DynaForms and the Validator

IN THIS CHAPTER

- DynaForms: Forms Without Java

- DynaBeans and Struts

- The Validator: Automating Field Checking

- Conclusions

It might sound like a comic book superhero, but a `DynaForm` can't leap a tall building in a single method invocation. On the other hand, it can reduce a lot of the drudgework of developing Struts applications.

Similarly, the Struts Validator framework can eliminate many common form validation tasks, leaving you to concentrate on the business logic. However, the validator must be used with care, because it doesn't handle all the more complex validations you could encounter during development.

By the end of this chapter, you'll have seen a few examples of how to create and use `DynaForms`, and how to integrate them with the Validator to create practically Java-less validating forms.

DynaForms: Forms Without Java

`DynaForms` are an extension of the Apache Commons Beanutils project. As part of the `org.apache.commons.beanutils` package, an interface called `DynaBean` was created. Unlike normal JavaBeans, which require explicit `getXXX()` and `setXXX()` methods to be written for each property, a DynaBean uses a generic `get()` and `set()` method with the property name as the first argument.

A fuller description of the DynaBean package can be found on the Jakarta Web site at `http://jakarta.apache.org/commons/beanutils.html`.

For example, in a traditional JavaBean, you would say

```
myBean.setType("kidneybean");
```

Buy This Book From informIT

"Use coupon code STRUTS when buying this book and save even more."

Using `DynaBeans`, you would say

```
myBean.set("type", "kidneybean");
```

This approach has both advantages and disadvantages. The major advantage is that you don't have to declare all your properties explicitly at compile-time; they can be loaded dynamically during execution. This can be very handy to quickly create new beans or new properties of beans.

The disadvantage is that you lose a lot of compile-time error checking. Let's say you are thick-fingered, and enter

```
myBean.set("tpye", "kidneybean");
```

The compiler won't catch this. As far as it's concerned, this is perfectly legal Java. It's not until run-time that you'll suddenly find yourself with a null pointer exception. Similarly, all `get()` and `set()` methods take and return the `Object` type, so you lose the strong type-checking of traditional beans.

DynaBeans and Struts

One place that DynaBeans make a lot of sense is in Struts. If you use DynaBeans correctly, you can reduce the size of your `ActionForms` by 80–90%. This is because you can use DynaBeans to eliminate all of your form getters and setters.

To see how this works, you'll rewrite the two `ActionForms` for the new account pages of the StockTrack application. To begin, take a look at one of the existing `ActionForms` and the same form rewritten with `DynaForms` (Listing 17.1 and 17.2).

LISTING 17.1 NewUserAddressForm.java

```
package stocktrack.struts.form;

import javax.servlet.http.*;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionError;
import org.apache.struts.action.ActionForm;
import stocktrack.struts.form.BaseForm;

/**
 * stocktrack.struts.form.NewUserAddressForm class.
 * this class used by Struts Framework to store data from newUserAddressForm
 *
```

LISTING 17.1   Continued

```
 * struts-config declaration:
 * <form-bean name="newUserAddressForm"
 *         type="stocktrack.struts.form.NewUserAddressForm" />
 *
 * @see org.apache.struts.action.ActionForm org.apache.struts.action.ActionForm
 * Generated by StrutsWizard.
 */

public class NewUserAddressForm extends BaseForm {
  public void reset(ActionMapping mapping, HttpServletRequest request) {
    streetAddress1 = "";
    streetAddress2 = "";
    city = "";
    state = "";
    postalCode = "";
    homePhone = "";
    workPhone = "";
    workExt = "";
  }
  public ActionErrors validate(ActionMapping mapping,
                               HttpServletRequest request) {
    ActionErrors errors = new ActionErrors();
    if (this.isBlankString(streetAddress1)) {
        errors.add("streetAddress1",
                  new ActionError("stocktrack.newuser.required"));
    }
    if (this.isBlankString(city)) {
        errors.add("city", new ActionError("stocktrack.newuser.required"));
    }
    if (this.isBlankString(state)) {
        errors.add("state", new ActionError("stocktrack.newuser.required"));
    } else {
      if (!this.isValidState(state)) {
        errors.add("state",
                  new ActionError("stocktrack.newuser.invalid.state"));
      }
    }
    if (this.isBlankString(postalCode)) {
        errors.add("postalCode",
                  new ActionError("stocktrack.newuser.required"));
```

LISTING 17.1 Continued

```
    } else {
      if (!this.isValidPostalCode(postalCode)) {
          errors.add("postalCode",
                    new ActionError("stocktrack.newuser.invalid.postalCode"));
      }
    }
    if (this.isBlankString(homePhone)) {
        errors.add("homePhone", new ActionError("stocktrack.newuser.required"));
    } else {
        if (!this.isValidPhone(homePhone)) {
          errors.add("homePhone",
                    new ActionError("stocktrack.newuser.invalid.phone"));
        }
    }
    if (this.isBlankString(workPhone)) {
        errors.add("workPhone", new ActionError("stocktrack.newuser.required"));
    } else {
        if (!this.isValidPhone(workPhone)) {
          errors.add("workPhone",
                    new ActionError("stocktrack.newuser.invalid.phone"));
        }
    }
    return errors;
  }
  private String streetAddress1;
  private String streetAddress2;
  private String city;
  private String state;
  private String postalCode;
  private String homePhone;
  private String workPhone;
  private String workExt;
  public String getStreetAddress1() {
    return streetAddress1;
  }
  public void setStreetAddress1(String streetAddress1) {
    this.streetAddress1 = streetAddress1;
  }
  public String getStreetAddress2() {
    return streetAddress2;
  }
```

LISTING 17.1 Continued

```java
  public void setStreetAddress2(String streetAddress2) {
    this.streetAddress2 = streetAddress2;
  }
  public String getCity() {
    return city;
  }
  public void setCity(String city) {
    this.city = city;
  }
  public String getState() {
    return state;
  }
  public void setState(String state) {
    this.state = state;
  }
  public String getPostalCode() {
    return postalCode;
  }
  public void setPostalCode(String postalCode) {
    this.postalCode = postalCode;
  }
  public String getHomePhone() {
    return homePhone;
  }
  public void setHomePhone(String homePhone) {
    this.homePhone = homePhone;
  }
  public String getWorkPhone() {
    return workPhone;
  }
  public void setWorkPhone(String workPhone) {
    this.workPhone = workPhone;
  }
  public String getWorkExt() {
    return workExt;
  }
  public void setWorkExt(String workExt) {
    this.workExt = workExt;
  }
}
```

LISTING 17.2   NewUserAddressForm.java as a DynaForm

```java
package stocktrack.struts.form;

import javax.servlet.http.*;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionError;
import org.apache.struts.action.ActionForm;
import stocktrack.struts.form.BaseForm;
import stocktrack.struts.form.BaseDynaForm;

/**
 * stocktrack.struts.form.NewUserAddressForm class.
 * this class used by Struts Framework to store data from newUserAddressForm
 *
 * struts-config declaration:
 * <form-bean name="newUserAddressForm"
 *          type="stocktrack.struts.form.NewUserAddressForm" />
 *
 * @see org.apache.struts.action.ActionForm org.apache.struts.action.ActionForm
 * Generated by StrutsWizard.
 */

public class NewUserAddressForm extends BaseDynaForm {

  public ActionErrors validate(ActionMapping mapping,
                               HttpServletRequest request) {
    ActionErrors errors = new ActionErrors();
    if (BaseForm.isBlankString(this.getString("streetAddress1"))) {
        errors.add("streetAddress1",
                   new ActionError("stocktrack.newuser.required"));
    }
    if (BaseForm.isBlankString(this.getString("city"))) {
        errors.add("city",
                   new ActionError("stocktrack.newuser.required"));
    }
    if (BaseForm.isBlankString(this.getString("state"))) {
        errors.add("state", new ActionError("stocktrack.newuser.required"));
    } else {
      if (!BaseForm.isValidState(this.getString("state"))) {
        errors.add("state",
                   new ActionError("stocktrack.newuser.invalid.state"));
```

LISTING 17.2   Continued

```
    }
  }
   if (BaseForm.isBlankString(this.getString("postalCode"))) {
      errors.add("postalCode",
                  new ActionError("stocktrack.newuser.required"));
   } else {
     if (!BaseForm.isValidPostalCode(this.getString("postalCode"))) {
        errors.add("postalCode",
                    new ActionError("stocktrack.newuser.invalid.postalCode"));
     }
   }
   if (BaseForm.isBlankString(this.getString("homePhone"))) {
      errors.add("homePhone", new ActionError("stocktrack.newuser.required"));
   } else {
      if (!BaseForm.isValidPhone(this.getString("homePhone"))) {
        errors.add("homePhone",
                    new ActionError("stocktrack.newuser.invalid.phone"));
      }
   }
   if (BaseForm.isBlankString(this.getString("workPhone"))) {
      errors.add("workPhone", new ActionError("stocktrack.newuser.required"));
   } else {
      if (!BaseForm.isValidPhone(this.getString("workPhone"))) {
        errors.add("workPhone",
                    new ActionError("stocktrack.newuser.invalid.phone"));
      }
   }
   return errors;
  }

}
```

The first thing to notice here is that the class has been changed from `BaseForm`
(which, as you might recall, is simply `ActionForm` with a few helper functions added
for validation) to `BaseDynaForm`, which is a different helper class that extends
`DynaActionForm` instead. All the getters and setters have been removed, and the
`validate` function now uses `this.getString` to get the values of the properties
rather than accessing the local variables of the class.

getString() is in fact not a part of DynaActionForm, which only defines generic get() and set() methods that take and return Objects. However, rather than cast to String all the time, I've created BaseDynaForm, which adds the getString() versions. Listing 17.3 shows this simple class.

LISTING 17.3   BaseDynaForm.java

```
package stocktrack.struts.form;

import org.apache.struts.action.DynaActionForm;

public class BaseDynaForm extends DynaActionForm {
  public String getString(String name) {
    return (String) this.get(name);
  }

  public String getString(String name, int ind) {
    return (String) this.get(name, ind);
  }
}
```

In addition, now that the class no longer derives from BaseForm, you need to call out explicitly to BaseForm to get helper functions such as nullOrVoid. Because they were originally defined nonstatic, you must go into BaseForm and declare them static. I suppose that we could have copied the code from BaseForm to BaseDynaForm, but that would have meant maintaining the same code in two places.

The <form-property> Tag

So, if all the getters and setters have been removed from the class, how does the class know what its legal properties are? The answer lies in the form-bean tag in struts-config.xml.

Until now, the forms you've defined in the file looked like this:

```
<form-bean name="newUserAddressForm"
           type="stocktrack.struts.form.NewUserAddressForm" />
```

But now, you're going to add some new tags, form-property tags, to the form-bean (see Listing 17.4).

LISTING 17.4   The Rewritten `newUserAddress` Form

```
<form-bean name="newUserAddressForm" type="stocktrack.struts.form.NewUserAddress-
Form">
  <form-property name="streetAddress1" type="java.lang.String"/>
  <form-property name="streetAddress2" type="java.lang.String"/>
  <form-property name="city" type="java.lang.String"/>
  <form-property name="state" type="java.lang.String"/>
  <form-property name="postalCode" type="java.lang.String"/>
  <form-property name="homePhone" type="java.lang.String"/>
  <form-property name="workPhone" type="java.lang.String"/>
  <form-property name="workExt" type="java.lang.String"/>
</form-bean>
```

Inside the `form-bean`, all the properties have been listed along with their type. This allows the `DynaForm` to populate the fields. You can use pretty much any class you want in the `type` value, as well as primitives such as `int`, `Boolean`, and so on.

You can also specify initializations for properties via the `initial` keyword. For example, the following `form-property` sets up a property called `opt-in`, which defaults to true:

```
<form-property name="homePhone" type="boolean" initial="true"/>
```

The `initial` keyword is also the only way to create an array or list of a given size. Imagine that you have a form with 10 lines to fill in the names, ages, and genders of the user's dependents. You'd define it like so:

```
<form-property name="depName" type="java.lang.String[]"
    initial="{'','','','','','','','','',''}"/>
<form-property name="depAge" type="java.lang.String[]"
    initial="{'','','','','','','','','',''}"/>
<form-property name="depGender" type="java.lang.String[]"
    initial="{'','','','','','','','','',''}"/>
```

Whatever the type requested, if you want to specify an initial value, there must be a converter defined from `String` to that class in the Commons Beanutils package. You can add your own converters, so just about any class can be initialized via a `form-property` tag.

String (and other) arrays are accessed using `get()` and `set()`, the same as any other `DynaBean`. However, you use a second argument, which specifies the index into the array.

The Validator: Automating Field Checking

Using DynaBeans has enabled you to drastically reduce the size of the form bean, but wouldn't it be nice to get rid of it all together?

After moving all the properties out of the bean and into the DynaForm definition in struts-config.xml, the only thing left in the bean is the validate function. By using the Struts Validation framework, which ties into the Commons Validator package, you can remove this last piece of code and get rid of the form bean.

WHAT IS COMMONS?

Commons (or more formally, the Jakarta Commons project) is an effort to gather a lot of reusable Java code in one place.

The idea is that there are any number of commonly coded tasks that can be generalized and placed in one place, so that no one ever needs to reinvent the wheel again.

Commons is divided into two pieces: the Commons proper and the sandbox. The Commons proper is where well-tested robust packages live; they have release cycles and beta testing. The sandbox is where the "not ready for prime time" packages live while they undergo their initial development, and is a much less formal environment.

You'll find that many of the Jakarta projects, from Torque to Struts and beyond, depend on various Commons packages to work. You can learn more about the Commons project by visiting the Jakarta Web site at

http://Jakarta.apache.org/commons/

Adding validation to Struts forms is done in a few steps. To begin, you need to add the Validator plug-in to the struts-config.xml file. This is just a piece of boilerplate XML, shown in Listing 17.5.

LISTING 17.5   Adding the Validator Plug-In to Struts

```
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
  <set-property
  property="pathnames"
  value="/WEB-INF/validator-rules.xml,/WEB-INF/validation.xml"/>
</plug-in>
```

This plug-in should be the very last thing in the file before the </struts-config> tag.

The plug-in entry does two things. First, it notifies Struts that the validator should be made available. Secondly, it defines the locations of the validator rules files, which are used to define the validations that will be run over the forms. The validator-rules.xml file comes standard with the validator. It defines the most

commonly used rules (such as credit cards, dates, strings, numbers, and so on). The `validation.xml` file is one that you will define, and contains specialized validations used by your application, as well as the actual validation-to-form field mappings.

First, take a look at a typical rule from `validator-rules.xml` as delivered by the Validator package (see Listing 17.6).

LISTING 17.6   A Rule from `validator-rules.xml`

```
<validator name="creditCard"
          classname="org.apache.struts.util.StrutsValidator"
          method="validateCreditCard"
          methodParams="java.lang.Object,
                      org.apache.commons.validator.ValidatorAction,
                      org.apache.commons.validator.Field,
                      org.apache.struts.action.ActionErrors,
                      javax.servlet.http.HttpServletRequest"
          depends=""
          msg="errors.creditcard">
   <javascript><![CDATA[
      function validateCreditCard(form) {
          var bValid = true;
          var focusField = null;
          var i = 0;
          var fields = new Array();
          oCreditCard = new creditCard();

          for (x in oCreditCard) {
              if ((form[oCreditCard[x][0]].type == 'text' ||
                  form[oCreditCard[x][0]].type == 'textarea') &&
                  form[oCreditCard[x][0]].value.length > 0) {
                if (!luhnCheck(form[oCreditCard[x][0]].value)) {
                  if (i == 0)
                      focusField = form[oCreditCard[x][0]];

                  fields[i++] = oCreditCard[x][1];

                  bValid = false;
                }
              }
          }

          if (fields.length > 0) {
              focusField.focus();
```

LISTING 17.6   Continued

```
            alert(fields.join('\n'));
        }

        return bValid;
    }

    /**
     * Reference: http://www.ling.nwu.edu/~sburke/pub/luhn_lib.pl
     */
    function luhnCheck(cardNumber) {
        if (isLuhnNum(cardNumber)) {
            var no_digit = cardNumber.length;
            var oddoeven = no_digit & 1;
            var sum = 0;
            for (var count = 0; count < no_digit; count++) {
                var digit = parseInt(cardNumber.charAt(count));
                if (!((count & 1) ^ oddoeven)) {
                    digit *= 2;
                    if (digit > 9) digit -= 9;
                };
                sum += digit;
            };
            if (sum == 0) return false;
            if (sum % 10 == 0) return true;
        };
        return false;
    }

    function isLuhnNum(argvalue) {
        argvalue = argvalue.toString();
        if (argvalue.length == 0)
           return false;
        for (var n = 0; n < argvalue.length; n++)
           if (argvalue.substring(n, n+1) < "0" || argvalue.substring(n,n+1) >
        ➥"9")
               return false;

        return true;
    }]]>
  </javascript>
</validator>
```

Unless you're a JavaScript wonk, only the first 10 lines of the definition are of any real importance to how the validator works with Struts. The first line of XML declares the official name of this validator—that's the name that other rules or field definitions will use to refer to it.

The next line of XML tells the validator which class defines the validation method for this rule. Unlike many Java-related (and especially JSP-related) tools, the validator doesn't call a standard method on each class. Instead, you can declare any number of methods in a single class and use different ones for different validations.

The `method` value tells the validator which method of the class that was just specified handles the validation for this rule. In this case, validation is handled by the aptly named `validateCreditCard()` method.

Adding to the ways in which the validator is different from everything else in Java, the validation method isn't defined as an interface or even a standard API. It can take a number of different arguments in a number of different orders. The validator code figures out which arguments go where by looking at the types of the arguments as specified in the `methodParams` arguments. They are

- `java.lang.Object`—The `ActionForm` that the value comes from, unless the property is a `String` array or `Collection`, (in which case it is the string itself).

- `org.apache.commons.validator.ValidatorAction`—The object created when the validation rules are parsed, one per rule.

- `org.apache.commons.validator.Field`—An object that holds information about the field being validated, such as its name.

- `org.apache.struts.action.ActionErrors`—The same old `ActionErrors` that you know and love; you add to it if there's a validation error.

- `javax.servlet.http.HttpServletRequest`—The request being processed.

- `org.apache.commons.validator.Validator`—Gives you a handle to the Validator itself, useful for accessing other fields values.

Given all that information, the method should be able to figure out whether there was a validation error or not, eh?

Next, the `depends` tag specifies any other rules that should be run before this rule runs. In this case, the file used to say that the `required` rule should be run first. But, in my opinion, that was wrong. It would mean that you couldn't have a form with both a credit card and bank account field and let the user fill out one or the other, because credit card would be a required field. Luckily, I was able to convince the Struts folks to remove these dependencies, so now you need to explicitly use the `"required"` tag if you need a field to be `required`.

Finally, the `message` tag enables you to define the error message, which will be drawn from the default `Resource`, to use for this validation error.

The remainder of the file defines the JavaScript that's used to provide client-side validation of the field, if requested. It's optional, and those who are interested are welcome to refer to the documentation for the validator to learn more.

Adding the Validator to NewUserAddress

By using the validator, you can completely eliminate the need to create an ActionForm. This can be demonstrated with the same NewUserAddress action you converted to DynaBeans in the first half of this chapter.

To start, you need to create a validation.xml file, which goes into the same directory (WEB-INF) as the validator-rules.xml file. Listing 17.7 shows the file with only the rules for this form placed in it.

LISTING 17.7   validation.xml

```
<form-validation>
 <global>
  <constant>
   <constant-name>phone</constant-name>
   <constant-value>^\(?(\d{3})\)?[-| ]?(\d{3})[-| ]?(\d{4})$</constant-value>
  </constant>
  <constant>
    <constant-name>states</constant-name>
    <constant-value>^(AL|AK|AS|AZ|AR|CA|CO|CT|DE|DC|FM|FL|GA|
➥GU|HI|ID|IL|IN|IA|KS|KY|LA|ME|MH|MD|MA|MI|MN|MS|MO|MT|NE|NV|NH|
➥NJ|NM|NY|NC|ND|MP|OH|OK|OR|PW|PA|PR|RI|SC|SD|TN|TX|UT|VT|VI|VA
➥|WA|WV|WI|WY)$</constant-value>
  </constant>
  <constant>
    <constant-name>zip</constant-name>
    <constant-value>^\d{5}(-\d{4})?$</constant-value>
  </constant>
 </global>
 <formset>
  <form name="newUserAddressForm">
   <field
     property="streetAddress1"
     depends="required">
     <arg0 key="newUserAddressForm.streetAddress1.label"/>
    </field>
   <field
     property="city"
     depends="required">
```

LISTING 17.7   Continued

```
     <arg0 key="newUserAddressForm.city.label"/>
    </field>
    <field
     property="state"
     depends="required,mask">
     <arg0 key="newUserAddressForm.state.label"/>
     <var>
      <var-name>mask</var-name>
      <var-value>${states}</var-value>
     </var>
    </field>
    <field
     property="postalCode"
     depends="required,mask">
     <arg0 key="newUserAddressForm.postalCode.label"/>
      <var>
        <var-name>mask</var-name>
        <var-value>${zip}</var-value>
      </var>
    </field>
  <field
     property="workPhone"
     depends="required,mask">
     <arg0 key="newUserAddressForm.workPhone.label"/>
      <var>
        <var-name>mask</var-name>
        <var-value>${phone}</var-value>
      </var>
    </field>
   <field
     property="homePhone"
     depends="required,mask">
    <arg0 key="newUserAddressForm.homePhone.label"/>
      <var>
        <var-name>mask</var-name>
        <var-value>${phone}</var-value>
      </var>
    </field>
  </form>
  </formset>
</form-validation>
```

This file is broken into two main sections. The top section (inside the globals tag) is a place to define any global values that will be used throughout the file. Normally, these are strings that will be used for the mask validation, but they could also be used in other places, such as for key values.

In this case, you're defining three constants, which are Perl-style regular expressions that will be used to validate phone numbers, ZIP codes, and states.

After the globals section comes the formset section. Each form begins with a form tag containing a single attribute, which is the name of the form (and must match the name as defined in struts-config.xml).

Inside the form tag are a number of field tags. This tag has several attributes. The first is the property of the form that this entry validates, which must match a form-property. Next is the depends attribute, which specifies one or more validations that must be confirmed for this field to pass. For example, because the postalCode field has required,mask for a depends, both the required and mask validation must pass for this field to validate.

Inside the form tag, two types of values are commonly defined. The arg*x* (that is, arg0, arg1, and so on) tags are used to provide values to the error message if one is generated. Usually, all that's done here is to provide the key that corresponds to the name of the field in the Resource file.

The second tag, var, is used to pass arguments to the validators themselves. In this example, the mask validation is handed several different masks, depending on which field is being validated. By using the ${var} syntax, values defined in the globals section can be used here.

For the error messages to work, a number of strings must be put in the ApplicationResources.properties file (see Listing 17.8).

LISTING 17.8   Additions to ApplicationResources.properties

```
# Validator errors
errors.required={0} is required.
errors.minlength={0} can not be less than {1} characters.
errors.maxlength={0} can not be greater than {1} characters.
errors.invalid={0} is invalid.
errors.byte={0} must be an byte.
errors.short={0} must be an short.
errors.integer={0} must be an integer.
errors.long={0} must be an long.
errors.float={0} must be an float.
errors.double={0} must be an double.
errors.date={0} is not a date.
```

LISTING 17.8   Continued

```
errors.range={0} is not in the range {1} through {2}.
errors.creditcard={0} is not a valid credit card number.
errors.email={0} is an invalid e-mail address

#New user address form
newUserAddressForm.streetAddress1.label=Street Address
newUserAddressForm.city.label=City
newUserAddressForm.state.label=State
newUserAddressForm.postalCode.label=Postal Code
newUserAddressForm.homePhone.label=Home Telephone
newUserAddressForm.workPhone.label=Work Telephone
```

The top section of strings is the generic validator error messages. You're free to alter them as you want; these are just the ones suggested in the code. The `{0}` will be replaced with the `arg0` value defined in the `field` tag, as will the others by other arguments (for example, the `{1}` will be replaced by the minimum required length in the `minlength` validation).

The lower section simply defines the labels (names) that were used as keys to the field names in the `validation.xml` file. You can also used them with the `bean:message` tag in your JSP to internationalize your forms.

A few more steps are needed to complete the transition. The `type` attribute of `newUserAddressForm` in `struts-config.xml` must be changed to `org.apache.struts.validator.DynaValidatorForm`. Similarly, the line in `NewUserAddressAction` where the Form is cast to from a generic form must be changed to read:

```
DynaValidatorForm uf = (DynaValidatorForm) form;
```

With those changes, you can start the application and test it. You can also delete `NewUserAddressForm.java` because you've totally eliminated it.

Last Minute Validator News

One of the frustrations I (James speaking here) had with Struts was that the validator was languishing for lack of development. Specifically, there was no way to say "validate the checking account number only if the Use Checking Account button is pushed," short of doing those validations in the action.

Also, the Commons Validator package, which is what the Struts Validator Framework depends on, had stalled short of a 1.0 release. I (and a lot of other Struts developers) felt nervous about depending on a package that hadn't even gotten out of the Commons Sandbox.

As this book was being completed, I was accepted as a committer for the Commons Validator package, and volunteered to spearhead a 1.0 release, which is currently scheduled for November of 2002. In addition, I refactored some of the underlying Validator code and added the needed hooks so that cross-form dependencies can be implemented. With that in mind, I submitted a patch against the Struts Validator that added a "requiredif" dependency rule, and which is now part of the core Struts Validator Framework.

Defining a New Validation

Before we leave the topic, it's a useful exercise to see how a new validation type could be written. A good example is the validation done on bank account routing numbers, known in the industry as the ACH (automated clearing house) routing number.

The basic algorithm for this validation is similar to the one done on credit card numbers, in that it's a simple checksum. The number itself is nine digits long. The digits are handled in groups of three starting from the left. For each group of three, the first digit is multiplied by thee, the second by seven, and the third left as is. All the results are added together, and if the resulting number is evenly divisible by 10, the number passes.

To implement this check, you must first define a new class, which in turn defines a method that does the check. Listing 17.9 shows this file.

LISTING 17.9   ACHCheck.java

```
package stocktrack.validator;

import java.io.Serializable;
import javax.servlet.http.HttpServletRequest;
import org.apache.commons.validator.Field;
import org.apache.commons.validator.ValidatorAction;
import org.apache.struts.action.ActionErrors;
```

LISTING 17.9   Continued

```java
import org.apache.commons.validator.GenericValidator;
import org.apache.commons.validator.ValidatorUtil;
import org.apache.struts.util.StrutsValidatorUtil;

public class ACHCheck implements Serializable {

    public static boolean validateACHRouting(Object bean,
                         ValidatorAction va, Field field,
                         ActionErrors errors,
                         HttpServletRequest request) {

        String value = null;
        boolean results = false;

        if (isString(bean)) {
            value = (String) bean;
        } else {
            value = ValidatorUtil.getValueAsString(bean, field.getProperty());
        }

         if (GenericValidator.isBlankOrNull(value)) {
          return true;
        }

        if (value.length() != 9) {
          errors.add(field.getKey(),
                   StrutsValidatorUtil.getActionError(request, va, field));
          return false;
        }

        int n = 0;
        for (int i = 0; i < value.length(); i += 3) {
          n += CharToInt(value.charAt(i)) * 3
                     +  CharToInt(value.charAt(i + 1)) * 7
                     +  CharToInt(value.charAt(i + 2));
        }
```

LISTING 17.9   Continued

```
      // If the resulting sum is an even multiple of ten (but not zero),
      // the aba routing number is good.

      if (n != 0 && n % 10 == 0)
          return true;
     else {
        errors.add(field.getKey(),
                   StrutsValidatorUtil.getActionError(request, va, field)));
        return false;
          }
  }

  public static int CharToInt(char chr)
   {
     return Integer.parseInt(CharToString(chr));
   }

  public static String CharToString(char chr)
   {
     return String.valueOf(chr);
   }

  private static Class stringClass = new String().getClass();

  private static boolean isString(Object o) {
    if (o == null) return true;
    return (stringClass.isInstance(o));
  }


}
```

The first thing to notice is that the arguments to the validation function should look very familiar because they're the same types as the values of the methodParams attribute in the validator-rules.xml file. That's because you're now defining the method that will be called by the validator and will follow the template described in the validator tag.

The first thing to do is see whether the bean value passed in is a string. If it is not a string, the function must have been called during validation of a string array.

If it's not a string (or null), the function must look up the property name in the bean. `ValidatorUtil` has a nice helper function to do this.

After the function has the value, it checks whether it's of the right length, and if so, whether it passes the checksum. If it fails, a helper function from `StrutsValidatorUtil` is used to generate the appropriate `ActionError` for the property.

In addition to adding to the `ActionErrors` variable, the method must also return true or false because the validator depends on this to determine whether further validations should be run on this field.

Next, you must add the validation rule. Technically, it could go in either `validation.xml` or `validator-rules.xml` because both files are the same format (and, in fact, you can load an arbitrary number of these files by changing the list in `struts-config.xml`). However, it's a good idea to put local extensions into `validation.xml` because the other file comes with Struts, and might be overwritten. Here are the additional lines of XML, placed right after the `global` tag:

```
<validator name="achRouting"
    classname="stocktrack.validator.ACHCheck"
    method="validateACHRouting"
    methodParams="java.lang.Object,
                  org.apache.commons.validator.ValidatorAction,
                  org.apache.commons.validator.Field,
                  org.apache.struts.action.ActionErrors,
                  javax.servlet.http.HttpServletRequest"
    depends=""
    msg="errors.achRouting"/>
```

Now you can add a new property to the new address form for testing:

```
<form-property name="bankRouting" type="java.lang.String"/>
```

Of course, there are new resource strings, too:

```
errors.achRouting={0} is not a valid routing number.
newUserAddressForm.bankRouting.label=Bank Routing Number
```

And, finally, the form itself needs to handle the field:

```
<tr>
  <td>Bank Routing</td><td>
      <html:text property="bankRouting" maxlength="9" size="9"/></td>
  <td><html:errors property="bankRouting"/></td>
</tr>
```

With all that work in place, you can now validate bank routing numbers. If you want a good one, 123123123 will pass.

IS THE VALIDATOR WORTH IT?

I have to say that I'm honestly of two minds in regards to the validator. On one hand, I like the way it enables you to eliminate many FormAction classes altogether.

On the other hand, you end up writing a lot of boilerplate XML for every form, and for every field of every form. In fact, a rough estimate showed that a 20-line Java class file that implemented the validations in a FormAction might be replaced by 40 or more lines of XML and properties to do the same thing.

On the other other hand, the validator does reduce the amount of validation logic you have to write. So, I can't say that there's a good answer one way or the other.

Conclusions

Using DynaBeans and the Validator framework, you can reduce or even eliminate the need to write an ActionForm. DynaForms enable you to define the properties of a form in the struts-config.xml file, rather than in the form bean itself. You then use generic get and set operations on the form, rather than explicitly defined accessors.

The Validator framework integrates Struts with the Commons Validator project. The validator enables you to define validation rules in XML files, and automatically run them against the form during submission. This eliminates the need for the validate method on the Form, and in combination with a DynaForm, it eliminates the need for a distinct class.

Because the framework doesn't handle all types of validations, it might be necessary to handle some of them in the Action, although this can confuse the user by presenting errors in two stages.

The Framework can also be extended by writing new validation rules, which are defined in one of the XML configuration files and implemented in a new class and method.