**Struts and MVC**

In Struts, **Model** code consists of plain old Java objects (POJOs).

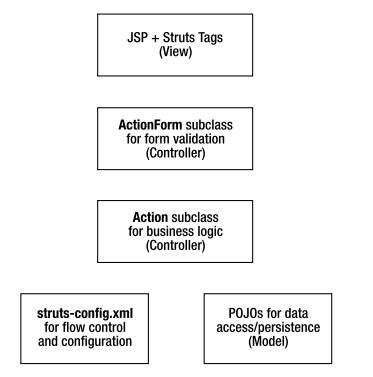
**View** code consists of JSPs and a set of custom tags supplied by Struts. These custom tags enable you to separate View from Controller.

With Struts, **Controller** code falls into three broad categories:

• **Simple validations** are performed by your subclasses of the Struts base class called ActionForm. Checks for password length or email address format are examples of this. Later in this book, you’ll see how Struts greatly simplifies validation with the Validator framework.

• **Complex validations and business logic** are done in your subclasses of the Struts base class called Action. The check for the duplicate user ID for the Registration webapp is an example of complex validation. An example of business logic is calculating the total amount due after a purchase in a shopping cart webapp.

• **Flow control** is also decided by your Action subclasses, restricted to paths declared in the Struts configuration file called struts-config.xml.



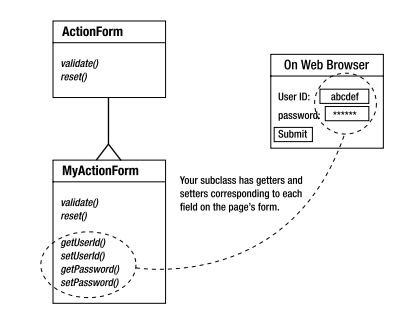
**Anatomy of ActionForm**

The central class for performing simple validations is org.apache.struts.action.ActionForm

Struts requires you to associate each form displayed to the user with your subclass of ActionForm. Your subclass of ActionForm does two things:

• It holds all form data: Your subclass of ActionForm must have getters and setters corresponding to each property of the form.

• It performs simple validation: Your subclass must override the validate() function, if you want to perform any simple checks. Figure 6-1 illustrates this relationship between ActionForm, your ActionForm subclass, and the input HTML form for which your subclass stores data.



Package bean;

import org.apache.struts.action.\*;

public final class RegistrationForm extends ActionForm{

private String \_userid = null;

private String \_pwd = null;

private String \_pwd2 = null;

/\*\* \* getXXX and setXXX functions \* corresponding to form properties \*/

public String getUserid(){ return \_userid; }

public void setUserid(String userid){ \_userid = userid; }

public String getPassword(){ return \_pwd; }

public void setPassword(String pwd){ \_pwd = pwd; }

public String getPassword2(){ return \_pwd2; }

public void setPassword2(String pwd2){ \_pwd2 = pwd2; }

/\*\* \* Validate the user input. Called automatically \* by Struts framework. \*/

public ActionErrors validate(ActionMapping mapping, HttpServletRequest request){

//create a blank ActionErrors

ActionErrors errors = new ActionErrors();

//check for a blank user id

if( null == \_userid ){

errors.add("userid", new ActionMessage("reg.error.userid.missing"));

}

//check password 1 == password 2

if( !\_pwd.equals(\_pwd2)){

errors.add("password", new ActionMessage("reg.error.password.mismatch"));

} return errors; }

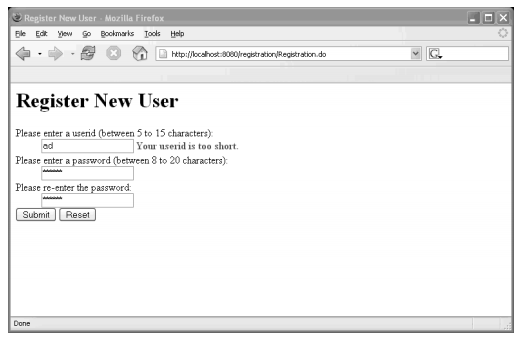
}

Let us consider a Form consist of three fields: a user ID, a password, and the password confirmation. The first thing to note from Listing 6-1 is that this ActionForm subclass contains getters and setters for each of these three fields. The body of validate() follows a simple pattern:

1. Create a blank ActionErrors instance.

2. Run a check for each field, and populate the ActionErrors instance with error messages.

3. Return the ActionErrors instance.



We could have include the following code for the existing user error.

//test for duplicate userid

if(User.exists(\_userid)){

errors.add("userid", new ActionMessage("reg.error.userid.exists"));

}

But we can’t include that code in validate method of ActionForm because User.exist (\_usedid) needs to go to model to access database and retrieve the data. For that we use Validator Framework.

**Using ActionErrors**

Struts uses the ActionErrors instance returned by validate() to display the error messages on the redisplayed form. ActionErrors (note the plural) is very much like a HashMap, and it is used to store error messages, indexed by error keys. The error message is not a String, but an instance of ActionMessage.

ActionMessages are added to an ActionErrors instance using the add() function:

public void add(String key, ActionMessage message)

error.add(“name”,new ActionMessage("reg.error.password.mismatch"))

Here, we’re passing in a key to an error message, stored in a properties file accessible to Struts.Properties files are just text files, which contain key/value pairs. In this case, the key is reg.error.password.mismatch, and the value might be The passwords you keyed in don't match!

# Properties file for Registration webapp

# Error messages:

reg.error.userid.missing=The user id is missing.

reg.error.userid.exists = The user id exists. Choose another.

reg.error.userid.bad = Use only alphanumerics for the userid.

reg.error.password.mismatch = The passwords you keyed in don't match!

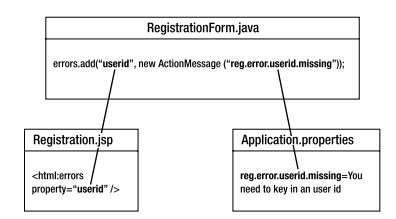
reg.error.password.long = The password is too long!

reg.error.password.short = The password is too short!

# Prompts: reg.prompt.userid=User ID

reg.prompt.password=Password

reg.prompt.password.confirmation=Password Confirmation



When the user submits a form, Struts calls validate() on the associated ActionForm subclass, which returns an ActionErrors instance. This is just a collection of ActionMessages, each of which references error messages stored in properties files. When Struts redisplays the page, it uses special tags you’ve embedded on the page to locate exactly where to display the error messages. The tags on the page are distinguished by their property attribute, which forms the link between the tag and a particular ActionMessage instance stored on the ActionErrors collection.

In addition to validate(), ActionForm exposes another function, called reset(), which resets the ActionForm’s values to defaults you define. reset() can be called by View code using a Reset button.