Web Application Design Patterns

Jim Yu

IBM
China Development Lab
Greater China Group

Web Characteristics

- Web is originally designed for **documents** instead of **applications**
  - Request–response model
    - Client (browser) initiates the request and server sends the response accordingly
    - No server push
  - Whole–page retrieval
    - The whole page is refreshed after the response is sent to the client
  - Stateless

Web as an Application Platform

- Enabling technologies
  - HTTP Cookie to remember user “states”
  - “server pages” such as ASP, PHP, JSP to generate dynamic contents
  - Client-side scripting (Javascript) to enhance client richness
- Often needs to access or integrate with other systems
  - Database, LDAP, another web application, etc

A Simple Web Application

- Intertwined HTML markup and application logic

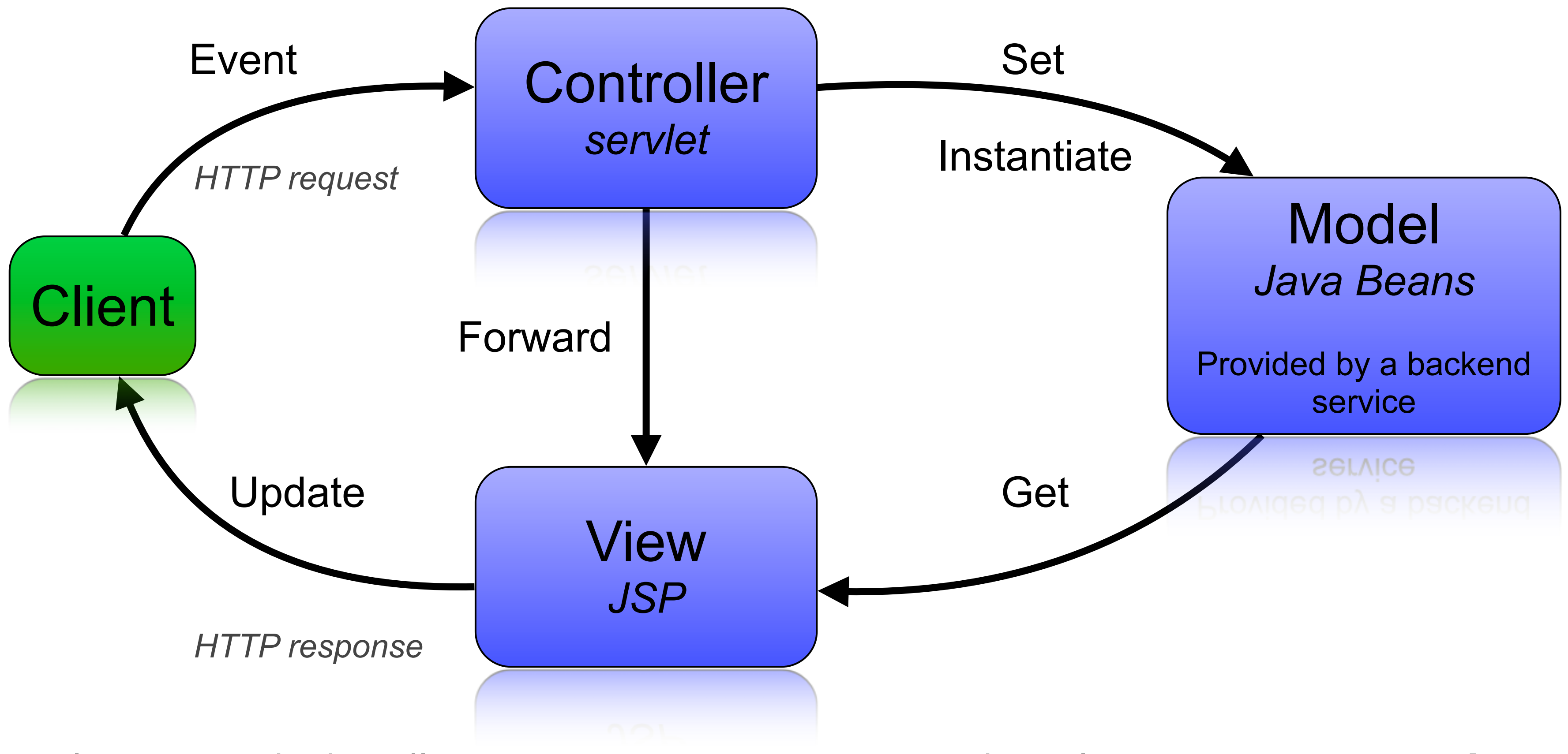
```
<%
  // application logic to handle the submitted request
%>

<form action="/some/web/page.jsp" method="post">
   User id: <input type="text" name="userid"
             value="<% request.getParameter("userid") %>">
   Password: <input type="password" name="password">
   <%  if (...) { /* if some condition is met */ %>
   <!-- some optional item is displayed here -->
   <%  } /* end of optional item */ %>
   <input type="submit" value="submit"
</form>
```

- Hard to maintain for large applications

MVC Model 2

- A Web adaptation of MVC
- An MVC Model 2 impl. using Servlet/JSP:



Web Mimicking Desktop Applications

- Client enhancements to make web applications richer
  - Asynchronous Javascript and XML (AJAX)
    - Asynchronous request/response with the server
      - No blocking of the client during request processing
      - Partial update of the web screen
  - Rich Internet Application (RIA) using Adobe Flash platform

Web Mimicking Desktop Applications

- Component-based web development:
  - Hides the underlying HTTP/HTML nature of web applications
  - Provide desktop-like development experience
    - Web widgets/components
    - Event notifications
- Web framework weaves the technologies together to simplify development
  - Some may provide additional features such as security or database access

Java EE Design Patterns

- Java Enterprise Edition (Java EE, formerly J2EE) is a platform for developing server-centric enterprise applications
  - Including Web, database, enterprise business component, etc.
- Java EE design patterns provides best practice and common solution to recurring problems in using Java EE
  - Some are Java EE-specific, while others apply to web/database development on other platforms

Intercepting Filter

Provide pluggable components to preprocess and post-process Web requests and responses

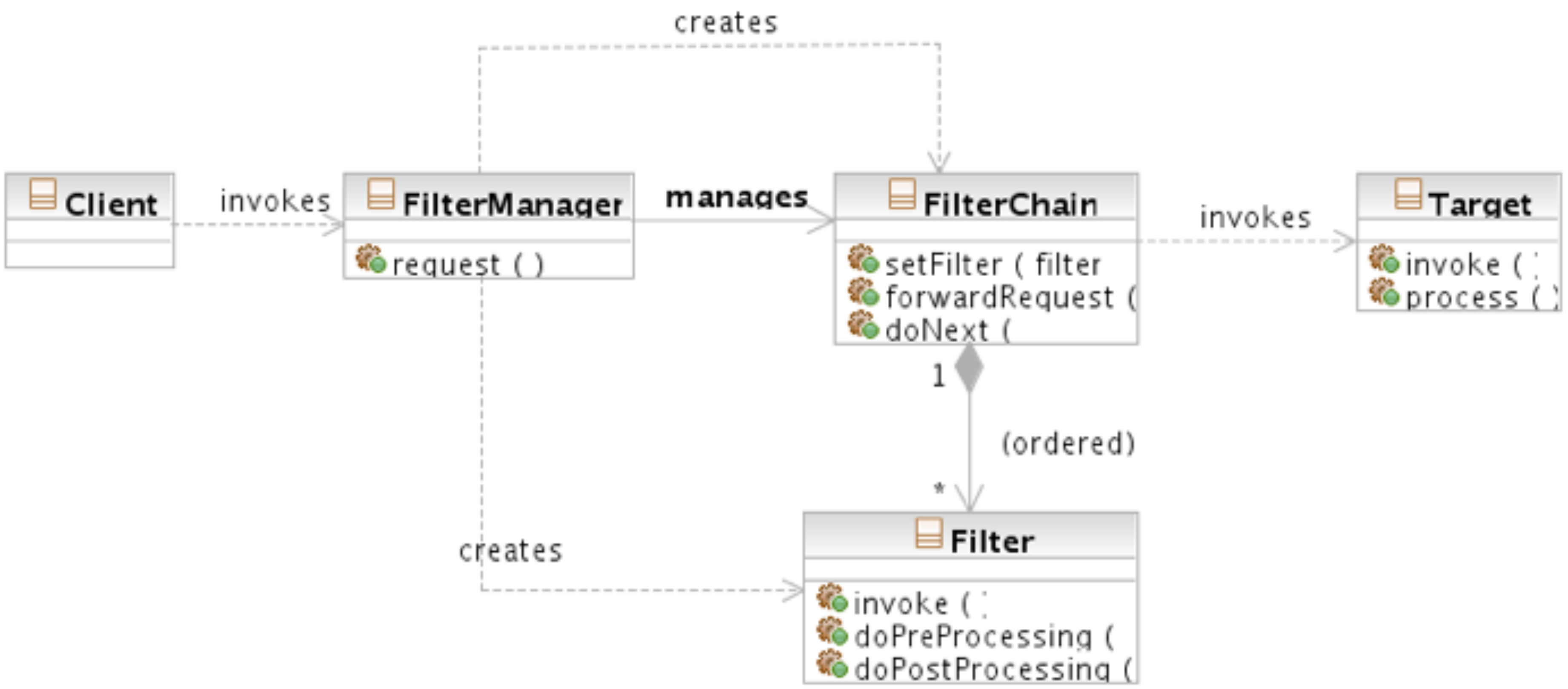
Problem

- We often have to preprocess and post-process Web requests and responses for:
  - Client authenticated?
  - Client authorized to access the resource?
  - Trusted client IP address?
  - Requirement for browser capabilities (flash player, JVM, audio/video player, etc.)
  - Client encoding?
- They are often shared services
- Request rejection or content manipulation needed

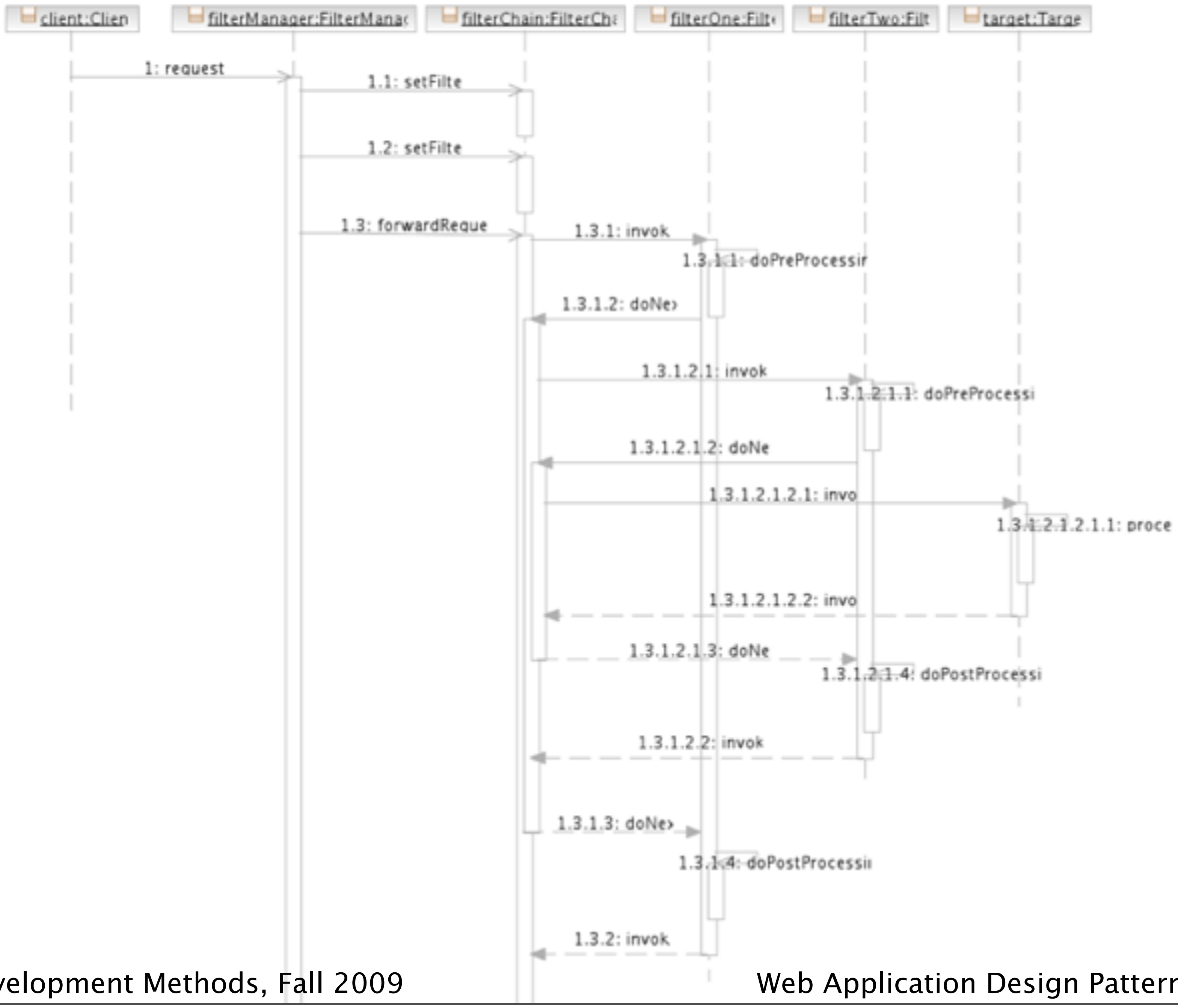
Solution

- Decorator pattern
- Standard and pluggable filters to process common/shared services
  - Independent of the main application logic
- Configured declaratively
  - Mapping from some URLs to a filter chain (of a set of filters)

Class Diagram



Sequence Diagram



Consequences

- Centralizes control with loosely coupled handlers
- Improves reusability
- Declarative and flexible configuration
- Information sharing (between filters) is inefficient

Composite View

Compose the final view with atomic subviews dynamically

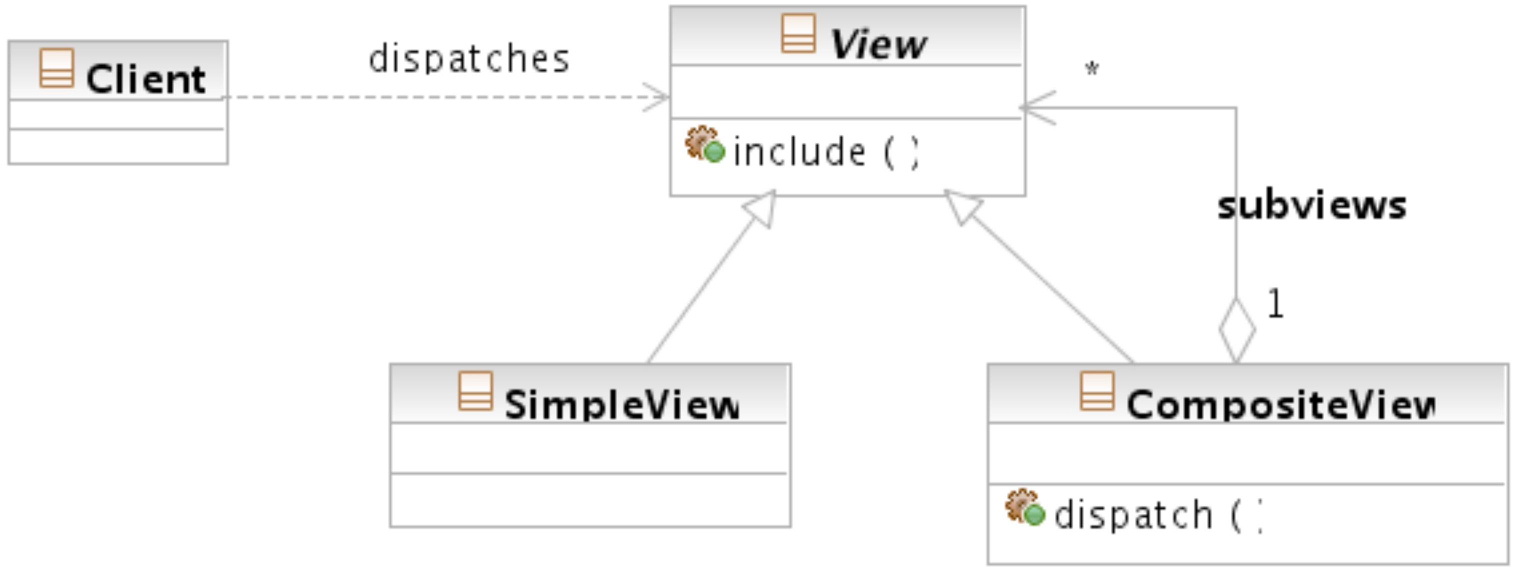
Problem

- Web views often have common visual components
  - Header, footer
  - Logo
  - Navigation toolbar, menu
- Statically embed them in each view is error prone and creates maintenance problems

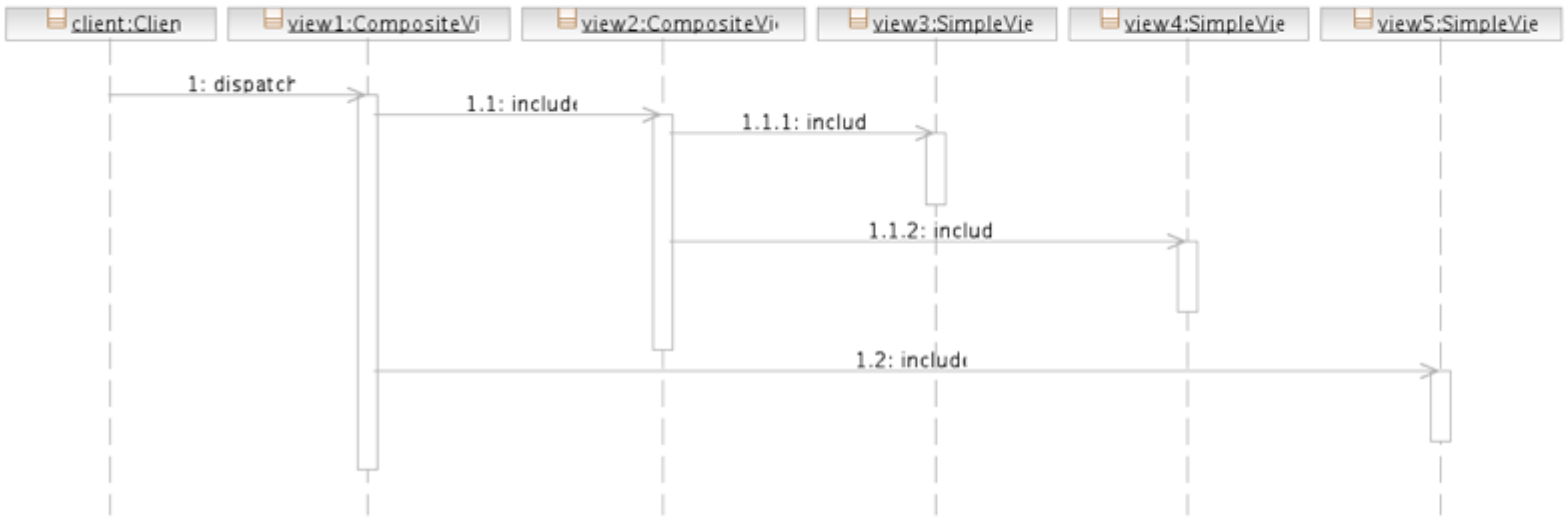
Solution

- Composite Pattern
- Composite views composed of atomic subviews
- Composite view to include composite views or atomic subviews dynamically
- Layout can be managed independently of the content

Class Diagram



Sequence Diagram



Consequences

- Improves modularity and reuse
- Enhances flexibility
- Enhances maintainability and manageability
- Reduces manageability
  - e.g. when subviews generates unbalanced html tags and make the final output invalid HTML page
- Performance impact

Data Access Object

Abstracts and encapsulates all access to the data source

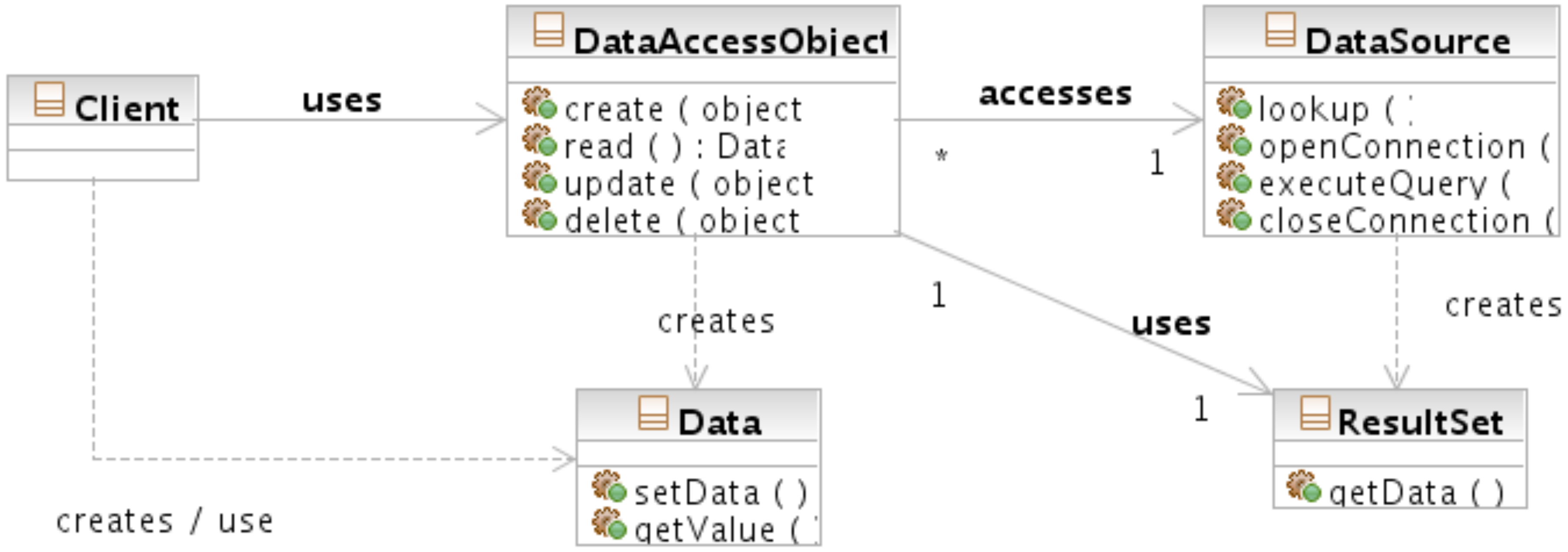
Problem

- Variety in how domain data are accessed
  - Persistent data from relational database, object-oriented database, XML database, LDAP, Flat files, etc.
    - Different APIs
  - Data from another system
    - Raw TCP socket, web service, etc.
- Some data access methods have constraints
  - e.g. connection number limit
- Hardcode data access API is inflexible

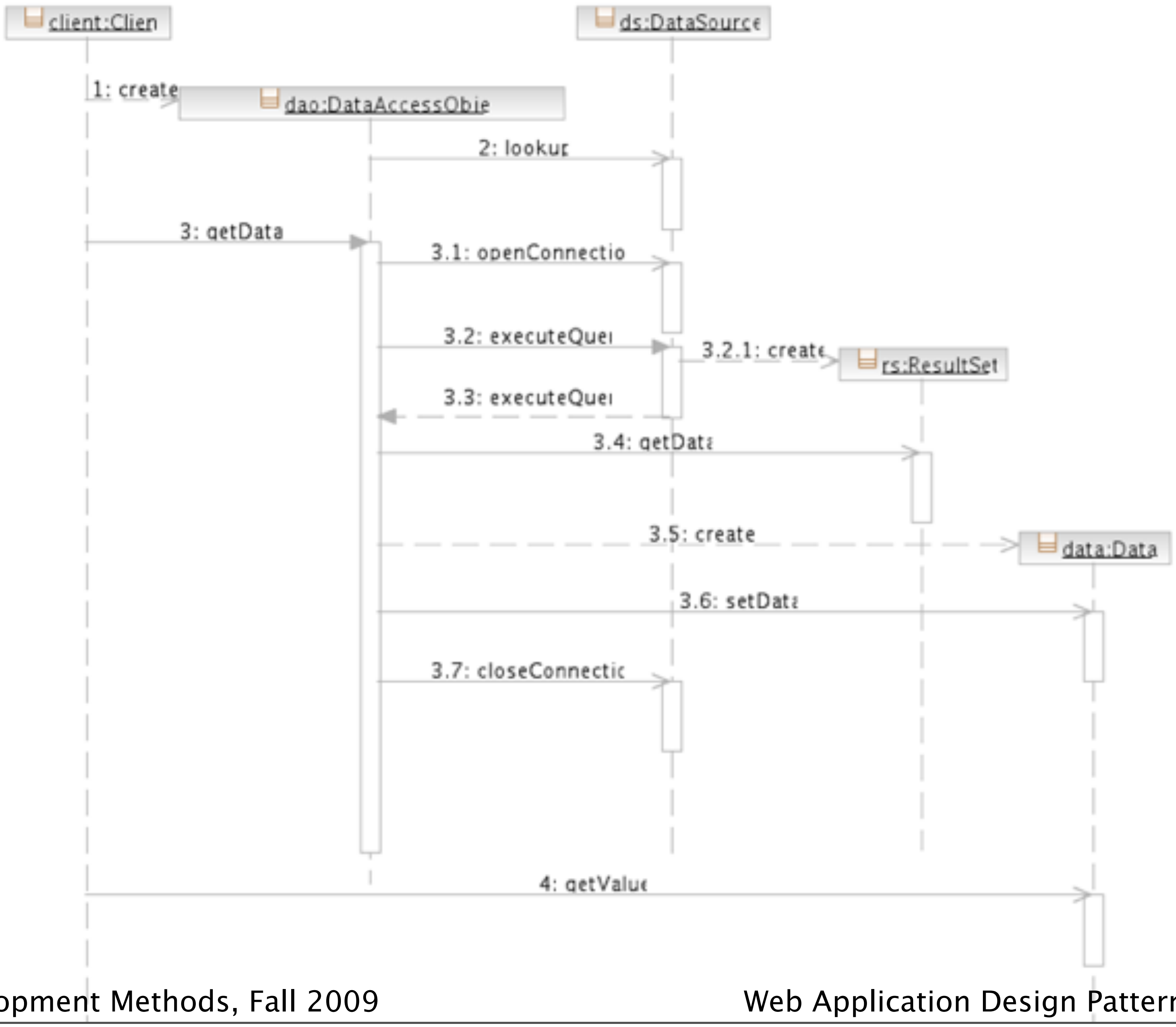
Solution

- Adapter Pattern
- Encapsulate all access to the data source in the Data Access Object (DAO)
- DAO provides a simplified and consistent API to hides data access details from the caller
- Change underlying data source without affecting the DAO user
  - just change the concrete adapter
- Easy to unit test with mock DAO impl.

Class Diagram



Sequence Diagram



Consequences

- Enables transparency
- Enables easier migration
- Reduces code complexity in business objects
- Centralizes all data access into a separate layer
- Adds extra layer

References

- Deepak Alur, John Crupi, Dan Malks, *Core J2EE Patterns*, Pearson Education, 2001
- Malcolm Davis, *Struts, an open-source MVC implementation*, http://www.ibm.com/developerworks/library/j-struts/