Presentation Layer Patterns

Lecture 03

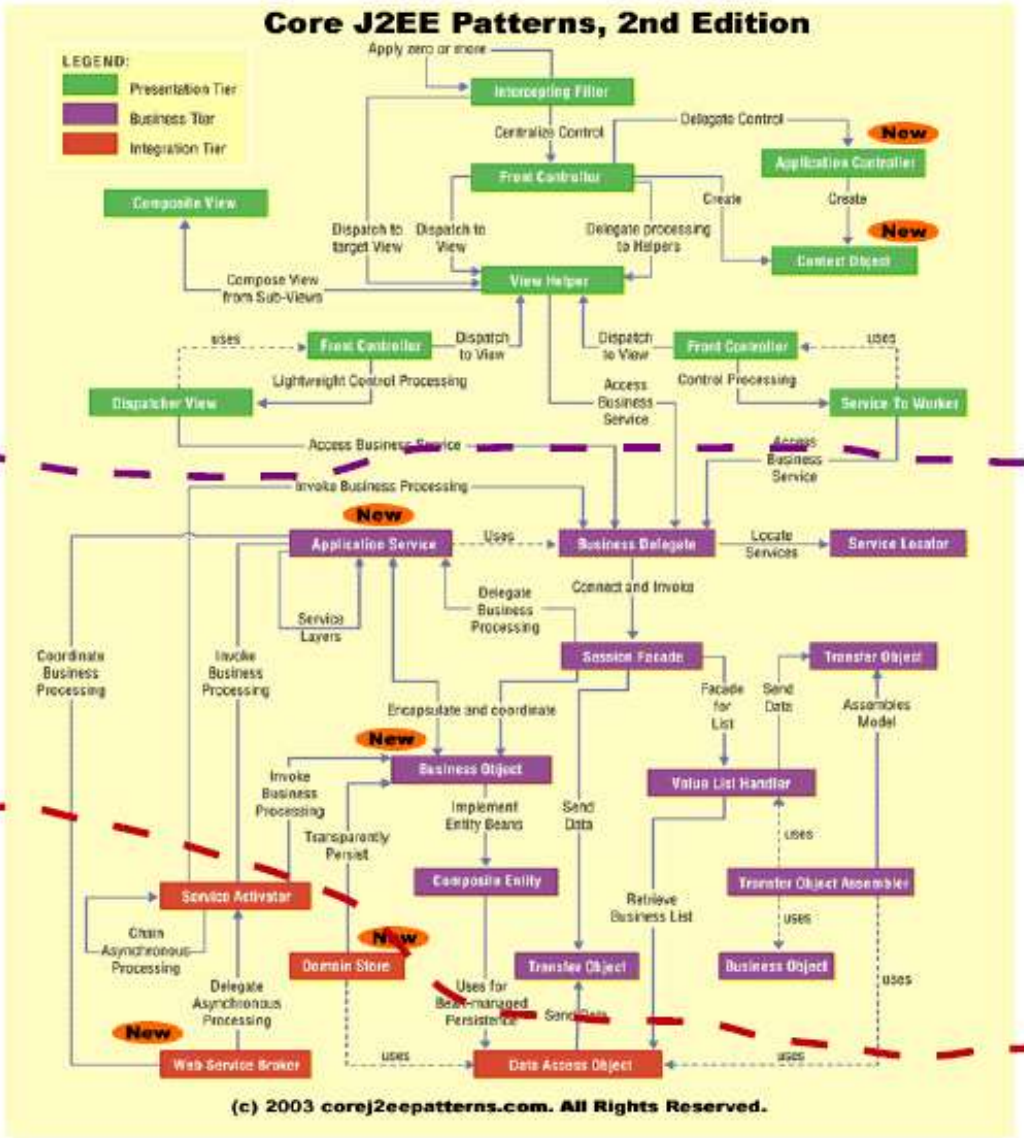
by Udara Samaratunge

J2EE Architecture Blueprint

Presentation Tier

Business Tier

Integration Tier

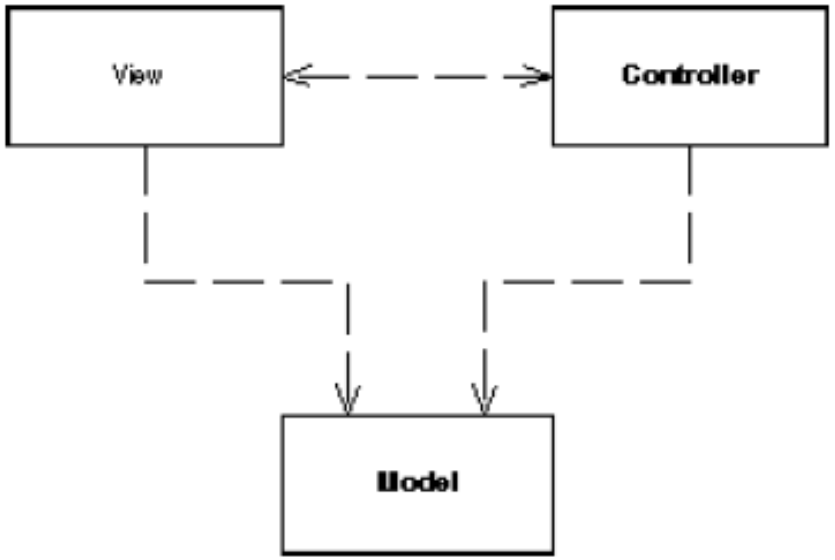


Reference: Core J2EE Patterns

Web Presentation Design Patterns

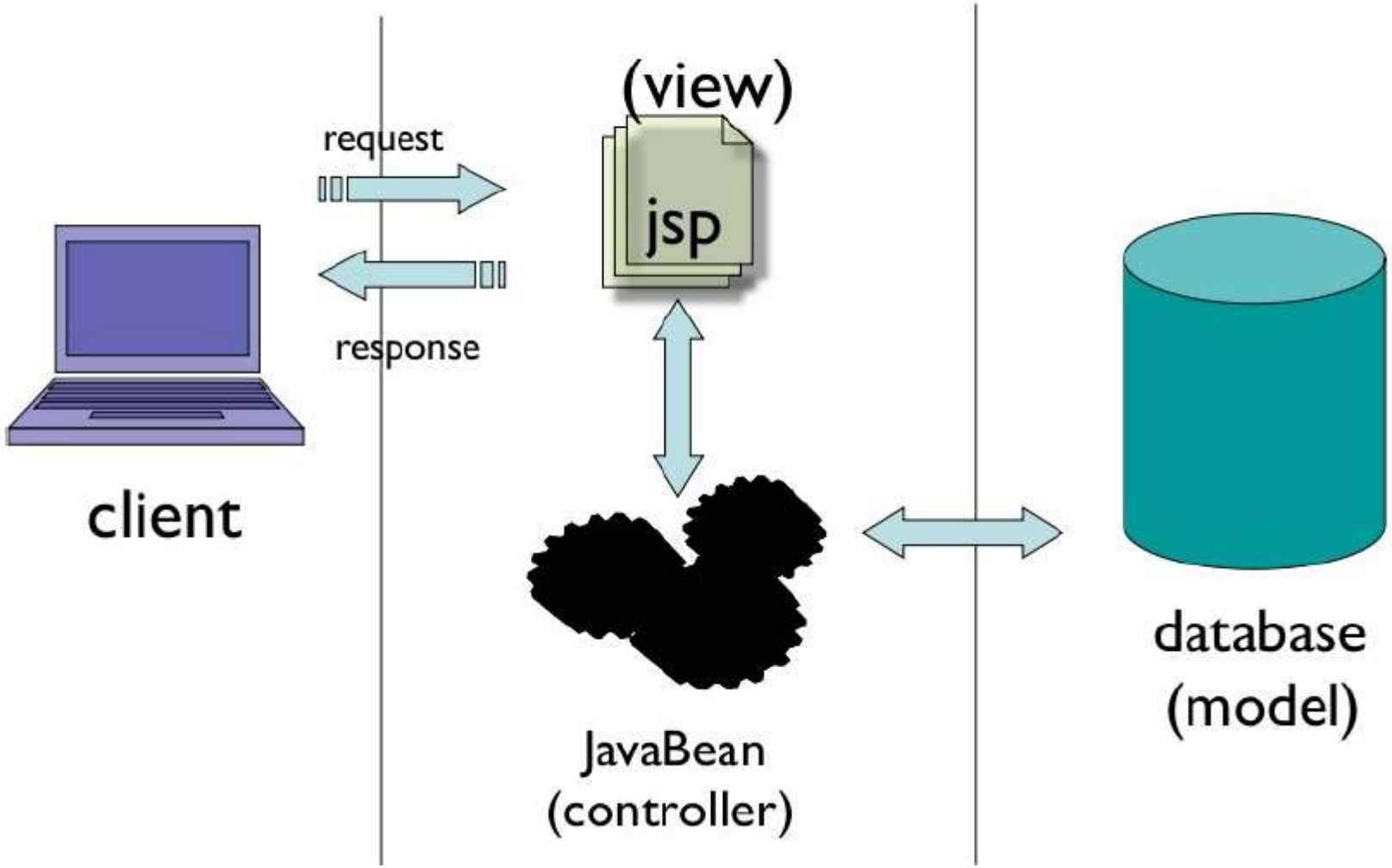
- Model View Controller (MVC)
- Intercepting Filter Pattern
- Front Controller
- View Helper
- Composite View
- Dispatcher View
- Service to Worker

Model-View Controller

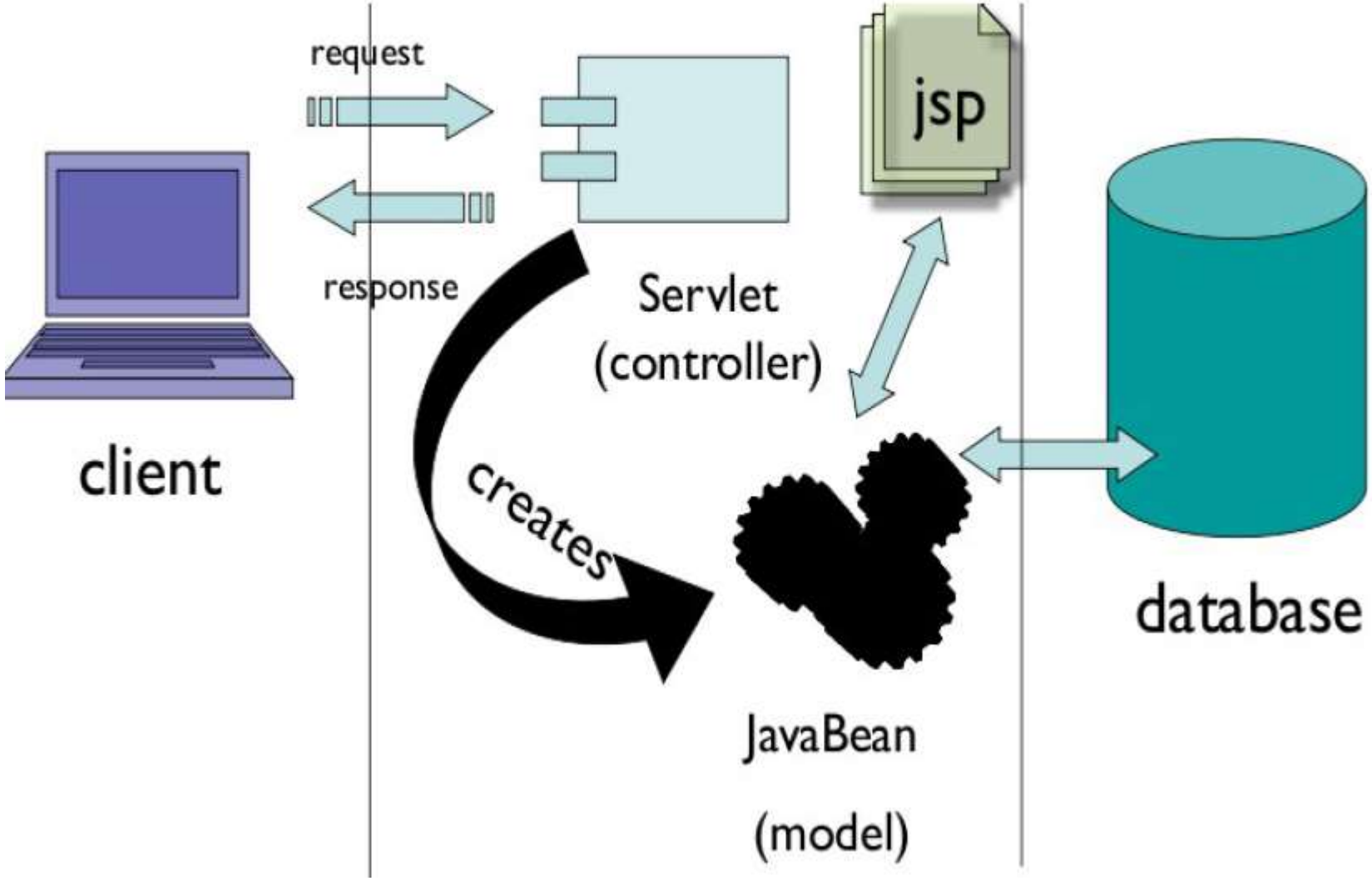


Splits user interface interaction into three distinct roles

Model 1 Architecture



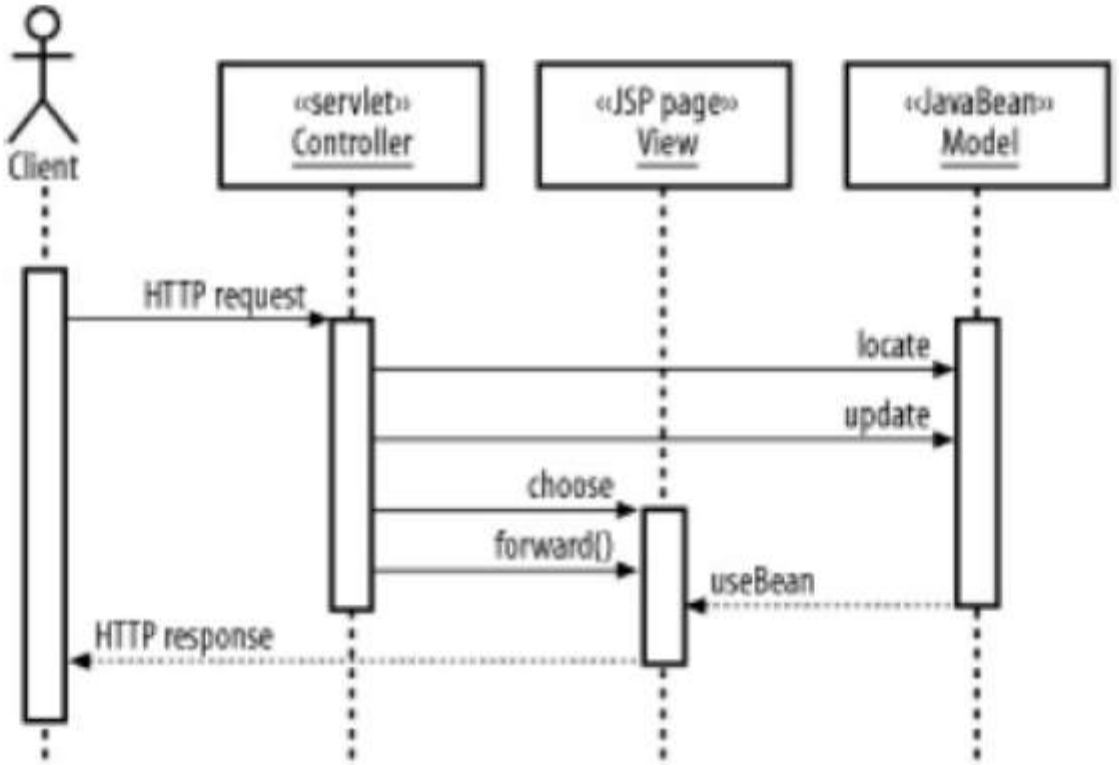
Model 2 Architecture



Model View Controller - (MVC)

- The MVC pattern provides the basic structure for a web application
- In J2EE, the following components define the structure of the web application
  - The JavaBeans provides the *Model*
  - The JSPs provide the *View*
  - The controller Servlet provides the *Controller*

Model View Controller - (MVC)



MVC Interaction in J2EE (Reference: *J2ee Design Patterns*)

Model View Controller - (MVC)

- **Model:**

  Holds the *data*, *state* and *application logic*

  Can send notification of *state* changes to observers

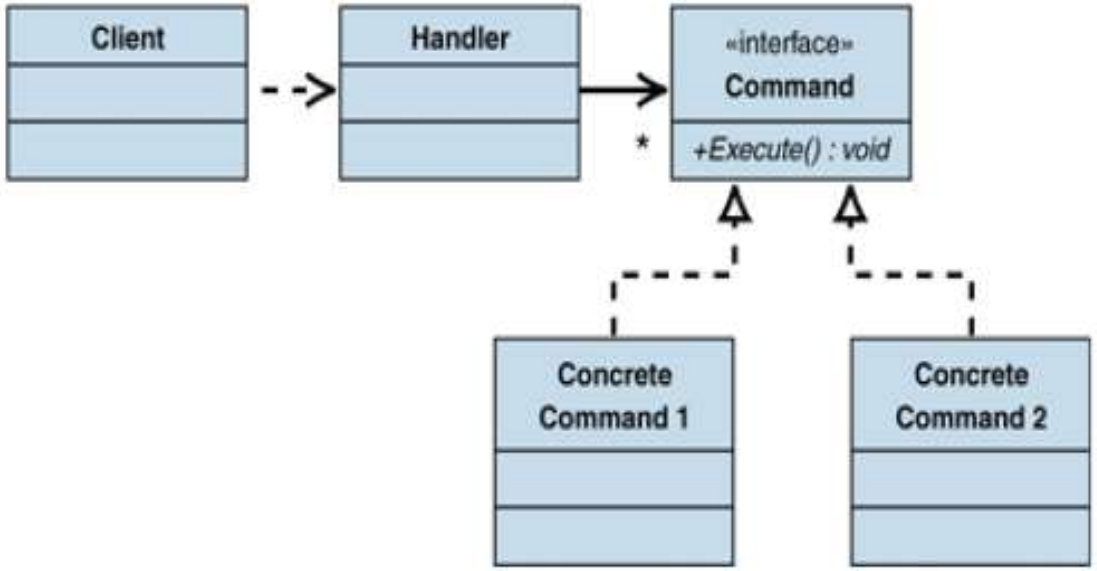
- **View:**

  Gives you the presentation of the model. This is *stateless*

- **Controller:**

  Takes user input and figure out what it means to the model

Front Controller



A controller that handles all the requests for the web application

Exercise 01 - In class Activity

- Assume that you have a **light** in both **Living Room** and **Kitchen** you use a **Remote Controller** to switch **on** and **off** lights. Command and Light interfaces are given with the Test class including final Output. Implement the **OnCommand, OffCommand**, **LivingRoomLight, KitchenLight**, and **RemoteController** classes.

```java
public interface Light {

    public void on();

    public void off();
}
```

```java
public interface Command {

    public void execute();
}
```

```java
package design.pattern.command;

public class Test {

    public static void main(String[] args) {

        Light livingRoomLight = new LivingRoomLight();
        Light kitchenLight = new KitchenLight();

        RemoteController remoteController = new RemoteController();

        Command lightOnCommand = new LightOnCommand(livingRoomLight);
        Command lightOffCommand = new LightOffCommand(livingRoomLight);
        remoteController.setCommand(lightOnCommand, lightOffCommand);
        remoteController.onButtonWasPushed();
        remoteController.offButtonWasPushed();

        Command lightOnCommand1 = new LightOnCommand(kitchenLight);
        Command lightOffCommand1 = new LightOffCommand(kitchenLight);
        remoteController.setCommand(lightOnCommand1, lightOffCommand1);
        remoteController.onButtonWasPushed();
        remoteController.offButtonWasPushed();
    }
}
```

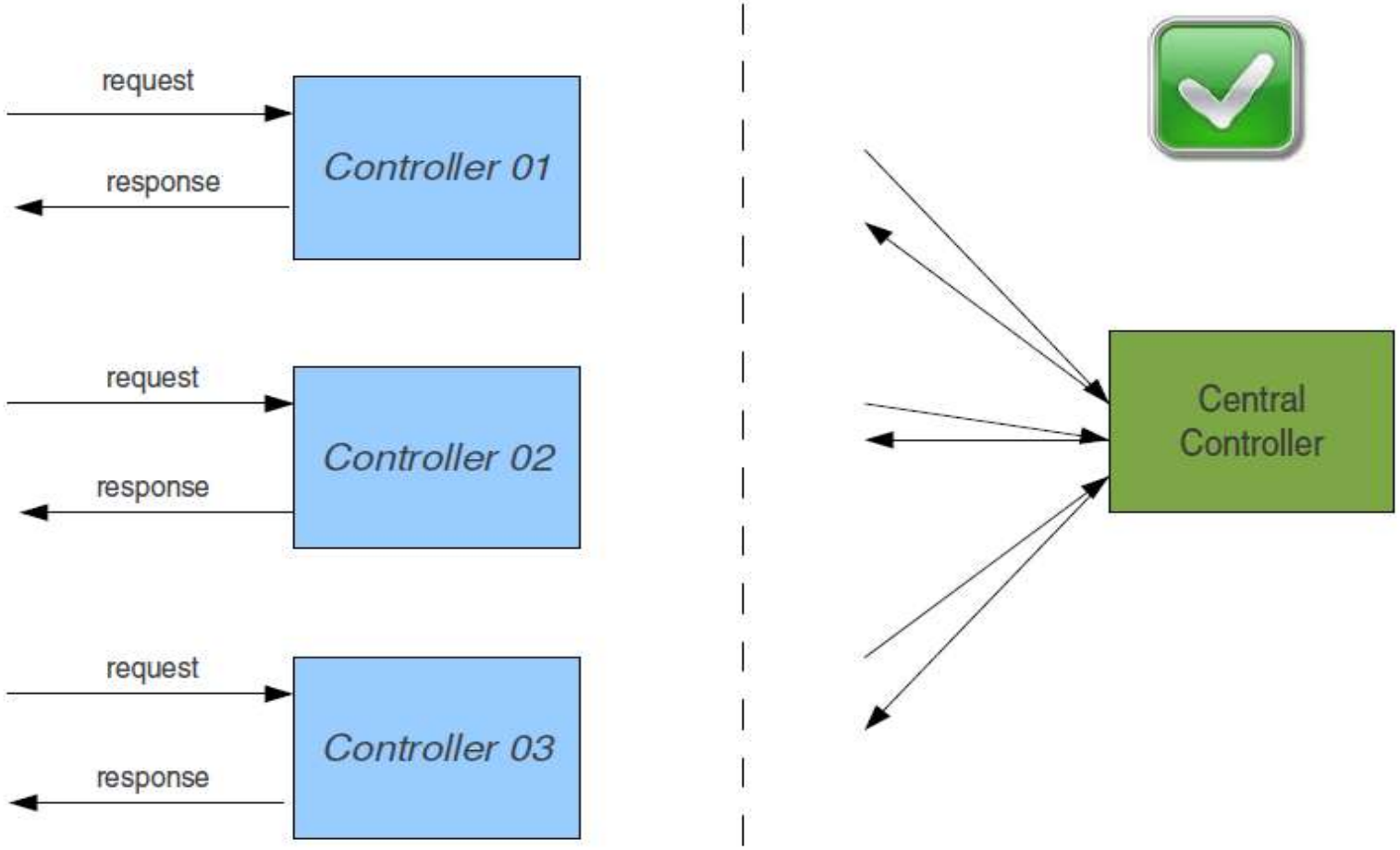
```
<terminated> Test (7) [Java Application] (
Switch on() Living Room Light
Switch off() Living Room Light
Swich on() Kitchen Light
Swich off() Kitchen Light
```

Front Controller

all incoming requests from users are first handled by the Front Controller before being passed on to other parts of the application.

- Front Controller is the central controlling point of a web application
- In a complex web site, there are many tasks need to be followed when handling a request. For example:
  - Authentication/ Authorization
  - Delegating to business Processors
  - Providing different view to the application, etc
- If we duplicate the input controller behavior (entry behavior) to all these tasks, the **behavior can be duplicated** across

Front Controller

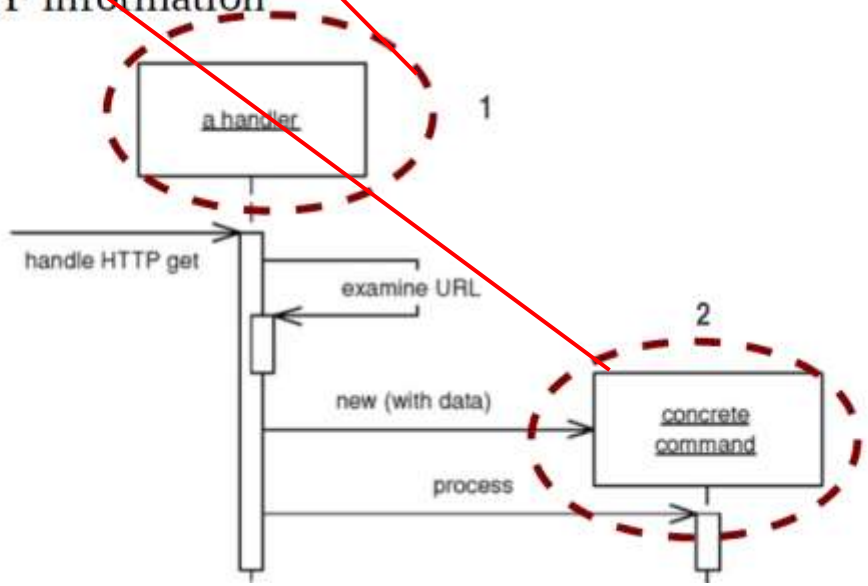


Front Controller - Benefits

- Provides a central entry point that controls and manages the web request
- Centralizing control in the controller and reducing business logic in the view promotes code reuse across requests
- Coordinates the request dispatching – Dispatchers are responsible for view management navigation

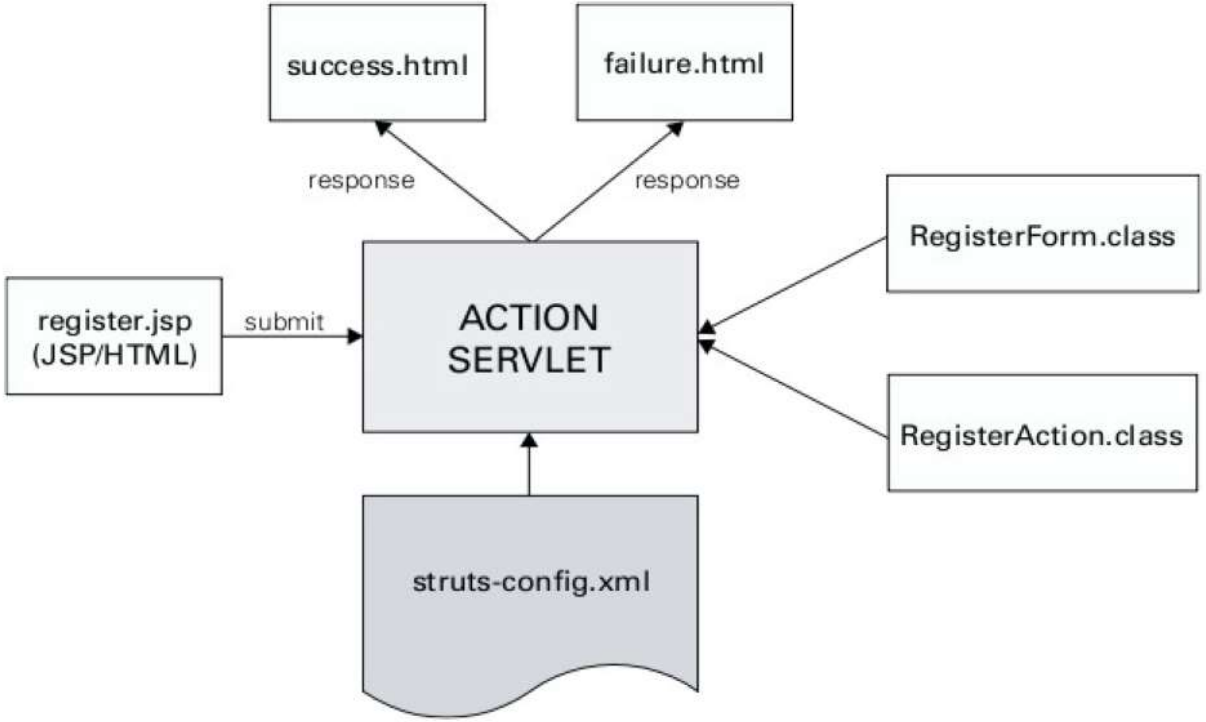
Front Controller

- The *Web Handler* is fairly a simple class that does nothing other than deciding which command to run
- The *commands* are also classes that are often passed with HTTP information

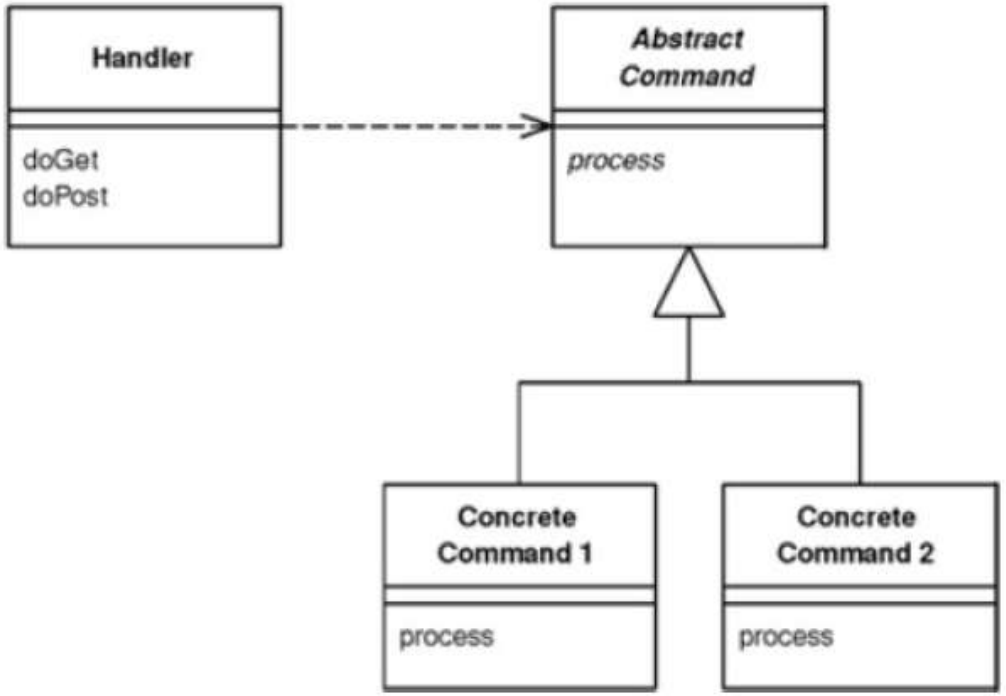


Front Controller – In Struts

- In Struts, *ActionServlet* is the Front Controller



The Front Controller consolidates all request handling by channeling requests through a single handler object.



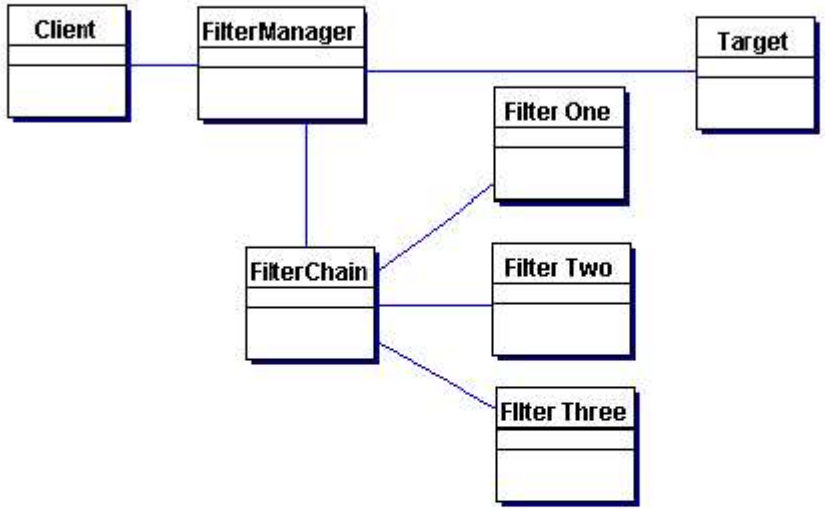
Front Controller

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {
    FrontCommand command = getCommand(request);
    command.init(getServletContext(), request, response);
    command.process();
}
private FrontCommand getCommand(HttpServletRequest request) {
    try {
        return (FrontCommand) getCommandClass(request).newInstance();
    } catch (Exception e) {
        throw new ApplicationException(e);
    }
}
private Class getCommandClass(HttpServletRequest request) {
    Class result;
    final String commandClassName =
        "frontController." + (String) request.getParameter("command") + "Command";
    try {
        result = Class.forName(commandClassName);
    } catch (ClassNotFoundException e) {
        result = UnknownCommand.class;
    }
    return result;
}
```

main entry point for GET requests

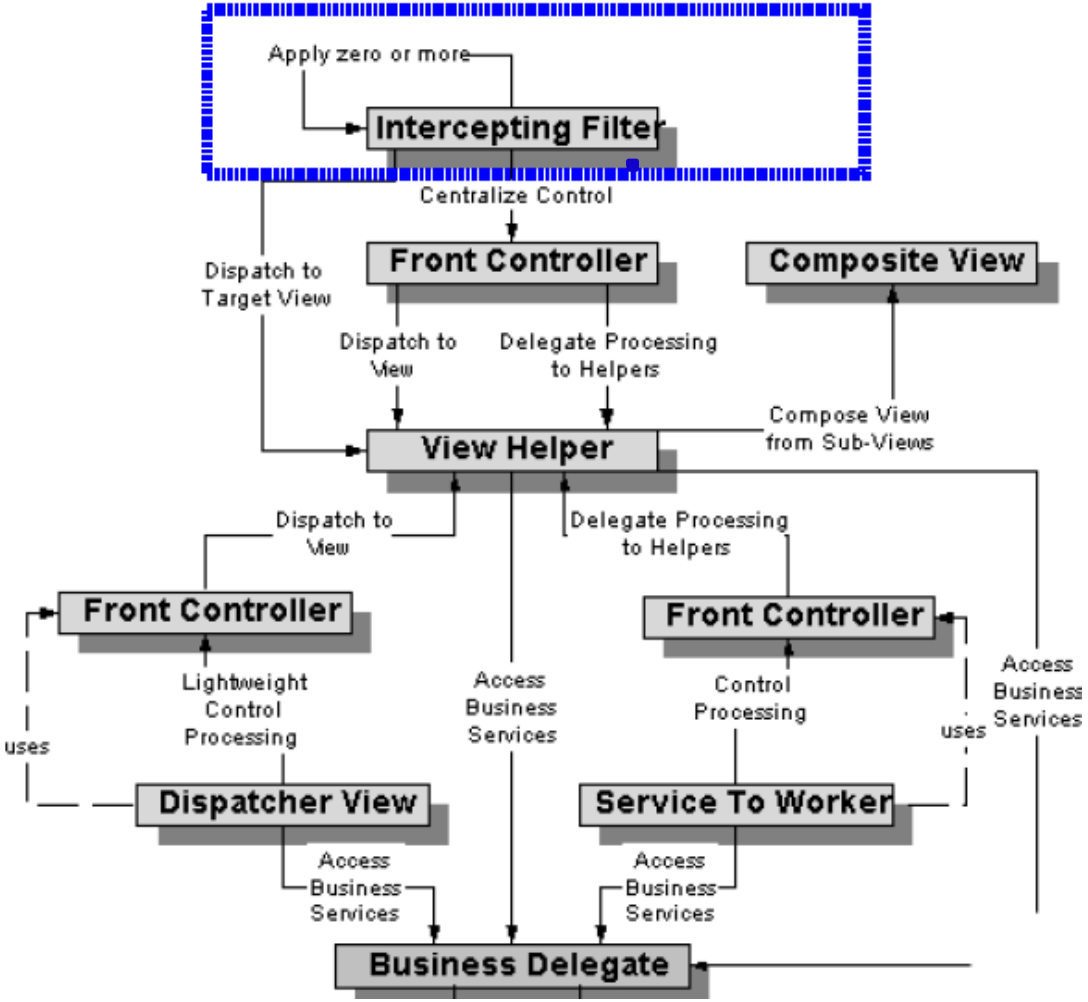
Instantiates the appropriate command object.

Intercepting Filter



Create pluggable filters to process common services in a standard manner without requiring changes to core request processing code. The filters intercept incoming requests and outgoing responses, allowing preprocessing and post-processing. We are able to add and remove these filters unobtrusively, without requiring changes to our existing code

Intercepting Filter Pattern



Intercepting Filter Pattern

- This is a presentation tier web pattern and is designed using several GOF design patterns

Gangs of Four (GoF) Design Patterns

Java => (Core J2EE Patterns)

http://java.sun.com/blueprints/corej2eepatterns/Patterns/InterceptingFilter.html

.NET => (MSDN)

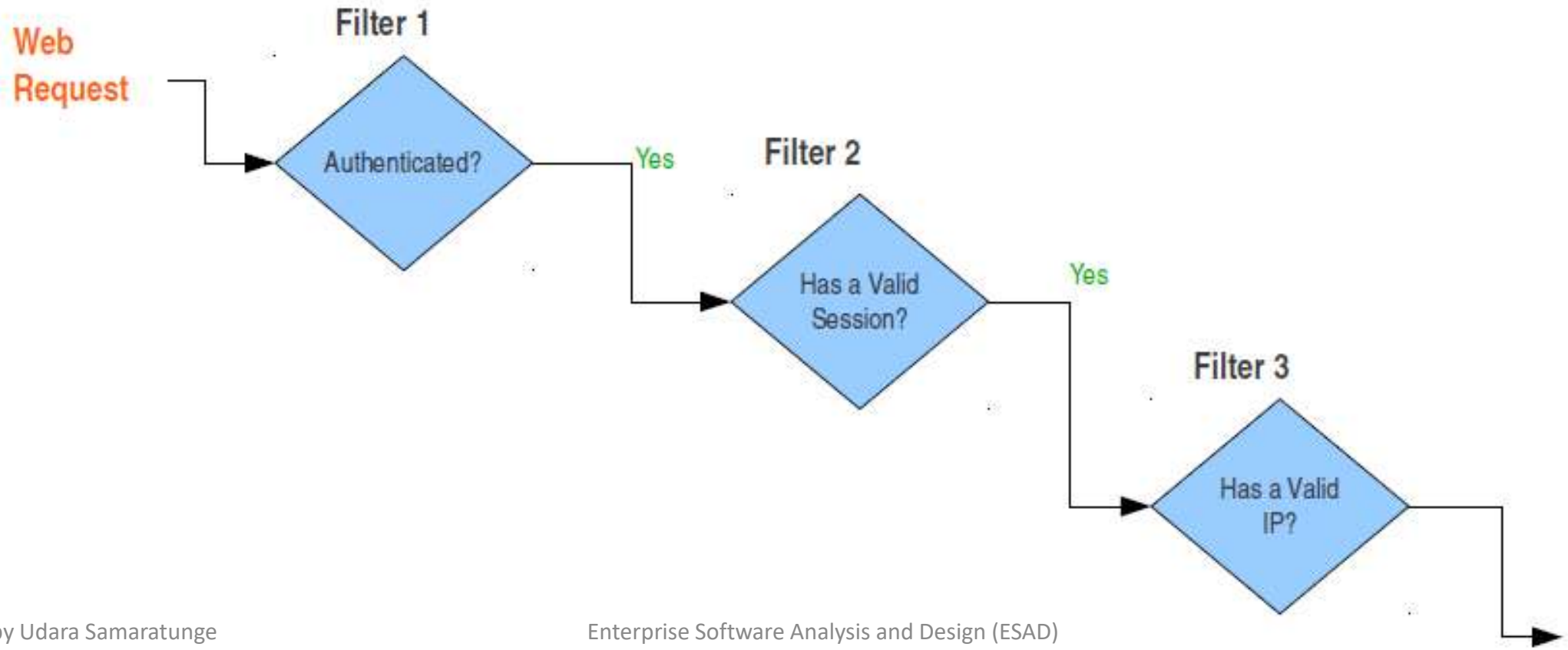
http://msdn.microsoft.com/en-us/library/ms978727.aspx

Intercepting Filter Pattern

- When a request enters a Web application, it often needs to pass a several **entrance tests** prior to the main processing stage. For example:
  - Has the client been authenticated?
  - Does the client have a valid session?
  - Is the client's IP address from a trusted network?
  - Does the request path violate any constraints?
  - What encoding does the client use to send the data?
  - Do we support the browser type of the client?

Intercepting Filter Pattern

The Solution: To have a simple mechanism to add or remove processing components(Filters), in which each component does a certain filtering.



Intercepting Filter Pattern

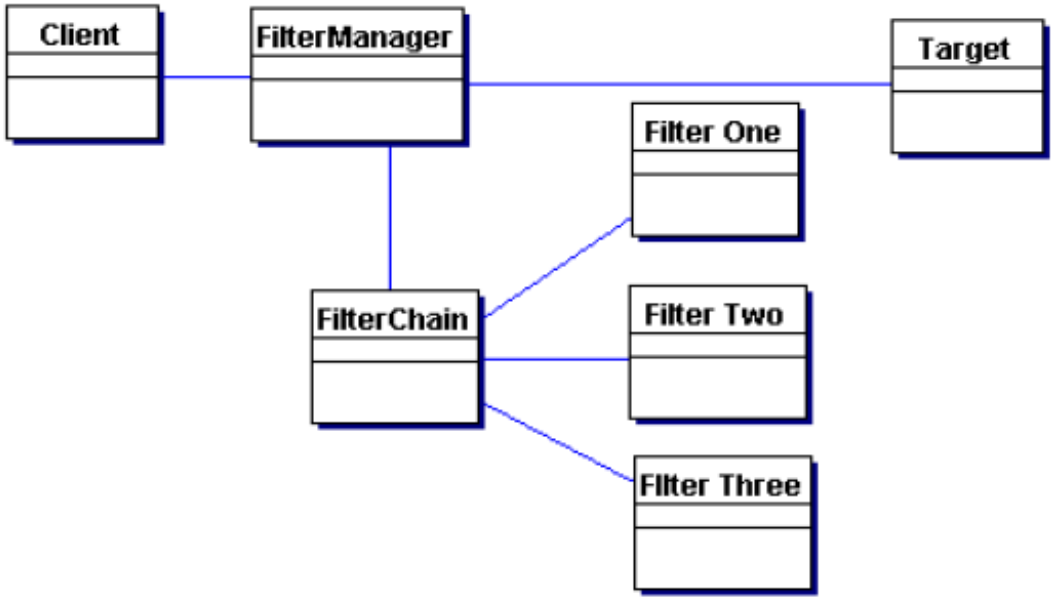
Creates pluggable filters to process common services in a standard manner without requiring changes to core request processing code.

The filters intercept incoming requests and outgoing responses, allowing preprocessing and post-processing.

We are able to add and remove these filters, without doing much changes to our existing code.

Intercepting Filter Pattern

These filters are components which are totally independent from the application code. They may be added or removed declaratively



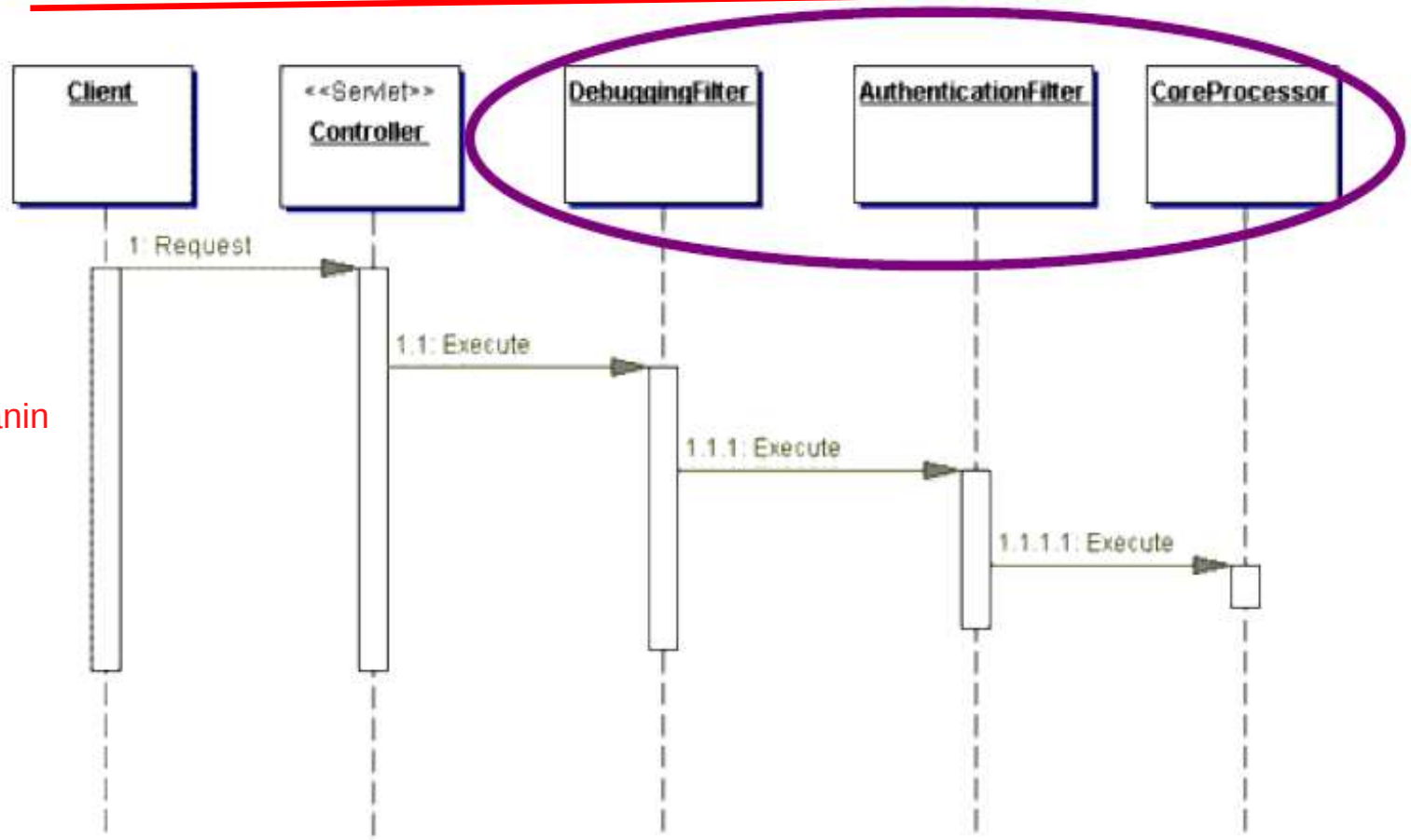
Intercepting Filter Pattern

(Custom Filters – Decorator implementation)

The Decorator Pattern [GoF] is used here

directly invoke next

no filter chanin



Source: Core J2EE Patterns

Exercise 02 - In class Activity

- Write the code for three filter classes **AuthenticationFilter**, **InputValidationFilter**, and **LoggingFilter**. You should implement the interface **IFilter** and override the method. As per the displayed output modify your filter classes accordingly.

```java
public interface IFilter {

    public void execute();
}
```

```java
package com.filter.decorator;

public class Test {

    public static void main(String[] args) {
        IFilter iFilter = new AuthenticationFilter(new DebuggingFilter(new InputValidationFilter()));
        iFilter.execute();

        System.out.println();
        new AuthenticationFilter(new DebuggingFilter()).execute();

        System.out.println();
        new AuthenticationFilter().execute();

        System.out.println();
        new InputValidationFilter(new DebuggingFilter(new AuthenticationFilter())).execute();

    }
}
```

```
<terminated> Test (4) [Java Ap
Authentication Filter
Debugging Filter
Input Validation Filter

Authentication Filter
Debugging Filter

Authentication Filter

Input Validation Filter
Debugging Filter
Authentication Filter
```

Exercise 03 - In class Activity

- Remodify the above program it should start printing inner object to outer object as depicted in the console output.

```java
public interface IFilter {

    public void execute();
}
```

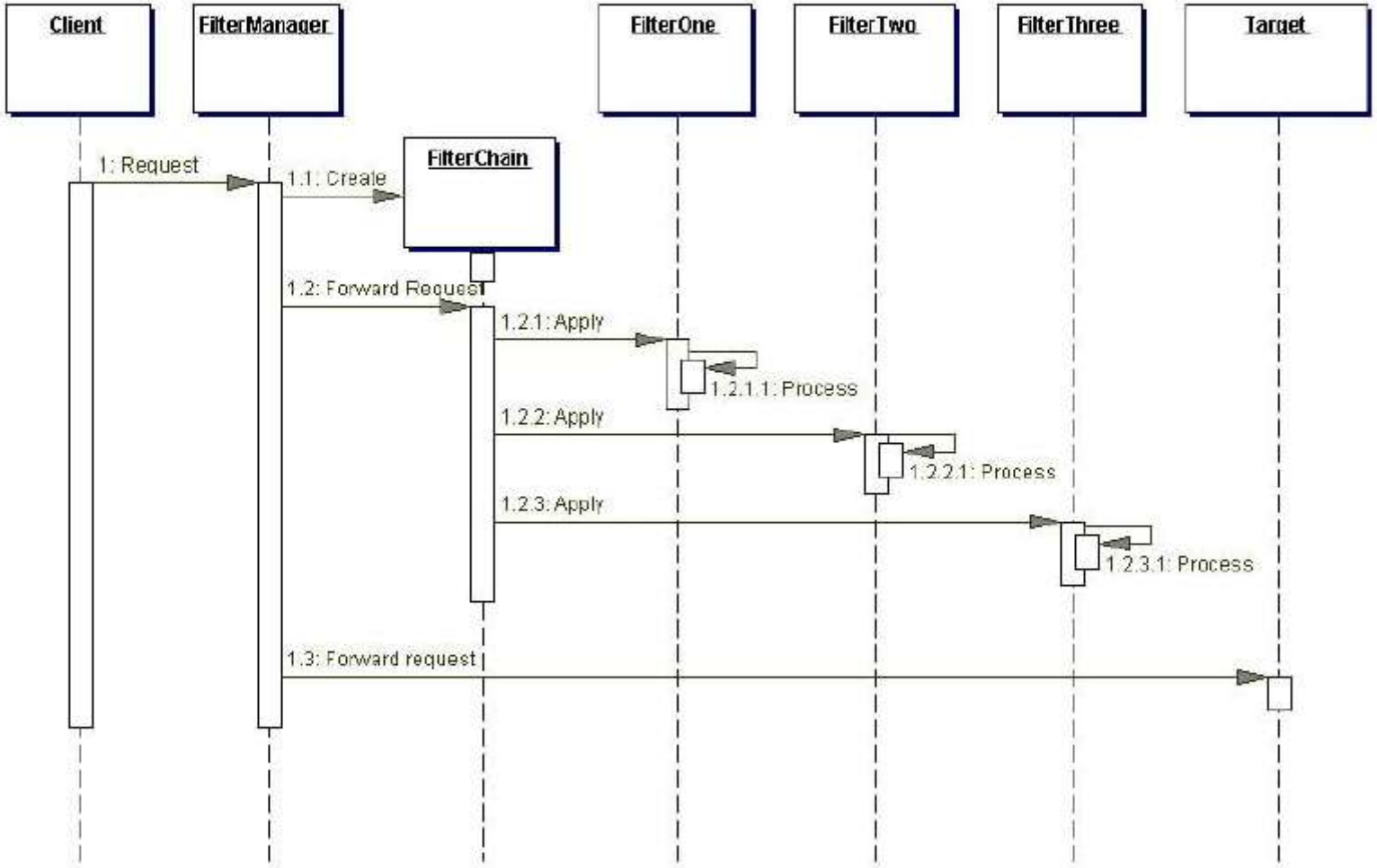
```java
package com.filter.decorator;
public class Test {

    public static void main(String[] args) {
        IFilter iFilter = new AuthenticationFilter(new DebuggingFilter(new InputValidationFilter()));
        iFilter.execute();

        System.out.println();
        new AuthenticationFilter(new DebuggingFilter()).execute();

        System.out.println();
        new AuthenticationFilter().execute();

        System.out.println();
        new InputValidationFilter(new DebuggingFilter(new AuthenticationFilter())).execute();
    }
}
```

```
<terminated> Test (4) [Java App
Input Validation Filter
Debugging Filter
Authentication Filter

Debugging Filter
Authentication Filter

Authentication Filter

Authentication Filter
Debugging Filter
Input Validation Filter
```

Intercepting Filter Non-Decorator Implementation

Intercepting Filter Pattern

(Custom Filters – Non-Decorator implementation)



Source: Core J2EE Patterns

Exercise 04 - In class Activity

- Remodify the above three filters according to the Non-Decorator Filter chain as per the below output. You should implement the same classes (**AuthenticationFilter**, **InputValidationFilter**, and **LoggingFilter)** and implement the interface **IFilter** and override the method. You can maintain chain of filters as ArrayList or Vector and use **FilterManager** class to invoke the Filter chain process. Filter Manager class is given below implement the **FilterChain** class. As per the displayed output modify your filter classes accordingly.

```java
package com.filter.chain;

public class FilterManager {

    public static void main(String[] args) {

        FilterChain filterChain = new FilterChain();
        filterChain.addFilter(new DebuggingFilter());
        filterChain.addFilter(new AuthenticationFilter());
        filterChain.addFilter(new InputValidationFilter());
        filterChain.processFilter();

        filterChain.addFilter(new AuthenticationFilter());
        filterChain.addFilter(new DebuggingFilter());
        filterChain.processFilter();
    }
}
```

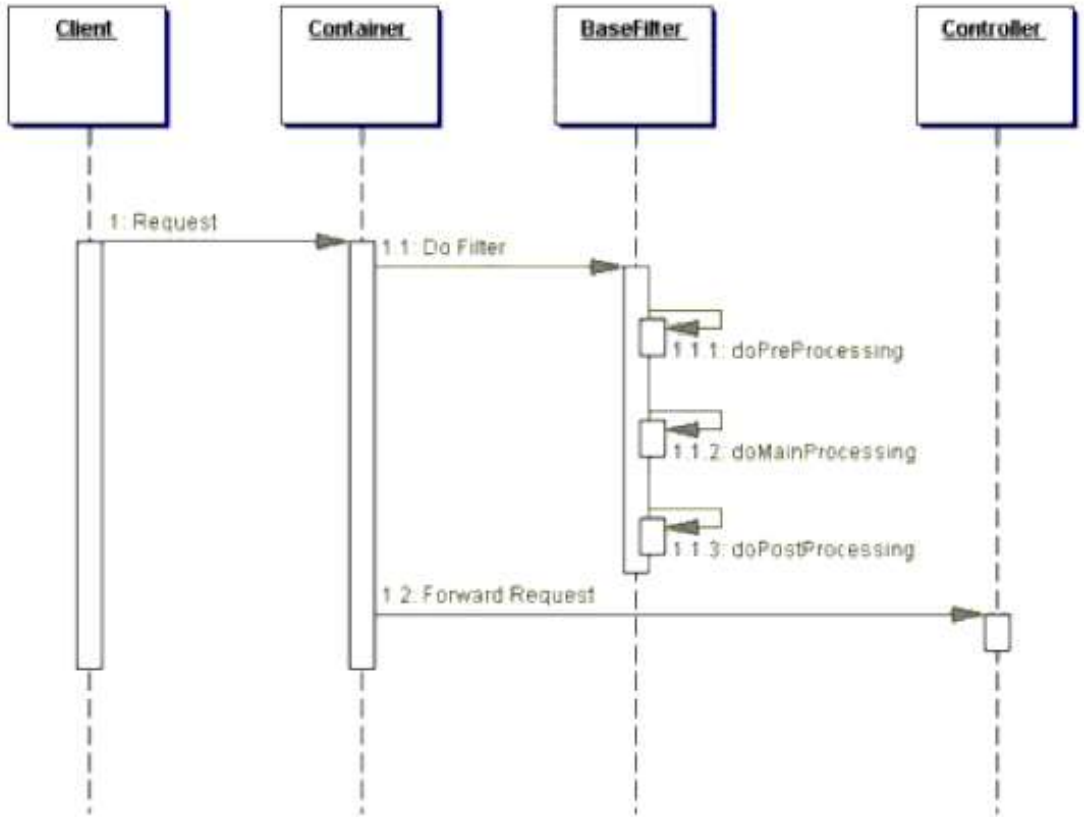
```
<terminated> FilterManager [Jav
Debugging Filter
Authentication Filter
Input Validation Filter

Debugging Filter
Authentication Filter
Input Validation Filter
Authentication Filter
Debugging Filter
```

```
public class FilterChain {
  // filter chain
  private Vector myFilters = new Vector();

  // Creates new FilterChain
  public FilterChain()  {
    // plug-in default filter services for example
    // only. This would typically be done in the
    // FilterManager, but is done here for example
    // purposes
    addFilter(new DebugFilter());
    addFilter(new LoginFilter());
    addFilter(new AuditFilter());
  }
  public void processFilter(
    javax.servlet.http.HttpServletRequest request,
    javax.servlet.http.HttpServletResponse response)
  throws javax.servlet.ServletException,
    java.io.IOException        {
    Filter filter;

    Iterator filters = myFilters.iterator();
    while (filters.hasNext())
    {
      filter = (Filter)filters.next();
      // pass request & response through various
      // filters
      filter.execute(request, response);
    }
  }
  public void addFilter(Filter filter)  {
    myFilters.add(filter);
  }
}
```

The wrapping is handled by HttpServletRequestWrapper implemented by the Custom Filter

Intercepting Filter Pattern

(Template Filters)



```java
public abstract class TemplateFilter implements
  javax.servlet.Filter {
  private FilterConfig filterConfig;

  public void setFilterConfig(FilterConfig fc) {
    filterConfig=fc;
  }

  public FilterConfig getFilterConfig() {
    return filterConfig;
  }

  public void doFilter(ServletRequest request,
    ServletResponse response, FilterChain chain)
    throws IOException, ServletException {
    // Common processing for all filters can go here
    doPreProcessing(request, response, chain);

    // Common processing for all filters can go here
    doMainProcessing(request, response, chain);

    // Common processing for all filters can go here
    doPostProcessing(request, response, chain);

    // Common processing for all filters can go here

    // Pass control to the next filter in the chain or
    // to the target resource
    chain.doFilter(request, response);
  }
  public void doPreProcessing(ServletRequest request,
    ServletResponse response, FilterChain chain) {
  }

  public void doPostProcessing(ServletRequest request,
    ServletResponse response, FilterChain chain) {
  }

  public abstract void doMainProcessing(ServletRequest
   request, ServletResponse response, FilterChain
   chain);
}
```

Template Method

Intercepting Filter Pattern
(Template Filters)

```
public class DebuggingFilter extends TemplateFilter {
  public void doPreProcessing(ServletRequest req,
    ServletResponse res, FilterChain chain) {
    //do some preprocessing here
  }

  public void doMainProcessing(ServletRequest req,
    ServletResponse res, FilterChain chain) {
    //do the main processing;
  }
}
```

This defines a specific processing by overriding the abstract *doMainProcessing* method and, optionally, *doPreProcessing* and *doPostProcessing*

Intercepting Filter Pattern

(Custom Filters – Non-Decorator implementation)

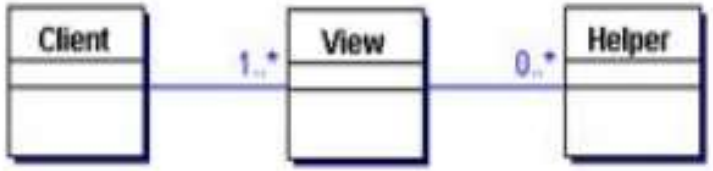
- **Limitations:**
  - Filters can only be added or removed programmatically
  - Not possible to wrap the request and response objects

Intercepting Filter Pattern
(Standard Filters)

- Filters are controlled declaratively using a deployment descriptor (web.xml)

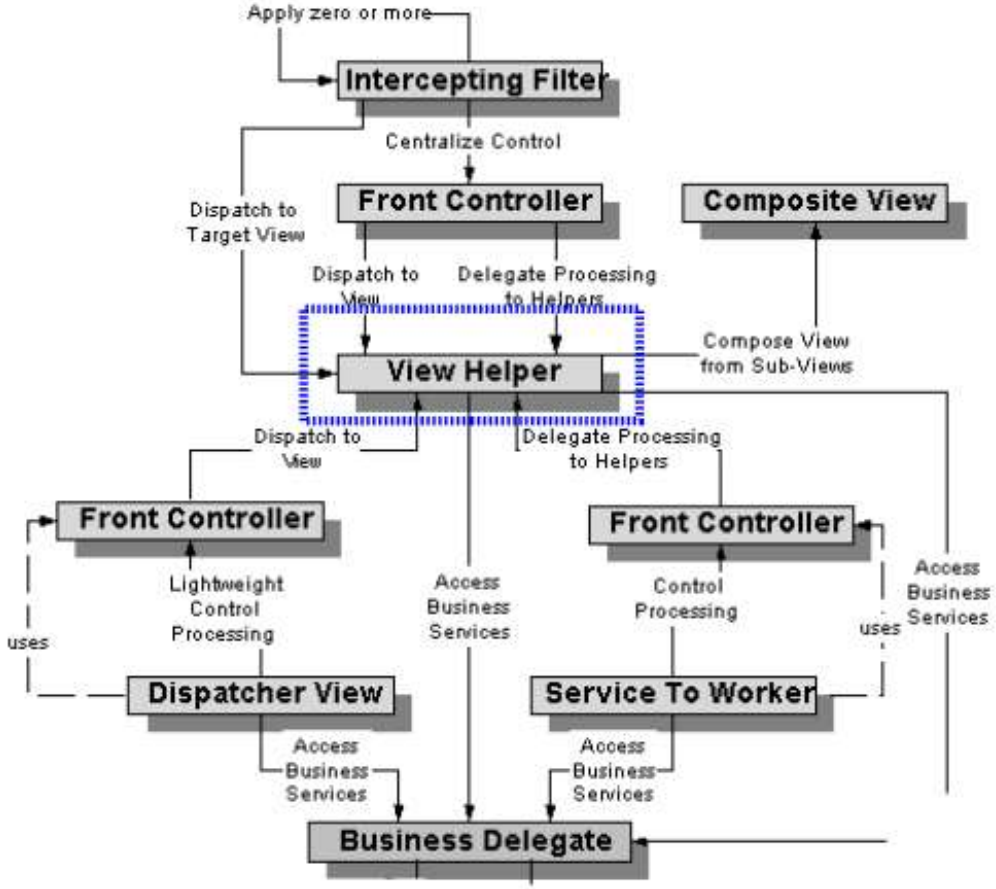
```
<filter>
    <filter-name>StandardEncodeFilter</filter-name>
    <display-name>StandardEncodeFilter</display-name>
    <description></description>
    <filter-class> corepatterns.filters.encodefilter.
            StandardEncodeFilter</filter-class>
  </filter>
  <filter>
    <filter-name>MultipartEncodeFilter</filter-name>
    <display-name>MultipartEncodeFilter</display-name>
    <description></description>
    <filter-class>corepatterns.filters.encodefilter.
            MultipartEncodeFilter</filter-class>
    <init-param>
      <param-name>UploadFolder</param-name>
      <param-value>/home/files</param-value>
    </init-param>
 </filter>
.
.
.
<filter-mapping>
    <filter-name>StandardEncodeFilter</filter-name>
    <url-pattern>/EncodeTestServlet</url-pattern>
  </filter-mapping>
  <filter-mapping>
    <filter-name>MultipartEncodeFilter</filter-name>
    <url-pattern>/EncodeTestServlet</url-pattern>
  </filter-mapping>
```

View Helper



The system creates presentation content, which requires processing of dynamic business data

View Helper Pattern



View Helper – why we need it?

- Presentation tier changes occur often and are difficult to develop and maintain when business data access logic and presentation formatting logic are interwoven
- This makes the system less flexible, less reusable

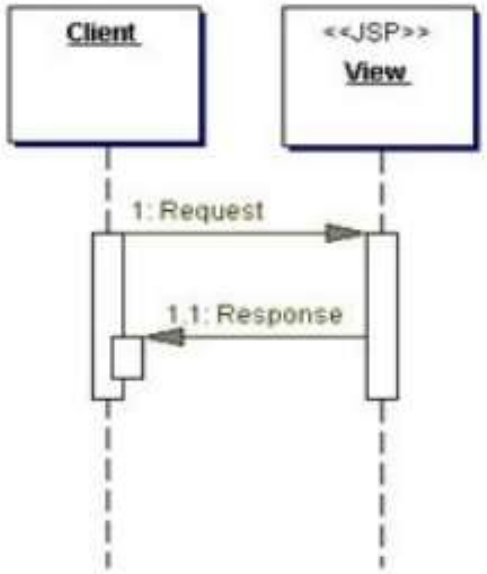
Hence,

- Encapsulating business logic in a helper instead of a view makes our application more modular and facilitates component reuse

  (Examples: Java Beans, JSP Tags, etc)

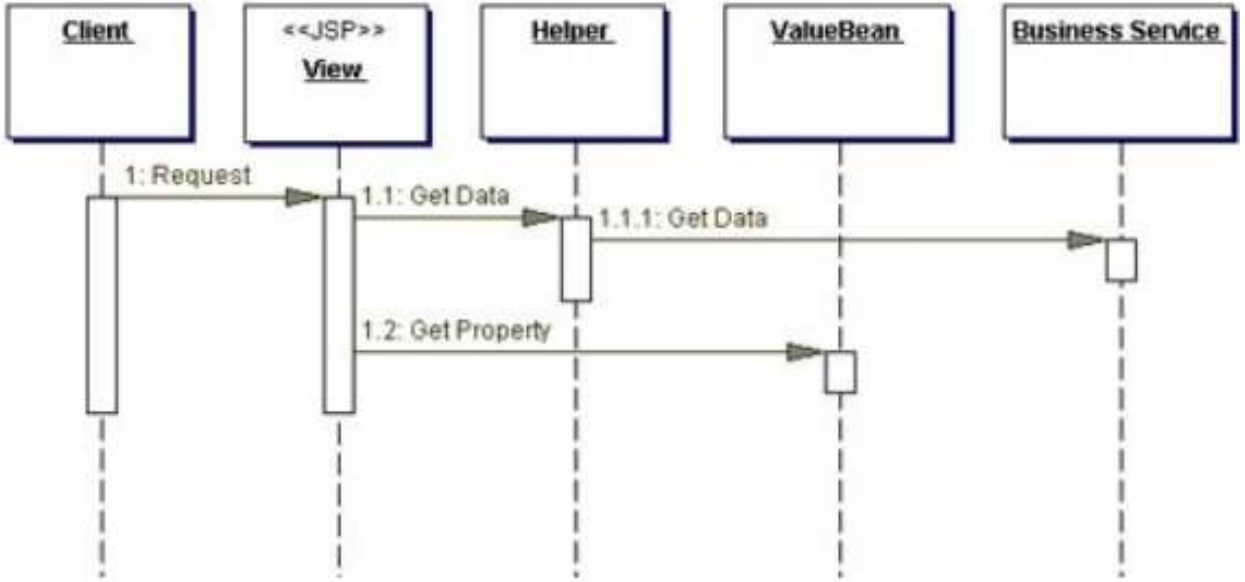
View Helper

The Simple Sequence Diagram



There are **no Helpers** here.

The page may be entirely static or with a little bit of scriplets



View Helper in J2EE

The Servlet View Strategy

Considers the Servlet as the View

```
public class Controller extends HttpServlet {
  public void init(ServletConfig config) throws
    ServletException {
    super.init(config);
  }

  public void destroy() { }

  /** Processes requests for both HTTP
   * <code>GET</code> and <code>POST</code> methods.
   * @param request servlet request
   * @param response servlet response
   */
  protected void processRequest(HttpServletRequest
    request, HttpServletResponse response)
    throws ServletException, java.io.IOException {
    String title = "Servlet View Strategy";
    try {
      response.setContentType("text/html");
      java.io.PrintWriter out = response.getWriter();
      out.println("<html><title>"+title+"</title>");
      out.println("<body>");
      out.println("<h2><center>Employees List</h2>");
      EmployeeDelegate delegate =
          new EmployeeDelegate();

      /** ApplicationResources provides a simple API
        * for retrieving constants and other
        * preconfigured values**/
      Iterator employees = delegate.getEmployees(
            ApplicationResources.getInstance().
```

Not a Good Strategy

View Helper in J2EE

The JSP/ Java Bean Helper View Strategy

Considers the Java Beans as the View Helpers to separate the business process logic

```
<jsp:useBean id="welcomeHelper" scope="request"
  class="corepatterns.util.WelcomeHelper" />

<HTML>
<BODY bgcolor="FFFFFF">
<% if (welcomeHelper.nameExists())
{
%>
<center><H3> Welcome <b>
<jsp:getProperty name="welcomeHelper" property="name" />
</b><br><br> </H3></center>
<%
}
%>

<H4><center>Glad you are visiting our
  site!</center></H4>

</BODY>
</HTML>
```

Helper Bean

A Good Strategy

The Custom-Tag Helper Strategy

```
<%@ taglib uri="/web-INF/corepatternstaglibrary.tld"
  prefix="corepatterns" %>
<html>
<head><title>Employee List</title></head>
<body>

<div align="center">
<h3> List of employees in <corepatterns:department
  attribute="id"/> department - Using Custom Tag
  Helper Strategy. </h3>
<table border="1" >
    <tr>
        <th> First Name </th>
        <th> Last Name </th>
        <th> Designation </th>
        <th> Employee Id </th>
        <th> Tax Deductibles </th>
        <th> Performance Remarks </th>
        <th> Yearly Salary</th>
    </tr>
    <corepatterns:employeelist id="employeelist_key">
    <tr>
        <td><corepatterns:employee
            attribute="FirstName"/> </td>
        <td><corepatterns:employee
            attribute= "LastName"/></td>
        <td><corepatterns:employee
            attribute= "Designation"/> </td>
        <td><corepatterns:employee
            attribute= "Id"/></td>
        <td><corepatterns:employee
            attribute="NoOfDeductibles"/></td>
        <td><corepatterns:employee
            attribute="PerformanceRemarks"/></td>
        <td><corepatterns:employee
            attribute="YearlySalary"/></td>
        <td>
     </tr>
    </corepatterns:employeelist>
</table>
</div>
</body>
</html>
```

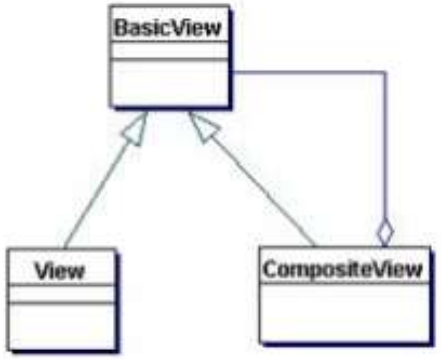
The Helper is implemented as a Custom Tag

Requires more effort to develop the Tag Library

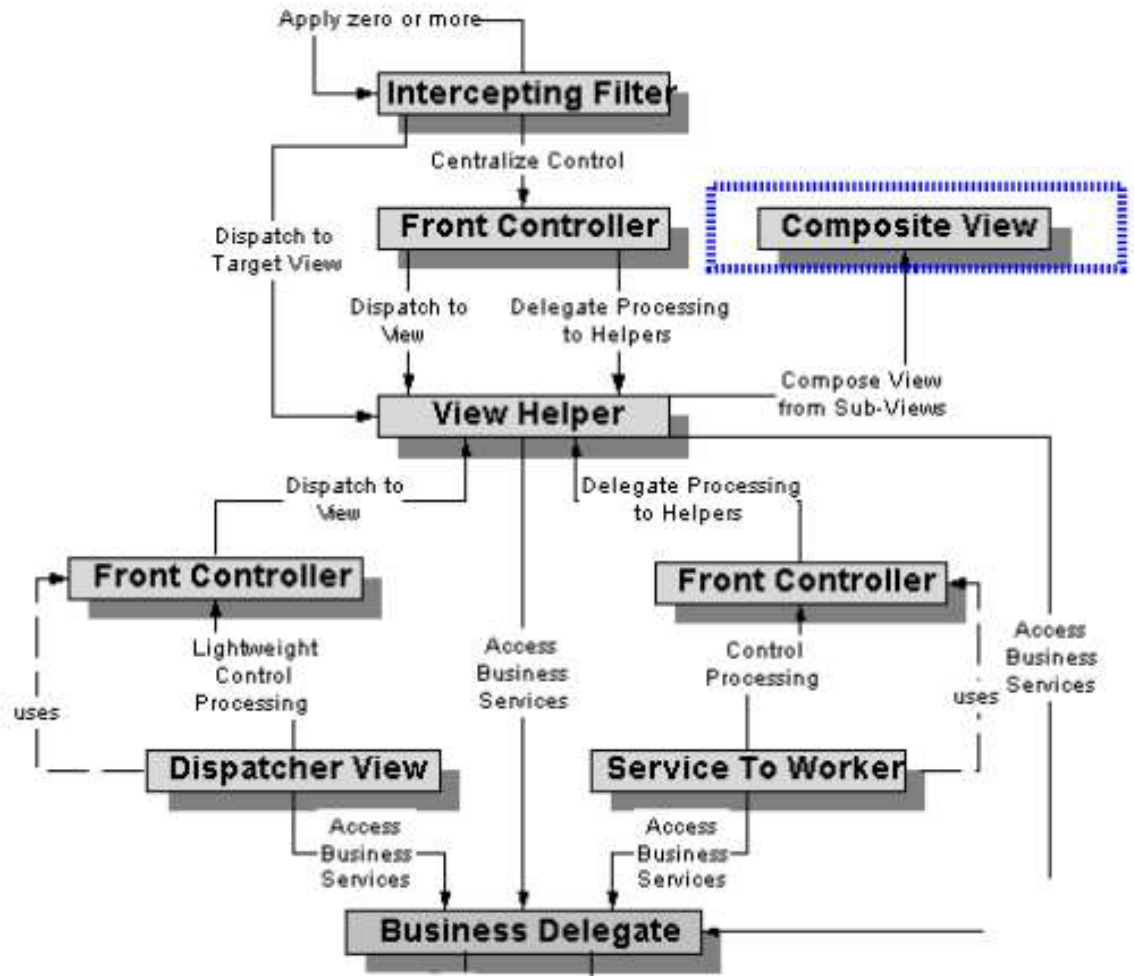
Good if you have more common business logic

Composite View Pattern

Composite View



Use composite views that are composed of multiple atomic sub-views. Each component of the template may be included dynamically into the whole and the layout of the page may be managed independently of the content



Composite View Pattern

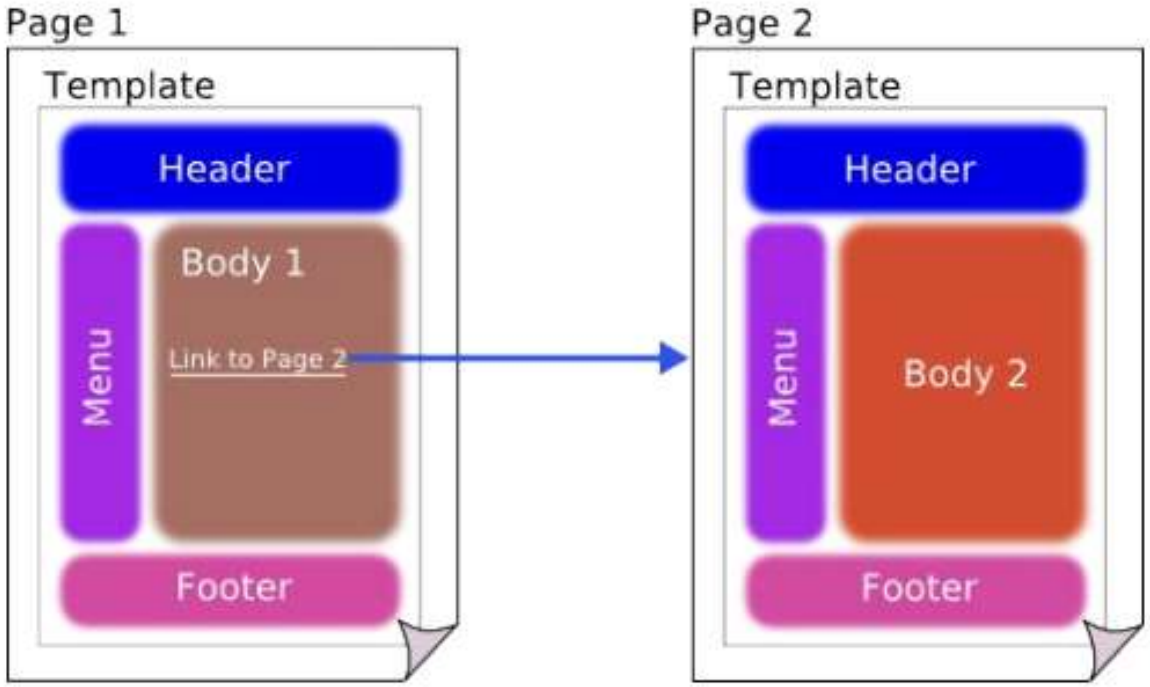
- This allows to create pages that have a similar structure, in which each section of the page vary in different situations

The Classic Page Layout

Composite View Pattern

Composite View is an Aggregation of multiple views

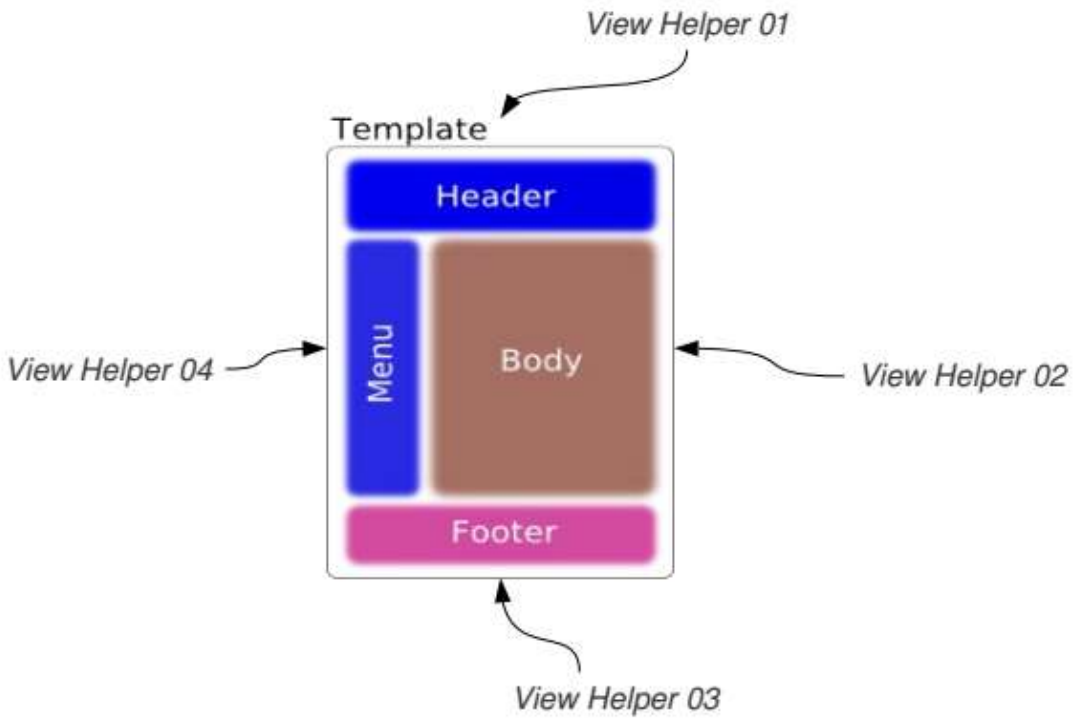
Composite View Pattern



Only the "Body" part is changed. Rest of the layout is preserved.
However the all pages related to the template are distinct.

Composite View Pattern

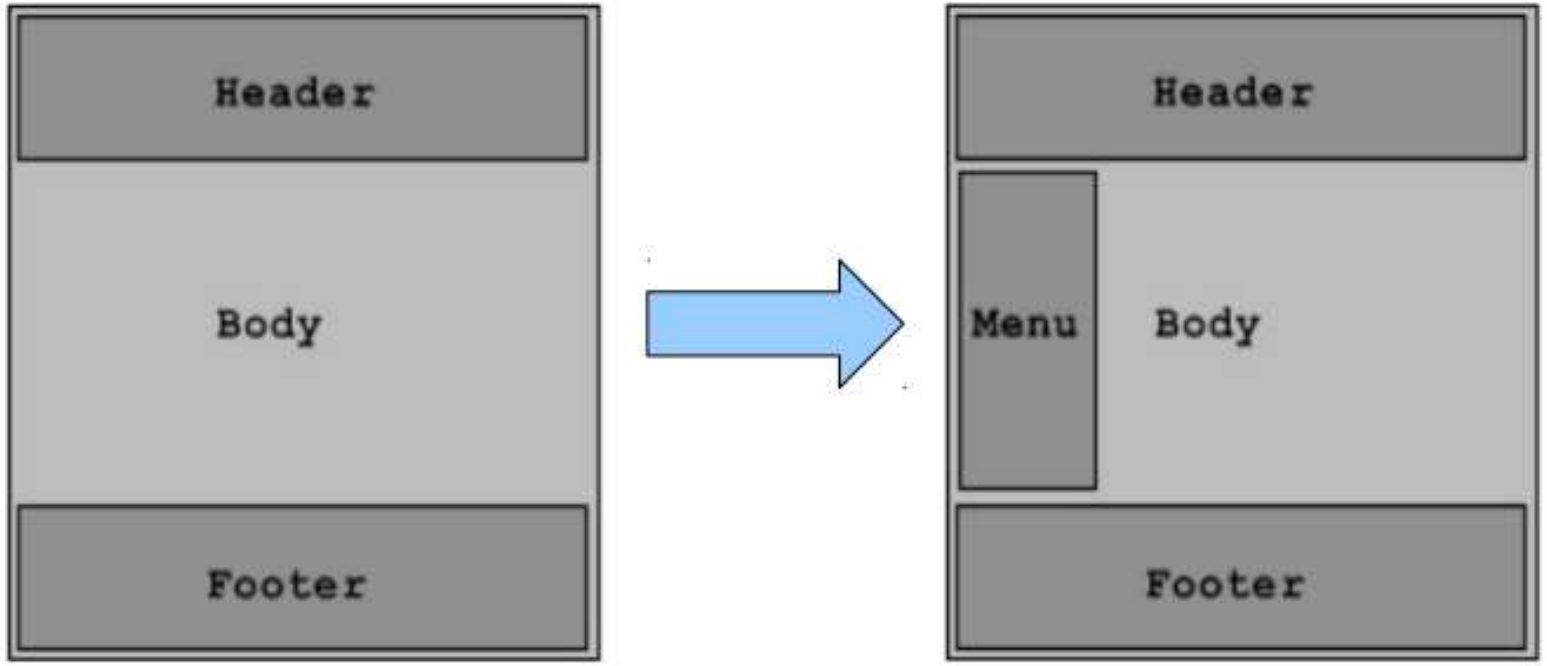
Each piece of the composed page can have a "view helper"



Composite View Pattern

Tiles Framework (http://tiles.apache.org)

What if you want to change the web application page layout like below?



Composite View Pattern

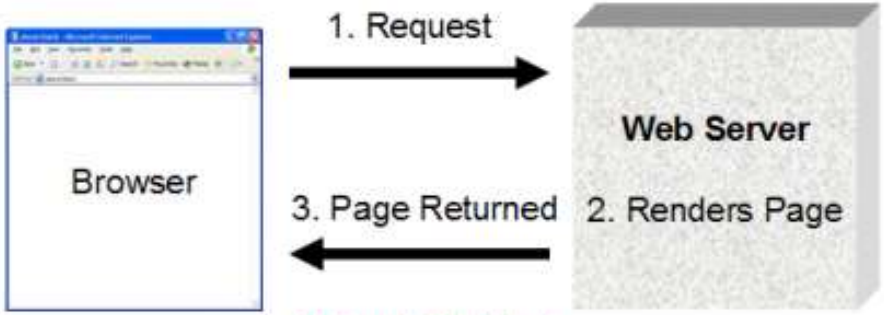
Tiles Framework (http://tiles.apache.org)

- Tiles uses a separate layout file
- When the layout of the application is changed, only this layout file and other tiles configuration files need to be changed

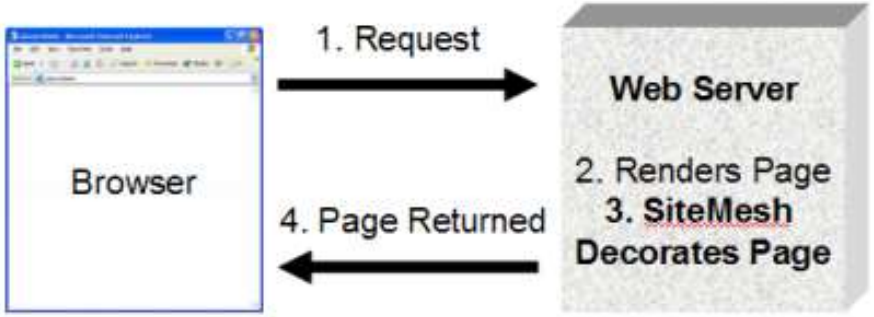
```
<%@ page language="java"%>

<%@ taglib uri="http://jakarta.apache.org/struts/tags-html" prefix="html" %>
<%@ taglib uri="http://jakarta.apache.org/struts/tags-tiles" prefix="tiles" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html:html locale="true">
  <head>
    <html:base />
    <title><tiles:getAsString name="title" /></title>
  </head>
  <body>
  <table border="1" width="600" cellspacing="5">
      <tbody>
            <tr><td colspan="2"><tiles:insert attribute="header" /></td></tr>
            <tr>
                  <td width="200"><tiles:insert attribute="navigation" /></td>
                  <td width="400"><tiles:insert attribute="body" /></td>
            </tr>
            <tr><td colspan="2"><tiles:insert attribute="footer" /></td></tr>
      </tbody>
  </table>
  </body>
</html:html>
```

SiteMesh Framework

http://www.opensymphony.com/sitemesh/



Without SiteMesh



With SiteMesh

A Recap

- **Requirement**: I need one place of control for handling all requests
  - Pattern: Front Controller Intercepting Filter
- **Requirement**: I need a generic command interface for delegating processing from a controller to various helper components
  - Pattern: Front Controller
- **Requirement**: I want to make sure data related to my presentation formatting logic is encapsulated correctly
  - Pattern: View Helper
- **Requirement**: I need to be able to create one View from a number of sub-Views
  - Pattern: Composite View

Enterprise Application Blueprints

- These are well-defined design patterns geared towards a specific technology
  - Sun's J2EE BluePrint for Java

    (Reference: Core J2EE Patterns Book)

  - .NET Blueprint for C# (.NET Pet Store example)

References

- Patterns of Enterprise Application Architecture (PoEAA): *Martin Fowler (http://martinfowler.com/eaaCatalog/)*
- J2EE Design Patterns: *William Crawford & Jonathan Kaplan*
- Core J2EE Patterns: *Deepak Alur, John Crupi, Dan Malks*
- *http://www.developer.com/design/article.php/3619786/Implementing-the-Intercepting-Filter-Pattern-in-Your-Enterprise-Java-Applications.htm*
- http://msdn.microsoft.com/en-us/library/ms998516.aspx
- http://struts.apache.org/1.x/userGuide/building_controller.html
- http://java.sys-con.com/node/36656