



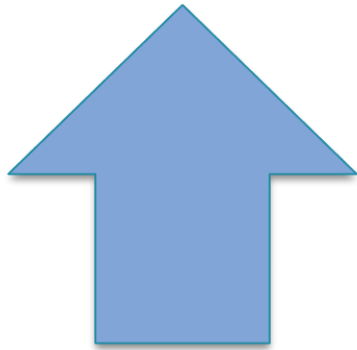
Desenvolvimento de Aplicações WEB

Padrão MVC

Profa. Joyce Miranda

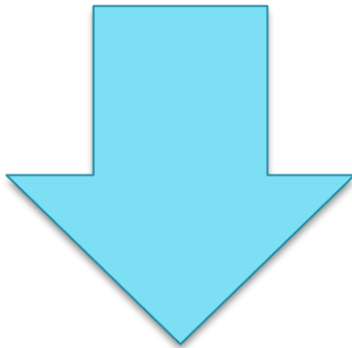
Materiais de Referência: <http://www.caelum.com.br/apostila-java-web>

Padrão MVC



Servlet

- Lógica de Negócio



JSP

- Lógica de Apresentação
sem acesso a BD e instanciação de objetos

- ▶ Arquitetura em camadas
 - ▶ Divisão de responsabilidades

Padrão MVC

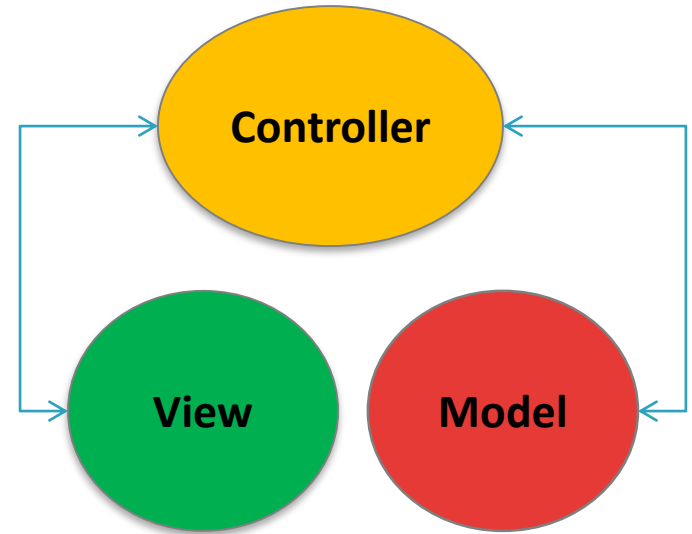
▶ MVC

▶ Fundamentos

- ▶ Padrão Arquitetural de Software
 - Não é um padrão de projeto
- ▶ Dividir a aplicação em camadas com responsabilidades específicas

▶ Vantagens

- ▶ Legibilidade
- ▶ Facilidade de manutenção
- ▶ Independência maior entre as camadas



Padrão MVC

► MVC

► Primeiramente: **Os Fundamentos**

Responsabilidades das Camadas

Cenário sem Fluxo de Dados

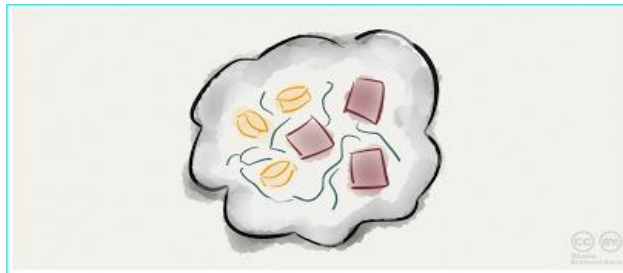
Cenário com Fluxo de Dados

Padrão MVC

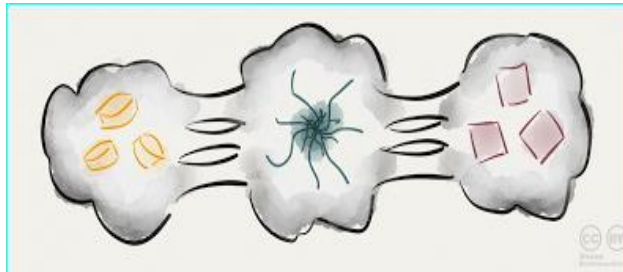
► MVC

► Fundamentos

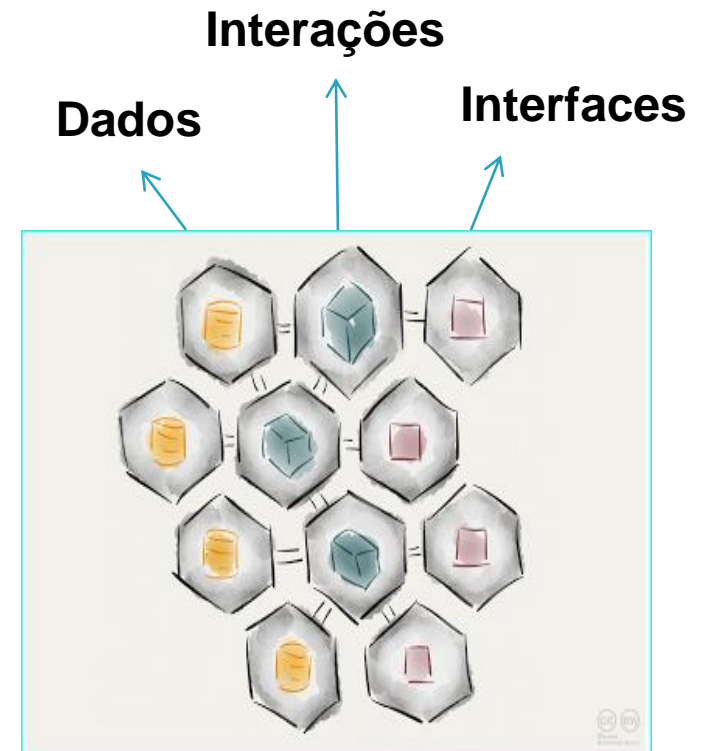
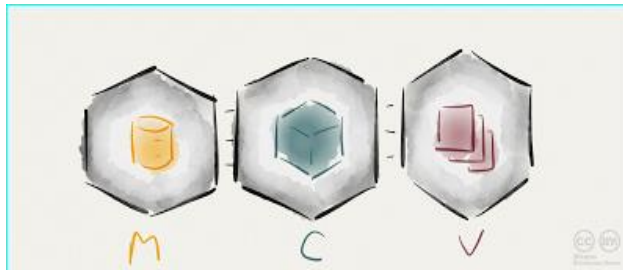
1º



2º



3º



N camadas M, V, C

* existindo regras de interação entre elas

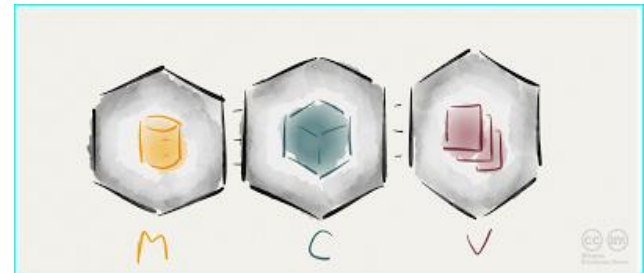
Padrão MVC

► MVC

► Fundamentos

► Regras

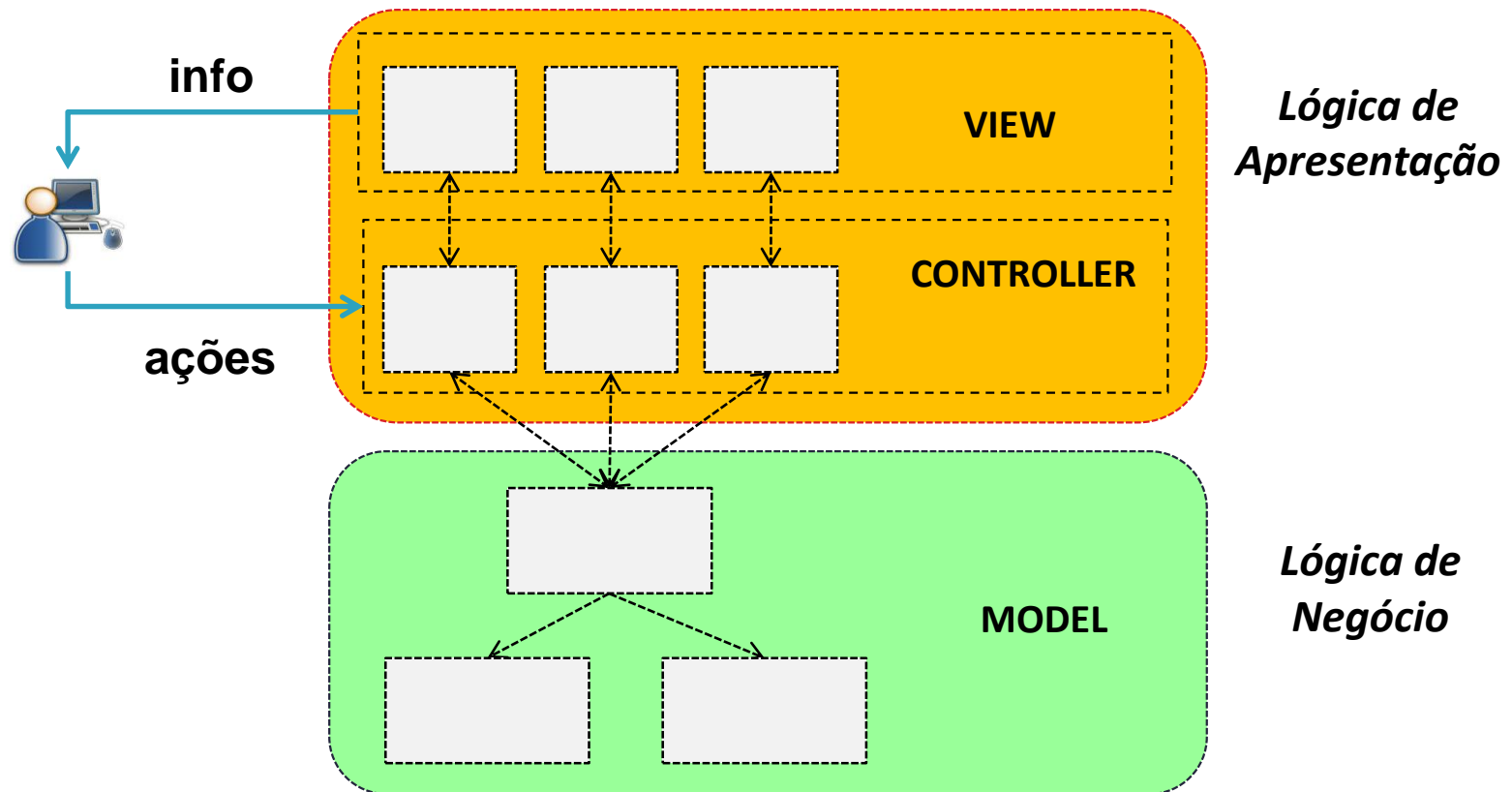
- A comunicação entre camadas deve ser sempre intermediada pela camada *Controller*
- Controladores não devem se comunicar entre si.



Padrão MVC

► MVC

► Fundamentos – Separação de Responsabilidades



Padrão MVC

► MVC

► Fundamentos

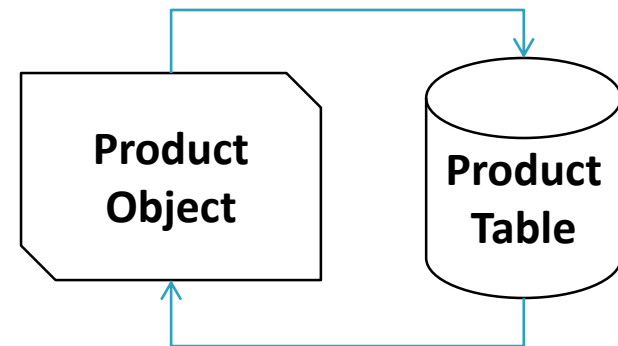
► **MODEL**

- Composta por classes que representam o domínio da aplicação

Regras de Negócio

Representação dos Dados

Camada de Acesso a Dados



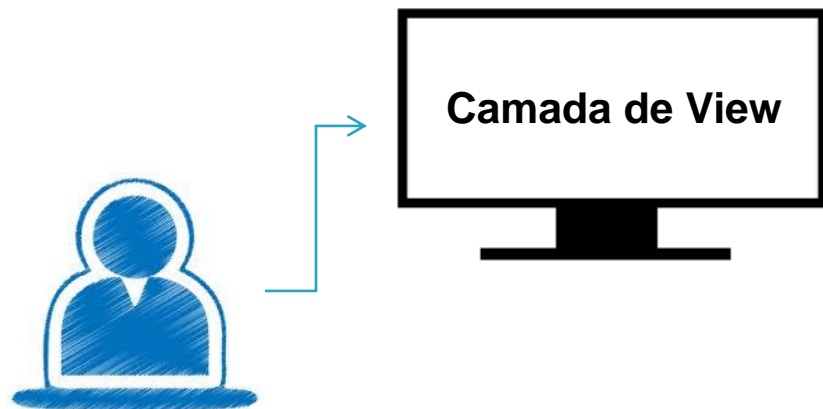
Padrão MVC

► MVC

► Fundamentos

► **VIEW**

- Representa a camada de interface com o usuário
- Invoca métodos do *Model* por meio do *Controller*
- Apresenta o resultado dado como resposta a uma requisição
- Monitora mudanças do *Model* e o apresenta atualizado



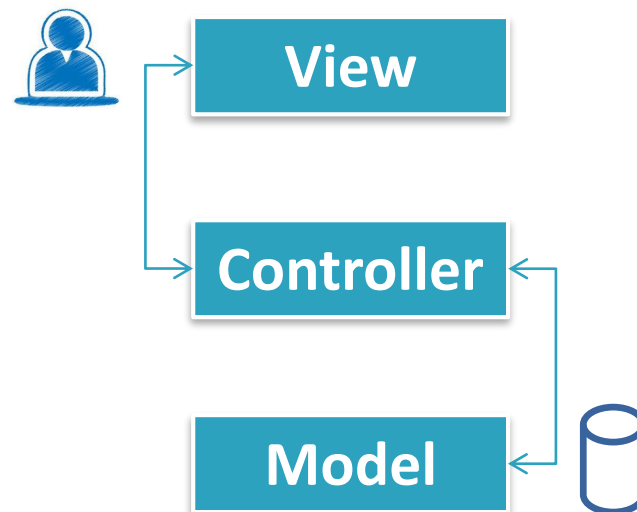
Padrão MVC

► MVC

► Fundamentos

► **CONTROLLER**

- ❑ Processa ações do usuário (invocadas pela View)
- ❑ Apresenta novas 'Views'
- ❑ Atualiza modelo

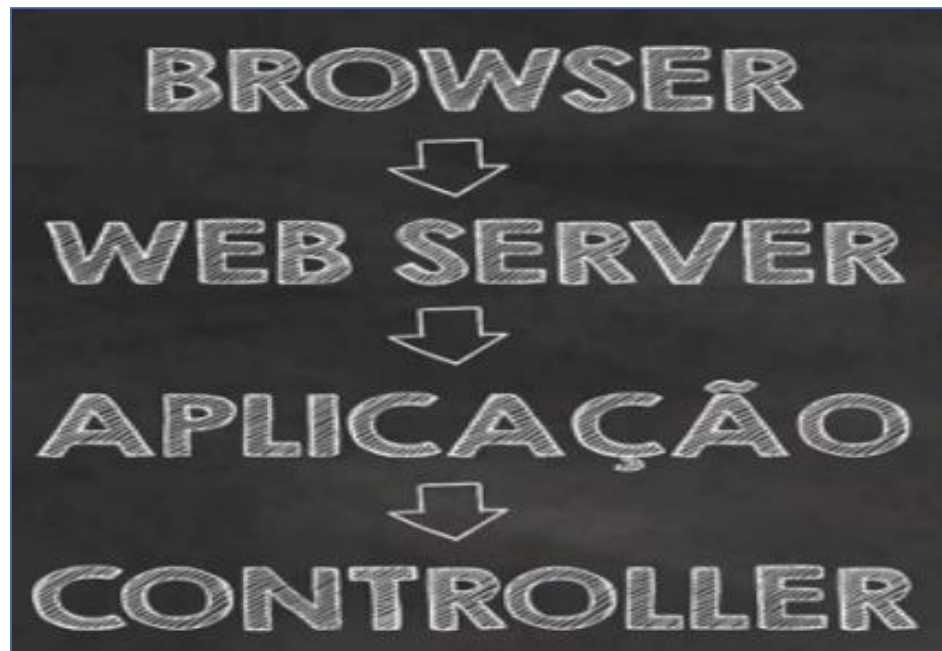


Padrão MVC

► MVC

► Fundamentos

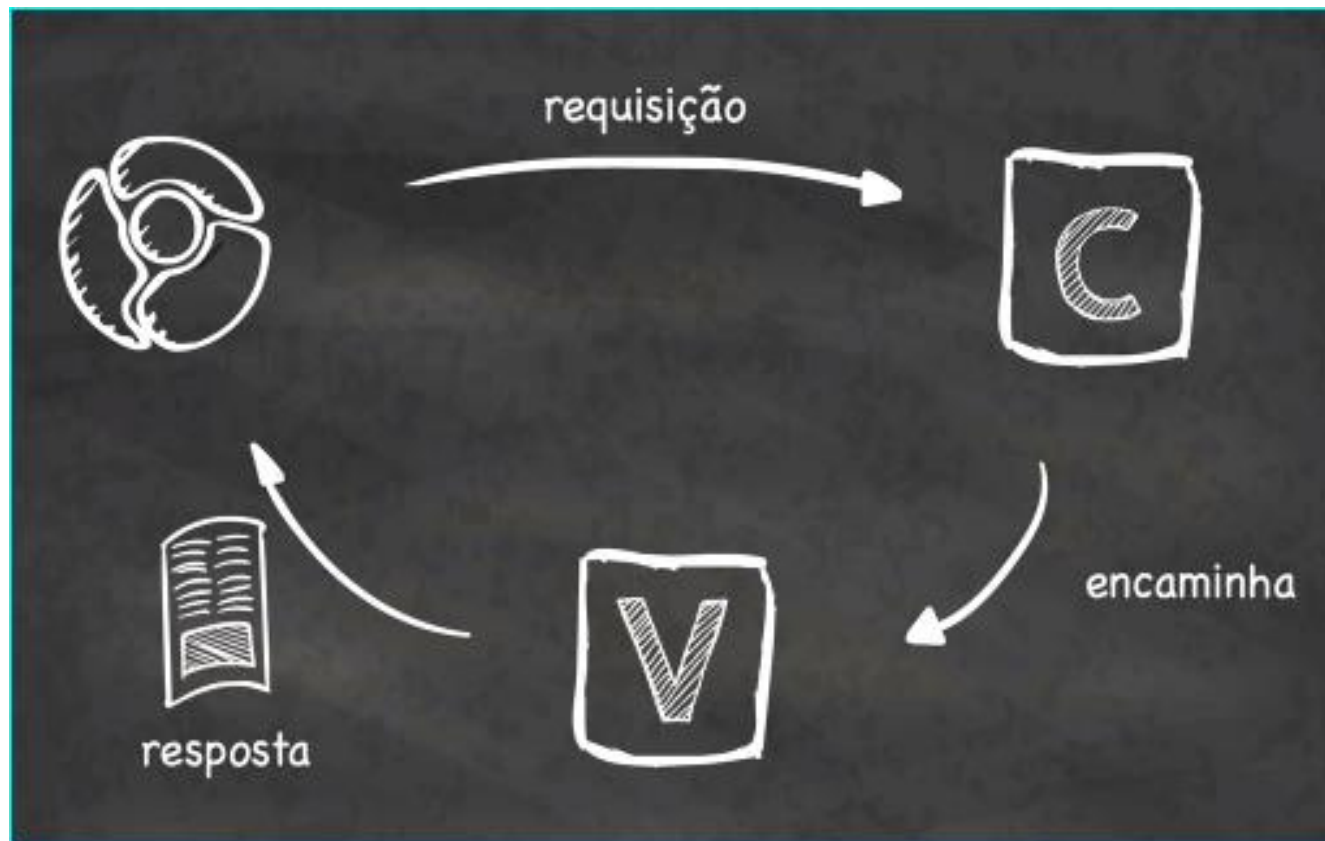
- A primeira camada que vai receber a requisição dentro do modelo MVC é a camada Controller, não é a View.



Padrão MVC

► MVC

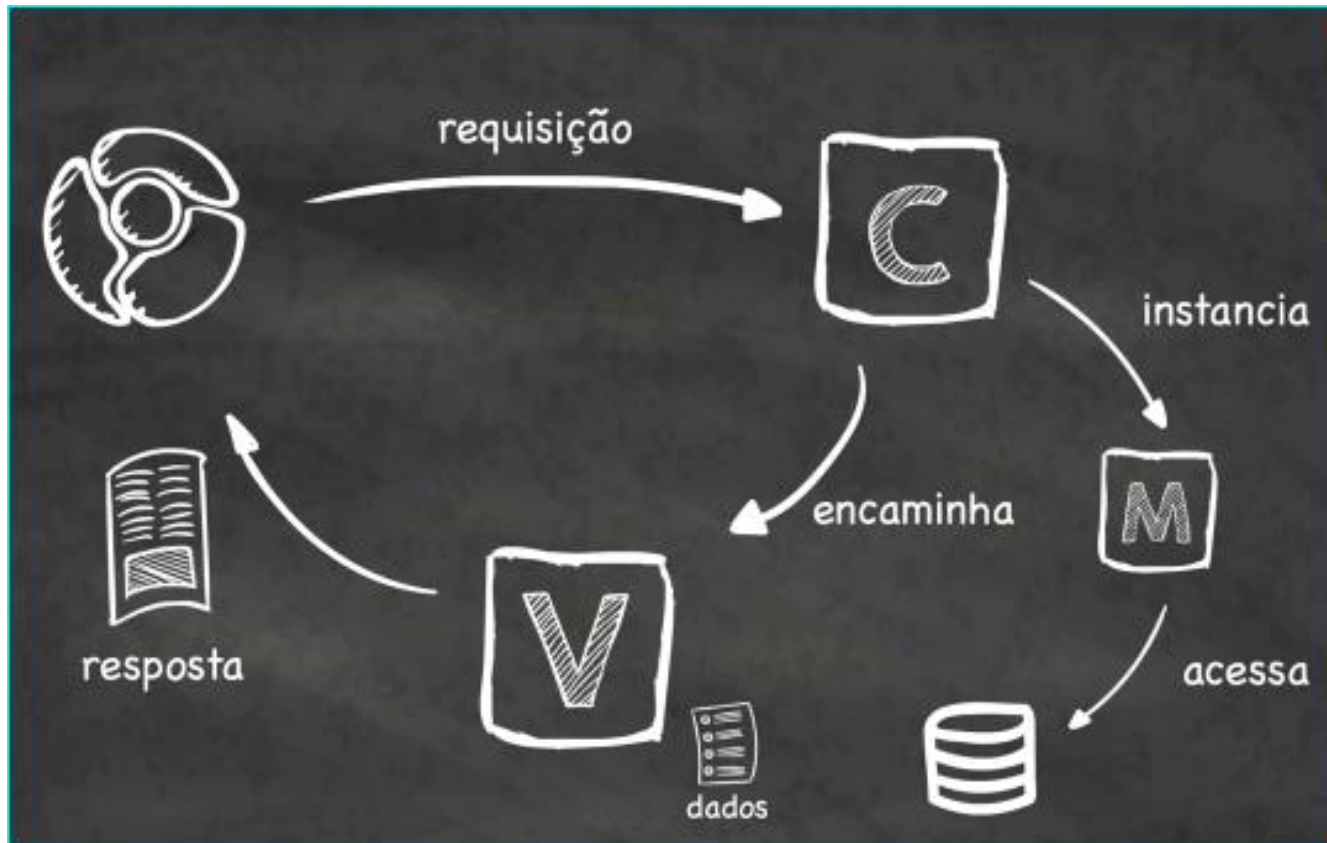
► Cenário **SEM** Fluxo de Dados



Padrão MVC

► MVC

► Cenário **COM** Fluxo de Dados



Padrão MVC

MVC na Prática

Código disponível em:

https://github.com/joyceMiranda/classCodes/tree/master/WEB_BASIC_JEE

Padrão MVC

► MVC na prática

Nome:

Email:

Telefone:

```
<html>
  <body>
    <form action="adicionaContato" method="post">
      Nome: <input type="text" name="nome">
      <br><br>
      Email: <input type="text" name="email">
      <br><br>
      Telefone: <input type="text" name="telefone">
      <br><br>
      <input type="submit" value="Adicionar">
    </form>
  </body>
</html>
```

```

@WebServlet("/adicionaContato")

public class AdicionaContatoServlet extends HttpServlet {

    protected void service(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        /***** log *****/
        System.out.println("Criando um novo contato");
        /***** acessando bean *****/
        Contato contato = new Contato(0,
            request.getParameter("nome"),
            request.getParameter("email"),
            request.getParameter("telefone"));
        /***** adicionando ao BD *****/
        ContatoDAO dao = new ContatoDAO();
        dao.addContato(contato);
        /***** ok *****/

        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");
        out.println("Contato: " + contato.getNome() + " adicionado com sucesso!");
        out.println("</body>");
        out.println("</html>");

    }
}

```

Exemplo do que não fazer!

Lógica de
apresentação

Padrão MVC

► MVC na prática

Boa prática!

► contato-adicionado.jsp

```
<html>
  <body>
    Contato {param.nome} adicionado com sucesso!
  </body>
</html>
```

► Request Dispatcher

- Redireciona a requisição do usuário para um outro recurso do navegador

```
RequestDispatcher rd =
    request.getRequestDispatcher("/contato-adicionado.jsp");
rd.forward(request, response);
```

```
@WebServlet("/adicionaContato")
```

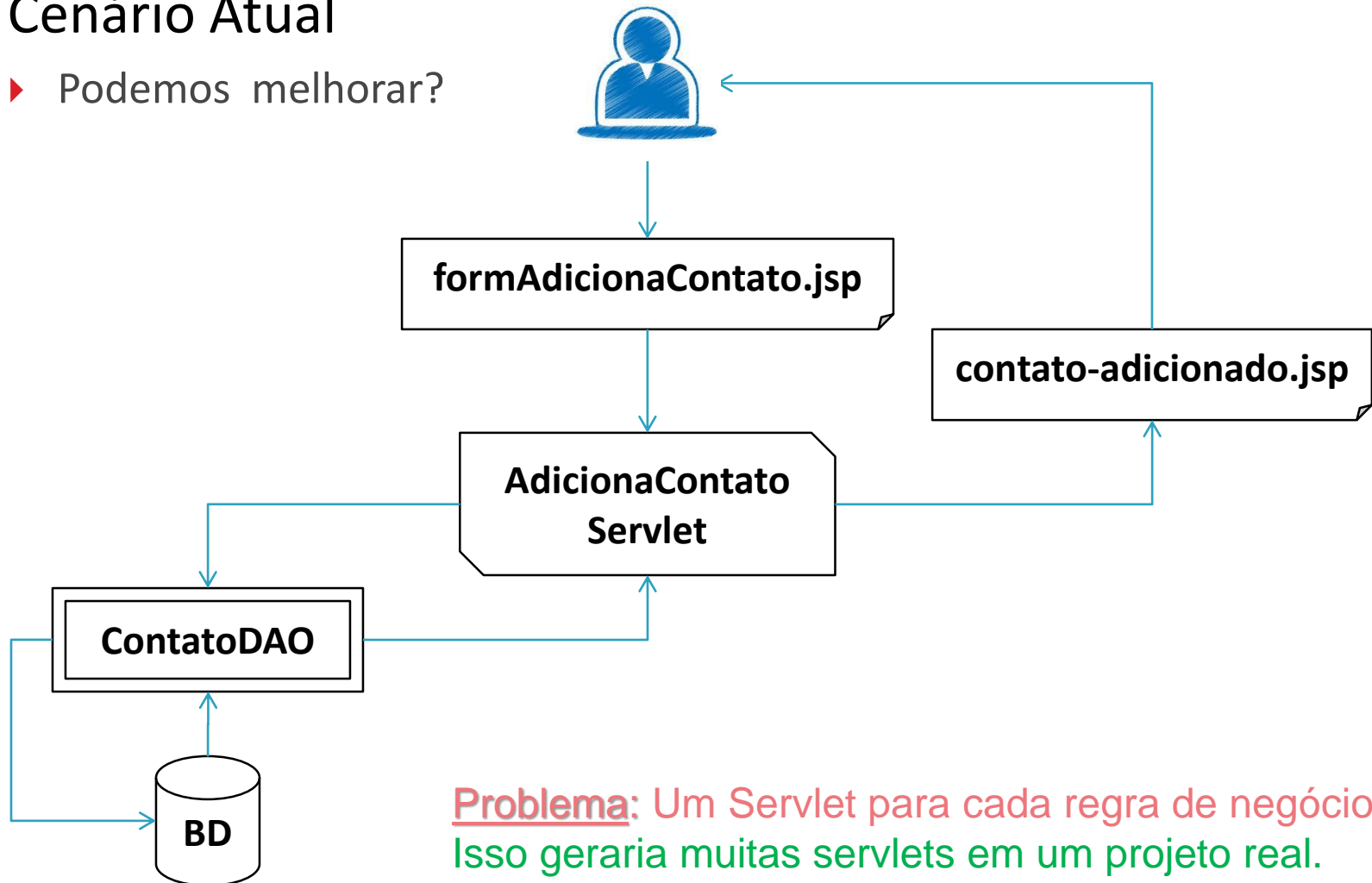
```
public class AdicionaContatoServlet extends HttpServlet {  
    protected void service(HttpServletRequest request, HttpServletResponse response)  
        throws ServletException, IOException {  
  
        /***** log *****/  
        System.out.println("Criando um novo contato");  
        /*****acessando bean *****/  
        Contato contato = new Contato(0,  
            request.getParameter("nome"),  
            request.getParameter("email"),  
            request.getParameter("telefone"));  
        /*****adicionando ao BD *****/  
        ContatoDAO dao = new ContatoDAO();  
        dao.addContato(contato);  
        /***** ok *****/  
        RequestDispatcher rd =  
            request.getRequestDispatcher("/contato-adicionado.jsp");  
        rd.forward(request, response);  
    }  
}
```

Melhorando o código!!

Padrão MVC

► Cenário Atual

- Podemos melhorar?

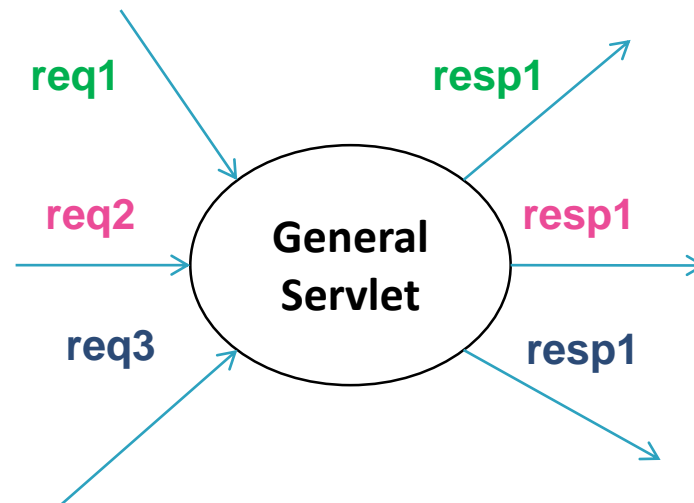


Problema: Um Servlet para cada regra de negócio
Isso geraria muitas servlets em um projeto real.

Padrão MVC

► Como podemos melhorar?

- Criar um único Servlet que decidirá o que fazer de acordo com os parâmetros de requisição do Cliente.



<http://myWebSitecom/sistema?logica=AdicionaContato>

<http://myWebSitecom/sistema?logica=ListaContatos>

<http://myWebSitecom/sistema?logica=RemoveContato>

```

@WebServlet("/sistema")
public class GeneralServlet extends HttpServlet{
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        String acao = req.getParameter("logica");
        ContatoDAO dao = new ContatoDAO();
        if(acao.equals("AdicionaContato")){
            Contato contato = new Contato(0,
                req.getParameter("nome"),
                req.getParameter("email"),
                req.getParameter("telefone"));
            dao.addContato(contato);
            RequestDispatcher rd =
                req.getRequestDispatcher("contato-adicionado.jsp");
            rd.forward(req, resp);
        }else if(acao.equals("ListaContatos")){
            //recupera lista do DAO
            //despacha para JSP
        }else if(acao.equals("RemoveContato")){
            //faz a remoção e redireciona para a lista
        }
    }
}

```

(*) Servlet muito grande, com todas as regras de negócio do sistema inteiro.

Padrão MVC

► Solução Elegante

- Padrões de Projeto aplicáveis ao desenvolvimento web

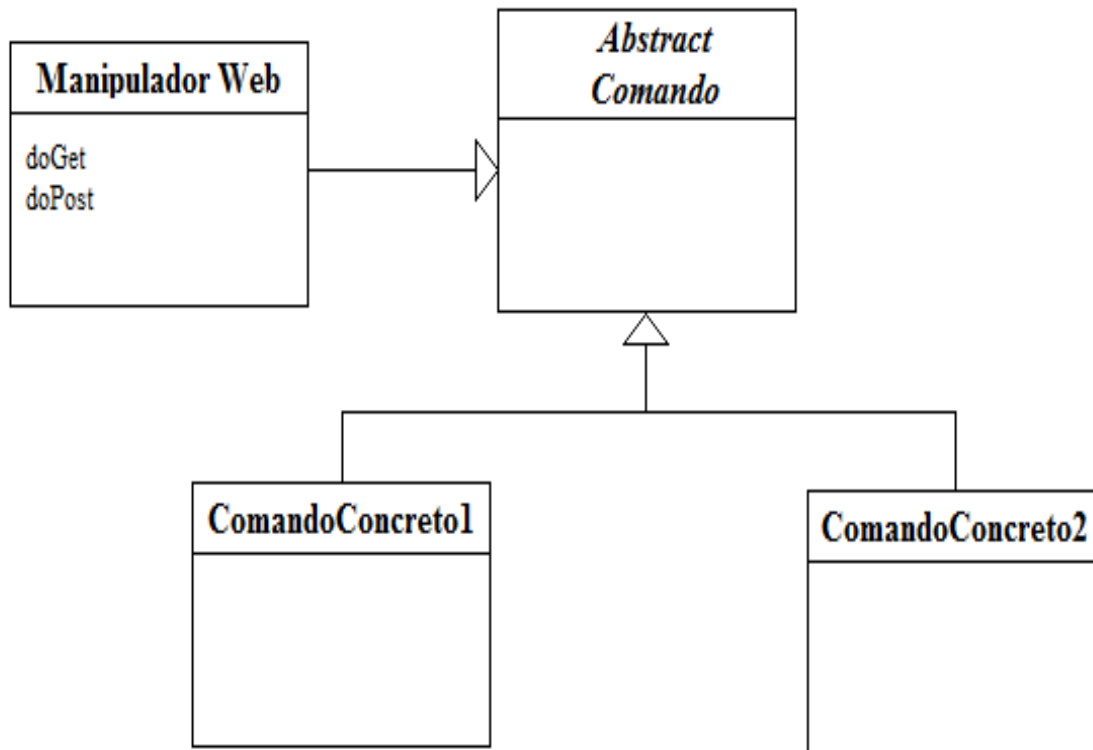


- Utilizados nos *frameworks* Struts, Spring e no JSF

Padrão MVC

► Pattern: Front Controller

- Controlador que recebe todas as requisições do site e as direciona para uma ação.



Versões de Implementação:

- Estática
 - Usa lógica condicional
- Dinâmica
 - Instanciação dinâmica para criar a classe Comando

Padrão MVC

► Pattern: Front Controller

► Versão Estática

- Colocar cada regra de negócio em uma classe separada

```
if (acao.equals("AdicionaContato")) {  
    new AdicionaContato().executa(request,response);  
} else if (acao.equals("ListaContato")) {  
    new ListaContatos().executa(request,response);  
}
```

► Vantagens

- Verificação de erros no despacho em tempo de compilação
- Flexibilidade na aparência das URLs

► Desvantagem

- Alteração da Servlet a cada inserção/remoção/alteração de lógica
- ***Observe o padrão nas diferentes requisições

Padrão MVC

► Pattern: Front Controller

► Versão Dinâmica

- Instanciação dinâmica para criar uma Comando.
 - Recupera parte da URL para criar uma instanciação dinâmica

```
String nomeDaClasse = request.getParameter("logica");  
new nomeDaClasse().executa(request,response);
```

* Chamada a um método sem
conhecer seu receptor e seu comportamento

► Vantagem

- Novos comandos podem ser adicionados sem a alteração do Manipulador

► Desvantagem

- Exige conhecimento específico relacionado à POO

Padrão MVC

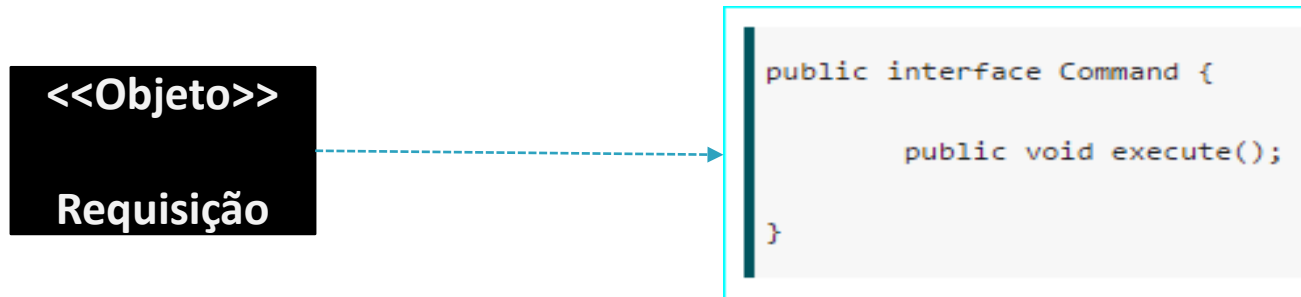
► *Pattern: Command*

► Motivação

- Necessidade de fazer chamada a métodos sem conhecer seu receptor e seu comportamento.

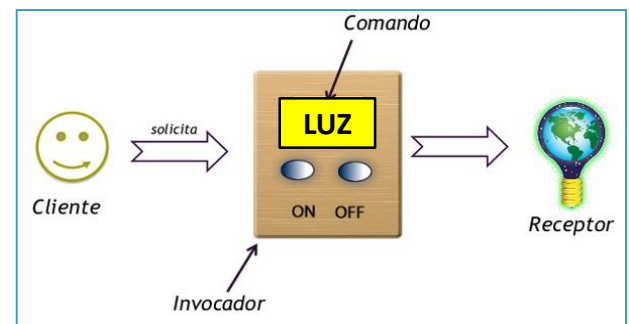
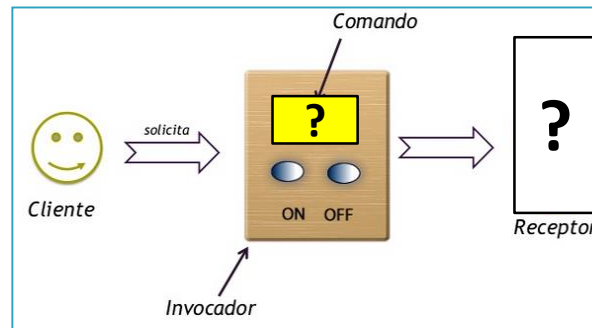
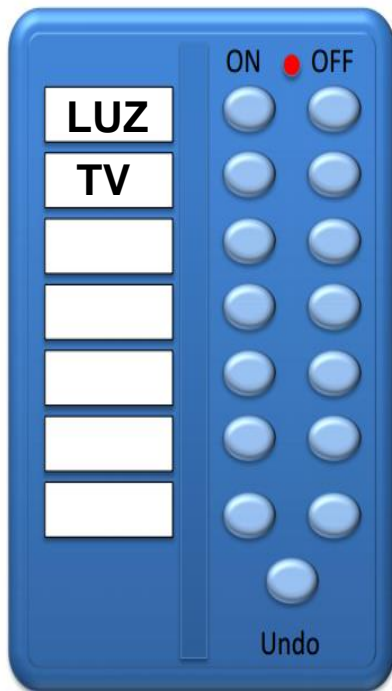
► Solução

- Tratar toda requisição como um objeto que implementa uma **interface** de comando



Padrão MVC

► Pattern: Command



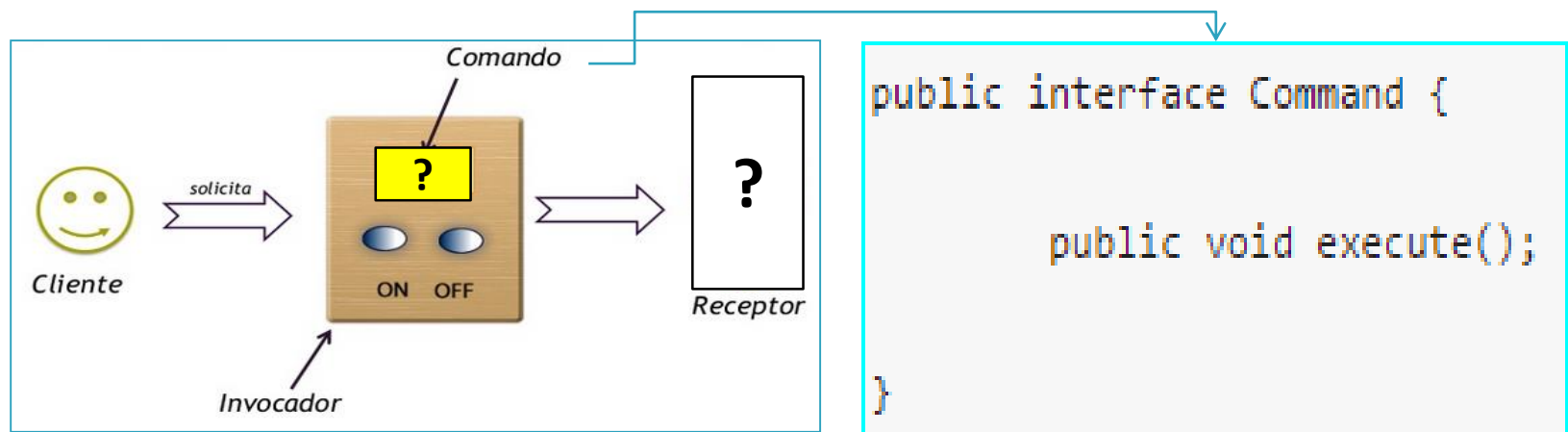
- O controlador passa a ser apenas um invocador
 - Não sabe como fazer, nem quem de fato faz
- A **requisição** é tratada como um objeto de comando
 - Determina o receptor e a operação a ser executada

Padrão MVC

► Pattern: Command

► Solução

- Desacoplar quem requisita a ação (o invocador) de quem de fato executa a ação (o receptor)



- O controlador vai encapsular o receptor e as operações na ideia abstrata de um comando com uma interface execute()

Padrão MVC

O que queremos?

```
String nomeDaClasse = request.getParameter("logica");  
new nomeDaClasse().executa(request, response);
```

► Como fazer isso?

► Aplicando polimorfismo

```
public interface Logica {  
    String executa(HttpServletRequest req, HttpServletResponse resp)  
        throws Exception;  
}
```

Retorna um objeto da
classe associada à String

```
Class<?> classe = Class.forName(nomeClasse);  
Logica logica = (Logica) classe.newInstance();  
String pagina = logica.executa(request, response);
```

```
public interface Logica {  
    String executa(HttpServletRequest req, HttpServletResponse resp)  
        throws Exception;  
}
```

Command

```
public class AdicionaContatoLogica implements Logica{  
    @Override  
    public String executa(HttpServletRequest request, HttpServletResponse response)  
        throws Exception {  
        /***** log *****/  
        System.out.println("Criando um novo contato");  
        /*****acessando bean *****/  
        Contato contato = new Contato(0,  
            request.getParameter("nome"),  
            request.getParameter("email"),  
            request.getParameter("telefone"));  
        /*****adicionando ao BD *****/  
        ContatoDAO dao = new ContatoDAO();  
        dao.addContato(contato);  
        /***** ok *****/  
        return "view/contato-adicionado.jsp";  
    }  
}
```

Receiver



ConcreteCommand

@WebServlet("/mvc")

public class **ControllerServlet** extends HttpServlet{

Invoker

String pacote = "mvc.logica.";

@Override

protected void **service**(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {

String acao = request.getParameter("logica");

String nomeClasse = pacote + acao;

System.out.println(nomeClasse);

try {

Class<?> classe = Class.forName(nomeClasse);

Logica logica = (Logica) classe.newInstance();

String pagina = logica.executa(request, response);

RequestDispatcher rd =

request.getRequestDispatcher(pagina);

rd.forward(request, response);

} catch (Exception e) {

throw new ServletException("Exceção gerada pela lógica de negócios", e);

}

}

Padrão MVC

► view/formAdicionaContato.jsp

```
<form action="mvc?logica=AdicionaContatoLogica" method="post">
  Nome: <input type="text" name="nome">
  <br><br>
  Email: <input type="text" name="email">
  <br><br>
  Telefone: <input type="text" name="telefone">
  <br><br>
  <input type="submit" value="Adicionar">
</form>
```


Padrão MVC

► Listando Contatos

► view/formListaContato.jsp

```
<jsp:useBean id="dao" class="mvc.dao.ContatoDAO" />
<table>
  <c:forEach var="contato" items="${dao.listaContatos}">
    <tr>
      <td>${contato.nome}</td>
    </tr>
  </c:forEach>
</table>
```

- (*antipattern*) Instanciar objetos da camada *Model* na camada *View* não é uma boa prática na arquitetura MVC.

Padrão MVC

► Lógica para Listar Contatos

```
public class ListaContatoLogica implements Logica {
    @Override
    public String executa(HttpServletRequest request, HttpServletResponse response)
        throws Exception {

        //Carrega lista de contato
        ContatoDAO dao = new ContatoDAO();
        List<Contato> listaContatos = dao.getListContatos();

        //armazena lista em um request
        request.setAttribute("listaContatos", listaContatos);

        return "view/formListaContato.jsp";
    }
}
```

localhost:8080/MyWebSite/mvc?logica=ListaContatoLogica

Padrão MVC

► Listando Contatos

► formListaContato.jsp

```
<table>
  <c:forEach var="contato" items="{listaContatos}">
    <tr>
      <td>${contato.nome}</td>
    </tr>
  </c:forEach>
</table>
```

Padrão MVC



▶ Seguindo o padrão MVC

▶ Implemente e teste

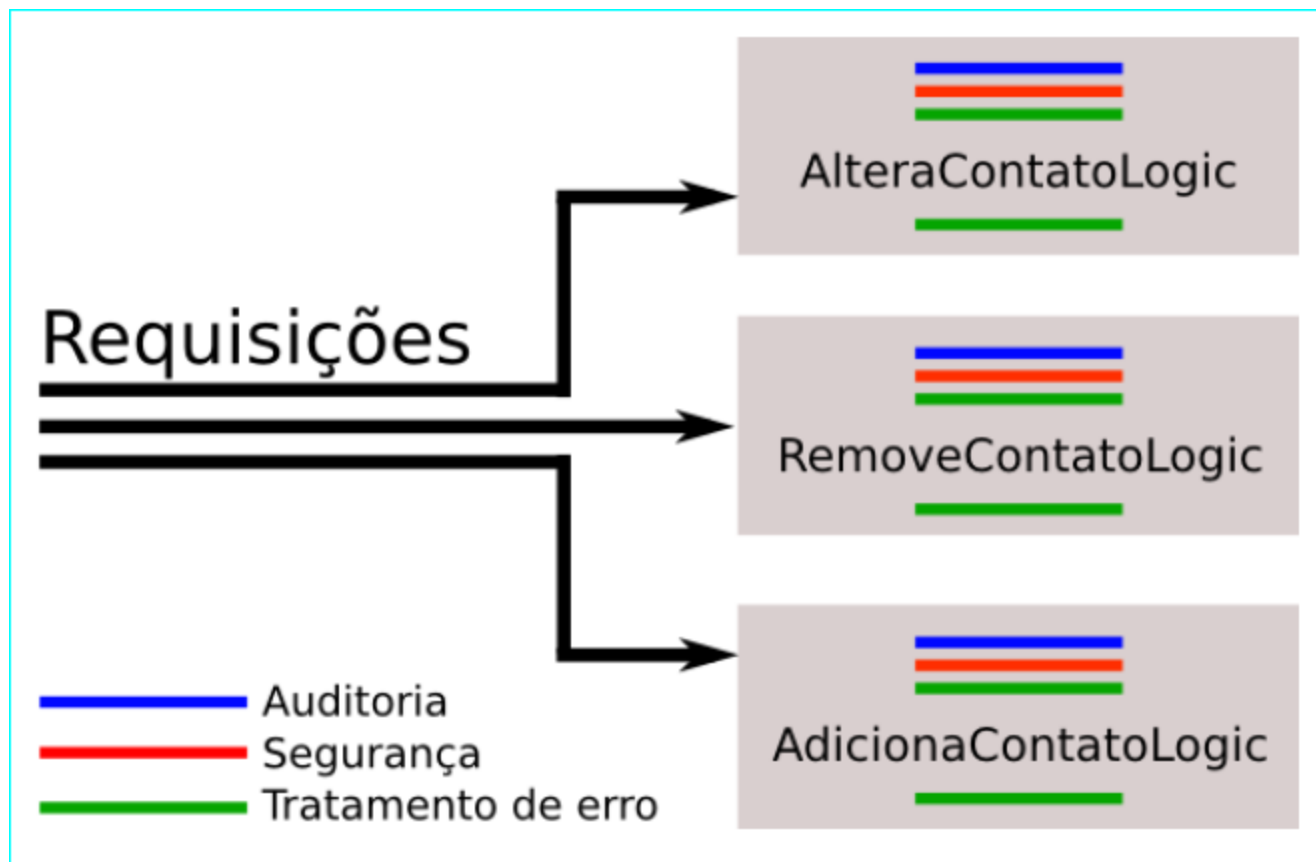
- ▶ Lógicas para: Cadastro, Listagem, Alteração e Remoção de Usuários

Usuario
- idUsuario : int - nome : String - login : String - senha : String
+ addUsuario(usuario : Usuario) : boolean + getListUsuario() : List<Usuario> + updateUsusario(usuario : Usuario) : boolean + deleteUsuario(idUsuario : int) : boolean

powered by Astah

Padrão MVC

- ▶ Como implementar requisitos não funcionais?
 - ▶ Forte acoplamento entre lógica e requisitos não funcionais



Padrão MVC



► Simple, but not smart!

```
public class AdicionaContatoLogica implements Logica{

    private static final Logger LOGGER =
        Logger.getLogger(AdicionaContatoLogica.class.getName());

    @Override
    public String executa(HttpServletRequest request, HttpServletResponse response)
        throws Exception {

        /***** auditoria *****/
        LOGGER.info("Usuário Fulano - Acessando : AdicionaContatoLogica");

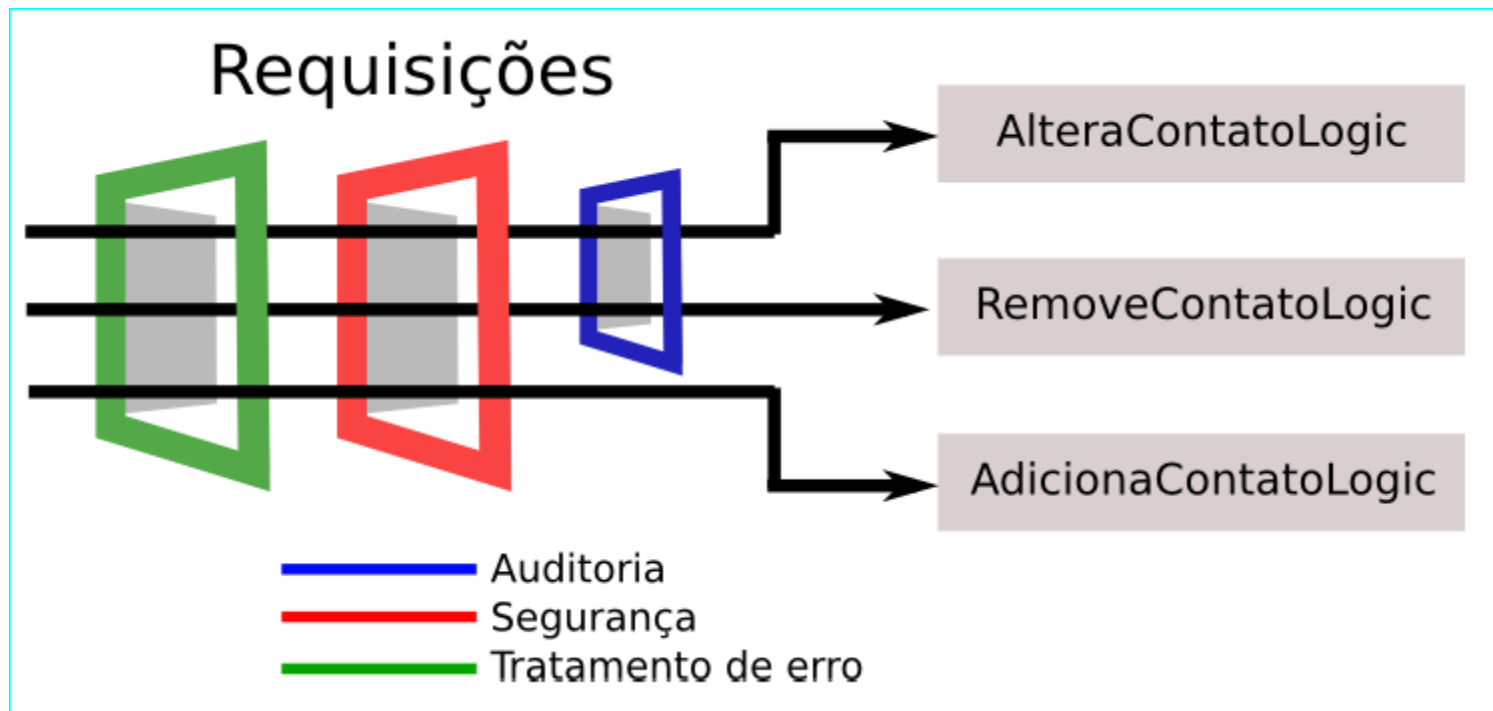
        /***** segurança *****/
        if(!usuario.ehCliente()){
            return "view/acesso-negado.jsp";
        }
        /***** continua lógica *****/
    }
}
```

Padrão MVC

► Como diminuir esse acoplamento?

► Filtros (API Servlet)

- Classes que permitem executar códigos antes da requisição e depois da resposta



Padrão MVC

► Filtros na Prática

► Filtros (API Servlet)

```
@WebFilter("/*")
public class MeuFiltro implements Filter {

    @Override
    public void init(FilterConfig filterConfig) {}

    @Override
    public void doFilter(ServletRequest request,
                        ServletResponse response,
                        FilterChain chain) {}

    @Override
    public void destroy() {}



}
```

```
@WebFilter("/mvc")
```


Padrão MVC

► Filtros na Prática

► Filtros (API Servlet)

```
public void doFilter(ServletRequest request,  
                    ServletResponse response, FilterChain chain)  
    throws IOException, ServletException {  
     // passa pela porta  
    chain.doFilter(request, response);  
      
}
```

Padrão MVC

► Filtros na Prática

► Filtro para medir tempo de execução

```
public void doFilter(ServletRequest request,
    ServletResponse response, FilterChain chain)
    throws IOException, ServletException {

    long tempoInicial = System.currentTimeMillis();

    chain.doFilter(request, response);

    long tempoFinal = System.currentTimeMillis();

    String uri = ((HttpServletRequest)request).getRequestURI();
    String parametros = ((HttpServletRequest) request).getParameter("logica");

    System.out.println("Tempo da requisicao de " + uri
        + "?logica="
        + parametros + " demorou (ms): "
        + (tempoFinal - tempoInicial));
}
```

Padrão MVC

► Filtros na Prática

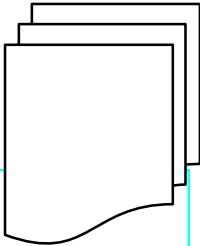
► Conexão com BD

```
public class ConnectionFactory {  
  
    public static Connection getConnection() {  
        try{  
            String host = "jdbc:mysql://localhost/sysControleAcademico";  
            String user = "root";  
            String password = "";  
            return DriverManager.getConnection(  
                host, user, password);  
        } catch (SQLException e) {  
            throw new RuntimeException(e);  
        }  
    }  
}
```

Padrão MVC

► Filtros na Prática

► Conexão com BD



```
public class ContatoDAO {  
  
    private Connection connection;  
  
    /** estabelece conexao **/  
    public ContatoDAO() {  
        connection = ConnectionFactory.getConnection();  
    }  
  
    /** continua **/  

```

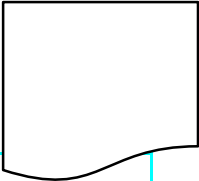
Forte Acoplamento!

Padrão MVC

► Filtros na Prática

► Conexão com BD

► Injeção de Dependência



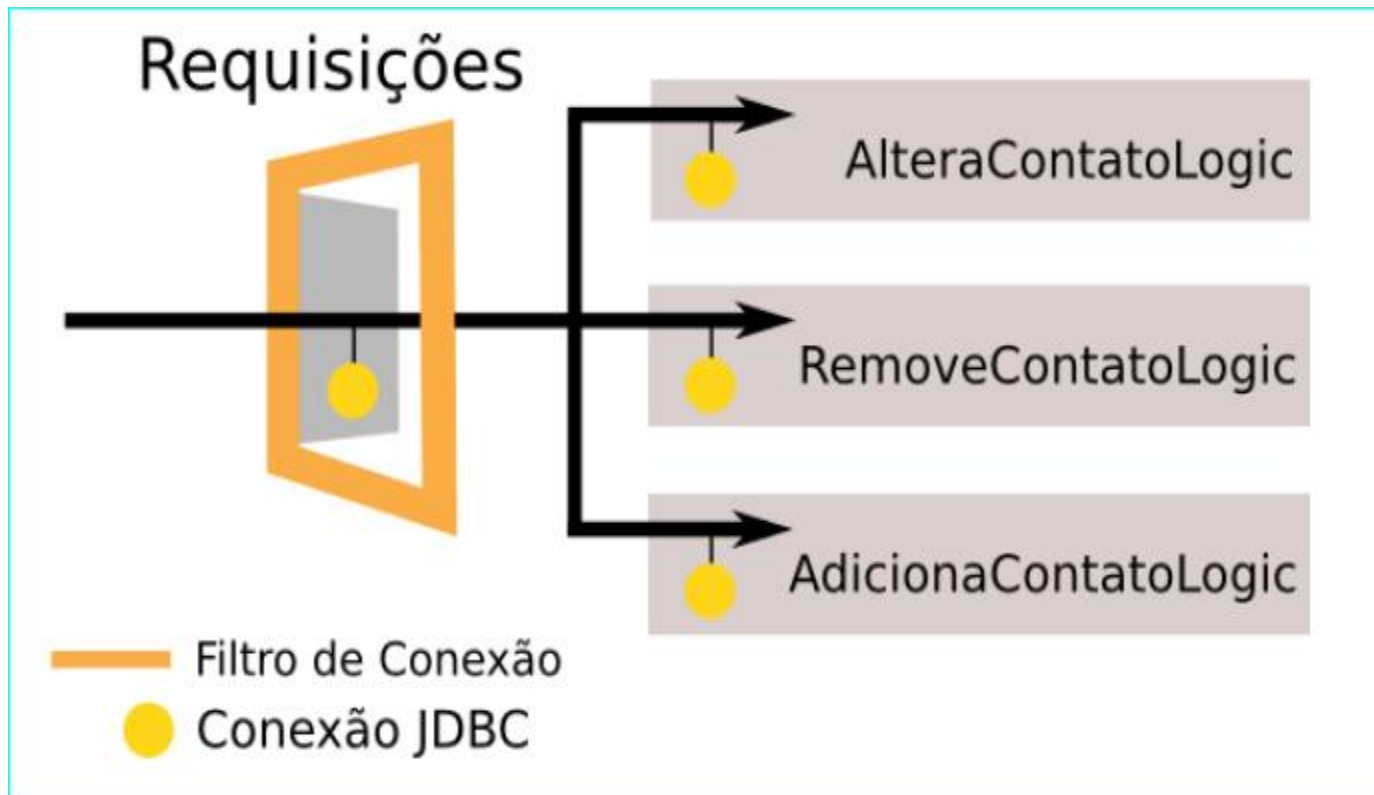
```
public class ContatoDAO {  
  
    private Connection connection;  
  
    /** recebe conexão **/  
    public ContatoDAO(Connection connection) {  
        this.connection = connection;  
    }  
}
```

Baixo Acoplamento!

Padrão MVC

► Filtros na Prática

- Filtro de Conexão com BD
 - Injeção de Dependência



```
public void doFilter(ServletRequest request,
    ServletResponse response, FilterChain chain)
    throws IOException, ServletException {

    try {
        //estabelece conexao
        Connection connection = ConnectionFactory.getConnection();

        //armazena objeto no request
        request.setAttribute("connection", connection);

        //prossegue execucao do request
        chain.doFilter(request, response);

        //fecha conexao
        connection.close();

    } catch (SQLException ex) {
        throw new RuntimeException(ex);
    }
}
```

```
public class AdicionaContatoLogica implements Logica{

    @Override
    public String executa(HttpServletRequest request, HttpServletResponse response)
        throws Exception {

        /*****acessando bean *****/
        Contato contato = new Contato(0,
            request.getParameter("nome"),
            request.getParameter("email"),
            request.getParameter("telefone"));
        /*****recuperando conexao *****/
        Connection connection = (Connection) request
            .getAttribute("connection");
        /*****adicionando ao BD *****/
        ContatoDAO dao = new ContatoDAO(connection);
        dao.addContato(contato);
        /***** ok *****/
        return "view/contato-adicionado.jsp";
    }
}
```