Introdução

- *A camada de apresentação encapsula toda a lógica relacionada com a visualização e comunicação com a interface do usuário*
  - *Requisições e respostas HTTP*
  - *Gerenciamento de sessão HTTP*
  - *Geração de HTML, JavaScript e recursos lado-cliente*
- *Principais componentes: Servlets e JSP*
  - *JSP é indicado para produzir interface do usuário*
  - *Servlets são indicados para processar dados recebidos e concentrar lógica de controle*

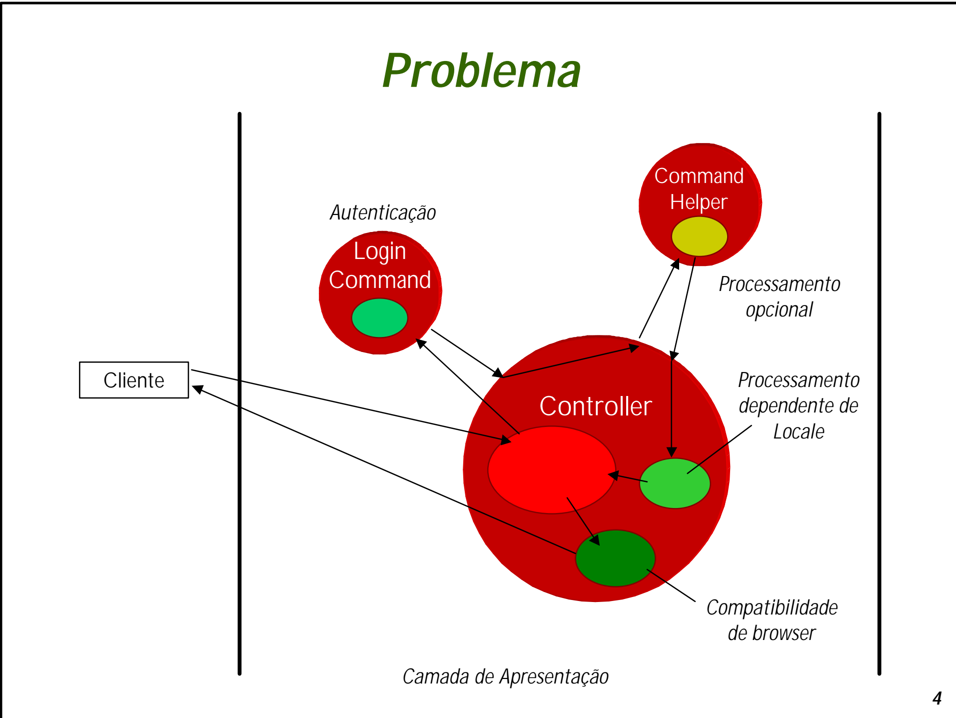
2

1

Intercepting Filter

Objetivo: permitir o pré- e pós processamento de uma requisição. Intercepting Filter permite encaixar filtros decoradores sobre a requisição ou resposta e remover código de transformação da requisição do controlador

Problema



Descrição do problema

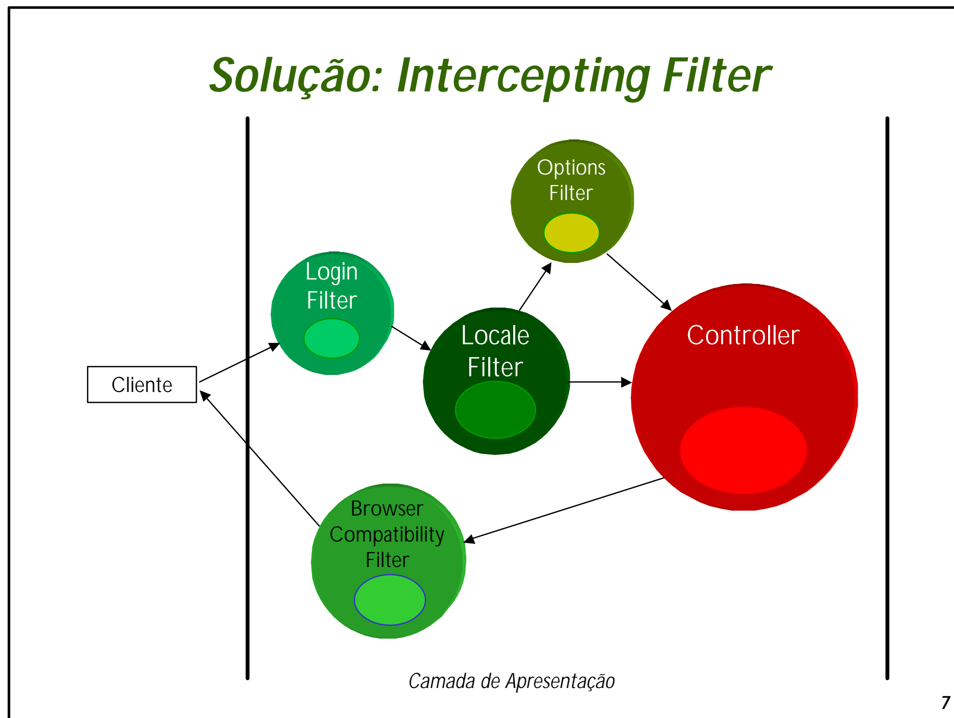
- *A camada de apresentação recebe vários diferentes tipos de requisições, que requerem processamento diferenciado*
- *No recebimento de uma requisição, várias decisões precisam ser tomadas para selecionar a forma de realização do processamento*
  - *Isto pode ser feito diretamente no controlador via estruturas if/else. Desvantagem: embute fluxo da filtragem no código compilado, dificultando a sua remoção ou adição*
  - *Incluir tratamento de serviços no próprio controlador impede que esse código possa ser reutilizado em outros contextos*

5

O que se deseja?

- *Interceptar e manipular uma requisição antes e depois do seu processamento*
- *Processamento centralizado e compartilhado através de requisições. Aplicações típicas:*
  - *Data encoding*
  - *Informações de logging*
  - *Compressão de dados*
- *Baixo acoplamento para facilitar reconfiguração e remoção*
- *Componentes independentes que possam ser ligados em cascata*

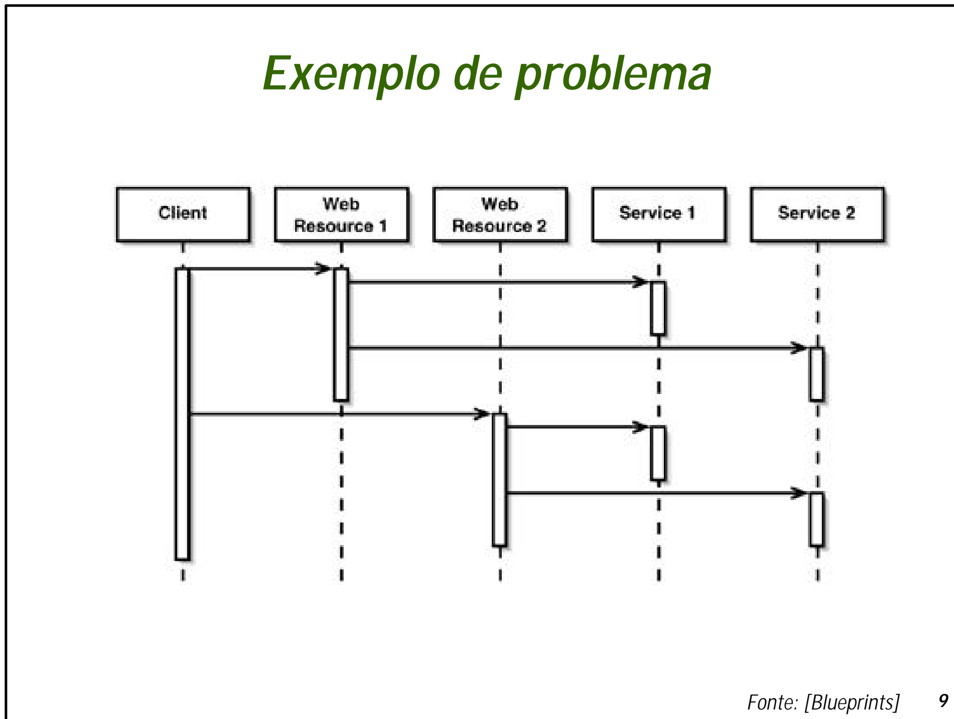
6

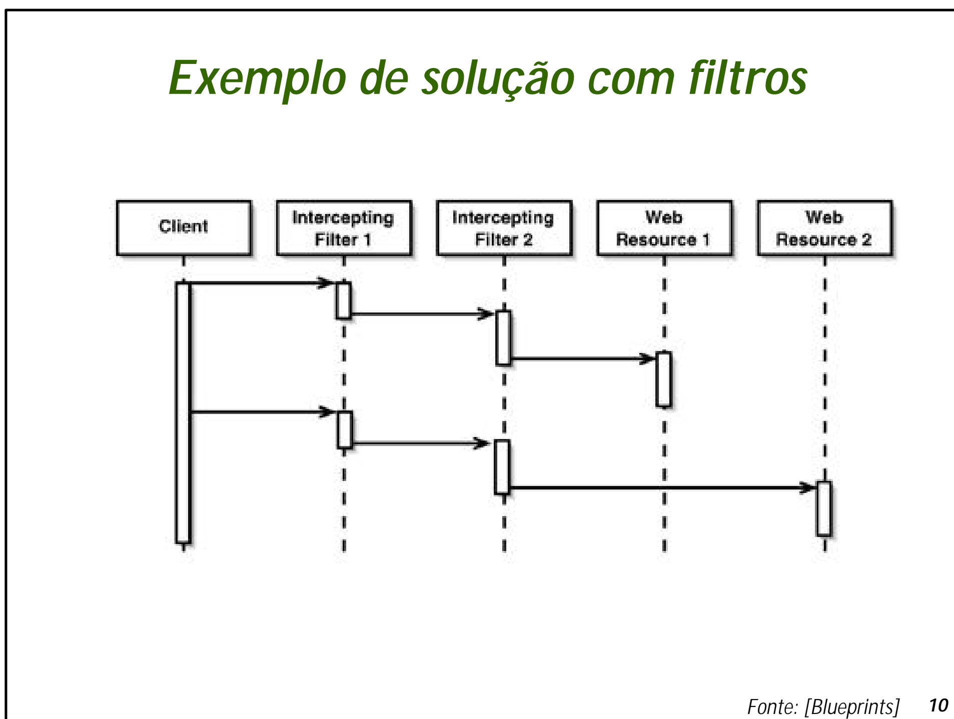


Descrição da solução

- *Criar filtros plugáveis para processar serviços comuns de forma padrão, sem requerer mudanças no código de processamento*
  - *Um gerenciador de filtros pode combinar filtros fracamente acoplados em uma corrente, delegando o controle ao filtro apropriado para cada serviço*
  - *Pode-se adicionar, remover e reorganizar a ordem de chamada dos filtros sem precisar alterar código existente*

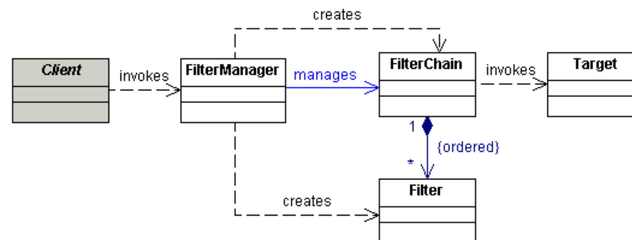
8





© 2003, Helder L. S da Rocha

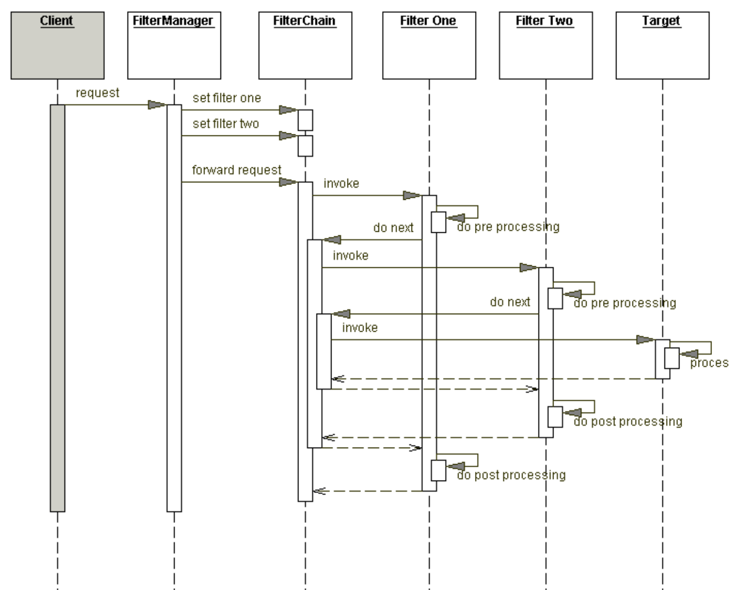
Estrutura UML



- *Client: envia requisição ao FilterManager*
- *FilterManager: gerencia processamento de filtros (web.xml)*
- *FilterChain: coleção ordenada de filtros independentes (API)*
- *Filter (FilterOne, FilterTwo, ...): representam filtros individuais mapeados a um alvo. O FilterChain coordena o processamento dos filtros (API).*

Fonte: [SJC] 11

Diagrama de Seqüência



Fonte: [SJC] 12

© 2003, Helder L. S da Rocha

Melhores estratégias de implementação

- Standard Filter Strategy
  - Solução padrão usando J2EE (javax.servlet.Filter)
- Base Filter Strategy e Template Filter Strategy
  - Implementam o padrão Template Method: filtro com operações genéricas e abstratas é implementado em diferentes instâncias
- Web Services*
  - Custom SOAP Filter Strategy (usando JAXM/SAAJ)
  - JAX RPC Filter Strategy (usando API JAX-RPC)

* Não abordado neste curso

13

Conseqüências

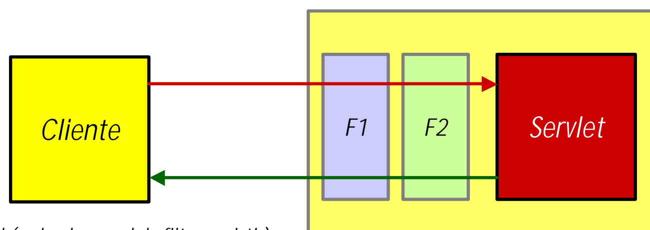
- Centraliza controle com processadores fracamente acoplados
  - Como um controlador, um filtro fornece um ponto centralizado para processamento de requisições
  - Podem ser removidos, adicionados, combinados em cascata
- Melhora reuso
  - Filtros são destacados do controlador e podem ser usados em outros contextos
- Configuração declarativa e flexível
  - Serviços podem ser reorganizados sem recompilação
- Compartilhamento ineficiente de informações
  - Se for necessário compartilhar muitas informações entre filtros, esta solução não é recomendada

14

© 2003, Helder L. S da Rocha

Mini-tutorial*: Filtros padrão (API Servlet 2.3)

- Um filtro é um componente Web que reside no servidor
  - Intercepta as requisições e respostas no seu caminho até o servlet e de volta ao cliente
  - Sua existência é ignorada por ambos. É totalmente transparente tanto para o cliente quanto para o servlet
- Filtros podem ser concatenados em uma corrente
  - Neste cenário, as requisições são interceptadas em uma ordem e as respostas em ordem inversa

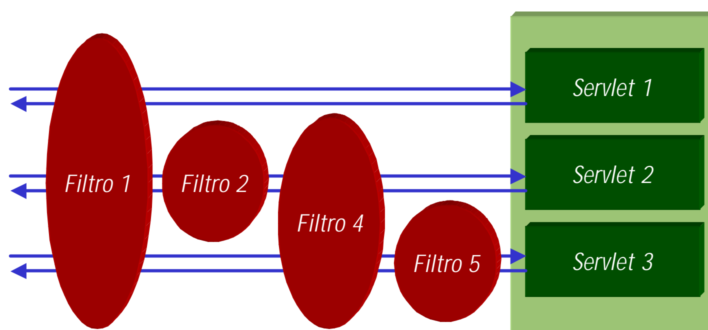


* Abordagem opcional (se background de filtros existir)

15

Filtros padrão

- Um filtro pode realizar diversas transformações, tanto na resposta como na requisição antes de passar esses objetos adiante (se o fizer)
- Filtros podem ser reutilizados em vários servlets



16

Como funcionam?

- Quando o container recebe uma requisição, ele verifica se há um filtro associado ao recurso solicitado. Se houver, a requisição é roteada ao filtro
  - Container é o FilterManager (configurado via web.xml)
- O filtro, então, pode
  1. Gerar sua própria resposta para o cliente
  2. Repassar a requisição, modificada ou não, ao próximo filtro da corrente, se houver, ou ao recurso final, se ele for o último filtro
  - Rotear a requisição para outro recurso
- Na volta para o cliente, a resposta passa pelo mesmo conjunto de filtros em ordem inversa

17

API Essencial

```
javax.servlet.Filter
   void init(FilterConfig),
   void doFilter(ServletRequest,
                 ServletResponse,
                 FilterChain)
   void destroy()
javax.servlet.FilterConfig
   String getFilterName()
   String getInitParameter(String name)
   Enumeration getInitParameterNames()
   ServletContext getServletContext()
javax.servlet.FilterChain
   void doFilter(ServletRequest, ServletResponse)
```

18

© 2003, Helder L. S da Rocha

API: Classes empacotadoras

- Úteis para que filtros possam trocar requisição por outra
  - Subclasse dessas classes empacotadoras pode ser passada em corrente de filtros no lugar da requisição ou resposta original
  - Métodos como getParameter() e getHeader() podem ser sobrepostos para alterar parâmetros e cabeçalhos
- No pacote javax.servlet

  ServletRequestWrapper implements ServletRequest: implementa todos os métodos de ServletRequest e pode ser sobreposta para alterar o request em um filtro

  ServletResponseWrapper implements ServletResponse: implementa todos os métodos de ServletResponse
- No pacote javax.servlet.http

  HttpServletRequestWrapper e **HttpServletResponseWrapper**: implementam todos os métodos das interfaces correspondentes, facilitando a sobreposição para alteração de cabeçalhos, etc.

19

Como escrever um filtro simples

1. Escreva uma classe implementando a interface Filter e todos os seus métodos

```
init(FilterConfig)
doFilter(ServletRequest, ServletResponse, FilterChain)
destroy()
```

2. Compile usando o JAR da Servlet API
3. Configure o filtro no deployment descriptor (web.xml) usando os elementos `<filter>` e `<filter-mapping>`
   - Podem ser mapeados a URLs, como servlets
   - Podem ser mapeados a servlets, para interceptá-los
   - A ordem dos mapeamentos é significativa
4. Instale o filtro da maneira usual no servidor (deploy)

20

Filtro simples que substitui servlet

```
package filtros;
import java.io.*;
import javax.servlet.*;

public class HelloFilter implements Filter {
  private String texto;
  public void init(FilterConfig config) {
    texto = config.getInitParameter("texto");
  }
  public void doFilter(ServletRequest request,
                       ServletResponse response,
                       FilterChain filterChain)
                        throws ServletException, IOException {
      PrintWriter out = response.getWriter();
      out.println("<HTML><HEAD><TITLE>Filter Response");
      out.println("</TITLE></HEAD><BODY>");
      out.println("<H1>Filter Response</H1>");
      out.println("<P>" + texto);
      out.println("</BODY></HTML>");
      out.close();
   }
   public void destroy() {}
}
```

21

Configuração

- *Os elementos <filter> e <filter-mapping> são quase idênticos aos equivalentes para <servlet>*
  - *A diferença é que <filter-mapping> é usado também para associar filtros a servlets, na ordem em que aparecem*
- *Filtro simples, que substitui um servlet*

```
<filter>
    <filter-name>UmFiltro</filter-name>
    <filter-class>filtros.HelloFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>UmFiltro</filter-name>
    <url-pattern>/filtro</url-pattern>
</filter-mapping>
```

- *Filtro que intercepta um servlet*

```
<filter-mapping>
    <filter-name>UmFiltro</filter-name>
    <servlet-name>UmServlet</servlet-name>
</filter-mapping>
```

22

Filtros "de verdade"

- *Filtros úteis podem ser encadeados em uma corrente. Para que isto seja possível, devem chamar doFilter() no objeto FilterChain - parâmetro no seu próprio doFilter()*

```
public void doFilter(...req,...res, FilterChain chain)
  {
   ...
   chain.doFilter(req, res);
   ...
}
```

- *Antes da chamada ao doFilter, o filtro pode processar a requisição e alterar ou substituir os objetos ServletRequest e ServletResponse ao passá-los adiante*

```
ServletRequest newReq = new ModifiedRequest(...);
chain.doFilter(newReq, res);
```

Estende ServletRequestWrapper

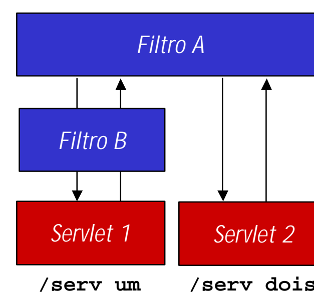
- *Na volta, opera sobre a resposta e pode alterá-la*

23

Configuração da corrente

- *A corrente pode ser configurada com definição das instâncias de filtros e mapeamentos em ordem*

```
<filter>
    <filter-name>FiltroA</filter-name>
    <filter-class>filtros.FilterA</filter-class>
</filter>
<filter>
    <filter-name>FiltroB</filter-name>
    <filter-class>filtros.FilterB</filter-class>
</filter>
<filter-mapping>
    <filter-name>FiltroA</filter-name>
    <url-pattern>/serv_um</url-pattern>
</filter-mapping>
<filter-mapping>
    <filter-name>FiltroA</filter-name>
    <servlet-name>Servlet2</servlet-name>
</filter-mapping>
<filter-mapping>
    <filter-name>FiltroB</filter-name>
    <servlet-name>Servlet1</servlet-name>
</filter-mapping>
```



24

Filtros que tomam decisões

- *Um filtro pode ler a requisição e tomar decisões como transformá-la, passar adiante ou retornar*

```
public void doFilter(.. request, ... response, ... chain) {
   String param = request.getParameter("saudacao");
   if (param == null) {
      PrintWriter out = response.getWriter();
      response.setContentType("text/html");
      out.println("<h1>Erro!</h1>");
   } else if (param.equals("Bonjour!")) {
      class MyWrapper extends ServletRequestWrapper { // ...
        public String getParameter(String name) {
          if (name.equals("saudacao")) {
             return "Bom Dia";
          }
        }
      }
      ServletRequest myRequest = new MyWrapper(request);
      chain.doFilter(myRequest, response);
   } else { chain.doFilter(request, response); }
}
```

Classe interna (o construtor (abaixo) foi omitido da declaração por falta de espaço)

25

Wrappers

- *Sobrepondo um HttpServletRequest*

```
public class MyServletWrapper
                          extends HttpServletRequestWrapper {
   public MyServletWrapper(HttpServletRequest req) {
      super(req);
   }
   public String getParameter(String name) {
      return super.getParameter(name).toUpperCase();
   }
}
```

- *Usando Wrappers em servlets HTTP*

```
HttpServletRequest req = (HttpServletRequest)request;
HttpServletResponse res = (HttpServletResponse)response;
HttpServletRequest fakeReq = new MyServletWrapper(req);
HttpServletResponse fakeRes = new TestResponse(res);

chain.doFilter(fakeReq, fakeRes);
```

26

Observações importantes

- *Filtros não são chamados quando o recurso que interceptam for chamado através de um RequestDispatcher*
  - *O recurso é acessado diretamente sem filtragem*
  - *Isto ocorre para evitar loops infinitos*
- *Filtros associados a páginas de erro também não são chamados*

27

6.1-6.13

Exercícios (preslayer/if/)

1. *Analise a implementação das estratégias de Intercepting Filter*
2. *Escreva e configure um filtro simples que leia a requisição e verifique se ela contém os parâmetros usuario e senha*
   - *Se não tiver, repasse a requisição para a página erro.html*
   - *Se tiver, abra o arquivo usuarios.txt usando a classe Properties. Ele possui uma lista de nome=senha, um por linha. Veja se o usuário coincide com a senha. Se sim, chame o próximo filtro. Se não, redirecione para acessoNegado.html (código está pronto)*
   - *Associe o filtro a um servlet qualquer (o SimpleServlet, por exemplo)*
   - *Acesse o servlet e verifique que ele passa pelo filtro*
3. *Escreva dois RequestWrappers que encapsulam HttpServletRequest e sobrepõem o método getParameter()*
   - *O primeiro, UpperCaseFilter, põe valores dos parâmetros em caixa-alta.*
   - *O segundo, ReverseFilter, inverte o texto dos parâmetros*
   - *Coloque os dois em cascata apontando para um servlet simples.*

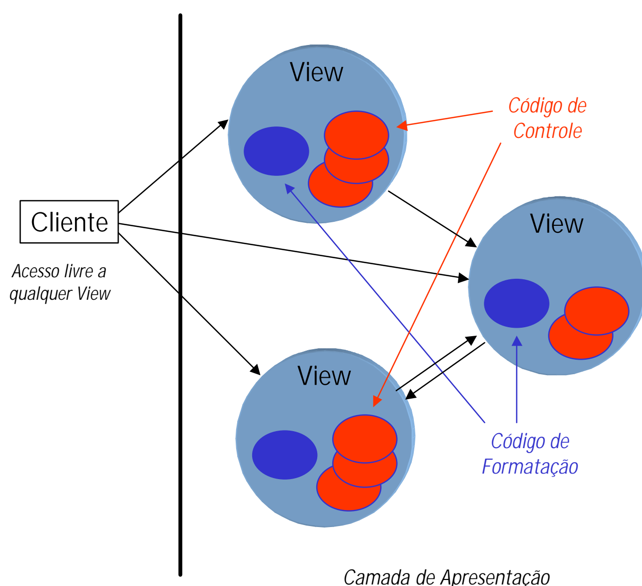
28

2

Front Controller

Objetivo: centralizar o processamento de requisições em uma única fachada. Front Controller permite criar uma interface genérica para processamento de comandos.

Problema



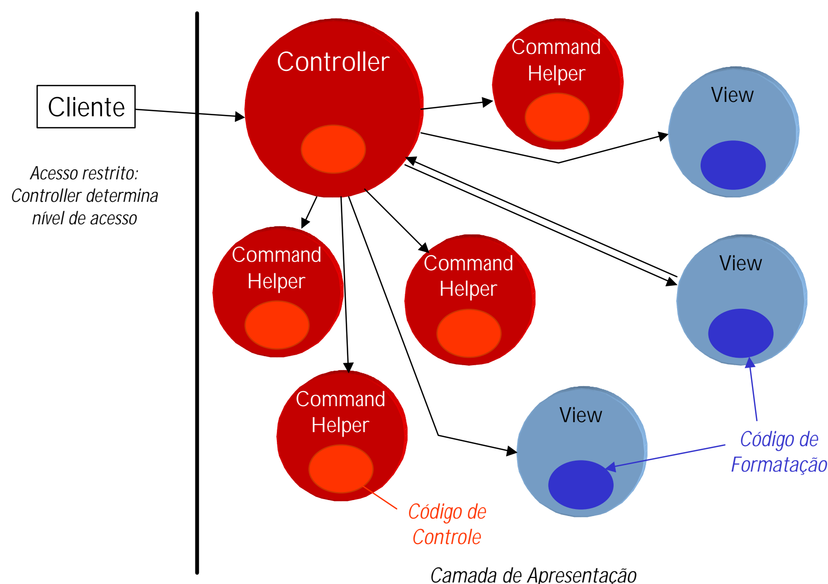
30

Descrição do problema

- *Deseja-se um ponto de acesso centralizado para processamento de todas as requisições recebidas pela camada de apresentação para*
  - *Controlar a navegação entre os Views*
  - *Remover duplicação de código*
  - *Estabelecer responsabilidades mais definidas para cada objeto, facilitando manutenção e extensão: JSP não deve conter código algum ou pelo menos não código de controle*
- *Se um usuário acessa um View sem passar por um mecanismo centralizado, código de controle é duplicado e misturado ao código de apresentação*
  - *Também não é possível controlar a navegação: cliente pode iniciar em página que não deveria ter acesso.*

31

Solução: Front Controller



32

Descrição da solução

- *Controlador é ponto de acesso para processamento de requisições*
  - *chama serviços de segurança (autenticação e autorização)*
  - *delega processamento à camadas de negócio*
  - *define um View apropriado*
  - *realiza tratamento de erros*
  - *define estratégias de geração de conteúdo*
- *O controlador pode delegar processamento a objetos Helper (Comandos ou ações, Value Beans, etc.)*
- *Solução depende do uso de um Application Controller*
  - *Usado para redirecionar para o View correspondente*

33

Application Controller

- *Um Front Controller tipicamente utiliza um Application Controller (veja adiante) que centraliza as operações de gerenciamento de ações e views*
  - *Gerenciamento de ações: localiza e roteia para ações específicas que irão processar uma requisição*
  - *Gerenciamento de views: localiza o view apropriado (este comportamento pode ser incluido em Front Controller)*

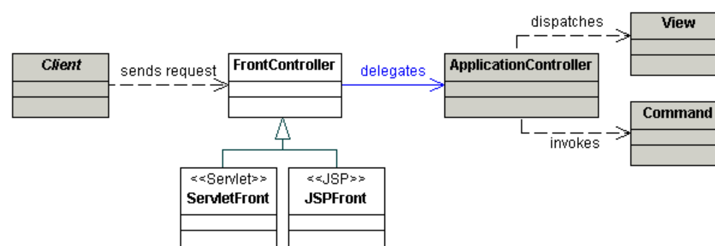
34

Processamento de uma requisição

- Envolve dois tipos de atividades
  - Manuseio da requisição
  - Processamento do View
- Durante o manuseio da requisição, é preciso realizar diversas atividades:
  - Manueio de protocolo e transformação de contexto
  - Navegação e roteamento
  - Processamento do serviço
  - Repasse da requisição

35

Estrutura UML

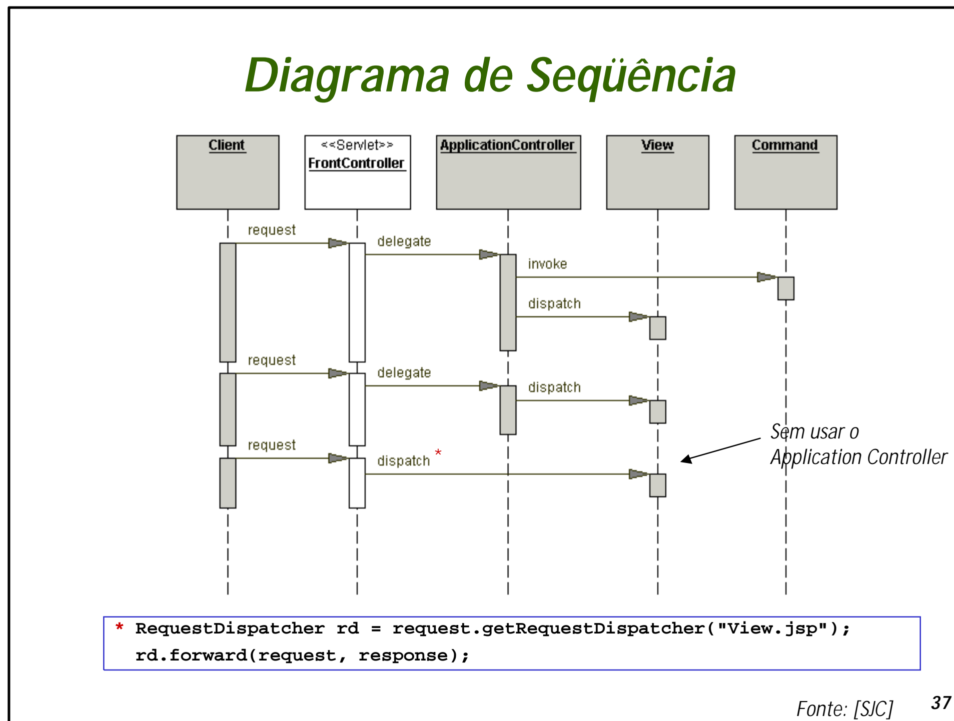


- FrontController: ponto de entrada para manuseio de requisições
- ApplicationController: gerencia de ações e views
- Command: encapsula uma ação específica para requisição
- View: representa a página retornada ao cliente

Fonte: [SJC]

36

© 2003, Helder L. S da Rocha

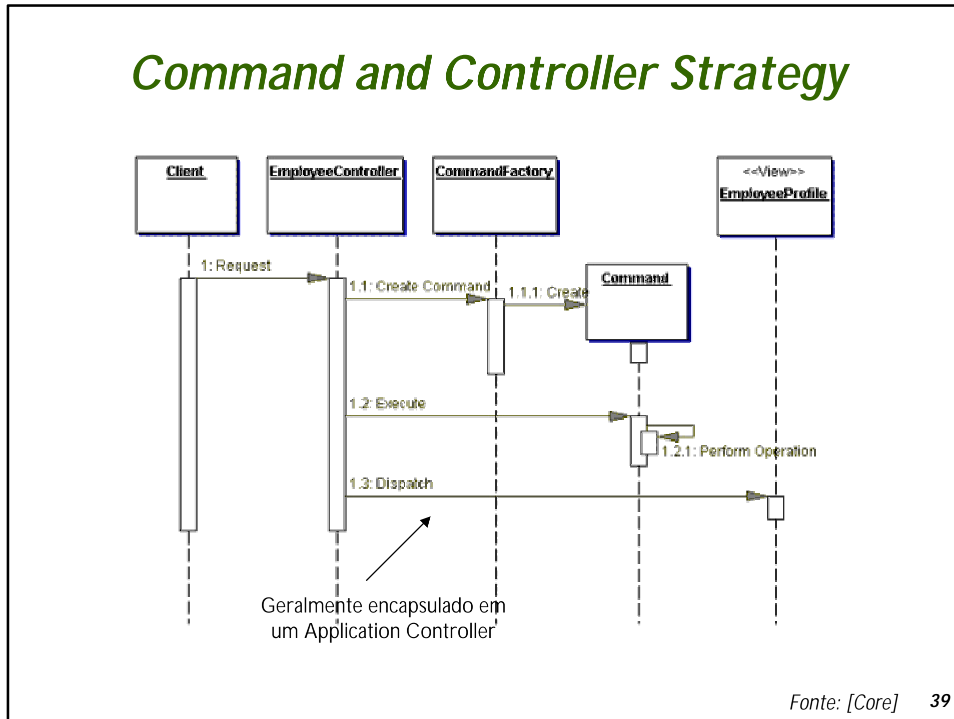


Melhores estratégias de implementação*

- Servlet Front Strategy
  - Implementa o controlador como um servlet
- Command and Controller Strategy
  - Interface baseada no padrão Command (GoF) para implementar Helpers para os quais o controlador delega responsabilidades via Application Controller
  - Esta é a estratégia mais comum
- Logical Resource Mapping Strategy
  - Requisições são feitas para nomes que são mapeados a recursos (páginas JSP, servlets) ou comandos (web.xml)
  - Multiplexed Resource Mapping Strategy usa wildcards para selecionar recursos a serem processados: *.do

* Veja exemplos de código em exemplos/preslayer/

38



Conseqüências

- *Controle centralizado*
  - *Facilidade de rastrear e logar requisições*
- *Melhor gerenciamento de segurança*
  - *Requer menos recursos. Não é preciso distribuir pontos de verificação em todas as páginas*
  - *Validação é simplificada*
- *Melhor possibilidade de reuso*
  - *Distribui melhor as responsabilidades*

40

Exercícios

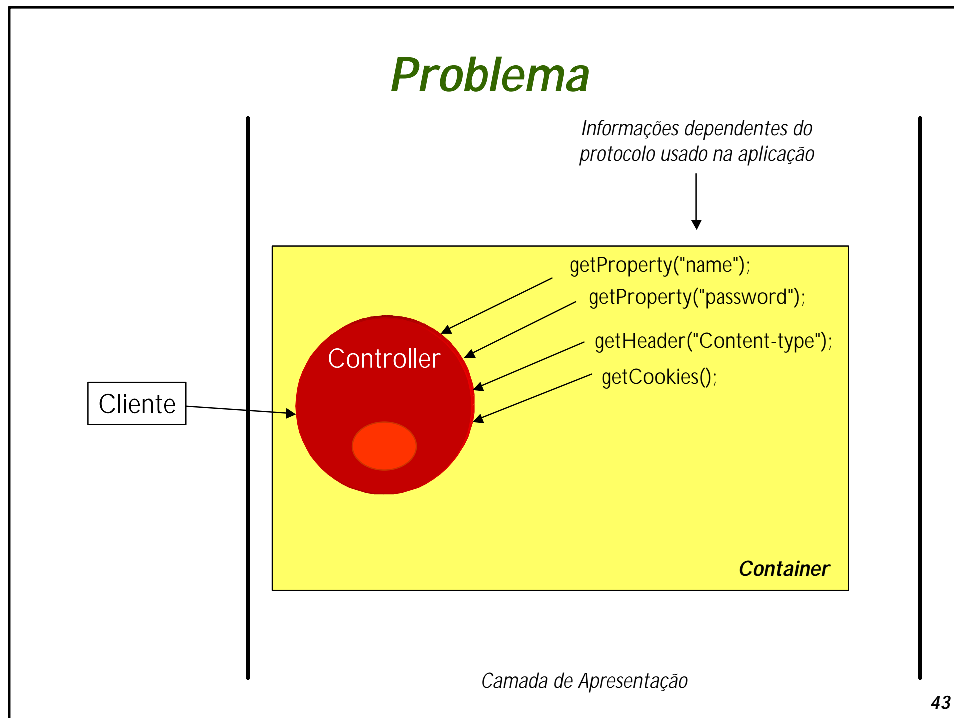
- *1. Analise a implementação das estratégias de FrontController*
- *2. Refatore a aplicação em preslayer/fc/ para que utilize FrontController. Empregue a estratégias FrontController com Command pattern:*
  - *a) Implemente o controlador usando um Servlet*
  - *b) Escreva um RequestHelper que mantenha uma tabela de comandos/nomes de classe de objetos Command e receba um request na construção. Seu método getCommand() deve retornar o comando correspondente recebendo newMessage, lastMessage, allMessages*
  - *c) Configure o web.xml para mapear todas as requisições ao controlador*

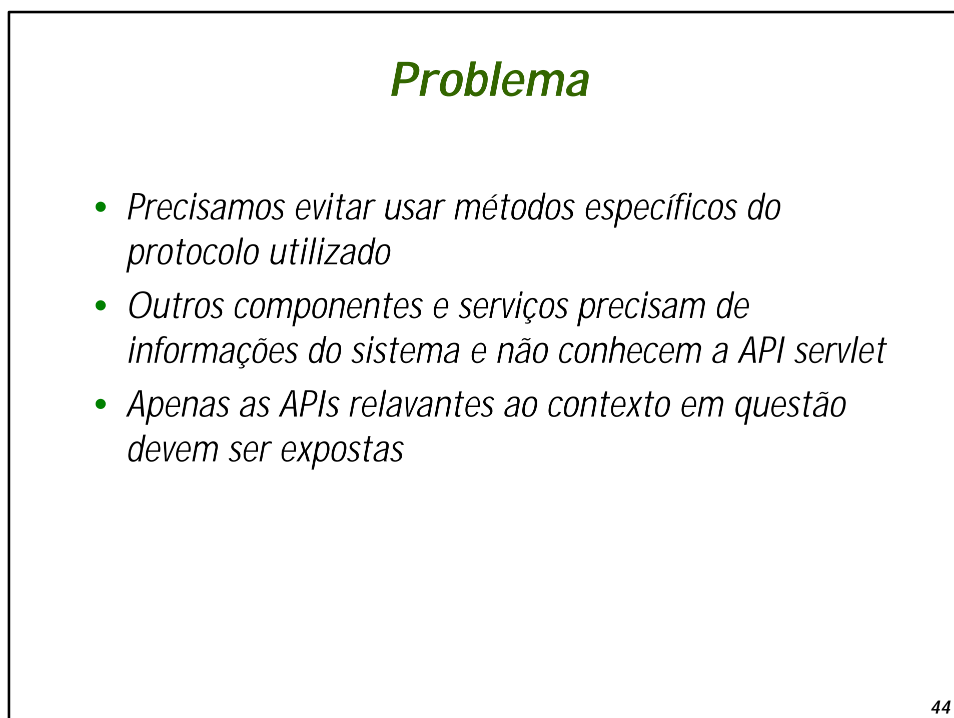
41

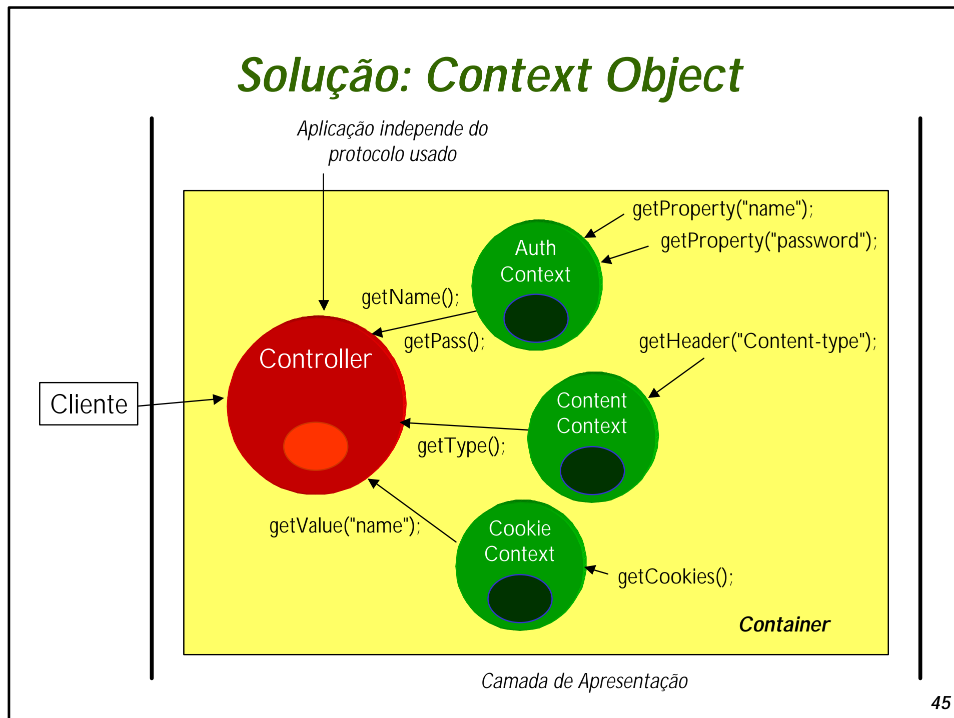
3

Context Object

Objetivo: encapsular estado de forma independente de protocolo para compartilhamento por toda a aplicação. Permite que aplicação use uma API que isole detalhes do protocolo (HTTP, por exemplo) permitindo que mesmos dados possam ser lidos em outros objetos.


Problema
Informações dependentes do protocolo usado na aplicação
getProperty("name");
getProperty("password");
getHeader("Content-type");
getCookies();
Controller
Cliente
Container
Camada de Apresentação
43

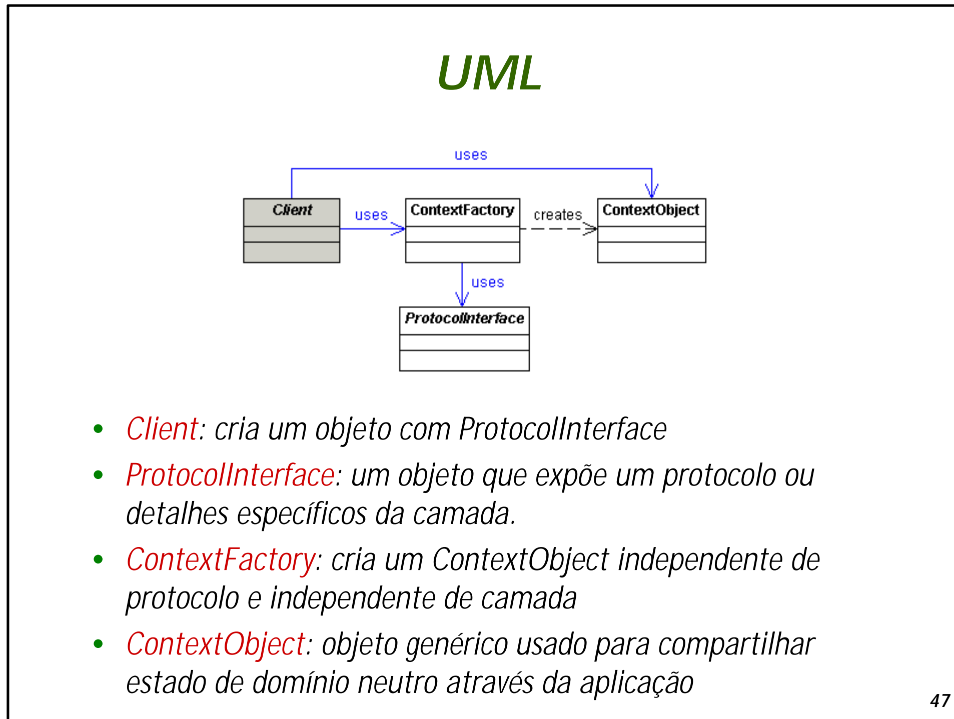

Problema
Precisamos evitar usar métodos específicos do protocolo utilizado
Outros componentes e serviços precisam de informações do sistema e não conhecem a API servlet
Apenas as APIs relavantes ao contexto em questão devem ser expostas
44

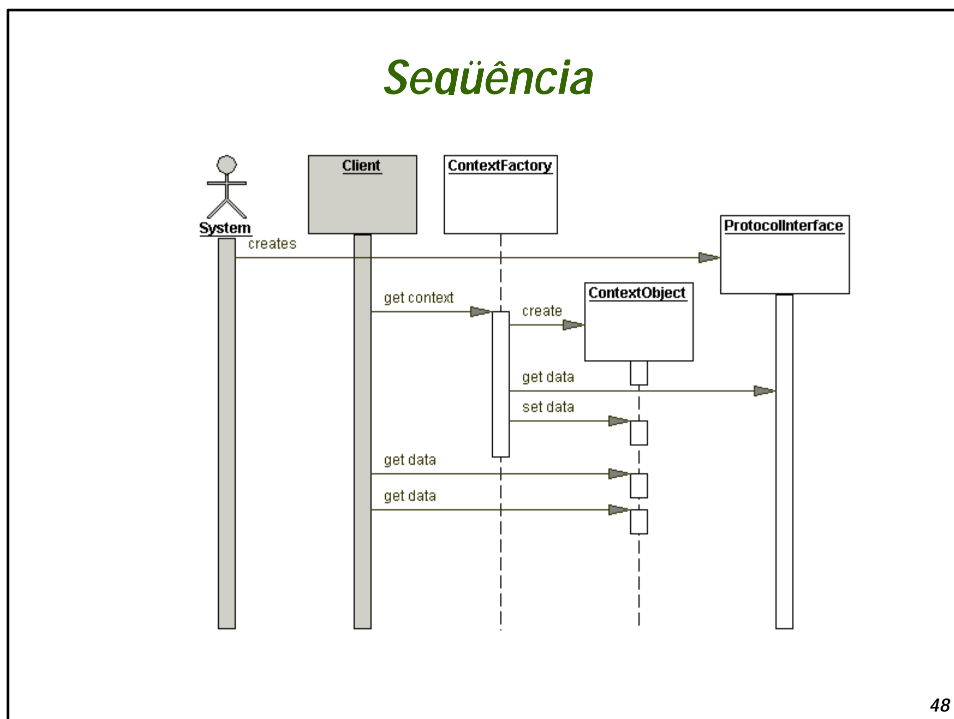


Solução

- *Um Context Object encapsula estado de forma independente de protocolo para que possa ser compartilhado por toda a aplicação.*
- *Não é a mesma coisa que Transfer Object ou Value Object*
  - *A intenção é diferente*
  - *Um Transfer Object é também usado para transferir estado, mas reflete o estado de objetos de negócio ou de repositórios de dados. Tem como objetivo reduzir tráfego na rede. Context Objects tem como objetivo isolar detalhes de implementação.*

46


UML
uses
Client
uses
ContextFactory
creates
ContextObject
uses
ProtocolInterface
Client: cria um objeto com ProtocolInterface
ProtocolInterface: um objeto que expõe um protocolo ou detalhes específicos da camada.
ContextFactory: cria um ContextObject independente de protocolo e independente de camada
ContextObject: objeto genérico usado para compartilhar estado de domínio neutro através da aplicação
47


Seqüência
System
Client
ContextFactory
ProtocolInterface
creates
ContextObject
get context
create
get data
set data
get data
get data
48

Estratégias

- Request Context Strategies (guarda estado da requisição)
  - Request Context Map Strategy (usa Map)
  - Request Context POJO Strategy (usa objeto comum)
  - Request Context Validation Strategy (vide: Struts ActionForm)
- Configuration Context Strategies (estado de configuração)
  - JSTL Configuration Strategy
  - Security Context Strategy (JAAS - veja J530)
- General Context Object Strategies (estados diversos)
  - Context Object Factory Strategy
  - Context Object Auto-Population Strategy

49

Conseqüências

- Melhora reuso e manutenção
  - Componentes são mais genéricos e podem ser usados em clientes que não conhecem a camada Web
- Facilita testes
  - Testes não precisam do container
- Reduz dependência da evolução de interfaces
  - Usando API própria "coarse-grained" evita depender de mudanças de uma API específica
- Reduz performance
  - Devido a transferência de estado entre objetos.

50

Exercícios

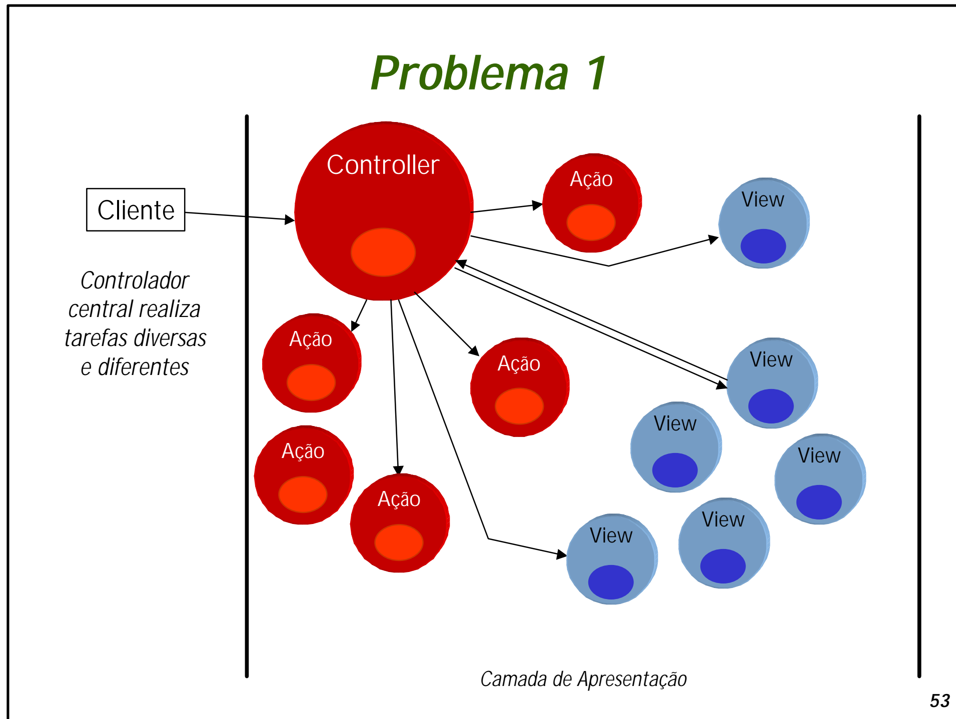
- *1. Analise o código com as diferentes estratégias para Context Object*
- *2. Use um Context Object para guardar o estado de login (nome, senha) utilizado em outras páginas da aplicação*

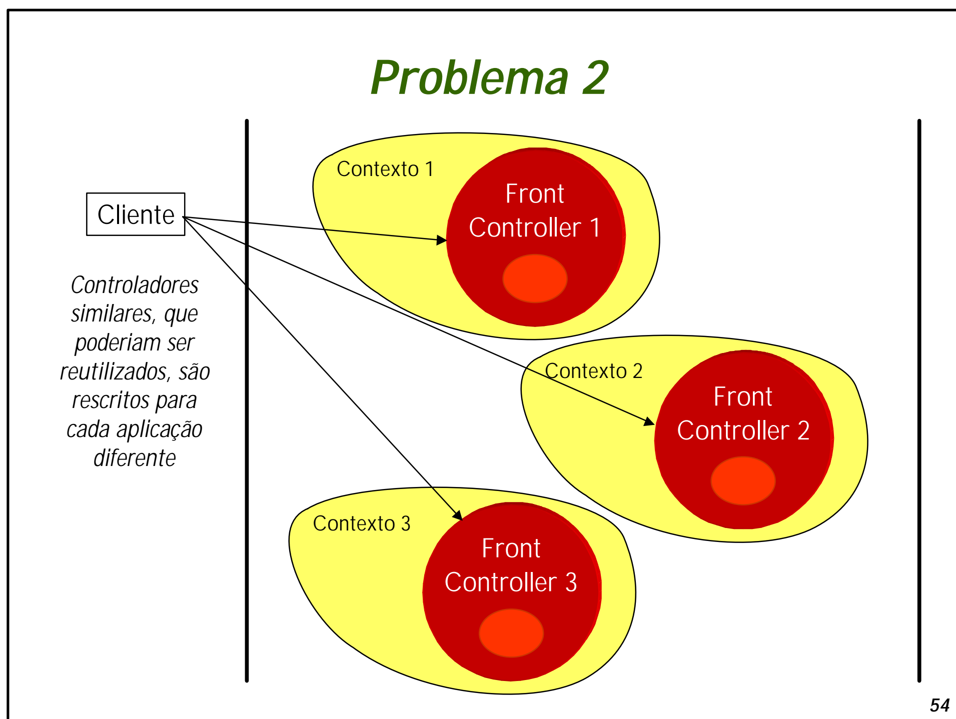
51

4

Application Controller

Objetivo: centralizar operações relacionadas com processamento e despacho de requisições, tais como redirecionamento para comandos (ações) e views para permitir o reuso do Front Controller


Problema 1
Cliente
Controlador central realiza tarefas diversas e diferentes
Controller
Ação
Ação
Ação
Ação
Ação
View
View
View
View
View
View
Camada de Apresentação
53

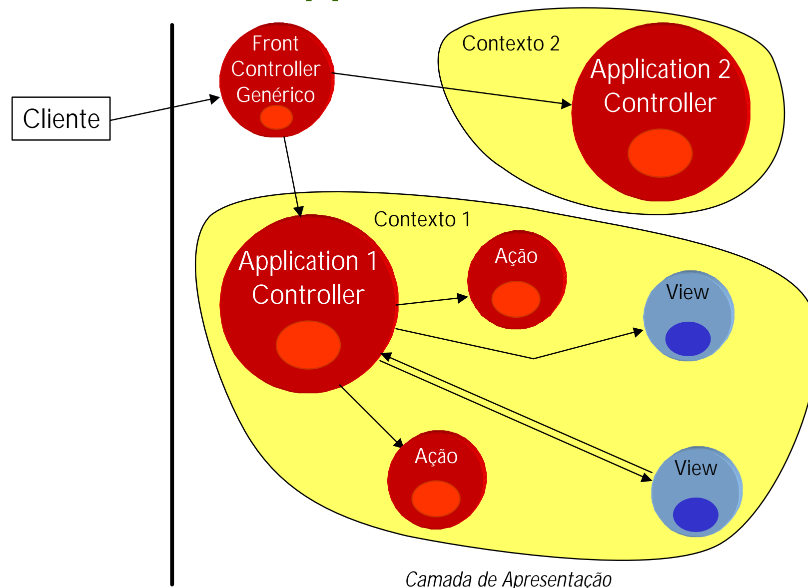

Problema 2
Cliente
Controladores similares, que poderiam ser reutilizados, são rescritos para cada aplicação diferente
Contexto 1
Front Controller 1
Contexto 2
Front Controller 2
Contexto 3
Front Controller 3
54

Problema

- *Front Controller tem que lidar com código de aplicações diferentes, ou*
- *Cada aplicação tem o seu Front Controller que faz basicamente a mesma coisa, com pequenas diferenças*
- *Desejamos centralizar e reutilizar operações padrão de requisição e resposta para todas as aplicações e reutilizar código de repasse para ações e views*
- *Desejamos melhorar modularidade do código, manutenabilidade, etc. facilitando a extensão da aplicação e o reuso do código de manuseio de requisições de forma independente do container*

55

Solução: Application Controller



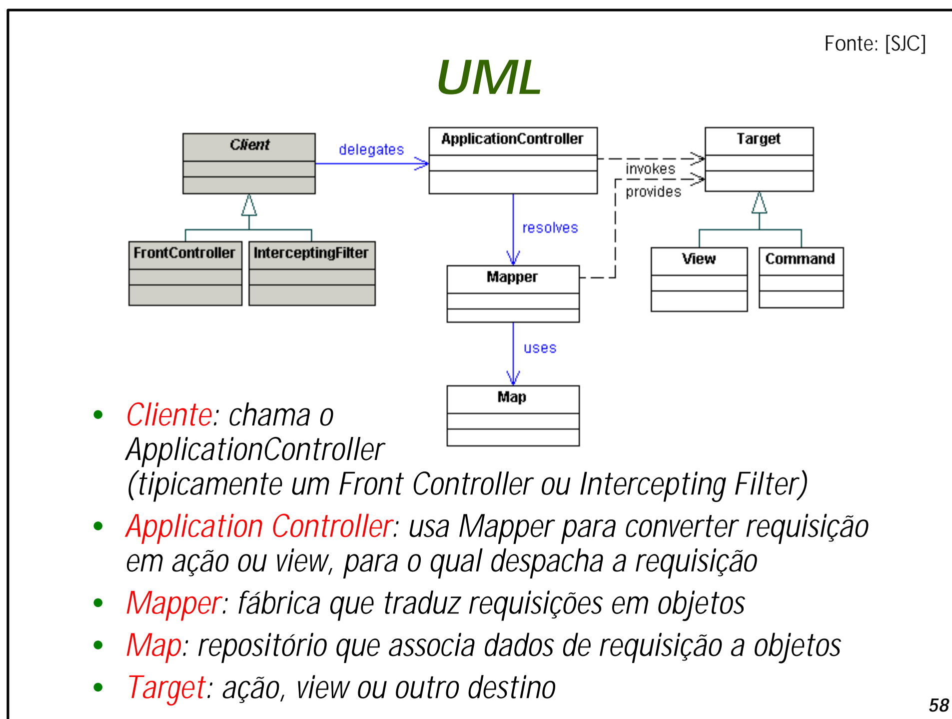
56

© 2003, Helder L. S da Rocha

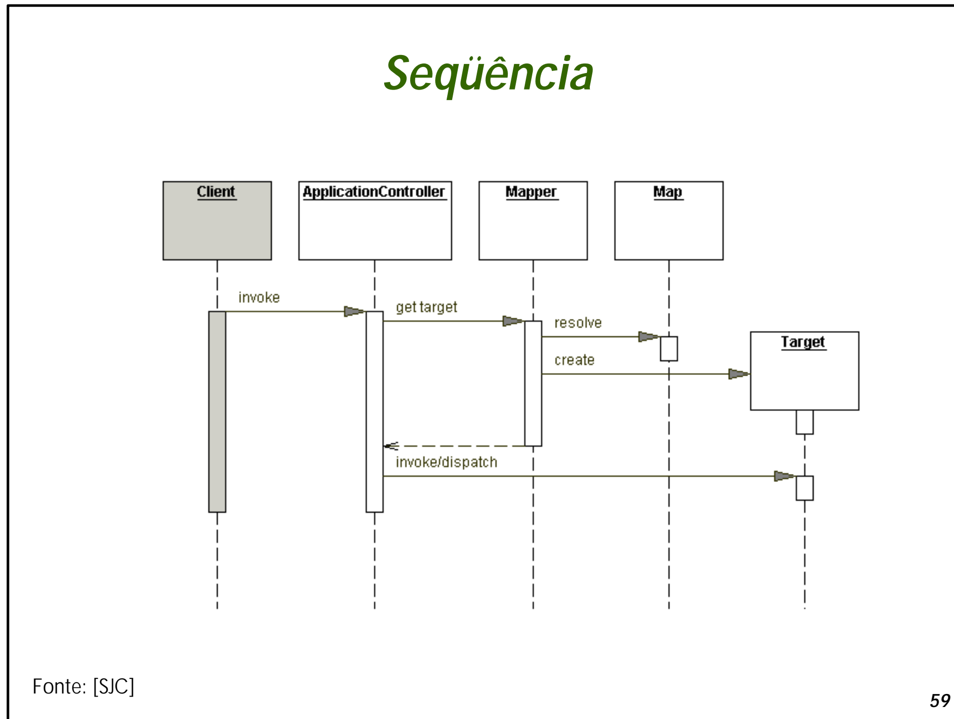
Solução

- *Um Application Controller centraliza as operações específicas de recuperação e chamada de componentes que processam requisições, para cada aplicação.*
- *Permite que se use um Front Controller genérico que seja mapeado a diferentes Application Controllers, por exemplo, um por aplicação (estratégia mais comum)*

57



Seqüência



Fonte: [SJC]

59

Estratégias

- *Command Handler Strategy*
  - *Redireciona para ações*
- *View Handler Strategy*
  - *Redireciona para páginas JSP (ou outro tipo de View)*
- *Transform Handler Strategy*
  - *Redireciona para folhas de estilo XSLT*
- *Navigation and Flow Control Strategy*
- *Message Handling Strategies*
- *Custom SOAP Message Handling Strategy*
- *JAX RPC Message Handling Strategy*

Podem ser combinados (vide Struts)

Veja [Core] para mais detalhes

60

Application Controller no Struts

- No Struts, a classe ActionServlet é um Front Controller que delega responsabilidades para um RequestProcessor, que é um Application Controller
- O RequestProcessor combina as operações de um Command Handler e View Handler
  - Mapeamento dos comandos a classes é feita no arquivo struts-config.xml (é o Command Mapper)
  - Objeto ActionMapping assume o papel de View Mapper
- O ActionServlet pode ser reusado em várias aplicações
  - Associação com RequestProcessor é configurado no web.xml

61

Conseqüências

- Melhora a modularidade
  - Facilita a separação da aplicação em diferentes módulos, que podem ser configurados separadamente e reutilizados
- Melhora o reuso
  - Permite que o FrontController seja compartilhado entre aplicações
- Melhora a extensibilidade
  - Permite incluir novos serviços ou aplicações sem alterar a estrutura geral da aplicação Web

62

Exercício

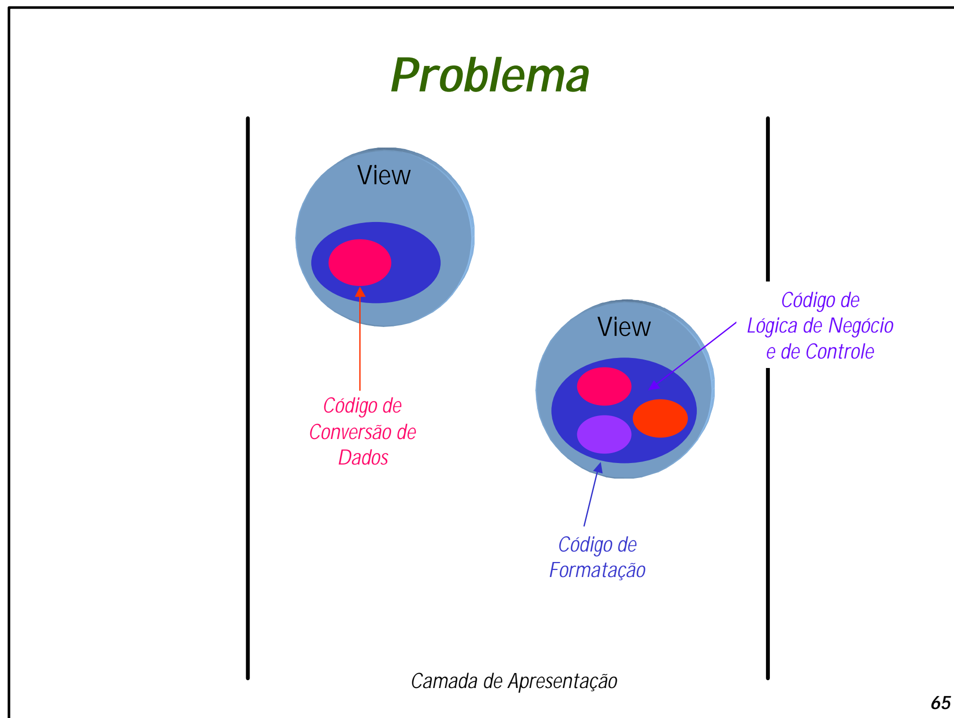
- *1. Analise as estratégias para Application Controller (veja o código)*
- *2. Altere o exercício 2 de Front Controller para que use um Application Controller para encapsular todas as operações de seleção de ações e escolha de Views*

63

5

View Helper

Objetivo: separar código e responsabilidades de formatação da interface do usuário do processamento de dados necessários à construção da View. Tipicamente implementados como JavaBeans e Custom Tags.



Descrição do problema

- *Mudanças na camada de apresentação são comuns*
  - *Alterações da interface do usuário*
- *Se código de apresentação (HTML, JavaScript) estiver misturado com código de processamento e controle (Java) as mudanças são dificultadas*
  - *Menos flexibilidade, menos reuso, menos modularidade e mistura de papéis em um mesmo componente*
- *É preciso identificar as responsabilidades de cada trecho de código e encapsulá-lo em objetos usados pela camada de apresentação*

66

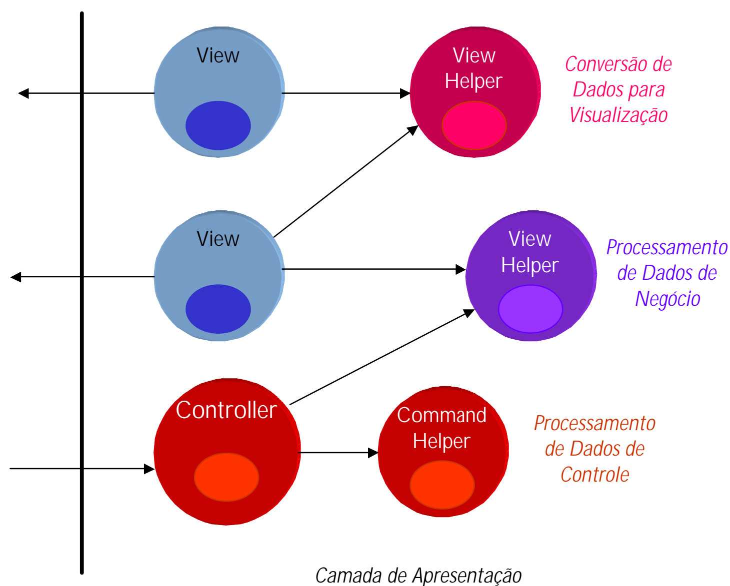
© 2003, Helder L. S da Rocha

O que desejamos

- *Queremos usar views baseados em templates, como JSP*
- *Queremos evitar colocar lógica de programação dentro do view*
- *Queremos separar lógica de programação do view para facilitar divisão de papéis entre desenvolvedores e web designers*

67

Solução: View Helpers



68

Descrição da solução

- *O padrão View Helper recomenda soluções para dividir as responsabilidades do View*
- *Uma View contém código relacionado apenas à formatação*
  - *Responsabilidades de processamento são delegados à classes ajudantes: View Helpers (implementadas como JavaBeans, como Custom Tags ou objetos Java comuns - POJOs)*
  - *Helpers também podem guardar modelo de dados intermediário usado pelo View e servir como adaptadores para dados oriundos da camada de negócios*
  - *Refatoramento pode sugerir separação de determinados trechos de código em objetos Controladores em vez de View Helpers*
- *Há várias estratégias para implementação de Views em associação com View Helpers*

69

Objetivo geral

- *Lógica de negócio tipicamente será mapeada a um modelo de objetos, como um Transfer Object ou Business Object*
- *Lógica de acesso a dados deve ser encapsulada em Data Access Objects (DAO)*
- *Lógica de controle mapeia-se a Front Controller e ações; lógica de formatação deve ficar com os View Helpers*

70

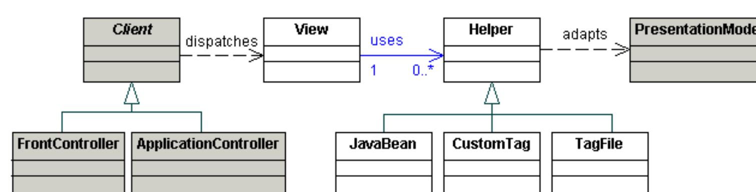
© 2003, Helder L. S da Rocha

Tecnologias

- *Bibliotecas de custom tags*
  - *Struts*
  - *JSTL*
- *Linguagem de expressões do JSTL*
  - *Oferece mais controle*
  - *Inclui linguagem de script na página, porém sem a burocracia do Java*
  - *Simplifica código de scriptlets (mas ainda complica em relação a custom tags)*

71

Estrutura UML

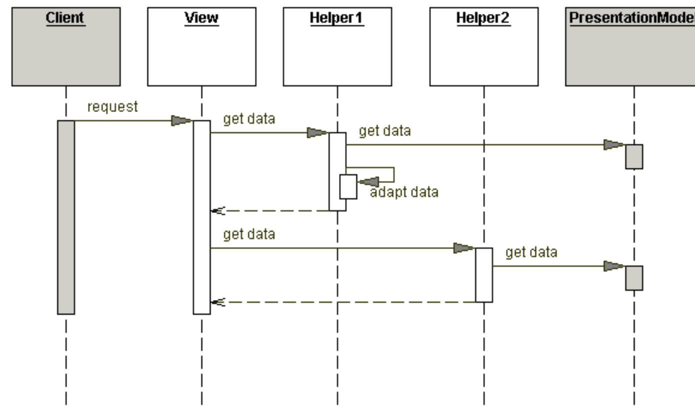


- *Client: repassa para o View*
- *View: representa e mostra informações ao cliente*
- *Helpers: encapsula lógica de processamento para gerar e formatar uma View. Tipicamente adapta um PresentationModel.*
- *PresentationModel: guarda os dados recuperados do serviço de negócios usado para gerar o View.*

Fonte: [SJC] 72

© 2003, Helder L. S da Rocha

Diagramas de Seqüência



Fonte: [SJC]

73

Melhores estratégias de implementação

- Template-Based View Strategy
  - JSP é componente de View
- JavaBean Helper Strategy
  - Helper implementado como JavaBean
- Custom Tag Helper Strategy
  - Mais complexo que JavaBean Helper
  - Separação de papéis maior (isola a complexidade)
  - Maior índice de reuso (pode-se usar custom tags existentes)
  - Evitar usar Custom Tags como linguagem!
- Business Delegate as Helper Strategy
  - Papéis de View Helper e Business Delegate podem ser combinados para acesso à camada de negócio
  - Pode misturar papéis J2EE

74

Conseqüências

- *Melhora particionamento da aplicação*
  - *Facilita o reuso*
  - *Facilita a manutenção*
  - *Facilita a realização de testes funcionais, de unidade e de integração*
- *Melhora separação de papéis J2EE*
  - *Reduz a complexidade para todos os participantes: Web Designer não precisa ver Java e Programador Java não precisa ver JavaScript e HTML*

75

Exercícios

- *1. Analise a implementação das estratégias de ViewHelper*
- *2. Refatore a aplicação em preslayer/vh/ para que utilize ViewHelper. Use JavaBean Helper Strategy:*
  - *a) Identifique código de conversão de formatos, código de negócio e código de controle (se houver)*
  - *b) Construa um Helper para cada responsabilidade encontrada*
- *3. Use Custom Tag Helper Strategy para encapsular a lógica de repetição*
  - *Use tags do Struts <logic:iterate> ou JSTL <c:forEach>*

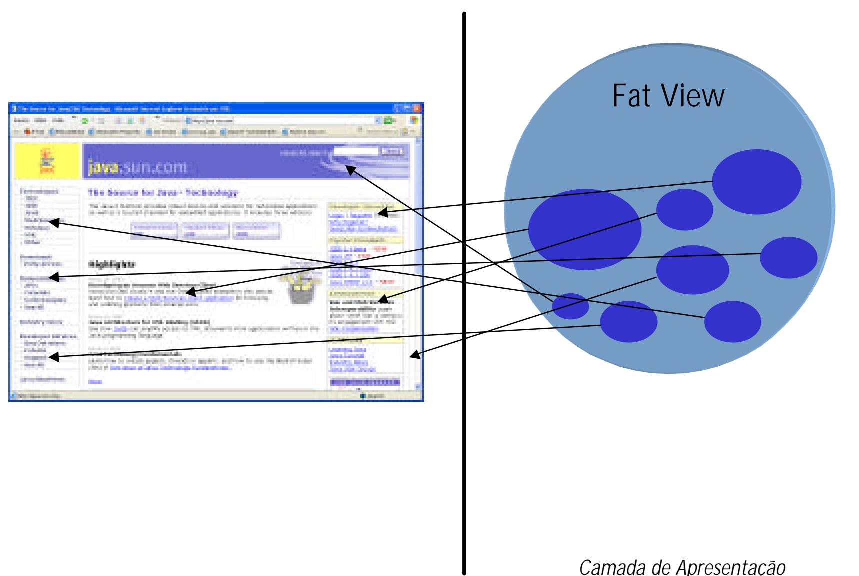
76

6

Composite View

Objetivo: criar um componente de View a partir de Views menores para dividir as responsabilidades, simplificar a construção da interface e permitir o reuso de componentes da View.

Problema



Camada de Apresentação

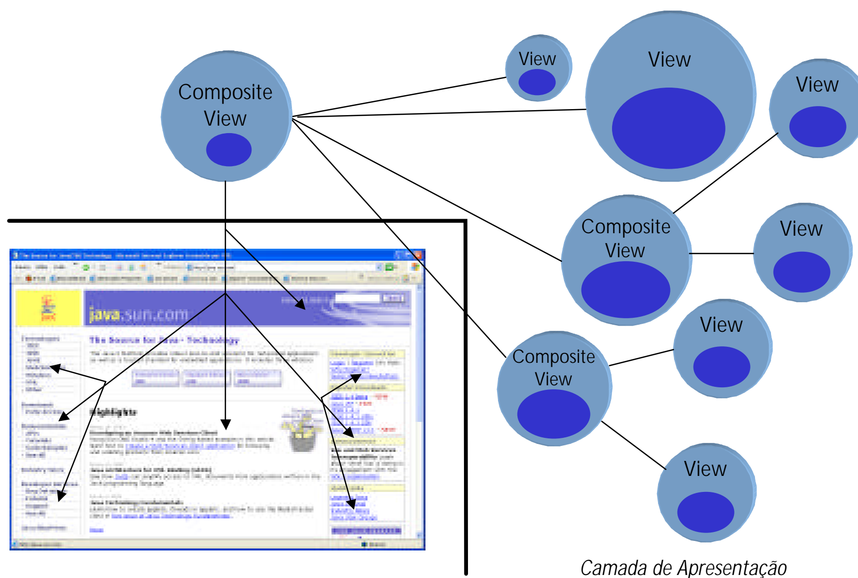
78

Descrição do problema

- *Páginas Web sofisticadas dividem seu conteúdo em várias partes, que tem função e tempo de vida diferentes*
  - *Podem ser manipuladas por pessoas diferentes já que cada seção tem finalidade, escopo e duração diferentes*
- *Se a página for gerada a partir de uma única View que concentra todo o código, a atualização da página é dificultada*
  - *É difícil identificar cada parte dentro de um documento*
  - *Há risco, quando se atualiza uma seção, de se invadir outra seção e afetar os dados*

79

Solução: Composite View



80

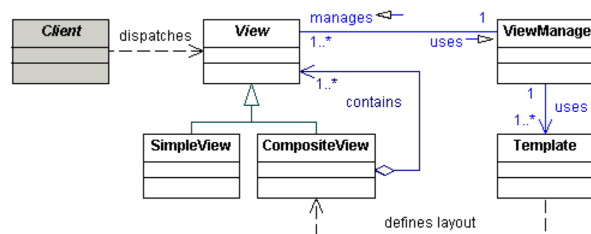
Descrição da solução

- *Usar views que consistem de composições de views menores*
  - *Os componentes do template podem ser incluídos dinamicamente e gerenciados separadamente*
  - *O layout geral da página pode ser manipulado independente do conteúdo*
  - *Componentes podem conter views ou coleções de views*

81

Fonte: [SJC]

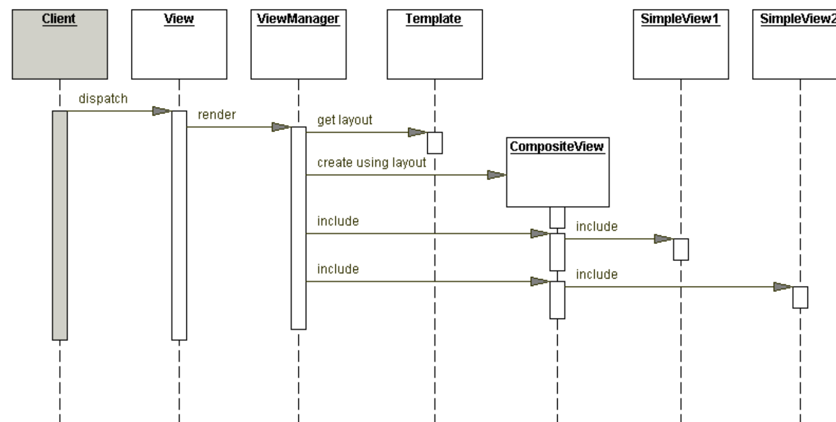
Estrutura UML



- *Cliente: repassa a requisição para a View*
- *View: representa o display*
- *SimpleView: representa uma porção atômica da composição*
- *CompositeView: é composta de múltiplas Views, que pode ser outra CompositeView ou SimpleView*
- *Template: representa o layout da View*
- *ViewManager: determina o layout dos Views. Ex: <jsp:include>*

82

Diagramas de Seqüência



Fonte: [SJC]

83

Melhores estratégias de implementação

- *JavaBean View Management Strategy**
  - *Utiliza JavaBeans para incluir outros views na página*
  - *Mais simples que solução com Custom Tags*
- *Early Binding Resource Strategy (Translation-time)*
  - *Usa diretivas padrão do JSP:* `<%@ include %>`
  - *Carga é feita em tempo de compilação: alterações só são vistas quando página for recompilada*
- *Late Binding Resource Strategy (Run-time)*
  - *Usa ação padrão do JSP: <jsp:include>*
  - *Carga é feita quando página é carregada: alterações são visíveis a cada atualização*
- *Custom Tag View Management Strategy*
  - *Utiliza Custom Tags: solução mais elegante e reutilizável*

** Veja exemplos de código de [Core]*

84

Early Binding Resource Strategy

- *Mais eficiente: fragmentos são incluídos em único servlet*
- *Indicada quando estrutura não muda com freqüência (conteúdo pode mudar)*
  - *Menus, Logotipos e Avisos de copyright*
  - *Telas com miniformulários de busca*
- *Implementada com <%@ include file="fragmento" %>*

```
<!-- Menu superior -->
<table>
<tr><td><%@ include file="menu.jsp" %></td></tr>
</table>
<!-- Fim do menu superior -->
```

Fragmento menu.jsp

```
<a href="link1">Item 1</a></td>
<td><a href="link2">Item 2</a></td>
<a href="link3">Item 3</a>
```

Se tela incluida contiver novos fragmentos, eles serão processados recursivamente

85

Late Binding Resource Strategy

- *Mais lento: fragmentos não são incluídos no servlet mas carregados no momento da requisição*
- *Indicada para blocos cuja estrutura muda com freqüência*
  - *Bloco central ou notícias de um portal*
- *Implementada com <jsp:include page="fragmento"/>*
- *Pode-se passar parâmetros em tempo de execução usando <jsp:param> no seu interior*

```
<!-- Texto principal -->
<table>
<tr><td>
<jsp:include page="texto.jsp">
   <jsp:param name="data" value="<%=new Date() %>">
</jsp:include>
</td></tr> </table>
<!-- Fim do texto principal -->
```

86

Conseqüências

- *Promove design modular*
  - *Permite maior reuso e reduz duplicação*
- *Melhora flexibilidade*
  - *Suporta inclusão de dados com base em decisões de tempo de execução*
- *Melhora facilidade de manutenção e gerenciamento*
  - *Separação da página em pedaços menores permite que sejam modificados e mantidos separadamente*
- *Reduz facilidade de gerenciamento*
  - *Possibilidade de erros na apresentação devido à composição incorreta das partes*
- *Impacto na performance*
  - *Inclusões dinâmicas fazem página demorar mais para ser processada*

87

Exercícios

- *1. Analise a implementação das estratégias de Composite View*
- *2. Refatore a aplicação em preslayer/cv/ para que utilize CompositeView (os blocos estão identificados com comentários no HTML em messages.jsp). Escolha as melhores estratégias entre Translation-time e Run-time Stategies*
  - *a) Qual a melhor estratégia para o navbar (raramente muda)?*
  - *b) E para o bloco principal?*
- *3. Implemente o menu usando Custom Tag View Management Strategy*
- *4. Implemente o bloco de mensagens usando JavaBean View Management Strategy (já está implementado)*

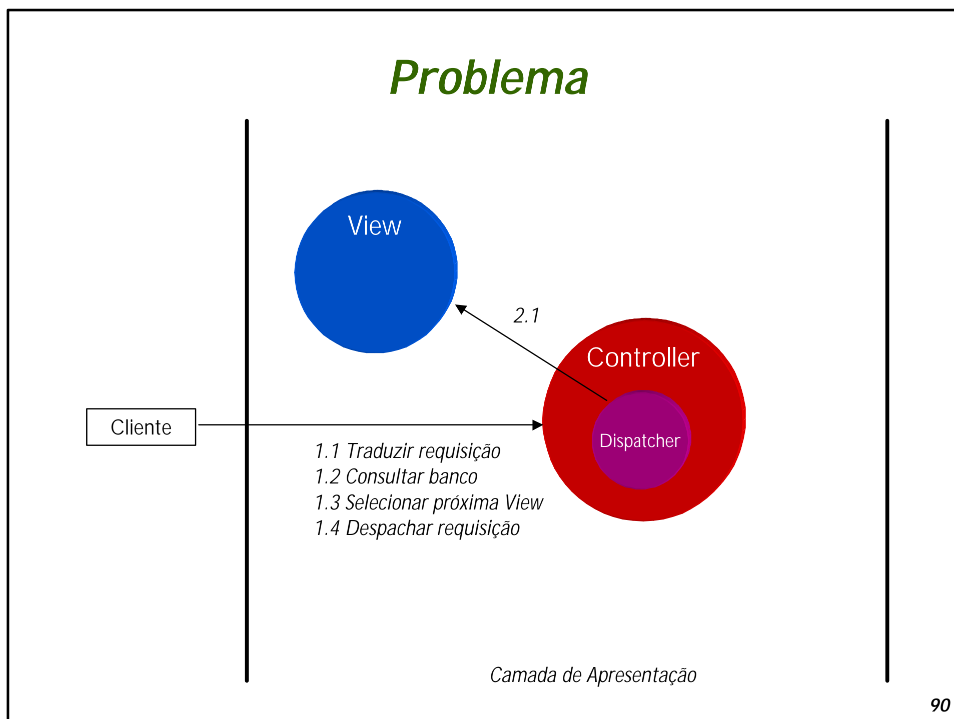
88

7

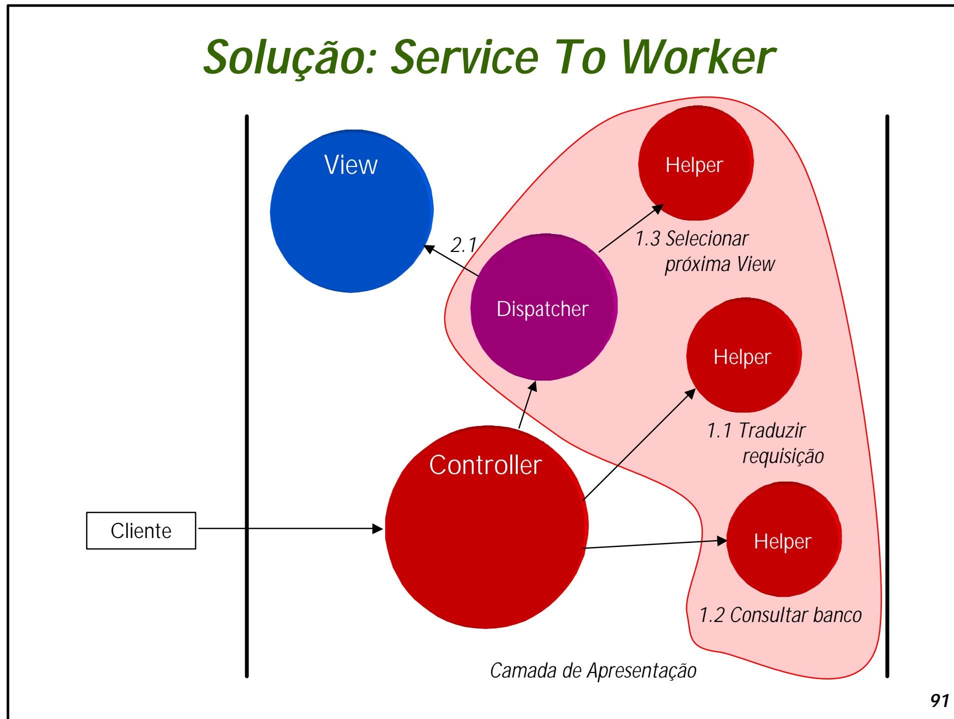
Service To Worker

Objetivo: combinar padrões de apresentação e encapsular lógica de navegação, que consiste em escolher uma View e despachar a requisição para ela. Service To Worker realiza mais processamento antes de despachar a requisição.

Problema



90

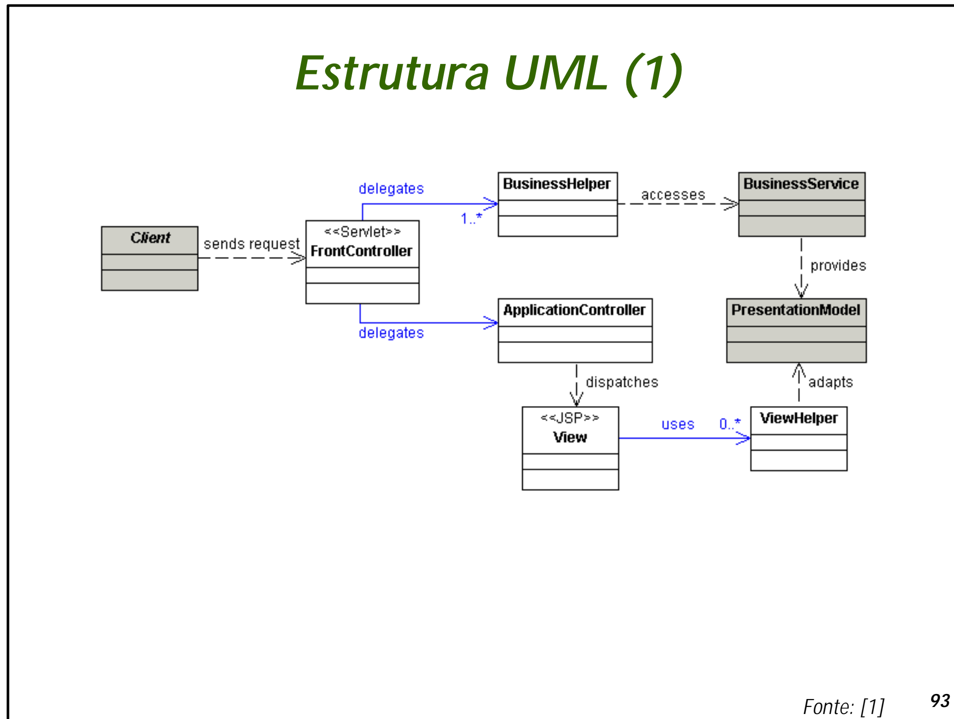


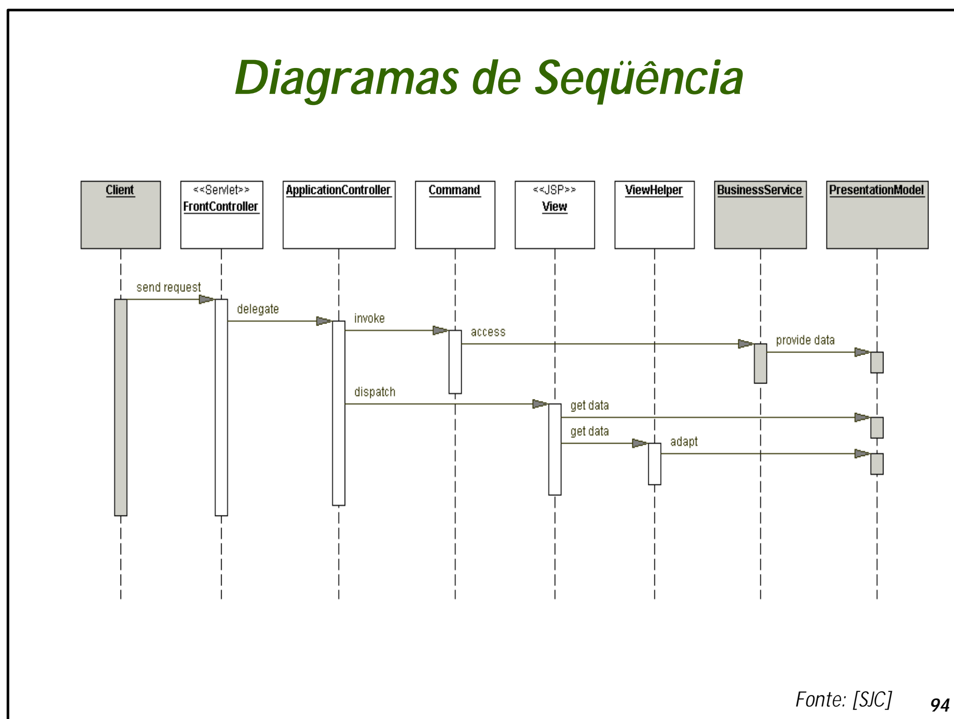
Descrição

- *Service To Worker combina Front Controller e View Helper com o objetivo de separar a lógica que tem responsabilidades diferentes*
  - *Maior parte das responsabilidades estão acumuladas entre os controladores e Dispatcher*
  - *Recuperação do conteúdo necessário para compor a View é obtido antes de despachar a requisição*

92

© 2003, Helder L. S da Rocha


Estrutura UML (1)
Client
sends request
<<Servlet>>
FrontController
delegates
1..*
BusinessHelper
accesses
BusinessService
provides
delegates
ApplicationController
PresentationModel
dispatches
adapts
<<JSP>>
View
uses
0..*
ViewHelper
Fonte: [1]
93

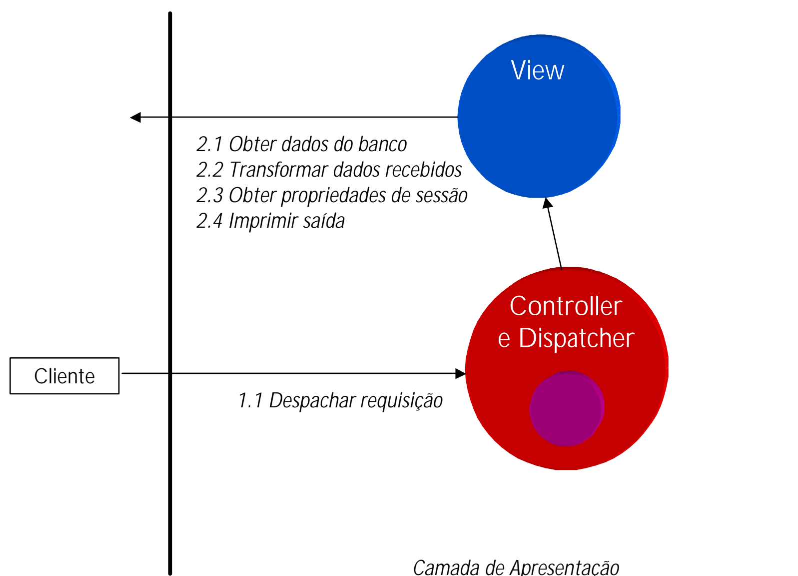

Diagramas de Seqüência
Client
<<Servlet>>
FrontController
ApplicationController
Command
<<JSP>>
View
ViewHelper
BusinessService
PresentationModel
send request
delegate
invoke
access
provide data
dispatch
get data
get data
adapt
Fonte: [SJC]
94

8

Dispatcher View

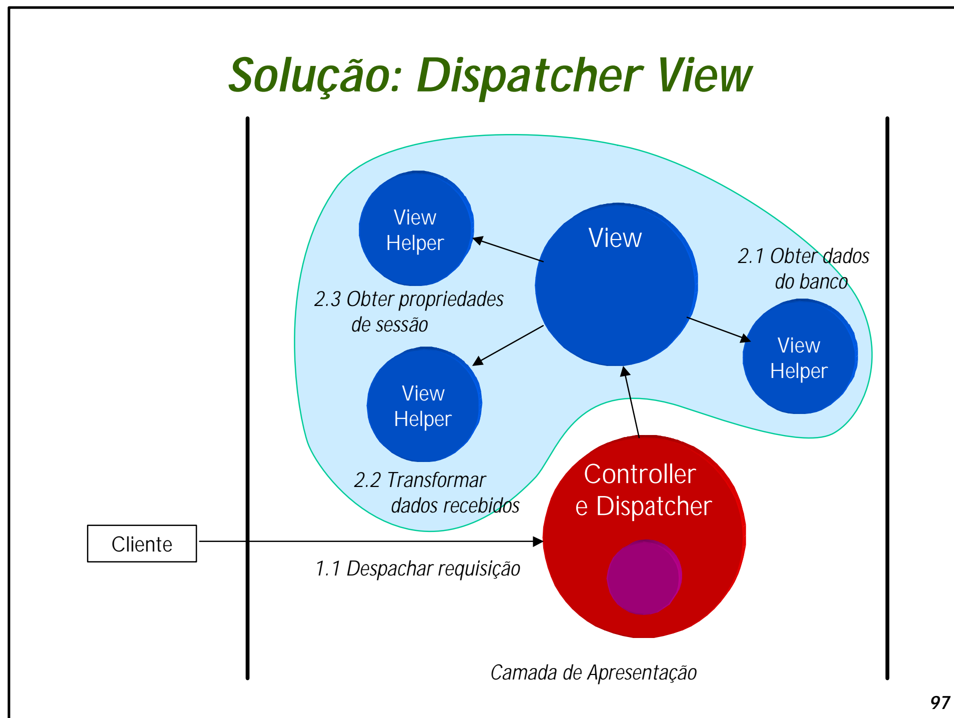
Objetivo: combinar padrões de apresentação e encapsular lógica de navegação, que consiste em escolher uma View e despachar a requisição para ela. Dispatcher View realiza mais processamento depois de despachar a requisição.

Problema



96

Solução: Dispatcher View



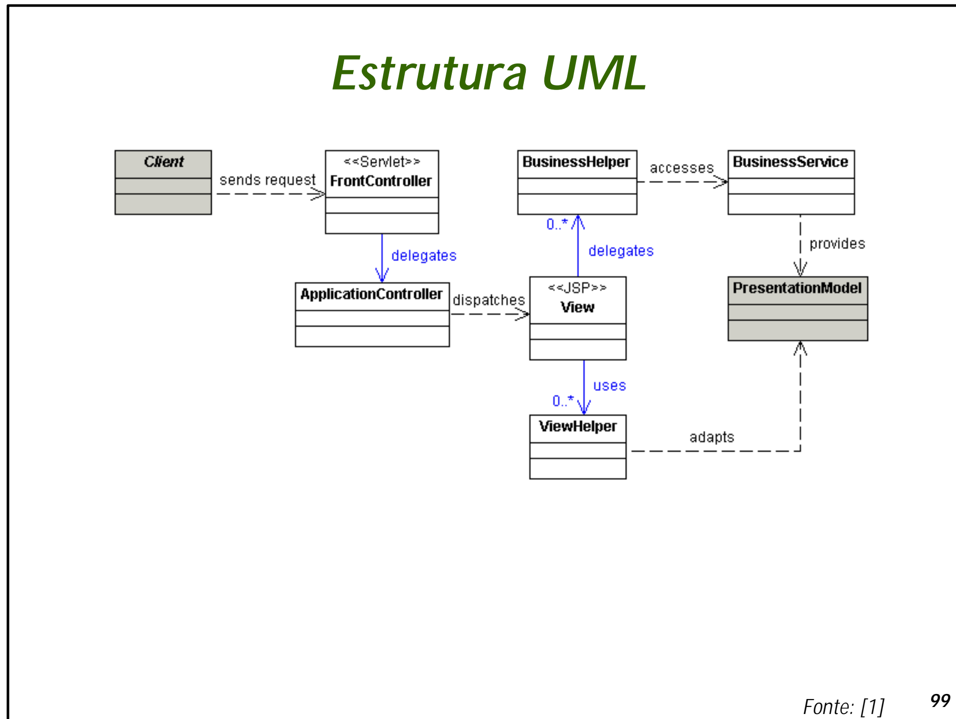
97

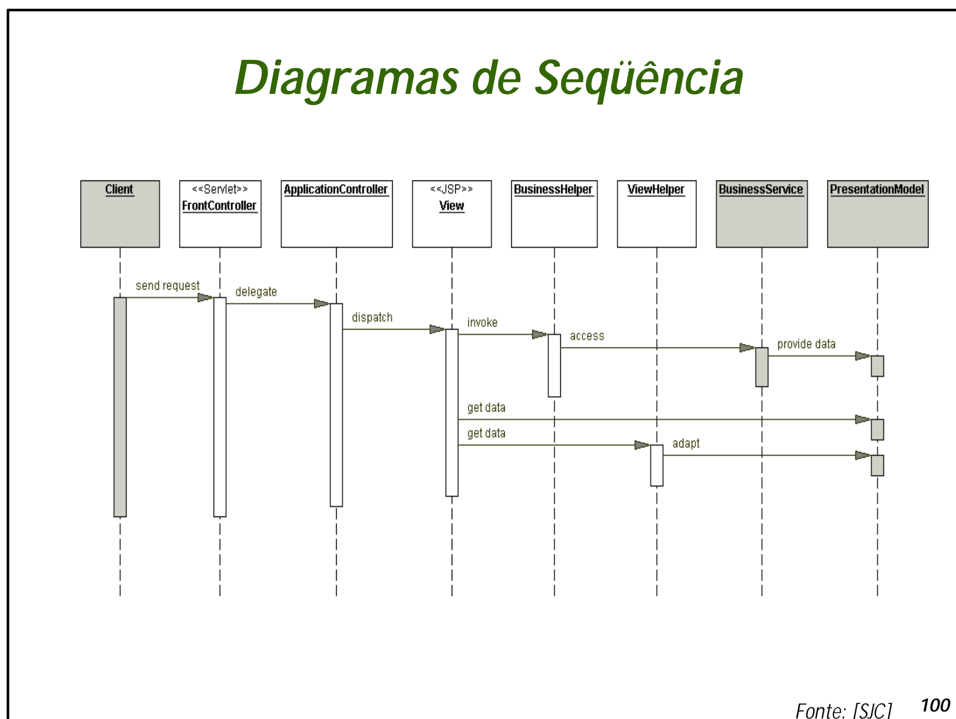
Descrição

- Dispatcher View combina Front Controller e View Helper com o objetivo de separar a lógica que tem responsabilidades diferentes
  - Maior parte das responsabilidades estão acumuladas entre o Dispatcher e View
  - Recuperação do conteúdo necessário para compor a View é obtido após despachar a requisição

98

© 2003, Helder L. S da Rocha


Estrutura UML
Client
sends request
<<Servlet>>
FrontController
delegates
ApplicationController
dispatches
BusinessHelper
accesses
BusinessService
0..*
delegates
<<JSP>>
View
provides
PresentationModel
uses
0..*
ViewHelper
adapts
Fonte: [1]
99


Diagramas de Seqüência
Client
<<Servlet>>
FrontController
ApplicationController
<<JSP>>
View
BusinessHelper
ViewHelper
BusinessService
PresentationModel
send request
delegate
dispatch
invoke
access
provide data
get data
get data
adapt
Fonte: [SJC]
100

Exercícios

- *1. Analise a implementação das estratégias de Dispatcher View e Service to Worker*
- *2. Discussão ("acadêmica"): você acha que esses dois últimos padrões devem ter o status de padrões ou são meramente "estratégias" de Front Controller?*

101

Fontes

[SJC] SJC Sun Java Center J2EE Patterns Catalog. http://developer.java.sun.com/developer/restricted/patterns/J2EEPatternsAtAGlance.html.

[Blueprints] J2EE Blueprints patterns Catalog. http://java.sun.com/blueprints/patterns/catalog.htm. Contém padrões extras usados na aplicação Pet Store.

[Core] Deepak Alur, John Crupi, Dan Malks. Core J2EE Patterns: Best Practices and Design Strategies, 2nd Edition. Prentice-Hall, 2001. http://www.corej2eepatterns.com

102

Curso J931: J2EE Design Patterns

Versão 1.0

www.argonavis.com.br

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