Universita' degli Studi di Bologna
Facolta' di Ingegneria

Anno Accademico 2007-2008

Laboratorio di Tecnologie Web

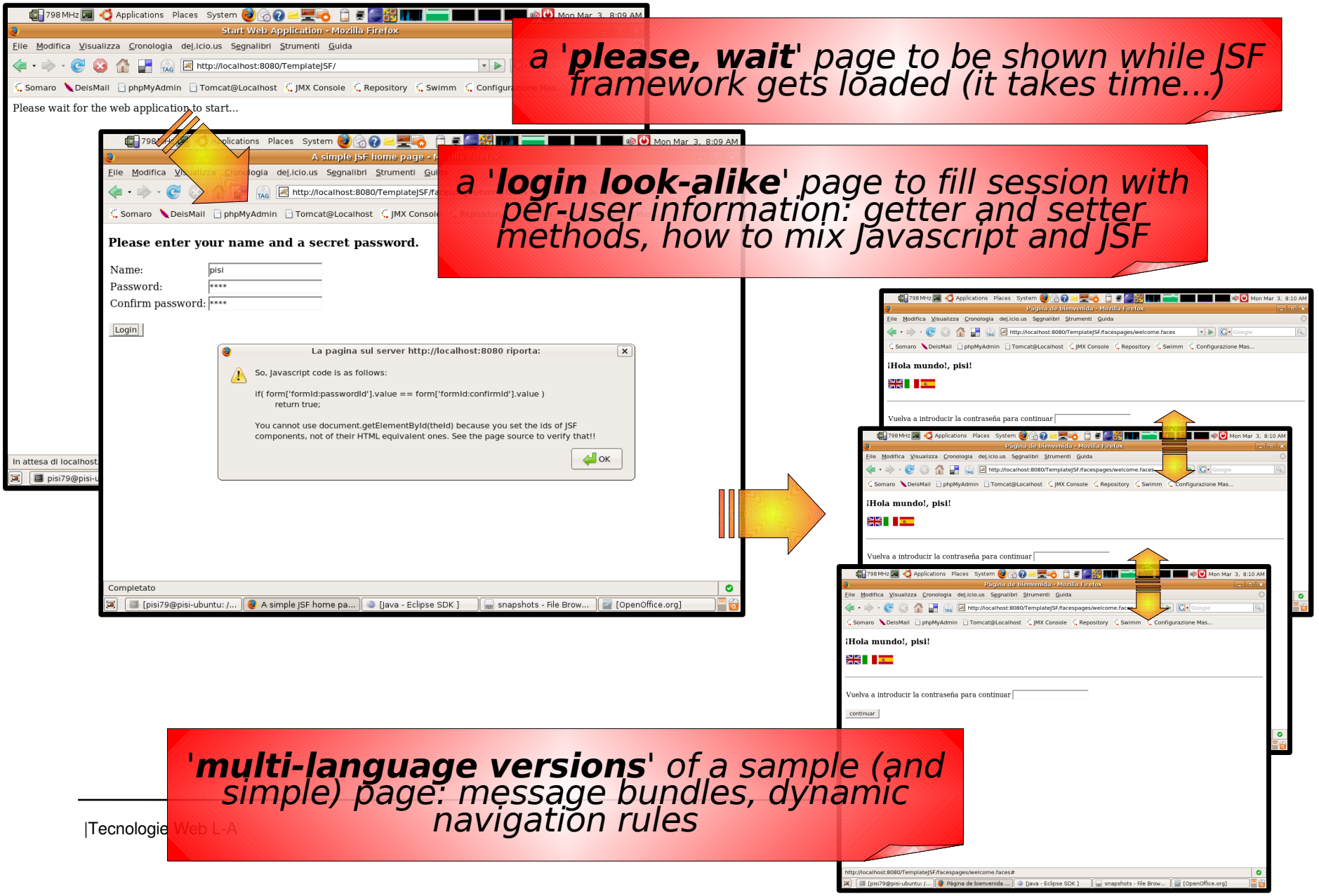
Sviluppo di applicazioni web

JSF

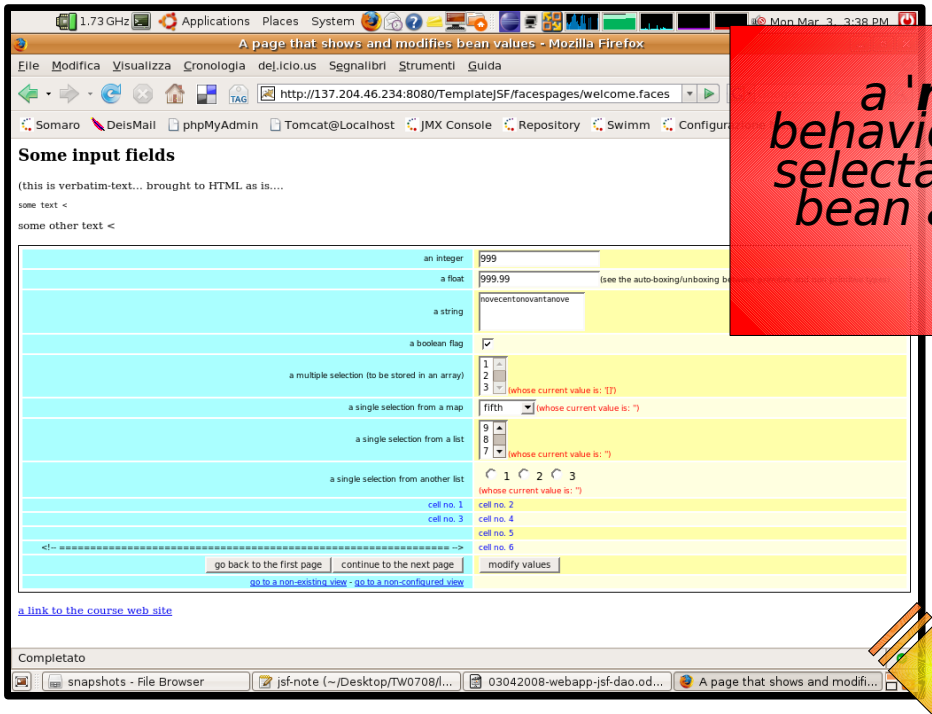
Pattern DAO

http://www-lia.deis.unibo.it/Courses/TecnologieWeb0708/

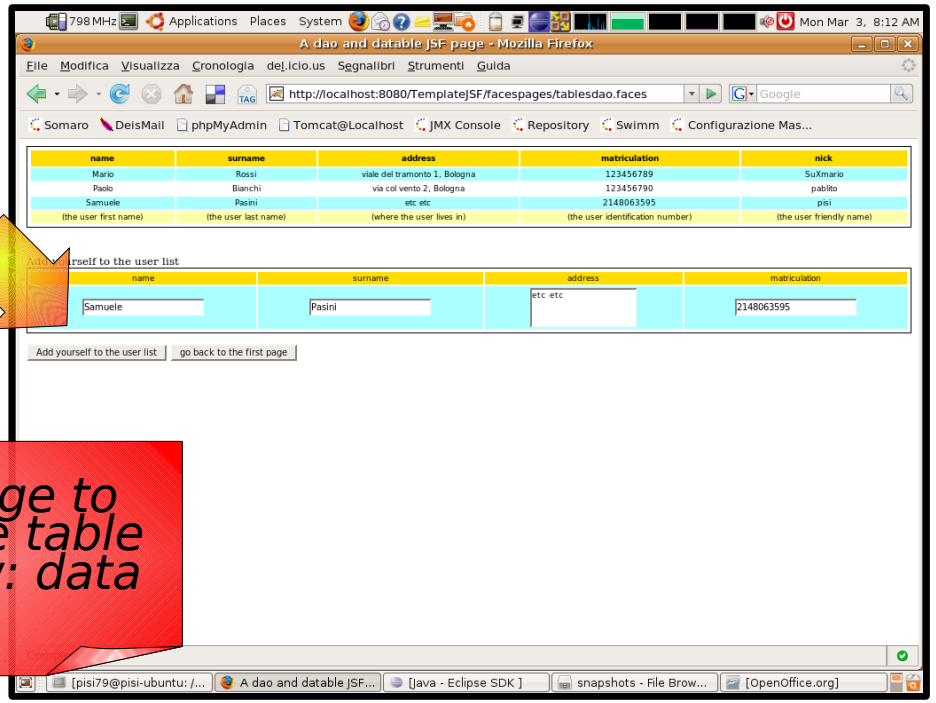
We will go through these pages (1/2)



We will go through these pages (2/2)



*a '**miscellaneous**' page to analyze behaviour of form controls and commands: selectable items, usage of a backing bean, bean and array initialization, panel grids, CSS styling*



*a '**database view-and-modify**' page to show content of an existing database table and add data to it, updating the view: data table, columns, DAO pattern*

To do so...

> Configure (if needed) and **launch Tomcat** (as usual)

> **Download** and **import** the '*TemplateJSF.zip*' project to your Eclipse workspace (as usual)

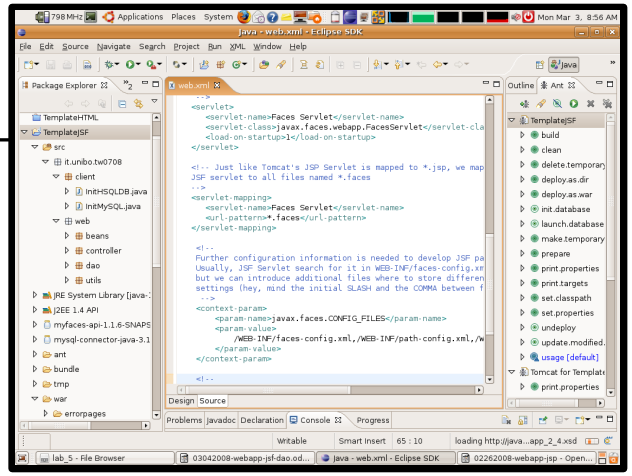
> This time, before deploying project to Tomcat...

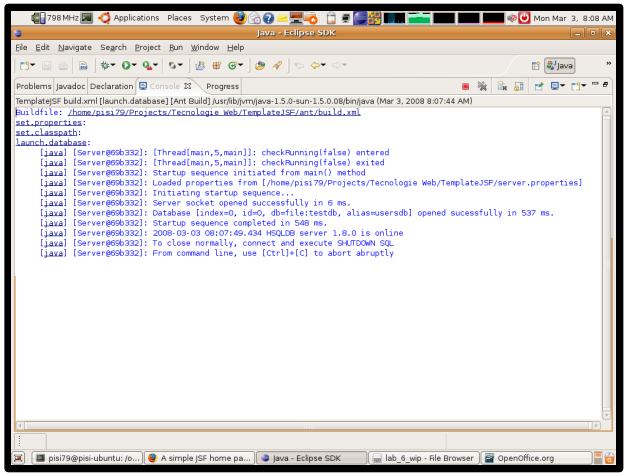
- **launch** the **db** server (*launch.database* target)
- **init** database **tables** (*init.database* target)

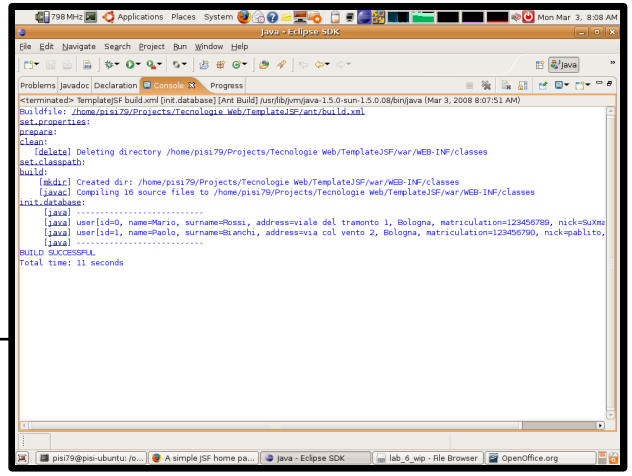
> Now you can **deploy** the web application

- how about doing that by using Tomcat build file?

(see dependencies and invocations across different ANT targets / build files and... further ANT extensions with non-core tasks such as try/catch...)







Project issues

> **Tomcat does not provide out-of-the-box support for the JSF framework** (at least, version 5.5 doesn't)

- project build-path must include both JSF API and implementations
- JSF framework gets deployed on Tomcat with the web application

> **Tomcat does not provide out-of-the-box support for database servers** (where the data gets stored) **and connectors** (the libraries used to access database from within Java code)

- project build-path must include the database connector (JDBC driver)
- the connector too gets deployed with the web application

> Of course you can tweak Tomcat to include your db connector and JSF support right from the start...

> **Database server runs apart** from Tomcat and any other kind of application...

- For the sake of simplicity, we use the in-memory Hypersonic DB (HSQLDB)
  - Incidentally, we start it from Eclipse (or, better, by calling an ANT target)
  - Incidentally, both server and connector are packaged in *hsqldb.jar*

> Notice the difference between ANT and Eclipse build-paths to run tasks....

JSF stuff

> JSF has these three parts:

- a set of pre-fabricated **UI components**
- an **event-driven programming model**
- a component model that enables **third-party extensions** and contributions

> You can run JSF 1.1 applications with any servlet container that supports the Servlet 2.3 and JSP 1.2 specifications

- Tomcat 5.5.x suits these needs...
- ...but, unfortunately, JSF 1.2 specs require JSP 2.1 support (.. only available in Tomcat 6.x) → so we're gonna use JSF 1.1 

> JSF specs are standard (and now part of J2EE), but **implementation can come from any of several vendors**

- whereas Apache MyFaces was prob'ly the best choice for JSF 1.1.x, Sun's implementation is best for the JSF 1.2 (**API commands what to do, but vendor implementations may sometime be defective... and it happens, really...**)

JSF programming

> To compile Java code that uses JSF classes (e.g., the `FacesContext`)...

- class-path must include the following JAR files:
  - *servlet-api.jar* (Servlet API specification *)
  - *jsp-api.jar* (JSP API specification *)
  - *jsf-api.jar* (JSF API specification *)
  - *jsf-impl.jar* (JSF API implementation from some vendor **)

** You only need API specification at compile-time (the web server's servlet container provides API implementation for run-time execution)*

*** Leveraging a legacy servlet container (like Tomcat 5 is), you are in charge of providing the JSF API implementation that the servlet container will use at run-time*

> We are gonna use Apache MyFaces 1.1.6

- Tomcat 5.5.x compatibility
- Help features thanks to Eclipse' JEE plugins...

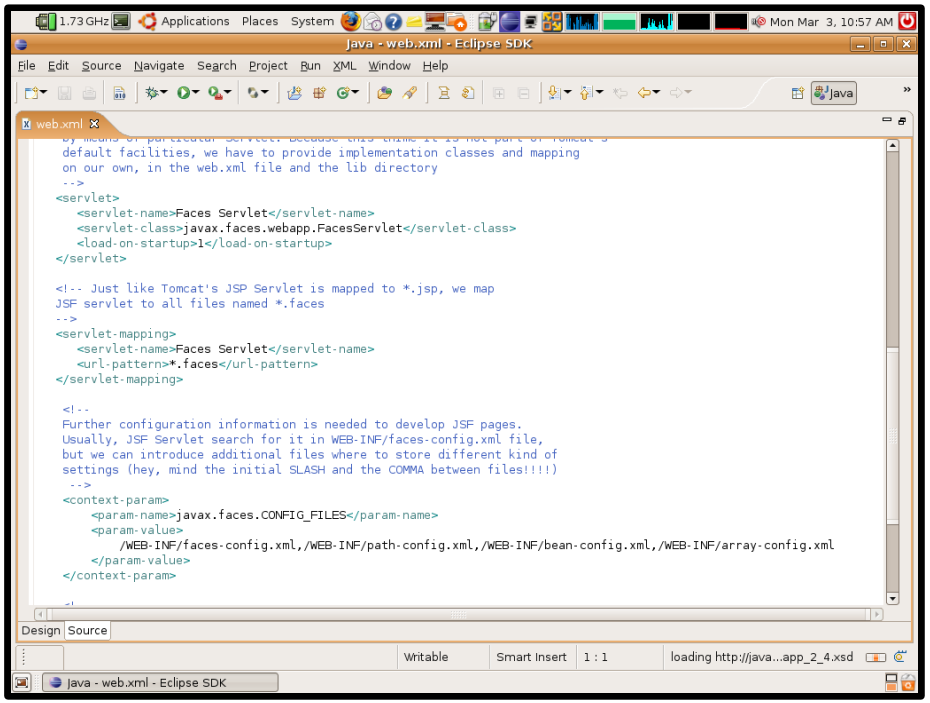
Ok, I see my Eclipse has by far more features and plugins than yours have... for your own convenience, on your home PCs, ***install Eclipse IDE for Java EE Developers*** *(http://www.eclipse.org/downloads)*

JSF and web app configuration files

> */WEB-INF/web.xml* maps **.faces* URIs to `Faces Servlet` handling (at least: this is the default configuration option...)

- when browser requests the *http://.../.../index.faces* URL, the extension *.faces* is mapped to the *.jsp* file with the same base name
- the *`Faces Servlet`* handles the jsp page and helps the container compiling it into a Servlet class
- container load this file

(other mappings are possible, but this process is necessary all the same to make JSF run on top of Servlet technology)



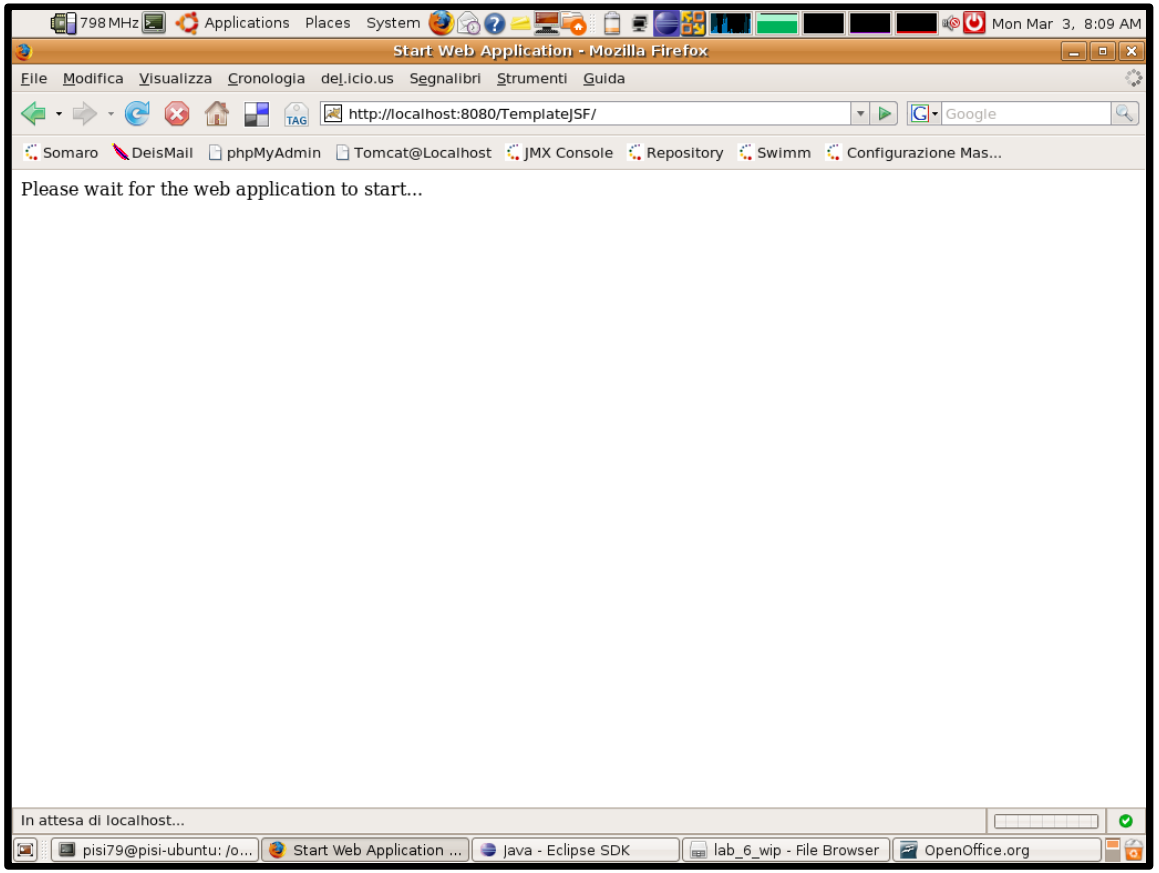
> By default, JSF configuration is kept in the */WEB-INF/faces-config.xml* file

- you can specify additional files in your *web.xml* to keep the various aspects separate (beans, navigation, resources, ..)

Have a start

> Conceptually, you need a JSF page for each browser screen
  - *.jsp* extension requires less configuration effort when used with Tomcat

> **JSF framework takes time to load....**
  - this is why we are not going to have users wait in front of a blank screen
  - web app first page is a traditional *index.html* one: the browser renders it immediately and then the page redirects user to the first JSF page (showing a '*please, wait*' message, meanwhile)

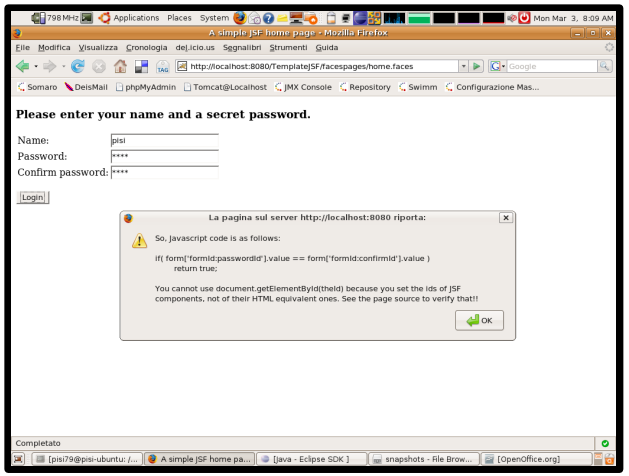


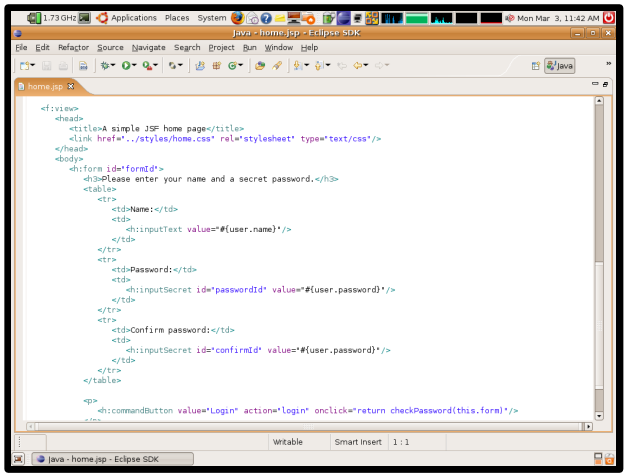
home.jsp

> **All JSF tags are contained inside an *f:view* tag**

> Much of the page is similar to an HTML form

- instead of HTML form tags, form controls are rendered by JSF tags inside the *h:form* one
- form has no associated action, but *JSF commands* inside it have, instead
- **Javascript code execution can still be fired by events** (let's have a look at DOM navigation inside it...)
- Text that is not part of JSF tags is simply passed through; JSF tags are converted to HTML by means of their corresponding Java components (provided by the *jsf-impl.jar* library)





> JSF defines two sets of tags: **core** and **html markup**:

<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>

- core tags are independent of the rendering technology being used

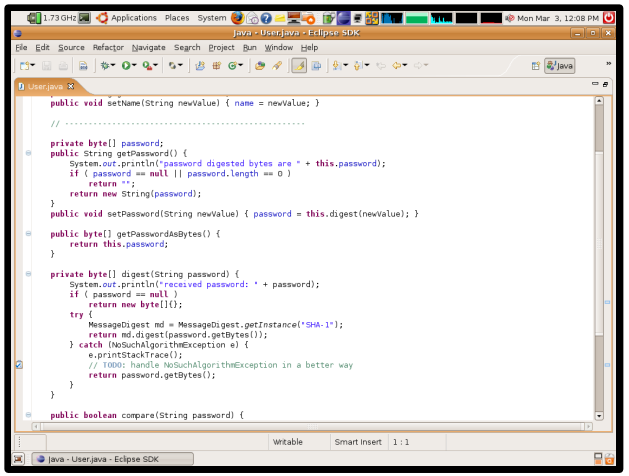
→ you need *<f:view>* in all cases

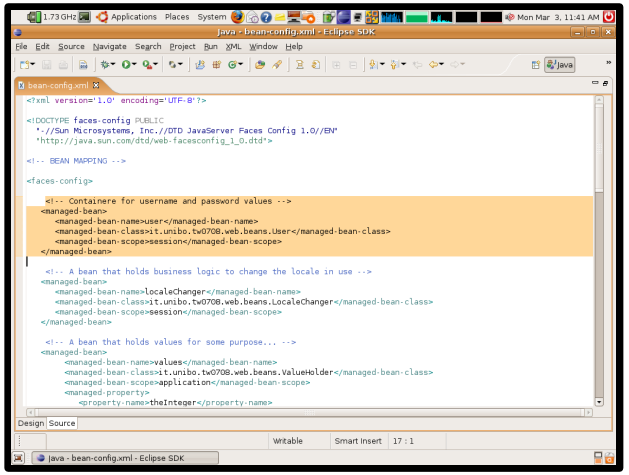
→ you may use a markup rendering library different than *<h:....>* to render pages on alternative client technology (e.g., HTML, PDF, ...)

The 'user' session-scoped bean

> *home.jsp* maps its form controls to the fields of a session-scoped bean

- an object that simply exposes properties (for **value binding expressions** by means of getters/setters) and event handling methods (for **method binding** expressions) according to a standard naming convention
- getters and setters can hold transformation logic *(or business logic too, though this is generally a bad bad habit...)* → see how to avoid storing password values





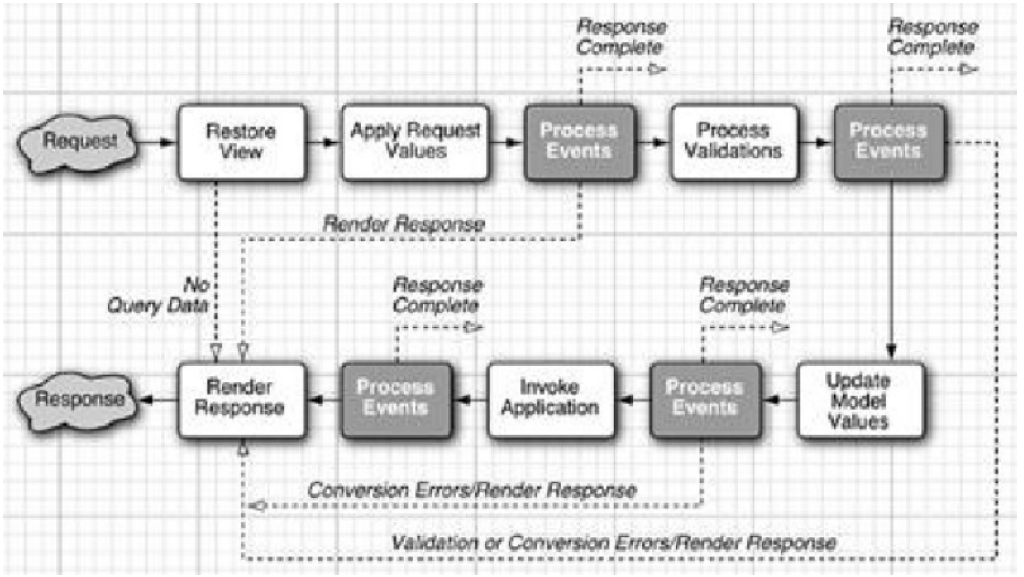
> Bean configuration (and initialization, when needed) is in */WEB-INF/bean-config.xml* file

> **Warning!** This is not a 'login' page: user password is just stored for future reuse, but validates against nothing. No security constraint is set up in web.xml

Model-view issues and navigation rules

> According to JSF page life-cycle, form command does not target another page to be rendered, but the current one:

- after validation and conversion, values are applied to the model
- then, navigation rules tell the JSF implementation which next page to send back to the browser



> Navigation rules are in */WEB-INF/path-config.xml* file

- in this case, navigation action is hard-coded in the page

<h:commandButton value="Login" ***action="login"*** *onclick="return checkPassword(this.form)"/>*

- alternatively, it can be the result of evaluating the bean method declared in the JSF command by means of the ***#{...}*** synopsis

> JSF command tags support **DHTML event** via **pass-through attributes**

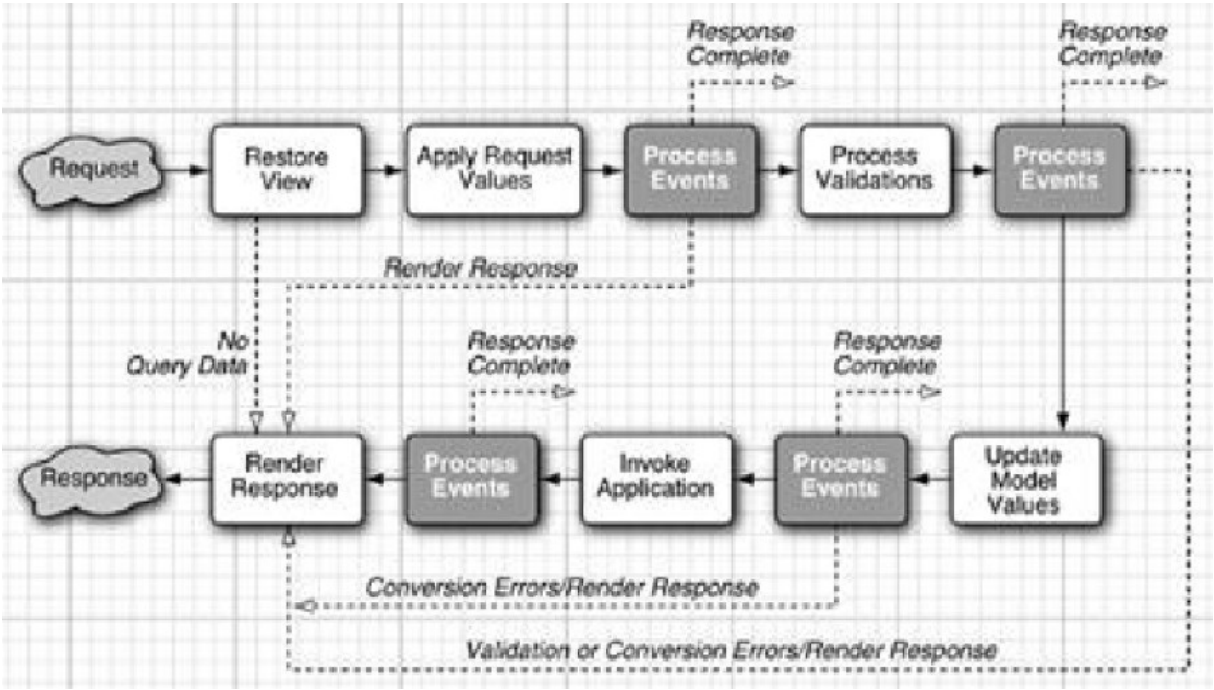
- e.g., JSF *onclick* is directly mapped to the DHTML's *onclick* attribute

More about JSF phases

> A request coming from a JSF page is an HTTP POST one and targets the same page that is issueing it

> JSF specs define **six distint phases** (the white blocks in the figure below) interleaved by processing stages

- on the server, page **view is restored** to find the beans, components and logic that are interested in processing
- request **values are applied to the view** (not the model!!!!)
- only after **validation** the framework can assert that it is safe to **update the model**
- **application is invoked** to tell the **response to render** back to the user



More about value binding expressions

> Many JSF **UI components** hava **attributes** that let you specify either **a value** or a **binding to a value** from a bean property:

```
<h:outputText value="Hello, World!"/>
<h:outputText value="#{user.name}"/>
```

...or even mix them up:

```
<h:outputText value="Hello, #{user.name}!"/>
```

> The **value binding** can be used **both for reading and writing** the bean property!

- when used in an input component (such as *h:inputText*), for instance...
  - → the getter is invoked when the component is rendered (***rvalue* mode**)
  - → the setter is invoked when the response is processed (***lvalue* mode)**
- JSF value binding expressions are different from JSTL/JSP ones (the latter ones, indeed, always invoke property getters)!
  - → this is why JSF uses the *#{...}* notation, instead of the *${....}* one
- you can use only a limited set of operators inside value binding expressions
- you cannot invoke bean methods inside attributes that expect value bindings

welcome.jsp

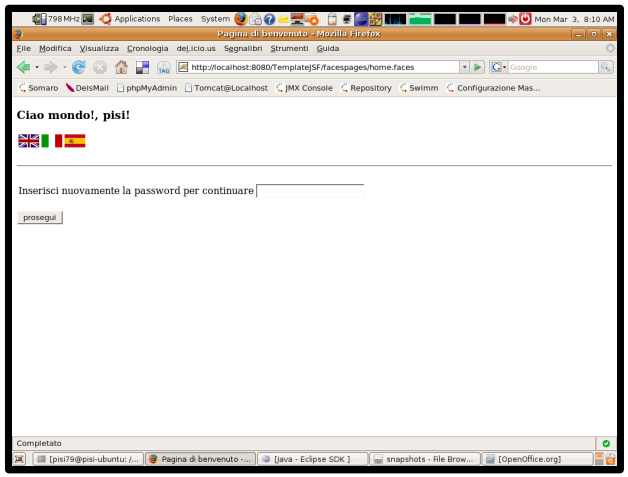
> A page that uses the same session-scoped bean to produce customized output

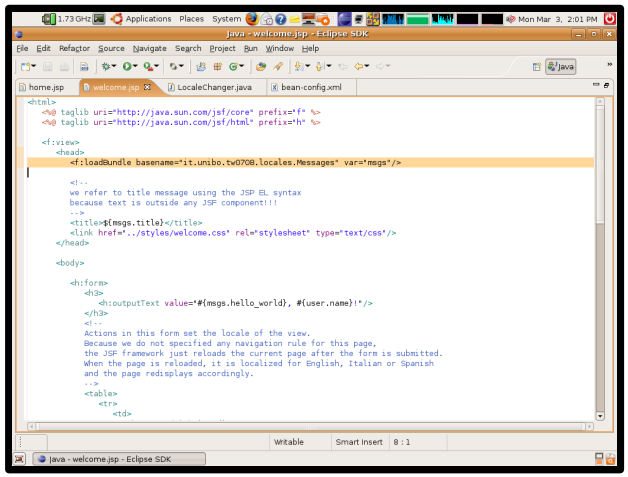
> Page also shows how easy is to load message bundles and achieve internationalization within JSF pages

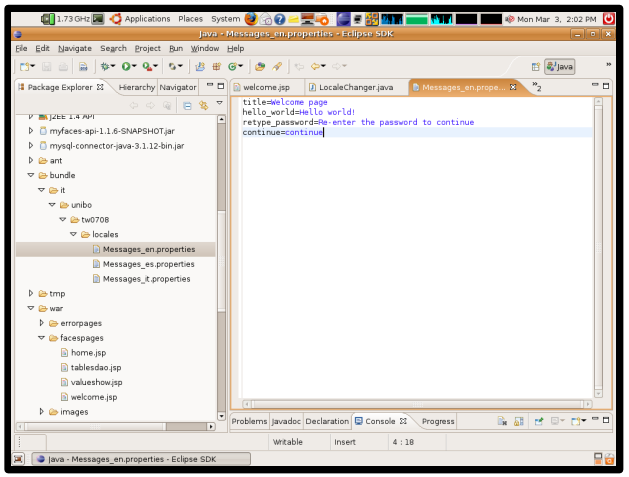
- bundles obey the same naming convention (and path structure) of Java classes

  <f:loadBundle basename="it.unibo.tw0708.locales.Messages" var="msgs"/>

- the locale in use corresponds to the suffix of the selected message bundle
- of course, you can load multiple bundles at the same time (for instance to keep page strings and error messages separated from each others)





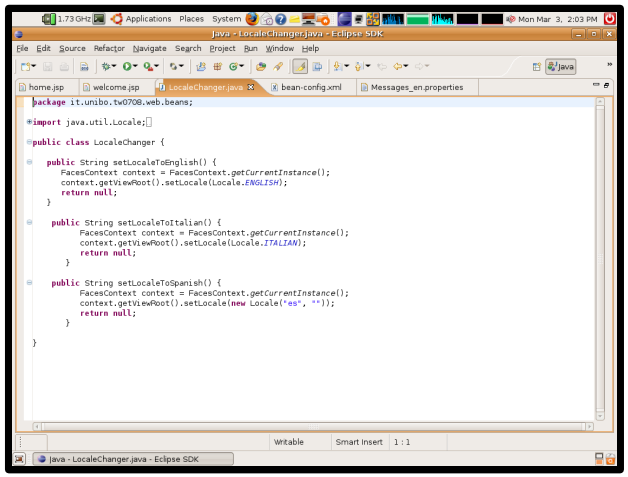


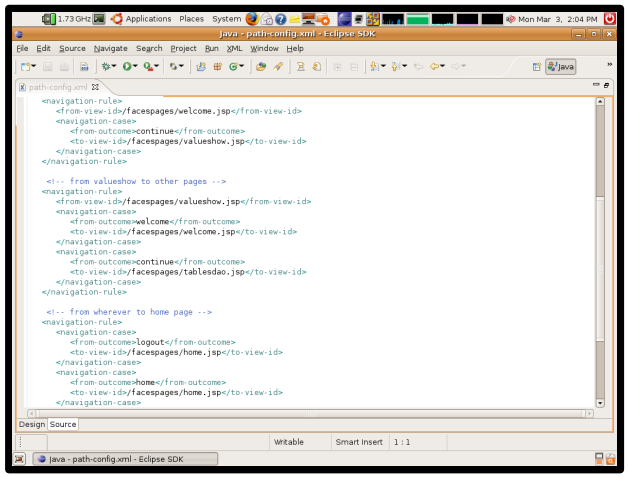
welcome.jsp

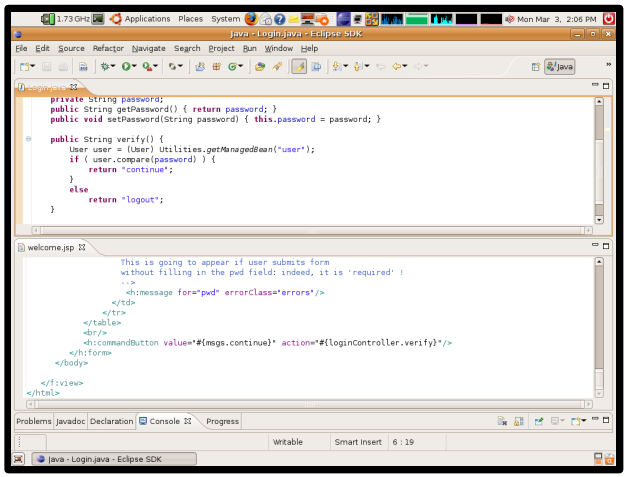
> *LocaleChanger* class shows how to programmatically manage locales
  - just another bean configured in */WEB-INF/bean-config.xml*
  - its methods are called upon executing JSF commands (that expect method binding expressions)

> See how page embeds graphic images within interactive UI components

> Finally, an example of **dynamic navigation**:
  - the **form command binds a method that generates different navigation actions**, according to password validation (against user bean that still is in session), by invoking a particular method from another JSF bean (*loginController*)



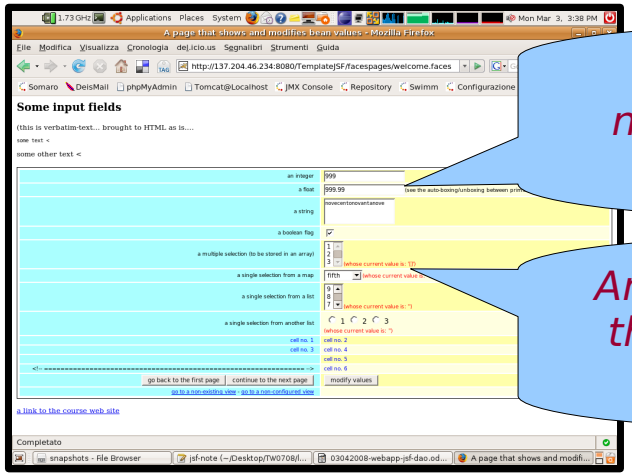




Miscellaneous page: valueshow.jsp (1/3)

> Just a gathering of JSF features *(sometimes unrelated with each other...)*:

- *<h:panelGrid>* allows easy placing of elements inside a grid
  - CSS styling turns useful in row and column presentation
  - every JSF tag constitutes a cell-grid: to group elements inside a single cell you MUST place them inside an *<h:panelGroup>*
- `ValueHolder` class provides a **backing bean** (named *values* in the (*/WEB-INF/bean-config.xml* file) for the page properties
  - the XML config file also shows how to initialize bean properties
- bean is application-scoped: properties are available to different clients



Backing beans are not only useful, but sometimes also necessary for validators and event handlers to access the actual components on a form

Anyway, they can also be abused easily: you should not use the user interface components as a repository for business data and logic

Miscellaneous page: valueshow.jsp (2/3)

> Just a gathering of JSF features *(sometimes unrelated with each other...)*

- *<h:select.... >* tags provide value binding with bean properties and let choose options from a set of *SelectItem* objects, defined in one of several ways...
  - HTML-hard coded:

    <f:selectItem itemLabel="1" itemValue="ONE"/>
  - Obtained from Java object methods:

    <f:selectItems value="#{values.listItems}"/> from method `SelectItem[] getListitems()` in the object of class `ValueHolder` that corresponds to the *values* bean
  - Obtained from static application resources, mapped in *array-config.xml*

*A single **f:selectItems** tag is generally better than multiple **f:selectItem** ones, but using **`SelectItem`** Java objects couples code to the JSF API!*

*Using values from a **java.util.Map** is an attractive alternative: JSF automatically turns map keys into item labels and map values into item values! But you have to pay attention to the item ordering !!*
(for instance, using a TreeMap can lead to alphabetical ordering of the days of the week, instead of presenting them in the order you wanted them to show).

Miscellaneous page: valueshow.jsp (3/3)

> The JSF framework fires action events and invokes actions when an *<h:commandButton>* or an *<h:commandLink>* is activated

- JSF generate JavaScript code to make controls act like submit buttons

> The *<h:outputLink>* tag, instead, generates an HTML anchor element that directly points to a resource (e.g., an image a or a web page):

- **clicking on the generated link takes you to the resource without further involving the JSF framework**
- the *value* attribute is used for the anchor's *href* attribute, and the contents of the *h:outputLink* body are used to populate the body of the HTML anchor element
- no Javascript code is generated

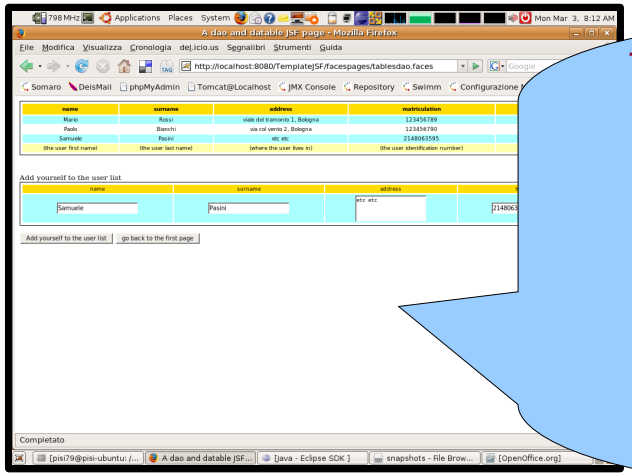
> Compare the use of *<f:verbatim>* tags to direct HTML writing to show non-JSF page elements:

- will the dash ' - ' appear the same, if it was just plain HTML between the two *h:outputLink*s?
- and what about text in the upper part of the page?
- how it comes?

using DAO and JSF: tablesdao.jsp

> A simple page that show elements of a database table by means of...
  - the JSF *<h:dataTable>* tag
  - a *bean* object (in the Java sense, not necessarily a JSF bean!) to represent a table tuple
  - CSS styling for the dataTable appearance

> Page, then, show another table to let you add new database entries by...
  - placing values in cells, thanks to tags that use **value-binding expressions**
  - providing input control elements for the fields to edit (value binding is r/w !!)
  - associating the 'add' command to a method from the *manager* bean object



*The **value** attribute of **h:dataTable** represents the data over which the tag iterates.*

That data must be one of the following:

- *an* **array**,
- *an instance of* **java.util.List**,
- *an instance of* **java.sql.ResultSet**,
- *an instance of* **javax.servlet.jsp.jstl.sql.Result**,
- *an instance of* **javax.faces.model.DataModel**

The DAO pattern: fundamentals

> The JDBC API enables standard access and manipulation of data in persistent storage, such as relational database, by using SQL statements

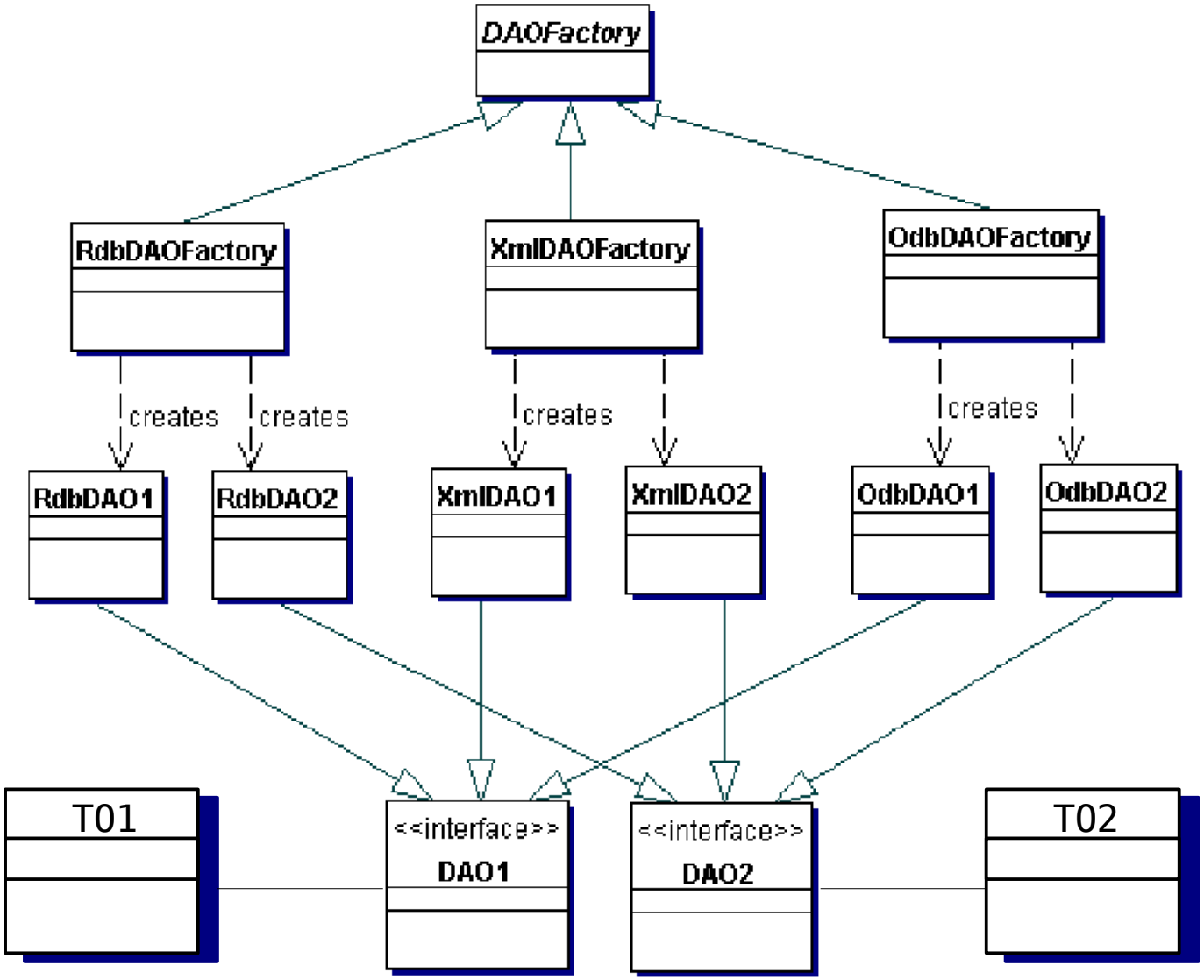
> However, **the actual syntax and format of SQL statements may vary depending on the particular database product in use**

- SQL statements get part of the application's Java code and these **dependencies make it difficult to migrate the application from one type of data source to another**

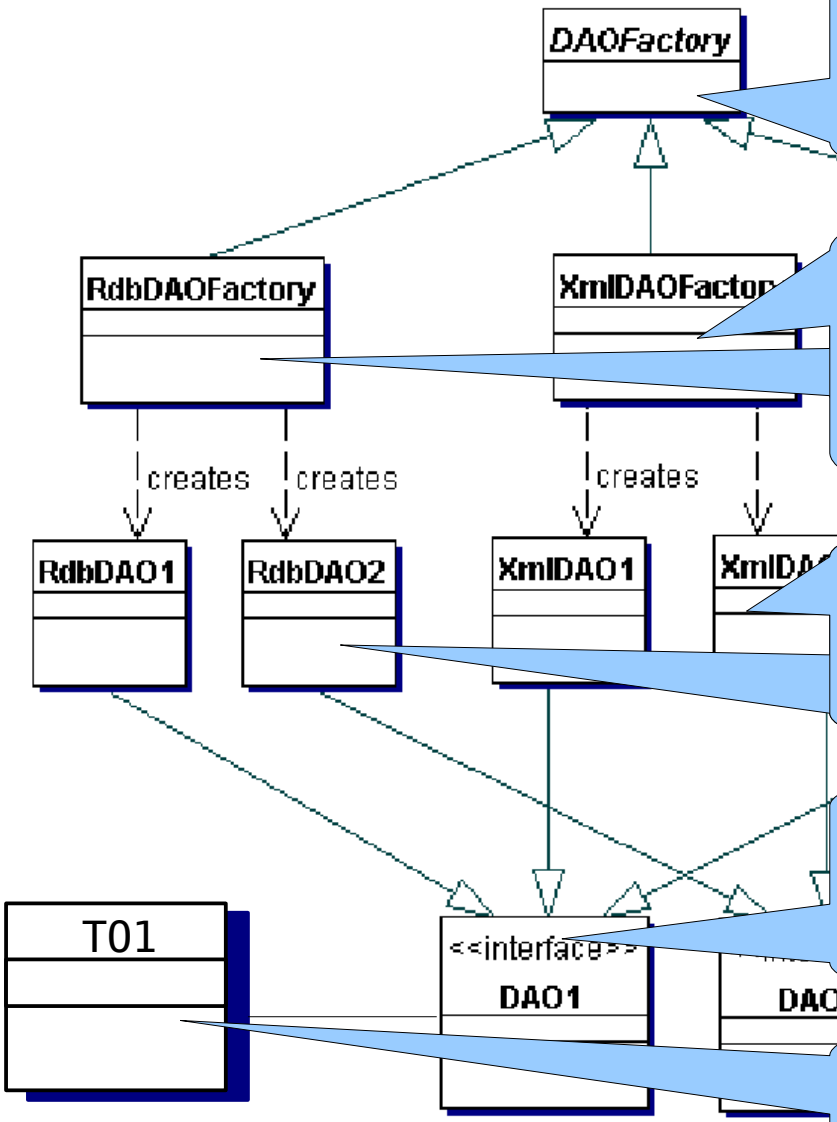
> Using ***Data Access Objects*** (DAO) abstracts and encapsulates access to the data source in a few, well-defined, components that can manage the connection with the data source to obtain and store data

- DAO pattern is often used together with the *Abstract Factory* (to **choose among different data source implementations**) and the *Factory Method* ones (to **instantiate DAO objects**)
- Besides, the different types of datasource-depending DAO objects (used to tie a web app to different data sources) make use of the *Transfer Object* pattern to provide a **common set of objects to trasport data** to and from their clients

The DAO pattern: UML schema



The DAO pattern: UML schema



`it.unibo.tw0708.web.dao.DAOFactory`

*(abstract factory: exposing **a static method for providing the right concrete factory;** extended by concrete factories that implement its abstract methods*

`it....dao.hsqldb.HsqldbDAOFactory`
`it....dao.myslq.MySqlDAOFactory`

*(concrete factories:, implement the **DAO object creational methods** demanded by their common abstract superclass; **manage database connectivity** as for credentials, connections, etc...)*

`it....dao.hsqldb.HsqldbUserDAO`
`it....dao.mysql.MySqlUserDAO`

*(concrete DAO objects holding **specific SQL code** to access and store data on the corresponding type of data source)*

`it.unibo.tw0708.web.dao.UserDAO`

*(DAO interface that describes **methods to access and store values for the corresponding type of object** - e.g., User: CRUD logic and other db logic)*

`it.unibo.tw0708.web.dao.UserTO`

*(transport object to **convey input arguments and output results of the** method in the **corresponding DAO interface**)*

DAO pattern and JSF framework: adding a new user to the db

> *UserTO* transfer object is a field of the *manager* bean

- *manager* provides getter and setter methods for its *userTO* field
- userTO follows Java bean naming convention too

- *tablesdao.jsp* page can bind userTO properties in the form controls by means of input tags and value-binding expressions like this:

```
<h:inputTextarea value="#{manager.user.address}"/>
```

> Upon invocation of the bean event handler method in the *tablesdao.jsp* page, *manager* **instantiates a DAO object by invoking factory methods** and **passes to it the transfer object to store**:

```
UserDAO userDAO =
    DAOFactory.getDAOFactory(DAOFactory.HSQLDB).getUserDAO();
...
userDAO.updateUser(this.userTO);
```

HSQLDB version of the DAO object is working properly.

Find out how different SQL statements can be: MySQL version has been created as a clone of the Hypersonic one → I already changed factory connection method (by using an alias and a password) and now objects have access to the DB (provided SQLServer is up and running), but still queries fail: can you fix them ??