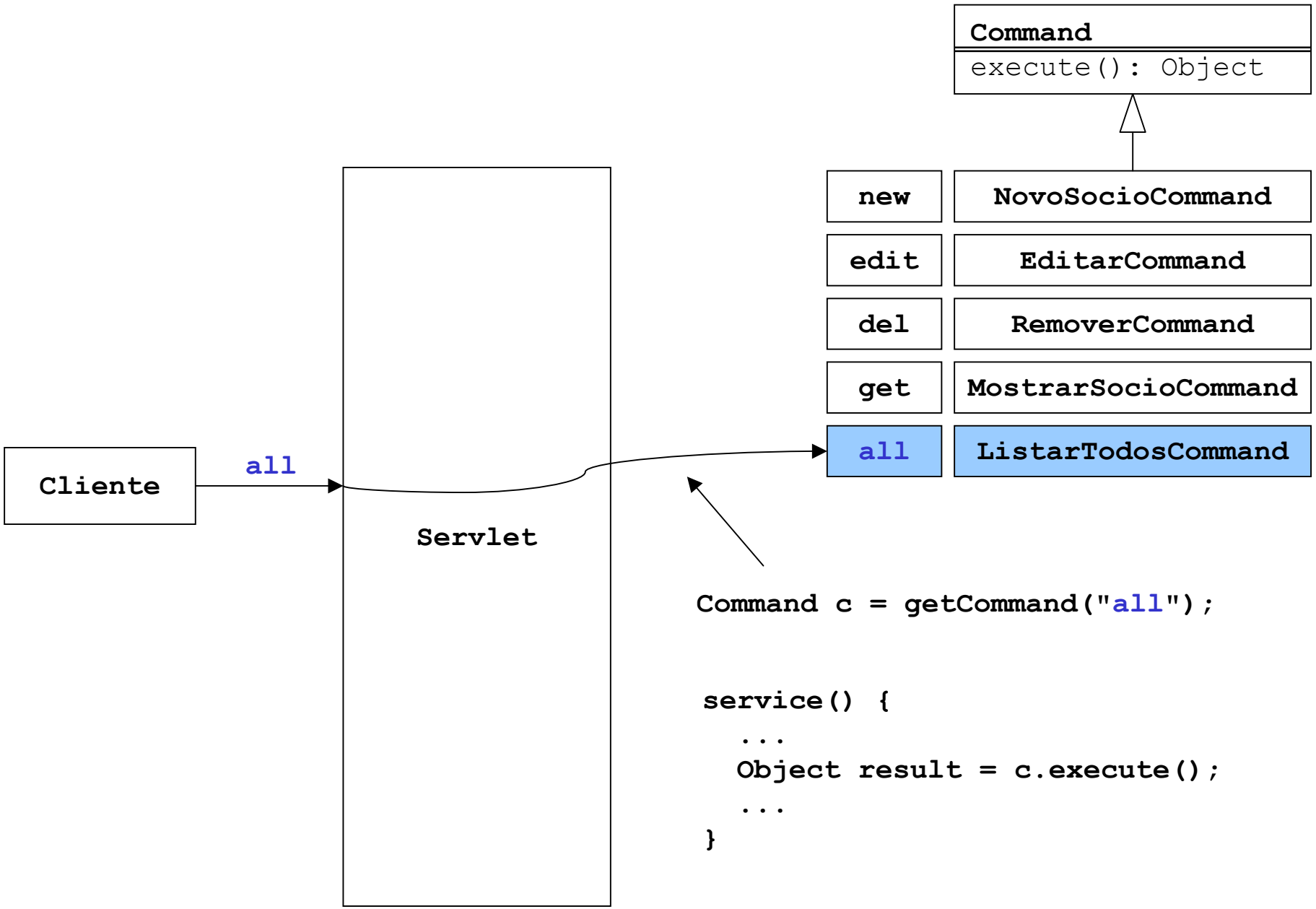
- Há várias estratégias
- Todas procuram isolar
  - As operações de controle de requisições em servlets e classes ajudantes,
  - Operações de geração de páginas em JSP e JavaBeans, e
  - Lógica das aplicações em classes que não usam os pacotes javax.servlet
- Uma estratégia consiste em se ter um único controlador (FrontController pattern) que delega requisições a diferentes objetos que implementam comandos que o sistema executa (Command pattern)

Command Pattern

- *É um padrão de projeto clássico catalogado no livro "Design Patterns" de Gamma et al (GoF = Gang of Four)*
  - *Para que serve: "Encapsular uma requisição como um objeto, permitindo que clientes parametrizem diferentes requisições, filas ou requisições de log, e suportar operações reversíveis." [GoF]*
- *Consiste em usar polimorfismo para construir objetos que encapsulam um comando e oferecer um único método execute() com a implementação do comando a ser executado*



Command Pattern



Command em Java

```
public interface Command {
   public Object execute(Object arg);
}
```

```
public class Server {
  private Database db = ...;
  private HashMap cmds = new HashMap();

  public Server() {
    initCommands();
  }

  private void initCommands() {
    cmds.put("new", new NewCommand(db));
    cmds.put("del",
                 new DeleteCommand(db));
    ...
  }

  public void service(String cmd,
                      Object data) {
    ...
    Command c = (Command)cmds.get(cmd);
    ...
    Object result = c.execute(data);
    ...
  }
}
```

```
public interface NewCommand implements Command {

  public NewCommand(Database db) {
    this.db = db;
  }

  public Object execute(Object arg) {
    Data d = (Data)arg;
    int id = d.getArg(0);
    String nome = d.getArg(1);
    db.insert(new Member(id, nome));
  }
}
```

```
public class DeleteCommand implements Command {

  public DeleteCommand(Database db) {
    this.db = db;
  }

  public Object execute(Object arg) {
    Data d = (Data)arg;
    int id = d.getArg(0);
    db.delete(id);
  }
}
```

FrontController com Command Pattern

- *Os comandos são instanciados e guardados em uma base de dados na memória (HashMap, por exemplo)*
  - *Pode-se criar uma classe específica para ser fábrica de comandos*
- *O cliente que usa o comando (o servlet), recebe na requisição o nome do comando, consulta-o no HashMap, obtém a instância do objeto e chama seu método execute()*
  - *O cliente desconhece a classe concreta do comando. Sabe apenas a sua interface (que usa para fazer o cast ao obtê-lo do HashMap*
- *No HashMap*

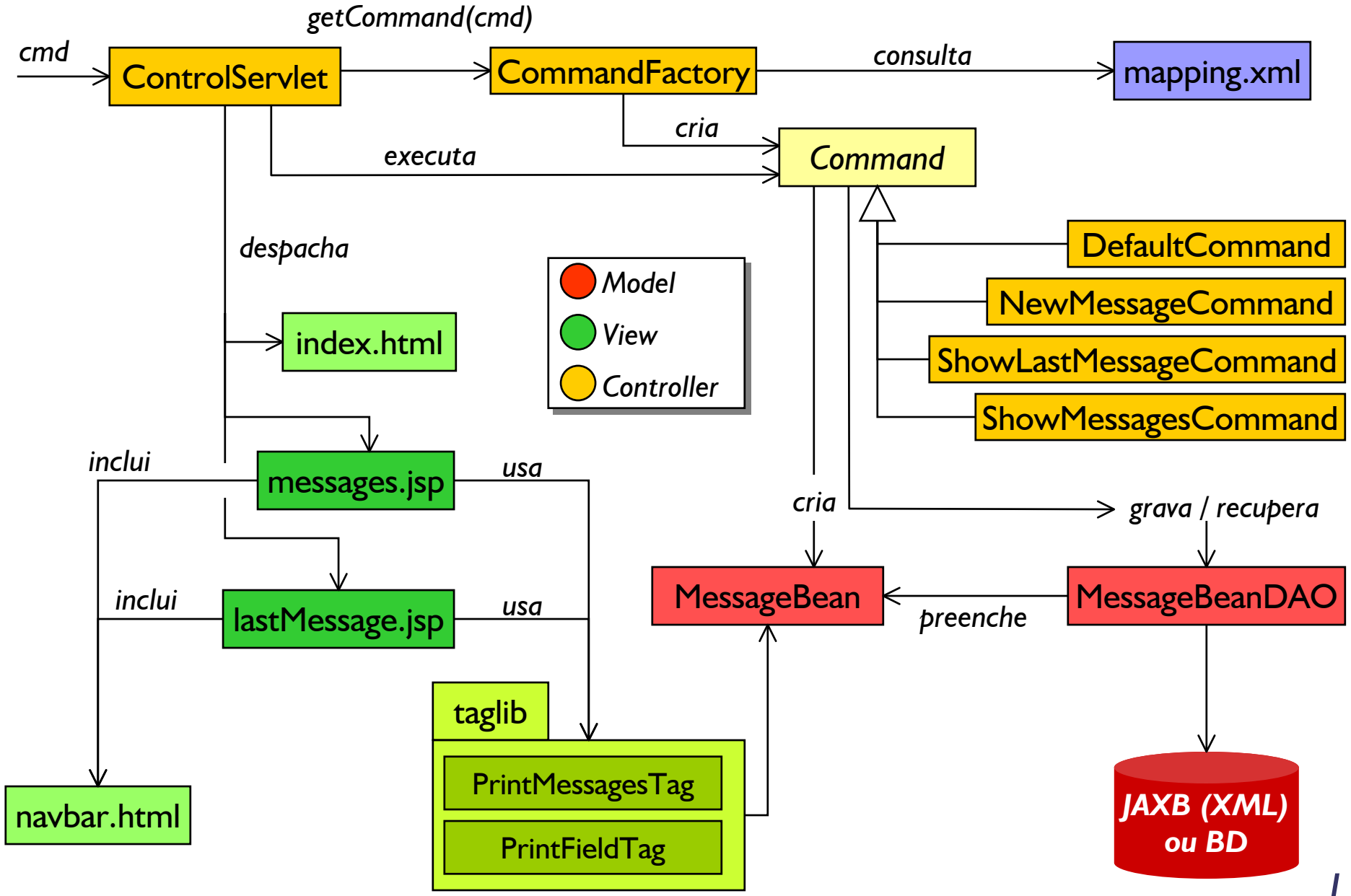
```
Comando c = new ComandoInserir();
comandosMap.put("inserir", c);
```

- *No servlet:*

```
String cmd = request.getParameter("cmd");
Comando c = (Comando)comandosMap.get(cmd);
c.execute();
```

Exemplo de implementação

cap12/mvc/hellojsp_2



Mapeamentos de comandos ou ações

- *No exemplo hellojsp_2, o mapeamento está armazendo 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/cap12/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)

Exercício

- *1. Coloque para funcionar o exemplo e analise suas classes*
- *2. Implemente a aplicação de mensagens que você criou nos capítulos anteriores em MVC*
  - *Crie um servlet controlador para interceptar todas as requisições*
  - *Crie páginas JSP que leiam os beans e que enviem comandos desejados através de um parâmetro cmd*
  - *Crie uma interface Comando e os comandos ComandoListar e ComandoAdicionar. Coloque-os em um HashMap inicializada no init() do servlet*
  - *Implemente o service que obtenha o parâmetro cmd, localize o comando desejado e o execute.*

helder@acm.org

argonavis.com.br