Streamline your JSF validation framework

by Benny Bottema - Sunday, July 13, 2008

http://www.bennybottema.com/2008/07/13/streamline-your-jsf-validation-framework/

I was working a JSF project which, as all JSF projects do, needed a bunch of validators. It was getting a little bit messy having a lot of boilerplate code and validation algorithms that couldn't be reused. So I decided to streamline the code and usage of validators in JSF in general, in combination with dealing with validation messages optionally using resource bundles.

The solution I'm suggesting isn't rocket science, but it serves its purpose in keeping everything maintainable, understandable and easy to work with.

Let me know what you think.

[Download all sources]

Contents

The general idea

So here's the general layout of what I had in mind:

1. Group all validation logic in methods in an independent utility class
2. Have an abstract validation class, which takes care of some logging and validation errors.
3. Add subclasses of this abstract validation class that merely invokes the validation logic on a separate utility class

The methods should be kept in a separate utility class, so that these validations can be used anywhere in the application and furthermore: outside JSF applications. Also, since all validations are now organized in one central location, we can deal with validation messages uniformly across all validation methods, within the utility class. We'll see about that later.

Setting up the Validation utility class

What I'm going to do is define constants with error codes, that should be able to be mapped directly to message id's in message resourcebundles. Realizing not all projects nescesarily use these bundles, we're going to create default messages as well. These default messages are there to be used by the various validators, but don't *have* to be used at all.

```
public static final String VALIDATIONERROR_BIRTHDATE_TOOYOUNG = "valid
ationerror.birthdate.tooyoung";
static final Map defaultMessages;
static {
 // set default error messages
 defaultMessages = new HashMap();
 defaultMessages.put(VALIDATIONERROR_BIRTHDATE_TOOYOUNG, "Minimum %s y
ears of age allowed");
}
```

I split up the validator utils into two classes. A ValidatorMessages class which contains all the constants with the error codes and default messages. And then ofcourse ValidatorUtils, which contains all validation algorithms, because I wanted to avoid having a validation God Class. I've thought of making the ValidatorMessages an interface, but I couldn't semantically justify that.

ValidatorMessages.java
ValidatorUtils.java

These classes use some utility functions from GenericUtils (such as the regexp helper method).

GenericUtils.java

Setting up the abstract validator class

The abstract validator class is really the engine behind the new validation framework. It should do some logging, some more logging in case of invalidation, getting the correct message from either the list of default messages or the message resourcebundle (if available) and then it should throw a JSF validation exception.

AbstractValidator.java

The abstract validator uses a method from yet another utility class, specific to JSF applications: JSFUtils. What it does is try to return a message from the default application resource bundle, and if unavailable (either the bundle or the specific message) it will return the given backup message.

JSFUtils.java

So the first thing you'll notice is that the abstract validator needs to be instantiated with a default field name. This is so that if you omit the corresponding attribute in the validator tag in your JSF page, there will be a default name in the error message to the end-user.

Furthermore, the abstract validator uses a utility method to get a message from the default application resource bundle, but returns a backup message if the resource bundle or message within is unavailable. Since the way the resource bundle is being fetched depends on a *FacesContext*, the utility is called JSFUtils and is specific to JSF applications, just as the JSF Validator shells are.

Setting up the validators

The actual validators being registered with JSF are now very easy to set up. In fact, they now function merely as glue between the AbstractValidator and the ValidatorUtils, while providing entry points for the JSF validation framework.

Here are two example validator classes:

PostalcodeValidator.java
AgeValidator.java

As you can see the subclasses are extremely lightweight and only contain some boilerplate code to optionally customize the error messages specific to that validator class and even then the error messages themselves are kept in message bundles or ValidatorMessages and handled by the AbstractValidator class.

To configure JSF with the concrete validators:

```
<faces-config>

 ...

 <validator>
  <validator-id>validatePostalcode</validator-id>
  <validator-class>nl.quintor.commons.validator.PostalcodeValidator</v
alidator-class>
  <property>
   <property-name>fieldName</property-name>
   <property-class>java.lang.String</property-class>
  </property>
 </validator>
 <validator>
  <validator-id>validateAge</validator-id>
  <validator-
```

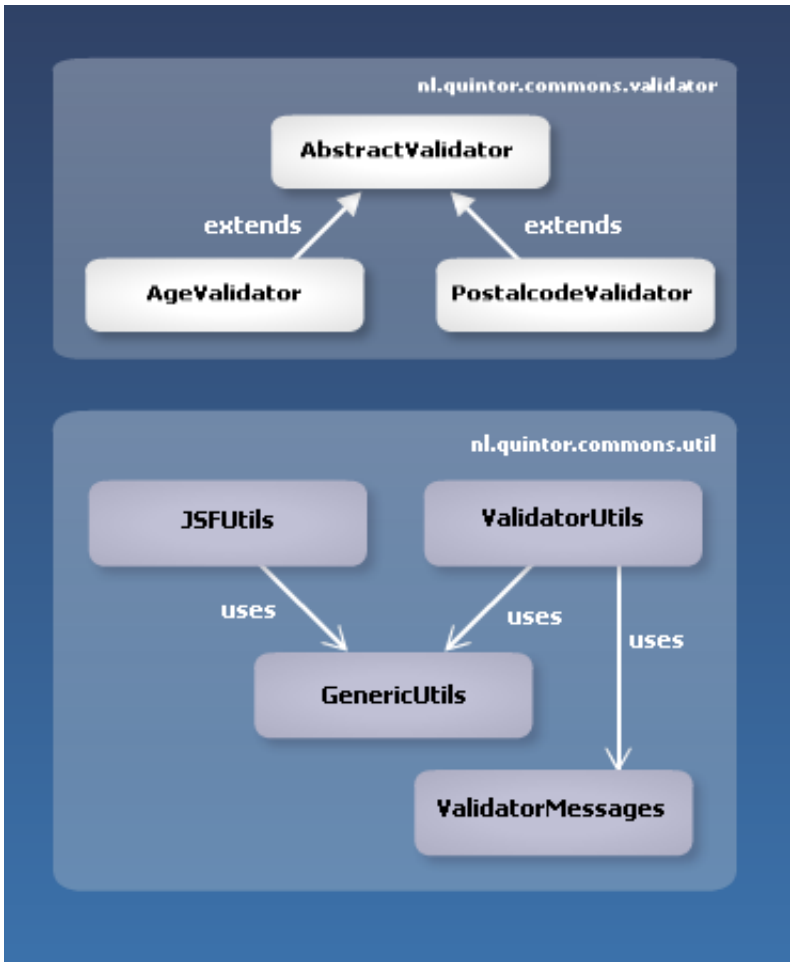
```
class>nl.quintor.commons.validator.AgeValidator</validator-class>
  <property>
   <property-name>fieldName</property-name>
   <property-class>java.lang.String</property-class>
  </property>
  <property>
   <property-name>min</property-name>
   <property-class>java.lang.Integer</property-class>
  </property>
  <property>
   <property-name>max</property-name>
   <property-class>java.lang.Integer</property-class>
  </property>
 </validator>
</faces-config>
```

And to actually use the validator in some page:

```
<!-- Put an age validator on the birthdate input field
fieldname 'age' will be used in the error messages too old/young -->
<h:inputText id="birthdate" required="true" maxlength="10" value="#{pe
rsonBean.birthdate}">
 <f:converter converterId="CalendarConverter" />
 <f:validator validatorId="validateAge" fieldName="Age" />
</h:inputText>
```

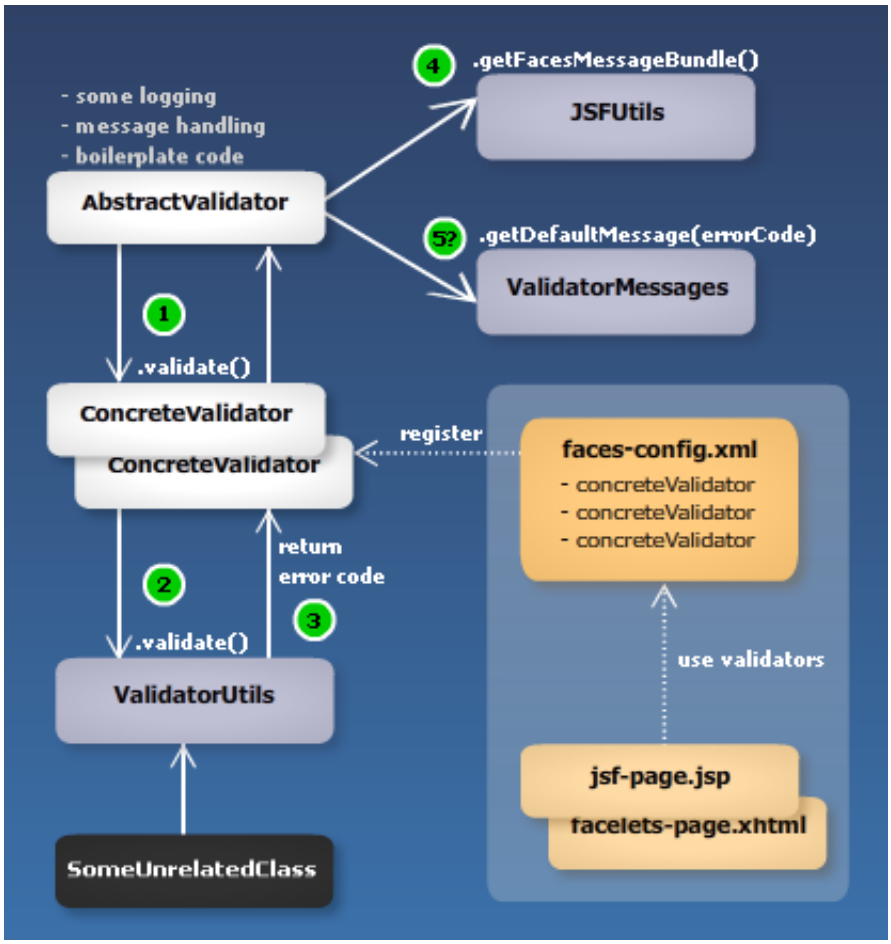
Roundup

In retrospect, here’s a short impression of the packages now in use:



Let’s summarize how the logic is spread out among the classes:

- GenericUtils: contains some generic functionality neutral to all applications, such as regexp methods
- JSFUtils: contains generic functionality specific to JSF applications, such as getting messages from a FacesContext resourcebundle
- ValidatorUtils: contains the validation methods, neutral to all applications, such as validateAge
- ValidatorMessages: encapsulates the error codes and their default messages that can be returned by validation methods in ValidatorUtils
- AbstractValidator: contains logic used for all validators, such as logging, determining validation message, throwing JSF validation exceptions
- ConcreteValidator.java: contains the default field name for end-user feedback purposes and calls the correct validation method in ValidatorUtils and returns its result to the AbstractValidator validation method (and optionally customizes the message)
- **faces-config.xml**: contains registration for concrete validators including properties that can be set (ie. limits for the age validator)
- **some-page.jsp/xhtml**: contains the input fields and nested validator tags, optionally with attributes to customize it



So to actually add a new validator, you can follow the next steps:

1. Add the validation codes and default messages you'll need to ValidatorMessages
2. Add a method to the ValidatorUtils that performs the validation and returns one of these messages
3. Create a subclass of AbstractValidator, call the super constructor with a default fieldname and implement the validation method, calling the corresponding validation method in ValidatorUtils
4. Register the validator with your faces config file

Benefits of this approach

The benefit is ofcourse that everything is being managed separately and can be reused; you can reuse the validation method anywhere across your projects and what's more: you can use the same validation methods in non-JSF applications. Also with this approach you can easily junit test the validation methods. When adding new validators, you add very little code but highly concise and cohesive code.

In this approach the Validator class acts as a shell around the actual validation logic and can therefor be very lightweight. Now you can add validators with the least amount of code possible while maintaining maximum flexibility.

Conclusion

So, coming to the conclusion, I hope you can use this stuff. I know it works beautiful for me, but If you have suggestions, as I have no doubt someone can do this all even better, let me know!
