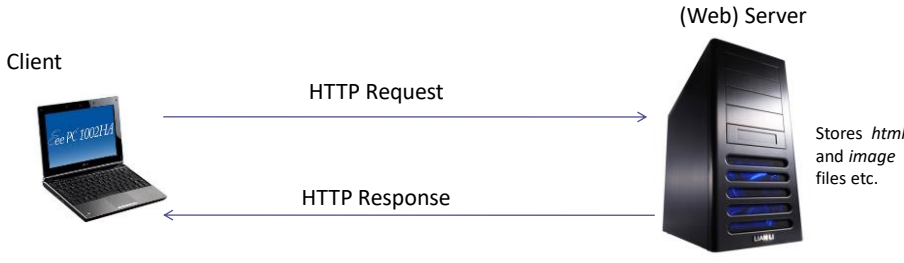
Software Engineering III

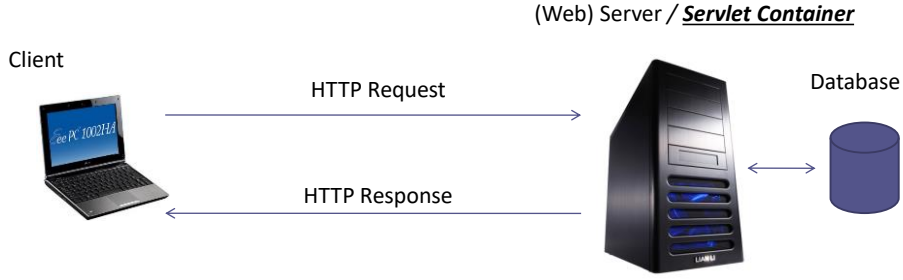
Example Patterns

Web Recap: *Client – Server (Static)*



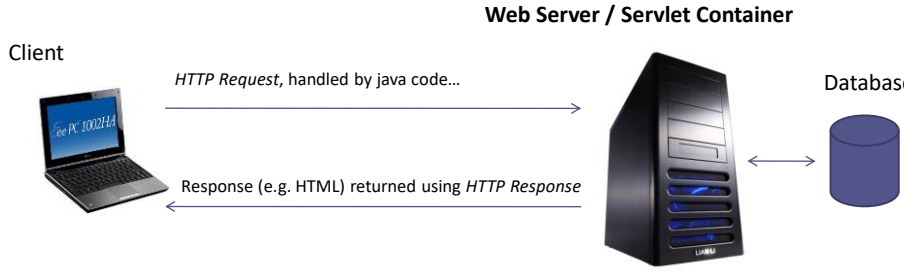
- **HTTP** – **H**yper**t**ext **T**ransfer **P**rotocol, rules used to transfer data across the internet
- **Client** – Any software – PC, Laptop, Smartphone, Tablet
- **Web Server** – Computer system that provides the service of responding to HTTP requests, can "serve" web site pages, images etc. using an HTTP response

Client – Server (Dynamic)



- Servlet Container:
  - A ***servlet*** is a java class that can be directly invoked by a http client –e.g. a web browser.
  - The java code executing in the servlet, for example, may access a relational database and retrieve information that can either directly returned to the http client or processed further into html and then returned to the client

Client – Server (Java)



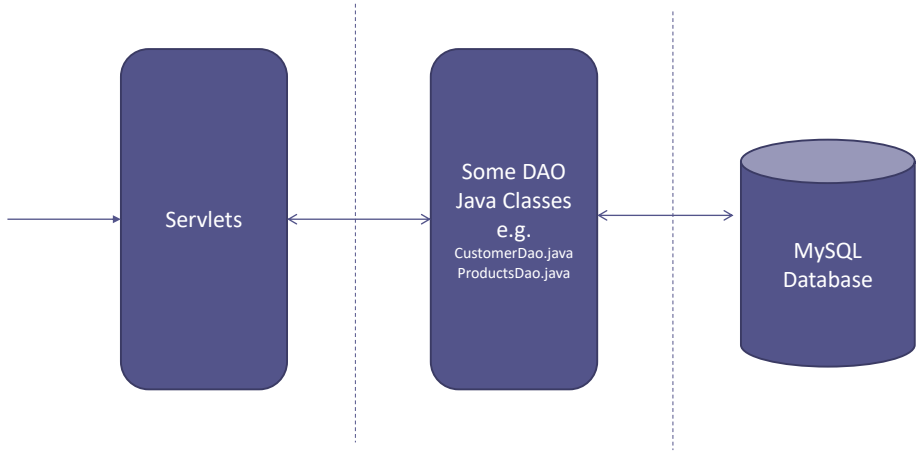
Example Servlet

```
public class UserController extends HttpServlet {


  public UserController() {
      super();
  }

  protected void doGet( HttpServletRequest request,
                        HttpServletResponse response)
                  throws ServletException, IOException
  {
      // code…
  }

  protected void doPost(  HttpServletRequest request,
                          HttpServletResponse response)
                  throws ServletException, IOException
  {
      //code…
  }
}
```

HTTP GET from client handled here

HTTP POST from client handled here

We can use the DAO's and other classes from a Servlet



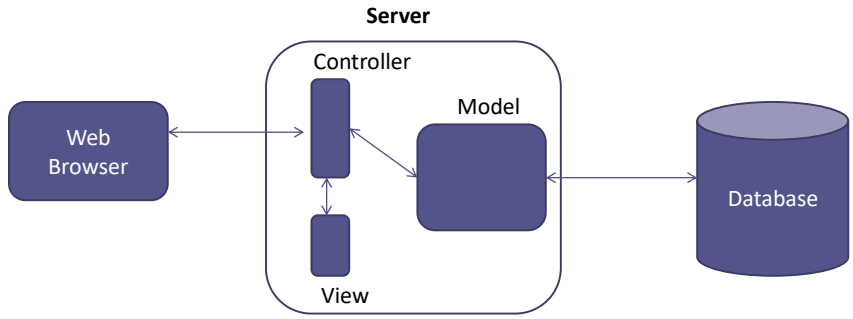
What do we do with the data retrieved / result of request?

The Model View Controller Pattern

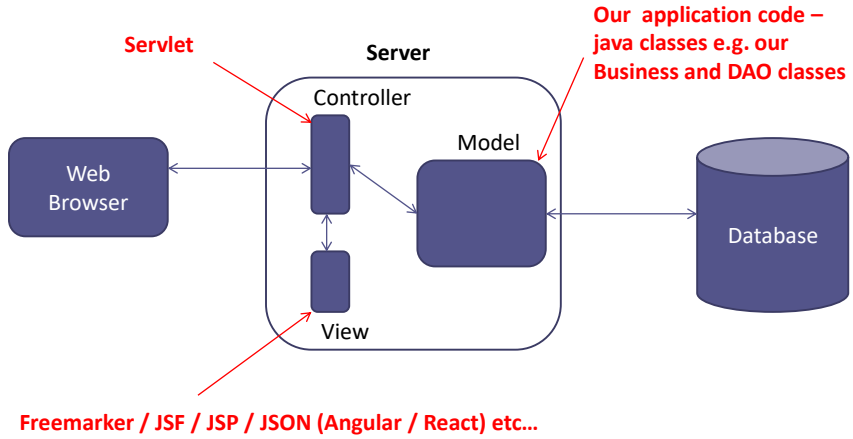
Model View Controller (MVC) Pattern

- A pattern for designing an architecture that separates the Application logic from the User Interface logic
- This allows development, testing and maintenance to be achieved independently

MVC Applied to Web Application Architecture



MVC Applied to Web Application Architecture



Using JSP for the View

Writing a JSP source file…

static_example.html

```
<html>
 <body bgcolor="yellow">
  <center>
   <h2>Hello World</h2>
  </center>
 </body>
</html>
```

static_example.jsp

```
<html>
 <body bgcolor="yellow">
  <center>
   <h2>Hello world</h2>
  </center>
 </body>
</html>
```

Using JSP for the View

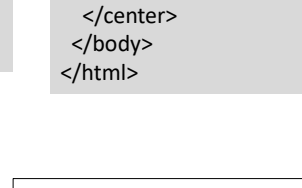
Writing a JSP source file…

static_example.html

```
<html>
 <body bgcolor="yellow">
  <center>
   <h2>Hello World</h2>
  </center>
 </body>
</html>
```

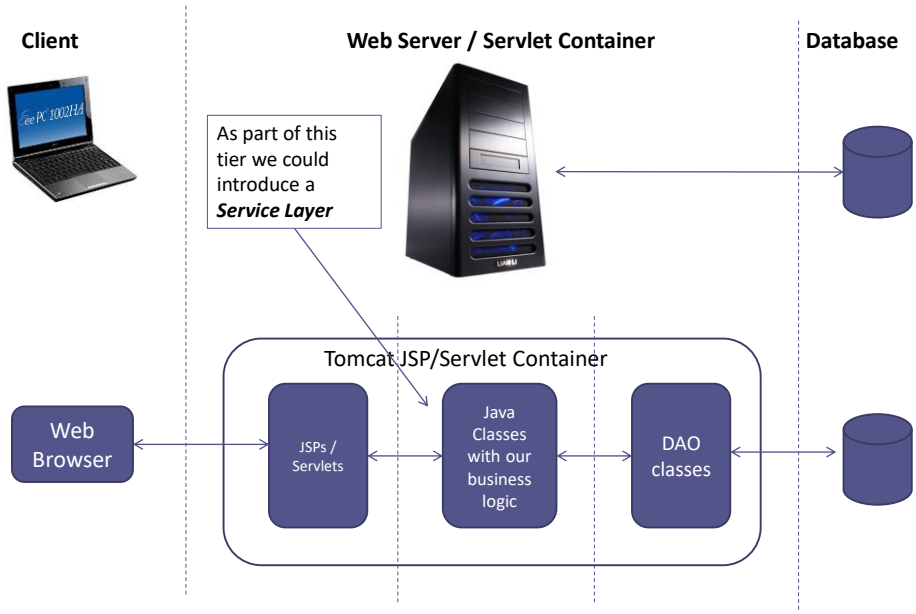
dynamic_example.jsp

```
<html>
 <body bgcolor="yellow">
  <center>
   <h2>Hello World</h2>
    <p> <%= new java.util.Date() %> </p>
  </center>
 </body>
</html>
```



Requesting a JSP file

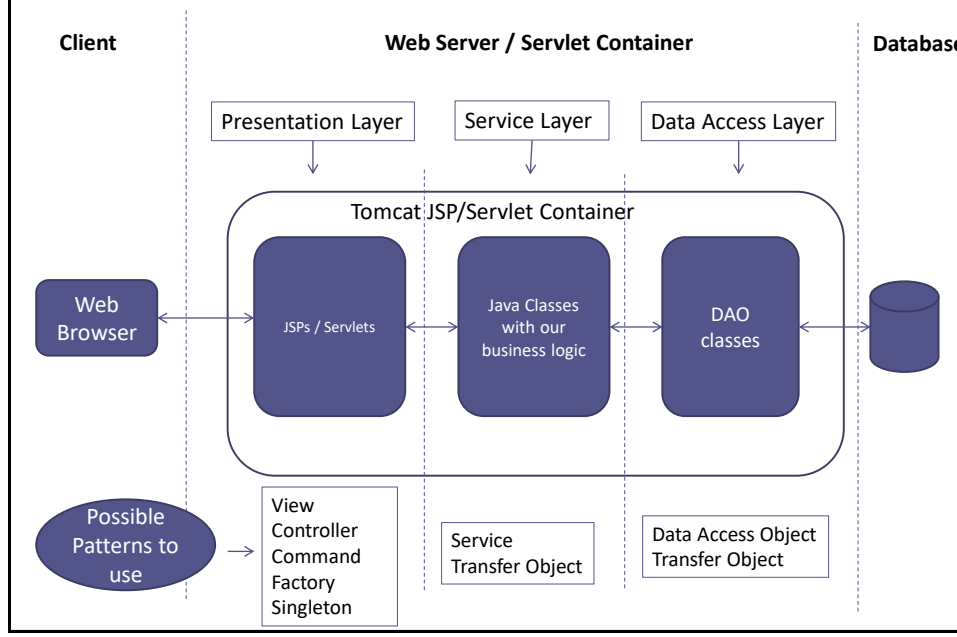
- Browser requests a *.jsp* file instead of a *.html* file from the server
- If you just renamed a *.html* to a *.jsp* file, the first time you loaded/requested the file it would take longer to load than the HMTL version…
- What's happening?
- The first time the .jsp page is requested by the browser, the server reads the file and converts it to a java class and compiles it. Once compiled, it is executed completely as a java program.
- This generated java class is a ***servlet***



Service Layer

- A set of classes, each of which provide a set of application operations.
  - E.g. *UserService* class provides a *login* operation.
- A typical implementation is a Service class that has a corresponding DAO attribute.
  - E.g. *UserService* has an attribute *userDao* which is of type *UserDao.*

```
public class UserService {

    UserDao userDao;

    public User login(String username, String password){

        User u = null;
        try {
```



Singleton Pattern

Singleton Pattern

- Intent: Ensure that a class has only one instance, and provide a global point of access to it
- Only one object will ever exist in memory. It is not possible to create a second object of the same class in memory

Singleton Pattern

```
public class MyServer {

   private static MyServer myServer = null ;

   private MyServer()       { }

   public static synchronized MyServer getInstance() {
      if (myServer == null){
            myServer = new MyServer() ;
      }
      return myServer ;
   }

   // other attributes, methods etc.

}
```

Must be static, as static methods can only access static attributes. Only one myServer object created, even if some other member method created another object of MyServer.

private constructor ensures only methods of the class can call the constructor.

Creates object first time. Returns the same myServer object each time the method is called.

Singleton Pattern

```
public class MyServerTest {

  // MyServer myServer = new MyServer() ;


  MyServer server1 = MyServer.getInstance() ;
  MyServer server2 = MyServer.getInstance() ;


  if (server1 == server2)
      System.out.println("Same server") ;
  else
      System.out.println("Different servers") ;
  }
}
```

Not permitted as constructor is private.

As clients cannot create objects, getInstance() must be a static method. This static method provides a global point of access. It always returns reference to the same object.

Command Pattern

Command Pattern

- Intent: Encapsulate a request as an object, thereby letting you parameterise clients with different requests, queue or log requests.
- E.g. Request from client browser to the server to list all users – we can encapsulate the code to handle this request on the server into a single object

Recall - Example Servlet

```
public class UserController extends HttpServlet {


  public UserController() {
      super();
  }

  protected void doGet( HttpServletRequest request,
                        HttpServletResponse response)
                  throws ServletException, IOException
  {
      // code…
  }

  protected void doPost(  HttpServletRequest request,
                          HttpServletResponse response)
                  throws ServletException, IOException
  {
      //code…
  }
}
```

Using the Command Pattern with JSP/Servlets

Example snippet from a simple *Servlet*

Code inside doGet() method

```
…
else if ( request.getParameter("action").equalsIgnoreCase("listUsers") ) {

        //The user wants a list if all users...

        //Use the UserService class to get all Users...
        List<User> users = new ArrayList<User>();
        users = userService.getAllUsers();

        //Put the list of users into the session so that JSP(the View) can display them...
        session.setAttribute("users", users);
        forwardToJsp = "/listUsers.jsp";

}
…
```

Action requested from web page

New *Servlet* code

```
…

if ( request.getParameter("action").equalsIgnoreCase("listUsers") ) {

    //The user wants a list of all users...
    Command command = new ListUsersCommand();
    forwardToJsp = command.execute(request, response);
}

…
```

Code to handle this is moved to the ListUsersCommand object.

Note the use of a *Command* Interface

```
public interface Command {
   public void execute();
}


public class ListUsersCommand implements Command {

    public String execute(HttpServletRequest request, HttpServletResponse response) {

        UserService userService = new UserService();
        String forwardToJsp = "";
        HttpSession session = request.getSession();

        //Make the call to the 'Model' by using the UserService class to get all Users...
        List<User> users = new ArrayList<User>();
        users = userService.getAllUsers();

        //Put the list of users into the session so that JSP(the View) can pick them up and display them...
        session.setAttribute("users", users);
        forwardToJsp = "/listUsers.jsp";

        return forwardToJsp;
    }

}
```

The Simple Factory Pattern

Simple Factory Pattern

- Intent: Provide a class for creating an instance of one of several possible classes depending on the data provided to it. Usually all classes that it returns have a common parent interface or class, but each performs a task differently.
- E.g. a **CommandFactory** object that can instantiate and return **Command** objects to the calling code.
- The objects that it returns are specific **Command** objects like **ListUsersCommand** – however, they are treated as Command objects because all specific **Command** objects (e.g. ListUsersCommand) implement the **Command Interface**.

Simple Factory Pattern

```
public class CommandFactory {

    ...

  private CommandFactory() {
  }
    ...
    ...

  public synchronized Command createCommand(String commandStr) {
    Command command = null;

    //Instantiate the required Command object...
    if (commandStr.equals("LoginUser")) {
            command = new LoginUserCommand();
    }
    if (commandStr.equals("ListUsers")) {
            command = new ListUsersCommand();
    }
    if (commandStr.equals("ViewUserProfile")) {
            command = new ViewUserProfileCommand();
    }

    //Return the instantiated Command object to the calling code...
    return command;// may be null

  }
}
```

Implemented as a singleton – private constructor

Note, this method returns the object as a *Command*, however, the specific *Command* implementation is what is returned (e.g. ListUsersCommand)

Depending on what information is passed to the *createCommand* method, the factory will create the specific command

Using the *CommandFactory* from our *Servlet*

```
...

if ( request.getParameter("action").equalsIgnoreCase("listUsers") ) {

    //The user wants a list if all users...
    CommandFactory factory = CommandFactory.getInstance();

    Command command = factory.createCommand("ListUsers");

    forwardToJsp = command.execute(request, response);

}

...
```

Front Controller Pattern

Front Controller Pattern

- Intent: Provide a centralised point of access for handling client requests in a web application.
- Our *servlet* acts as *Front Controller* – it accepts all *User* related requests from the client (browser), calls the relevant *command.execute()* method and forwards the server to the next page.

Example: UserController *Servlet*

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException {
        processRequest (request, response);
}

protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException {
        processRequest(request, response);
}

/**
 * Common method to process all client requests (GET and POST)
 */
private void processRequest(HttpServletRequest request, HttpServletResponse response) {
    //Execute the appropriate command…
}
```

Sending a request through doGet()

```
<c:set var="user" value="${sessionScope.user}"/>
<b>Hello <c:out value="${user.firstName}"/>, you are now logged in...</b>
<b>What would like to do?</b>

<br/><br/>

<form action="UserController" method="post">
    <input type="hidden" name="action" value="list" />
        <input type="submit" value="List Users" />
</form>

<a href="UserController?action=viewProfile">View My Profile</a>
```

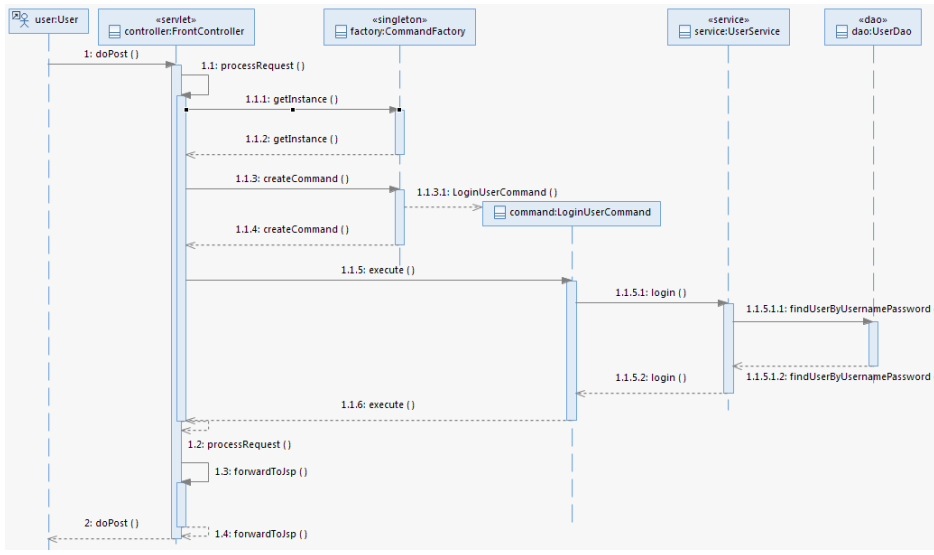
This sends the data using Http POST

This sends the data using Http GET

Application Example using multiple patterns

- Web Application example
- Start with a Login User use case
  1 User enters *username* and *password*
  2 User clicks *Login* button
  3 System validates *username / password*
  4a *Normal flow* - user presented with options page
  4b *Alternative flow* - if invalid, authentication message displayed with option to return to login page

Application Example



Login Page – static html

```
<html>

<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>

<body>

            <form action="FrontController" method="post">
                <table>
                    <tr>
                        <td> Username  : </td><td> <input name="username" size=15 type="text" /> </td>
                    </tr>
                    <tr>
                        <td> Password  : </td><td> <input name="password" size=15 type="password" /> </td>
                    </tr>
                </table>

                <input type="hidden" name="action" value="LoginUser" />
                <input type="submit" value="login" />
            </form>

</body>
</html>
```

This action (command) drives the request through the back end

FrontController

```
import java.io.IOException;

/**
 * Servlet implementation class FrontController
 */
@WebServlet(urlPatterns={"/FrontController"})
public class FrontController extends HttpServlet {
    private static final long serialVersionUID = 1L;
    private static final String LOGIN_ACTION = "LoginUser";

    /**
     * @see HttpServlet#HttpServlet()
     */
    public FrontController() {
        super();
        // TODO Auto-generated constructor stub
    }

    /**
     * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
     */
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        processRequest (request, response);
    }


    /**
     * @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse response)
     */
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        processRequest(request, response);
    }
```

This method processes any *http POST* requests

FrontController

```
/**
 * Common method to process all client requests (GET and POST)
 */
private void processRequest(HttpServletRequest request, HttpServletResponse response) {

    String forwardToJsp = null;
    String action = request.getParameter("action");


    //Now we can process whatever the request is...
    //We just create a Command object to handle the request...
    CommandFactory factory = CommandFactory.getInstance();
    Command command = null;

    try {
        command = factory.createCommand(action);
        forwardToJsp = command.execute(request, response);
    } catch (CommandCreationException e) {
        e.printStackTrace();
        forwardToJsp = "/errorPage.jsp";
    }

    forwardToPage(request, response, forwardToJsp);
}
```

Ask the command factory for the appropriate command object.

Note the controller has no knowledge of which command implementation it is invoking.

The action String from the web page is passed directly to the factory. In our example this would contain "LoginUser".

FrontController

```
/**
 * Forward to server to the supplied page
 */
private void forwardToPage(HttpServletRequest request, HttpServletResponse response, String page){

    //Get the request dispatcher object and forward the request to the appropriate JSP page...
    RequestDispatcher dispatcher = getServletContext().getRequestDispatcher(page);
    try {
        dispatcher.forward(request, response);
    } catch (ServletException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

}
```

For completeness, here is the *forwardToPage* method implementation.

CommandFactory

```
public class CommandFactory {

    private static CommandFactory factory = null;

    private CommandFactory() {
    }

    /**
     * Get an instance of the CommandFactory
     * @return The singleton CommandFactory object
     */
    public synchronized static CommandFactory getInstance() {
        if (factory == null) {      // first time

            factory = new CommandFactory();
        }
        return factory;
    }
```

Implements the singleton pattern

CommandFactory

```
    /**
     *
     * @param commandStr Identifier for the exact Command object required
     * @return The specific Command object requested
     * @throws CommandCreationException
     */
    public synchronized Command createCommand(String commandStr) throws CommandCreationException {

        Command command = null;
        String packageName = "com.sampleapp.command.";

        try {
            commandStr = packageName + commandStr + "Command";

            //...
            Class<?> theClass = Class.forName(commandStr);

            //...
            Object theObject = theClass.newInstance();

            command = (Command) theObject;
        } catch (InstantiationException e) {
            throw new CommandCreationException("CommandFactory: " + e);
        } catch (IllegalAccessException e) {
            throw new CommandCreationException("CommandFactory: " + e);
        } catch (ClassNotFoundException e) {
            throw new CommandCreationException("CommandFactory: " + e);
        }

        //Return the instantiated Command object to the calling code...
        return command;     // may be null
```

The *CommandFactory* object itself still has no explicit knowledge of the command implementation it is instantiating.

It uses the reflective method *theClass.newInstance()* which invokes the zero argument constructor.

LoginUserCommand

```
public class LoginUserCommand implements Command {

    @Override
    public String execute(HttpServletRequest request, HttpServletResponse repsonse){

        UserService userService = new UserService();
        String forwardToJsp = "";

        String username = request.getParameter("username");
        String password = request.getParameter("password");

        //Check we have a username and password...
        if (username != null && password != null){

            //Make call to the 'Model' using the UserServive class to login...
            User userLoggingIn = userService.login(username, password);

            if (userLoggingIn != null){

                //If login successful, store the session id for this client...
                HttpSession session = request.getSession();
                String clientSessionId = session.getId();
                session.setAttribute("loggedSessionId", clientSessionId);

                session.setAttribute("user", userLoggingIn);

                forwardToJsp = "/loginSuccess.jsp";
            }
            else{
                forwardToJsp = "/loginFailure.jsp";
            }
        }
        else {
            forwardToJsp = "/loginFailure.jsp";
        }
        return forwardToJsp;
    }

}
```

Accepts the client request and invokes the business service

UserService

```
public class UserService {

    public User login(String username, String password){

        User u = null;
        try {
            UserDao dao = new UserDao();
            u = dao.findUserByUsernamePassword(username, password);
        }
        catch (DaoException e) {
            e.printStackTrace();
        }
        return u;

    }

    public List<User> getAllUsers(){

        List<User> users = null;

        try {
            UserDao dao = new UserDao();
            users = dao.findAllUsers();
        }
        catch (DaoException e) {
            e.printStackTrace();
        }
        return users;

    }
}
```

In this instance, the service just calls through to the DAO object.

UserDao

```
public User findUserByUsernamePassword(String uname, String pword) throws DaoException {

    Connection con = null;
    PreparedStatement ps = null;
    ResultSet rs = null;
    User u = null;
    try {
        con = this.getConnection();

        String query = "SELECT * FROM USER WHERE USERNAME = ? AND PASSWORD = ?";
        ps = con.prepareStatement(query);
        ps.setString(1, uname);
        ps.setString(2, pword);

        rs = ps.executeQuery();
        if (rs.next()) {
            int userId = rs.getInt("ID");
            String username = rs.getString("USERNAME");
            String password = rs.getString("PASSWORD");
            String lastname = rs.getString("LAST_NAME");
            String firstname = rs.getString("FIRST_NAME");
            u = new User(userId, firstname, lastname, username, password);
        }
    } catch (SQLException e) {
        throw new DaoException("findUserByUsernamePassword " + e.getMessage());
    } finally {
```

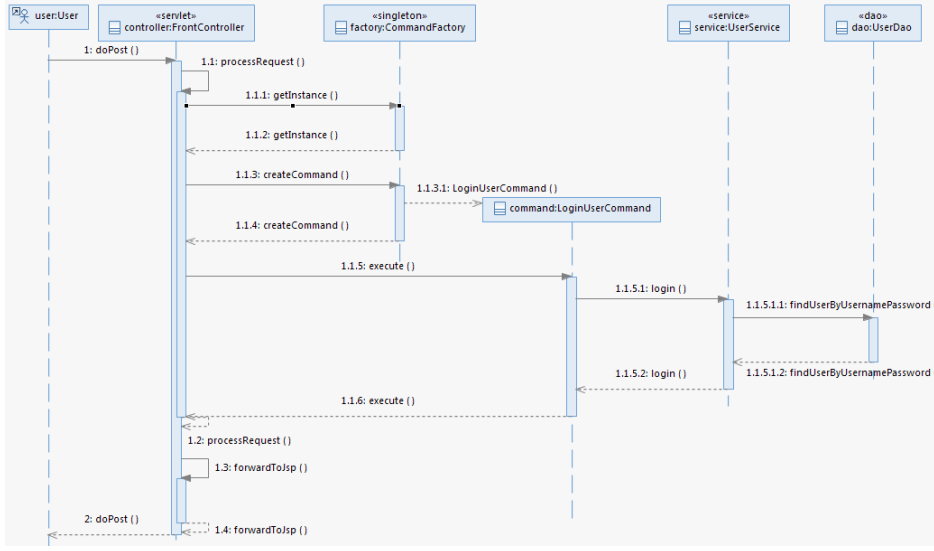
In this example, we are simply using JDBC, however, we could implement code here that utilises an *Object Relational Mapping* framework.

Login Successful View – *loginSuccess.jsp*

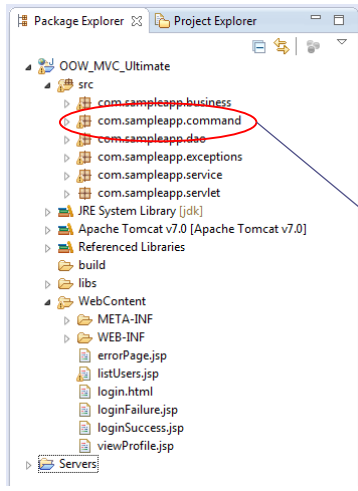
```
<html>

<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>

<body>
    Servlet forwarded to me... loginSuccess.jsp

    <br/><br/>

    <c:set var="user" value="${sessionScope.user}"/>
    <b>Hello <c:out value="${user.firstName}"/>, you are now logged in...</b>
    <b>What would like to do?</b>

    <br/><br/>

    <form action="FrontController" method="post">
        <input type="hidden" name="action" value="ListUsers" />
        <input type="submit" value="List Users" />
    </form>
    <form action="FrontController" method="post">
        <input type="hidden" name="action" value="ViewUserProfile" />
        <input type="submit" value="View My Profile" />
    </form>

</body>

</html>
```

The view accesses the data made available by the model

This gives us a separation of the layers



Eclipse Dynamic Web Project – example codebase



Our package structure reflects our layered design

Each package contains the set of classes appropriate to that layer