3.3: EIS Patterns

One solution to complexity of EIS design is to provide 'patterns' or standard solutions for EIS design problems. Design patterns define microarchitecture within an EIS, to implement particular required property/functionality of system or to rationalise system structure.

These apply at different tiers. Presentation tier patterns include:

- **Intercepting Filter**: defines a structure of pluggable filters to add pre and post-processing of web requests/responses, eg: for security checking.
- **Front Controller**: defines a single point of access for web system services, through which all requests pass. Enables centralised handling of authentication, etc.
- **View Helper**: separates presentation and business logic by taking responsibility for visual presentation (eg, as HTML) of particular business data.

- **Composite View**: uses objects to compose a view out of parts (subviews).
- **Service to Worker**: this combines front controller and view helper to construct complex presentation content in response to a request.
- **Dispatcher View**: similar, but defers content retrieval to the time of view processing.

Business tier patterns include:

- **Business Delegate**: provides an intermediary between presentation tier and business services, to reduce dependence of presentation tier on details of business service implementation.
- **Value Object**: an object which contains attribute values of a business entity (entity bean), this object can be passed to presentation tier as a single item, so avoiding cost of multiple *getAttribute* calls on the entity bean.

- **Session Facade**: use a session bean as facade to hide complex interactions between business objects in one workflow/use case.
- **Composite Entity**: use an entity bean to represent and manage a group of interrelated persistent objects, to avoid costs of representing group elements in individual fine-grained entity beans (eg, group a master object with its dependents).
- **Value Object Assembler**: builds a model using possibly several value objects from various business objects.
- **Value List Handler**: provide efficient interface to examine a list of value objects (eg, result of a database search).
- **Service Locator**: abstracts details of service/resource lookup, bean creation, etc.

Integration tier patterns include:

- **Data Access Object**: provides abstraction of persistent data source access.
- **Service Activator**: implements asynchronous processing of business service components.

Many of these patterns are not specific to Java, and could be used with other EIS application platforms such as .Net.

Here we will consider: Intercepting filter; Front controller; Composite view; Value object; Session facade; Composite entity; Value list handler; Data access object.

Intercepting Filter

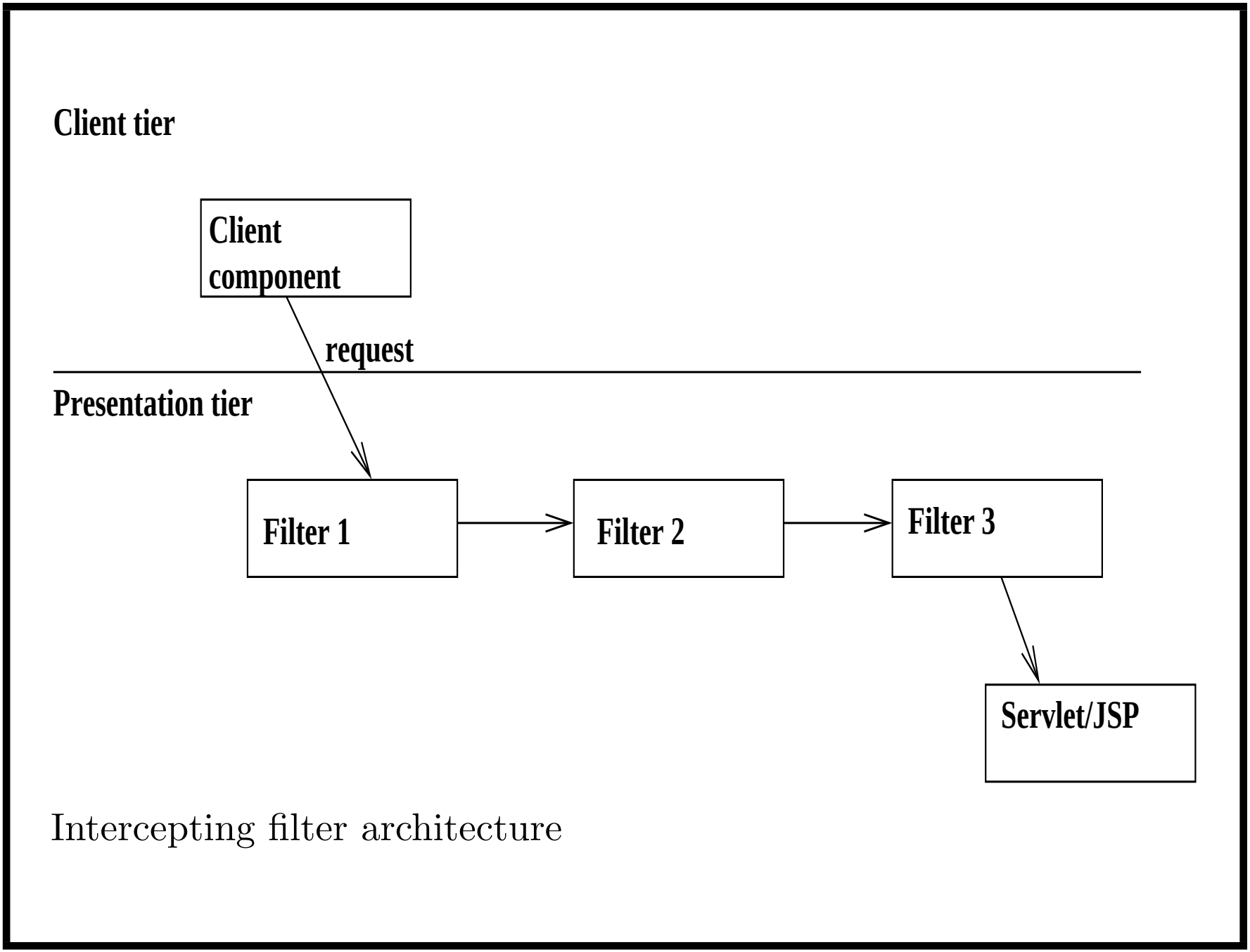
This presentation tier pattern has purpose to provide a flexible and configurable means to add filtering, pre and post processing, to presentation-tier request handling.

When a client request enters a web application, it may need to be checked before being processed, eg:

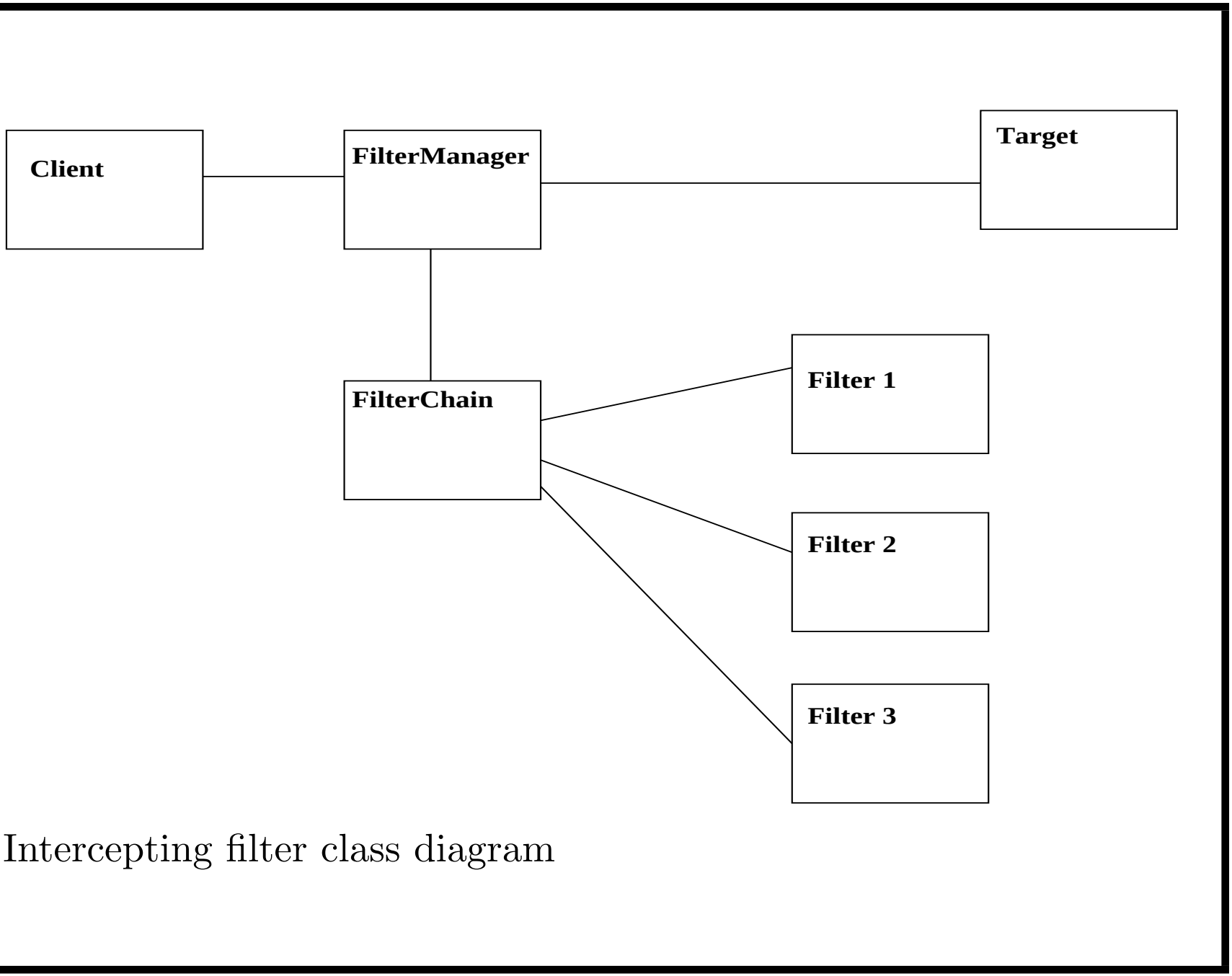
- Is the client's IP address from a trusted network?
- Does the client have a valid session?
- Is the client's browser supported by the application?

and so forth.

It would be possible to code these as nested *if* tests, but is more flexible to use separate objects in a chain to carry out successive tests. (Cf: the Chain of Responsibility pattern).



Intercepting filter architecture



Intercepting filter class diagram

The elements of the pattern are:

- *Filter Manager*: sets up filter chain with filters in correct order. Initiates processing.
- *Filter One*, *Filter Two*, etc: individual filters, which each carry out a single pre/post processing task.
- *Target*: the main application entry point for the resource requested by the client. It is the end of the filter chain.

An example of this pattern in Java, with two filters, could be:

```
public interface Processor
{ public void process(ServletRequest req,
                      ServletResponse res)
  throws IOException, ServletException;
}

public class Filter1 implements Processor
```

```
{ private Processor target;

  public Filter1(Processor t) { target = t; }

  public void process(ServletRequest req,
                      ServletResponse res)
  throws IOException, ServletException
  { // do filter 1 processing, then forward request
    ....
    target.process(req,res);
  }
}

public class Filter2 implements Processor
{ private Processor target;

  public Filter2(Processor t) { target = t; }
```

```
  public void process(ServletRequest req,
                      ServletResponse res)
  throws IOException, ServletException
  { // do filter 2 processing, then forward request
    ....
    target.process(req,res);
  }
}

public class Target implements Processor
{ public void process(ServletRequest req,
                      ServletResponse res)
  throws IOException, ServletException
  { // do main resource processing  }
}

public class FilterManager
{ Processor head;
```

```
  public void setUpChain(Target resource)
  { Filter2 f2 = new Filter2(resource);
    head = new Filter1(f2); }

  public void processRequest(ServletRequest req,
                             ServletResponse res)
  { head.process(req,res); }
}
```

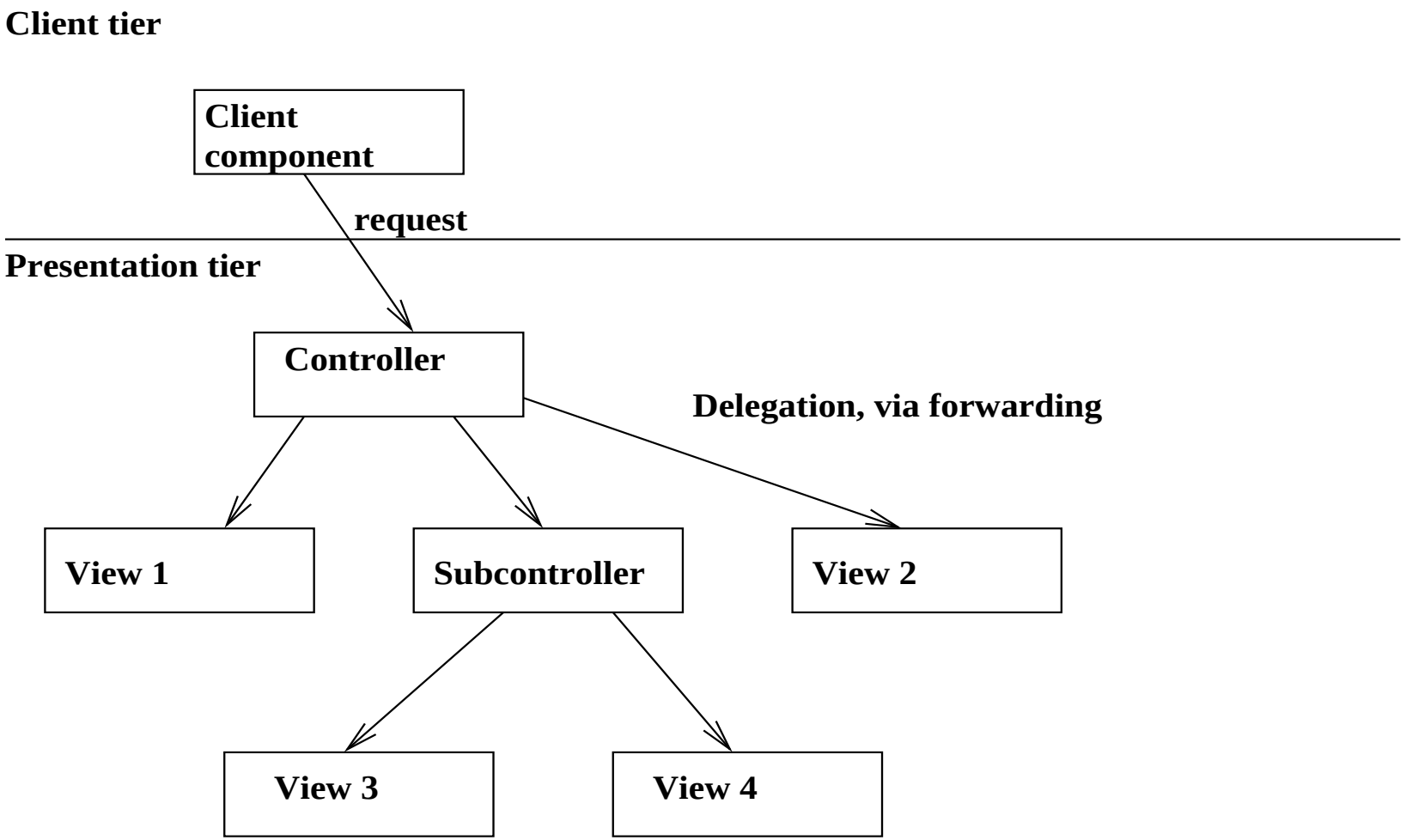
This pattern is provided using standard interfaces and components in Java EE.

Front Controller

This presentation tier pattern has the purpose to provide a central entry point for an application that controls and manages web request handling. The controller component can control navigation and dispatching.

The pattern factors out similar request processing code that is duplicated in many views (eg, the same authentication checks in several JSP's).

It makes it easier to impose consistent security, data, etc, checks on requests.



Architecture of front controller

The elements of the pattern are:

- *Controller*: initial point for handing all requests to the system. It forwards requests to subcontrollers and views.
- *Subcontroller*: responsible for handling a certain set of requests, eg, all those concerning entities in a particular subsystem of the application.
- *View1*, *View2*: components which process specific requests, forwarded to them by the controller.

Some example Java code could be:

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

  public void doGet(HttpServletRequest rq,
                    HttpServletResponse rs)
  throws ServletException, IOException
  { // carry out any common security/authentication checks

    String regC = rq.getParameter("Register");
    if (regC != null)
    { // pass request to register servlet
      dispatch(rq,rs,"RegisterUserServlet");
      return;
    }
```

```
    String editC = rq.getParameter("Edit");
    if (editC != null)
    { // pass request to edit servlet
      dispatch(rq,rs,"EditUserServlet");
      return;
    }
    ...
  }
}
```

This pattern helps to improve security and flexibility, by disallowing direct access to specific components of the system: all requests must pass through the controller.