Zend_Validate_Builder - Bryce Lohr

<ac:macro ac:name="unmigrated-inline-wiki-markup"><ac:plain-text-body><![CDATA[

<ac:macro ac:name="unmigrated-inline-wiki-markup"><ac:plain-text-body><![CDATA[

Zend Framework: Zend_Validate_Builder, Zend_Filter_Builder Component Proposal

Proposed Component Name	Zend_Validate_Builder, Zend_Filter_Builder
Developer Notes	http://framework.zend.com/wiki/display/ZFDEV/Zend_Validate_Builder, Zend_Filter_Builder
Proposers	Bryce Lohr
Revision	0.3 (wiki revision: 50)

Table of Contents

1. Overview
2. References
3. Component Requirements, Constraints, and Acceptance Criteria
4. Dependencies on Other Framework Components
5. Theory of Operation
6. Milestones / Tasks
7. Class Index
8. Use Cases
9. Class Skeletons

1. Overview

This proposal is simply an alternative to the Zend_Filter_Input proposal. Bill Karwin and others have put a lot of time and energy into that work, and I'd like to take this opportunity to thank them. This work is built upon those, and I'd like to acknowledge that up front.

This was originally inspired by Christopher Thompson's Zend_FilterChain proposal, and provides a vaguely similar interface. This takes quite a different tact than the array-driven structure used by Zend_Filter_Input; some people may find the code-driven interface preferable.

Here's a quick rundown of the major parts:

- Zend_Filter_Builder and Zend_Validate_Builder classes
- Facades providing a fluent interface for both Builder classes
- An Error Manager class to facilitate configuring error messages for many validators at once, as well as retrieving raised errors

2. References

- Zend_FilterChain Zend_Validator - Christopher Thompson
- Zend_Filter Design Proposal - Darby Felton
- Zend_Filter_Input redesign - Bill Karwin
- Zend_Filter static usage - Bill Karwin
- Zend_Validate_Array Proposal

- Zend_Form - Simon Mundy & Ralf Eggert

3. Component Requirements, Constraints, and Acceptance Criteria

- Zend_Validate_Builder manages exactly one responsibility: validating sets of data in arrays
- Zend_Filter_Builder manages exactly one responsibility: filtering sets of data in arrays
- Zend_Validate_Builder_ErrorManager is responsible for defining and reporting error messages for validators
- Neither component handles any other aspect of form processing
- No mechanism for statically calling filters or validators is provided (out of scope)
- This solution is not designed to make it easier to use single filters or validators; rather, it automates using many filters/validators together on whole sets of data
- This is compatible with the existing Zend_(Validate|Filter)_Interface API; i.e., existing Framework filters and validators will work without being modified

4. Dependencies on Other Framework Components

- Zend_Validate_Interface
- Zend_Filter_Interface
- Zend_Validate_*
- Zend_Filter_*
- Zend_Validate_Exception

5. Theory of Operation

Essentially, Zend_Validate_Builder and Zend_Filter_Builder use the Builder and Composite patterns to construct a recursive tree structure of Zend_Validate_* or Zend_Filter_* instances, which is then applied wholesale to an array of input data. In the case of Zend_Filter_Builder, the result is the original input array after having each of the constituent filters applied to each input array element. In the case of Zend_Validate_Builder, the result is a boolean value indicating whether any single element in the input data was deemed invalid by any of the constituent Zend_Validate_* instances.

In addition to the Builder classes, there are also Zend_Validate_Builder_FluentFacade and Zend_Filter_Builder_FluentFacade. These classes provide a very elegant and easy to use fluent interface to the Builder classes, and their use is totally optional. The Builder classes in no way depend on these.

Error messages are a vital part of data validation. If your code decides that something the user entered is not acceptable, you must be able to tell the user why. The Zend_Validate_Builder_ErrorManager class addresses this concern. It provides a way for the Zend_Validate_* classes, Zend_Validate_Builder, and your application code to define and communicate error messages. The Zend_Validate_* classes all have built-in error messages which, for a variety of reasons, are often not suitable to display directly to users. Therefore, the error messages for validators can be overridden. However, this must be done manually, on each individual instance with the built-in validators. Zend_Validate_Builder_ErrorManager allows you to define all the error messages for any arbitrary set of validators, by data field. This way, you can easily set up each error message for a field, and also easily retrieve any error messages by field for display.

6. Milestones / Tasks

- Milestone 1: [DONE] Design interface and test feasibility
- Milestone 2: [DONE] Write proposal
- Milestone 3: [DONE] Gather feedback and revise design as necessary (done, but more feedback is always welcomed)
- Milestone 4: [DONE] Develop full implementation and unit tests
- Milestone 5: Write documentation (this proposal actually *is* the current documentation)

7. Class Index

- Zend_Validate_Builder
- Zend_Validate_Builder_FluentFacade

- Zend_Validate_Builder_FluentAdder
- Zend_Validate_Builder_ValidatorFactory
- Zend_Validate_Builder_ValidatorFactory_Interface
- Zend_Validate_Builder_ErrorManager
- Zend_Validate_Builder_ErrorManager_Interface
- Zend_Filter_Builder
- Zend_Filter_Builder_FluentFacade
- Zend_Filter_Builder_FluentAdder
- Zend_Filter_Builder_FilterFactory
- Zend_Filter_Builder_FilterFactory_Interface

8. Use Cases

For all of these use cases, assume that the code shown is extracted from a class that is processing a form, with the intention of saving the data to the database. Assume that methods referenced after `$this->` exist and are valid; variables such as `$post` pretty obviously refer to the user-submitted data. In most of these examples, I refer to Zend_Validate_* classes that don't exist in the Framework. Many of them are available in the Laboratory, along side ZVB and ZFB; as for the others--just assume they exist for now.

UC-01

Some very basic examples. First, Zend_Filter_Builder:

```
$filter = new Zend_Filter_Builder;
$filter->add(new Zend_Filter_Digits, 'phone_no');
$filter->add(new Zend_Filter_StripTags, 'description');
$filteredPost = $filter->filter($unfilteredPost);
```

This filters out all non-digit characters from the `phone_no` field, and strips HTML tags from the `description` field. The `filter()` method takes an array of data, and runs the specified filters on the matching keys.

Next, Zend_Validate_Builder:

```
$validate = new Zend_Validate_Builder;
$validate->add(new Zend_Validate_IsNumeric, 'phone_no');
$validate->add(new Zend_Validate_Between(1000000, 9999999), 'phone_no');
$validate->add(new Zend_Validate_StringLength(255), 'description', Zend_Validate_Buider::OPTIONAL);
if ($validator->isValid($post)) {
    $this->saveData($post);
}
```

Here, the `phone_no` field is checked to make sure it's numeric, and `description` field is checked to make sure it's less than 255 characters long. Since the Zend_Validate_* classes will usually fail an empty value, all fields are required by default. The `add` method takes an optional third argument, which are flags for that specific combination of field and validator. We're passing the OPTIONAL flag for the `description` field, so Zend_Validate_Builder will skip over it when it's empty, and the StringLength validator will never see it unless it has a value.

Note the `phone_no` example: the order the validators are added to the field is preserved; that is, the IsNumeric validator will be run before the Between validator. All the validators assigned to a field will always be executed. Therefore, a single field might generate several error messages, depending on how many validators it fails. Since these error messages are built up in the Error Manager object, your application code can decide which errors to display to the user; you need not display all of them.

UC-02

This demonstrates a little more advanced usage.

Pattern matching:

```
// Decode HTML entities and strip tags from all POST fields
// Zend_Filter_HtmlEntityDecode is in the Laboratory
$filter = new Zend_Filter_Builder;
$filter->add(new Zend_Filter_HtmlEntityDecode, '/./', Zend_Filter_Builder::PATTERN);
$filter->add(new Zend_Filter_StripTags, '/./', Zend_Filter_Builder::PATTERN);
$filteredPost = $filter->filter($post);

// Require all fields on the form to be Alpha-only characters
$validate = new Zend_Validate_Builder;
$validate->add(new Zend_Validate_Alpha, '/./', Zend_Validate_Builder::PATTERN);
if ($validate->isValid($filteredPost)) {
    $this->saveData($filteredPost);
}

// Say we had a bunch of sequentially-numbered checkboxes starting with 'item_', and we wanted to make
sure they all had {{int}} values:
$validate = new Zend_Validate_Builder;
$validate->add(new Zend_Validate_Int, '/^item_\d+$/', Zend_Validate_Builder::PATTERN);
if ($validate->isValid($formData)) {
    $this->saveData($formData);
}
```

Specifying the PATTERN flag causes the Builder classes to interpret the field name argument as a PCRE regex to be matched against input field names. The given validator or filter is applied to each matching field.

Validating each field in a list:

```
// Filter out non-digit characters from three fields
$filter = new Zend_Filter_Builder;
$filter->add(new Zend_Filter_Digits, array('phone_no', 'ss_no', 'cc_no'));
$filteredPost = $filter->filter($post);

// Make sure none of the address fields are longer than 100 characters
$validate = new Zend_Validate_Builder;
$validate->add(new Zend_Validate_StringLength(100), array('address1', 'address2', 'address3'));
if ($validate->isValid($filteredPost)) {
    $this->saveData($filteredPost);
}
```

Note
Processing arrays of patterns (i.e., array in 2nd parm, with PATTERN flag in 3rd) is currently not supported. However, it is possible (and likely) to be added in the future.

UC-03

Since this follows the Composite pattern, you can nest validators to do some neat things. In this example, assume we're trying to validate a mailing address. Suppose, for organizational purposes, you structured the form so the mailing address was in a sub-array, something like this:

```
$post = array(
    'name' => 'Jimmy',
    'email' => 'jimmy@chickenshack.com',
    'address' => array(
        'addr1' => '123 N Main St',
        'addr2' => '',
        'addr3' => '',
        'city'  => 'Townsville',
        'state' => 'NC',
        'zip'   => '12345',
    )
);
```

You could validate this nested structure like so:

```
$outer = new Zend_Validate_Builder;
$outer->add(new Zend_Validate_StringLength(50), 'name');
$outer->add(new Zend_Validate_EmailAddress, 'email');

$string100 = new Zend_Validate_StringLength(100);

$inner = new Zend_Validate_Builder;
$inner->add($string100, array('addr1', 'city'));
$inner->add($string100, array('addr2', 'addr3'), Zend_Validate_Builder::OPTIONAL);
// Assume $states is a valid array of US states
$inner->add(new Zend_Validate_InArray($states), 'state');
// Another custom validator for Zip codes
$inner->add(new My_Zipcode_Valiadator, 'zip');

// ZVB will pass the entire array in the 'address' field to the inner validator, which will operate
just like the outer does normally.
$outer->add($inner, 'address');
if ($outer->isValid($post)) {
    $this->saveData($post);
}
```

If you have a situation where the entire address is optional, you might be temped to pass the OPTIONAL flag to the `address` field. However, that won't work: when the OPTIONAL flag is set, the Builder checks the value of the current field, and if it's blank, the Builder never calls the validator. Here, the current field would be a six-element array, which will never be considered empty. The elements might be empty, but the array itself won't be. Therefore, the entire `address` field will always be validated against the sub-validator. In this situation, what you actually need is *conditionally* optional fields, which requires completely different logic.

UC-O4

Password confirmation, catpcha validation, etc. This uses the PASS_GROUP flag, and a pair of similar validators called Zend_Validate_Equals and Zend_Validate_AllEqual. Zend_Validate_Equals takes a scalar value and compares it to a standard value passed to the constructor. Zend_Validate_AllEqual takes an array, and returns true if all of its elements equal each other. Both of these can be found in the Laboratory, and both optionally support strict type checking, which is false by default. The PASS_GROUP flag allows you to tell the Builder to pass all the fields named in the second parameter as a single array to the validator. This is handy for validators like Zend_Validate_AllEqual, which examine the contents of several fields in the input to decide if they are valid.

Password Confirmation:

```
$validator = new Zend_Validate_Builder;
// My_Password_Validator is a user-defined (i.e., non-ZF) class the app uses to validate passwords
$validator->add(new My_Password_Validator, 'password1');
$validator->add(new Zend_Validate_AllEqual, array('password1', 'password2'),
Zend_Validate_Builder::PASS_GROUP);
if ($validator->isValid($post)) {
    $this->_addUser($post);
}
```

Catpcha validation:

```
$validator = new Zend_Validate_Builder;
$validator->add('captcha', new Zend_Validate_Equals($_SESSION['captcha']));
if ($validator->isValid($post)){
    $this->_addComment($post);
}
```

UC-05

In order to handle a situation like the optional address mentioned above, you'd need something like this. This uses another new validator called Zend_Validate_AllOrNone. It allows you to make some fields conditionally required. You pass it a set of fields, and will consider them valid under two distinct cases: if either all of them are empty, or if all of them are filled. If only some are filled, then it considers the set invalid. To get the most benefit from this, combine it with other validators. The example may clarify things a bit. This validator can be found in the Laboratory, with the Builder classes.

```
// Put all the conditionally required address fields into an array
$fields = array('address1', 'city', 'state', 'zip')
$reqd   = array_intersect_key($post, array_flip($fields));

$string100 = new Zend_Validate_StringLength(100);

// In this example, the POST data doesn't have the nested array as it did previously
$validate = new Zend_Validate_Builder;
// This is where the magic happens: based on the set of fields passed to its constructor, AllOrNone
will return
// valid if the entire set is either all empty values or all non-empty values. Otherwise, it returns
invalid.
$validate->add(new Zend_Validate_AllOrNone($reqd), $fields);
// After the AllOrNone check, other validators are used to check the validity of the specific fields
$validate->add($string100, array('address1', 'address2', 'address3', 'city'),
Zend_Validate_Builder::OPTIONAL);
// Assume $states is a valid array of US states
$validate->add(new Zend_Validate_InArray($states), 'state', Zend_Validate_Builder::OPTIONAL);
// Another custom validator for Zip codes
$validate->add(new My_Zipcode_Valiadator, 'zip', Zend_Validate_Builder::OPTIONAL);
```

Here, the OPTIONAL flag is passed to all the validators after the AllOrNone validator. This is so those validators don't fail the field in the case when the field is omitted (the fields are still opitonal; albeit *conditionally* optional).

UC-06

Zend_Validate_Builder and Zend_Filter_Builder really promote composing together validators to do interesting things. Here are some examples that show some of neat things that can be done with custom "Decorator" validators. Keep in mind that a Zend_Validate_Builder instance counts as a validator that can be decorated; if you're creative, you can probably come up with some really cool things to do with this.

Zend_Validate_Array
Most validators only take a scalar value to `isValid`. `Zend_Validate_Array` turns an exisiting scalar validator into an array validator. You pass a validator to the constructor, and an array value to `isValid`. It will then iterate over all the elements of the array, performing the given validator on each one. Obviously, you could nest these as much as you needed to, provided that you know the structure of the input array ahead of time.

```
// Suppose the 'selected' field is an array of IDs, which were checkboxes in the UI. The user
// can select any number of them. We just want to make sure they're all int values, and haven't
// been tampered with.
$validate = new Zend_Validate_Builder;
$validate->add(new Zend_Validate_Array(new Zend_Validate_Int), 'selected');
if ($validate->isValid($post)) {
    $this->deleteSelected($post);
}
```

Zend_Validate_AtLeastN and **Zend_Validate_AtMostN**
These two are decorating array validators. They expect an array passed to `isValid`, a validator instance passed to the constructor, and check that at least (or at most) *n* elements in the array pass the validator. This works great with the PASS_GROUP flag when your input data doesn't already have an array in it.

```
// This example is pretty contrived, but bear with me. Assume there are seven fields. The user is
// required to choose at least any three of them, but they can choose no more than five.
$isInt  = new Zend_Validate_Int;
$fields = array('field1', ..., 'field7');

$validate = new Zend_Validate_Builder;
$validate->add(new Zend_Validate_AtLeastN(3, $isInt), $fields, Zend_Validate_Builder::PASS_GROUP);
$validate->add(new Zend_Validate_AtMostN (5, $isInt), $fields, Zend_Validate_Builder::PASS_GROUP);
if ($validate->isValid($post)) {
    $this->saveData($post);
}
```

I haven't come up with a good, realistic example for this, but there's a lot of creative potential here. Also remember that you don't have to always use these two together as I have here. All three of these validators are in the Laboratory.

UC-07

I used to build forms that mimiced my table structure, with the ability to edit multiple records for multiple tables in one form. In this example, assume we want to validate a CRUD form for a simple contact table, but the user can add multiple contacts on the form (with the help of some Javascript in UI). The form fields are laid out in a three-dimensional array, where the first key is the table name, the second key is the row number, and the third key is the field name. It would result in post data that looks something like this:

```
$post = array(
    'contacts' => array(
        0 => array(
            'name' => 'Billy Bob',
            'phone_no' => '555-1212',
            'email' => 'billy@bob.com',
            'notes' => 'Owner of Billy Bob\'s Widgets',
        ),
    ),
);
```

Assume that the user actually entered several contacts, so the 'contacts' array would have several sub-arrays, one for each record, just like a DB query result set.

Here's how to set up a filter/validator that would process all of the records:

```
// The filter structure for the contact "record"
$filterContacts = new Zend_Filter_Builder;
$filterContacts->add(new Zend_Filter_StringTrim, '/./', Zend_Filter_Builder::PATTERN);
$filterContacts->add(new Zend_Filter_HtmlEntityDecode, '/./', Zend_Filter_Builder::PATTERN);
$fitlerContacts->add(new Zend_Filter_StripTags, '/./', Zend_Filter_Builder::PATTERN);
$fitlerContacts->add(new Zend_Filter_Digits, 'phone_no');

// Use the Array filter to validate all the "records" in the set
$filterForm = new Zend_Filter_Builder;
// Zend_Filter_Array is the Filter analog to Zend_Validate_Array (look in the Laboratory)
$filterForm->add(new Zend_Filter_Array($filterContacts), 'contacts');
$post = $filterForm->filter($post);

// The validator structure for the contact "record"
$valContacts = new Zend_Validate_Builder;
$valContacts->add(new Zend_Validate_StringLength(50), 'name');
$valContacts->add(new Zend_Validate_IsNumeric), 'phone_no');
$valContacts->add(new Zend_Validate_EmailAddress, 'email');
$valContacts->add(new Zend_Validate_StringLength(2000), 'notes', Zend_Validate_Builder::OPTIONAL);

// Use the Array validator to validate all the "records" in the set
$valForm = new Zend_Validate_Builder;
$valForm->add(new Zend_Validate_Array($valContacts), 'contacts');
if ($valForm->isValid($post)) {
    $this->saveData($post);
}
```

UC-08

So where have the error messages from all these validators been going all this time? To the Error Manager! This shows how to use the Error Manager both to retrieve raised errors, and define custom error messages.

The `Zend_Validate_Interface` interface defines the method `getMessages`, which returns an array of all the error messages raised by the validator, indexed by error code. `Zend_Validate_Builder` implements this method (albeit, with an incompatible return value), which you can use to retrieve all the error messages that have been raised during the entire validation run. Internally, it delegates this work to the Error Manager; so for retrieving messages, you don't need to touch it directly. You only need to access the Error Manager directly when you want to use it set up custom error messages, or want more control over which errors are returned.

Retreiving errors
Normally, calling `getMessages` on a validator returns a flat array of error messages. But because the Builder manages a whole collection of fields and validators, it's return value is slightly different. It returns a three-dimensional array, indexed by field name, validator class name, and failure reason code. Say the user entered a value in the `name` field that was too short:

```
$userInput = array(
    'name' => 'bal'
);

$zvb = new Zend_Validate_Builder;
$zvb->add(new Zend_Validate_StringLength(4), 'name');
if (!$zvb->isValid($userInput)) {

    // Retrieve the error message
    $messages = $zvb->getMessages();

    // $messages would look like this:
    $messages = array(
        'name' => array(
            'Zend_Validate_StringLength' => array(
                'stringLengthTooShort' => "'bal' is less than 4 characters long"
            )
        )
    );
}
```

Of course, if there were other fields and validators that failed, their messages would all be in the returned array, under the appropriate index.

Custom error messages
You can grab an instance of the Error Manager object to set any number of custom error messages for any combination of field, validator, and failure reason code.

```
$zvb = new Zend_Validate_Builder;

// Get an instance of the Error Manager
$em = new Zend_Validate_Builder_ErrorManager;
// This also equally valid; it creates a new instance if one isn't already present
// $em = $zvb->getErrorManager();

// The error messages can contain the same substitution parameters allowed by the validators
themselves, which is very useful
$em->setTemplate('name', 'Zend_Validate_StringLength', 'stringLengthTooShort', 'Name must be less than
%min% characters long.');
$em->setTemplate('phone_no', 'Zend_Validate_IsNumeric', 'isNumericNot', 'Phone number can only contain
numeric digits.');
$em->setTemplate('phone_no', 'Zend_Validate_Between', 'notBetween', '"%value%" does not appear to be a
valid US phone number.');
// You don't have to override all the possible messages in a validator. It will fall back to the
built-in messages when necessary
$em->setTemplate('email', 'Zend_Validate_EmailAddress', 'emailAddressInvalid', 'Please enter a valid
email address');

$zvb->add(new Zend_Validate_StringLength(100), 'name');
$zvb->add(new Zend_Validate_IsNumeric, 'phone_no');
$zvb->add(new Zend_Validate_Between(1000000, 9999999), 'phone_no');
$zvb->add(new Zend_Validate_EmailAddress, 'email');

if (!$zvb->isValid($post)) {
    // Get the error messages for all the failed fields
    $messages = $zvb->getMessages();
}
```

The Error Manager provides a `hasMessages` method, in addition to `getMessages`. Both of these methods take optional arguments that allow you to filter down to a particular subset of the error messages you might be interested in.

One key benefit of all this is that it allows you to completely separate your error messages from the code that does validation. You could put all your messages in a separate file, or use a Zend_Config object to load all the error messages for a form at once. Among other things, this should make I18N much easier.

Here's one possible way to store errors in an INI file, and set them via Zend_Config. The `setTemplates` method makes this possible, since it accepts an entire array structure of error messages.

errors.ini

```
[Errors]
name.Zend_Validate_StringLength.stringLengthTooShort = Name must be less than %min% characters long.
phone_no.Zend_Validate_IsNumeric.isNumericNot = Phone number can only contain numeric digits.
phone_no.Zend_Validate_Between.notBetween = %value% does not appear to be a valid US phone number.
email.Zend_Validate_EmailAddress.emailAddressInvalid = Please enter a valid email address
```

SomeClass.php

```
// Inside whatever appropriate method...
$zvb = new Zend_Validate_Builder;
$em  = $zvb->getErrorManager();
$cfg = new Zend_Config_Ini('/path/to/errors.ini');
$em->setMessages($cfg->Errors->toArray());
```

UC-09

This use case is more real-world. I'm writing a financial system which consists of individual applications for General Ledger, Accounts Payable, Accounts Receiveable, etc., along with some applications, such as Privilege Licenses, which are designed for local governments. These examples are adapted from some of the Privilege Licenses code (obviously, it's not the actual application code).

This is from a form that allows entry of new License Codes. There are quite a few complex validation requirements on this form. The `amount` and `rate_schedule` fields are mutually exclusive. There are seven account number fields, where each account number field actually consists of two fields: one for the Entity (Company) and one for the Account Number. Account numbers are only valid by Entity.

The seven account fields have several complex requirements. The Cash and Revenue accounts are always required. The Due To and Due From accounts are required only if the Cash Ent and Revenue Ent are different, or (if given) the Cash Ent and AR Ent are different. If present, the Cash Ent and Due To Ent must match each other, and the AR Ent, Reserve AR Ent, Due From Ent, and Prepaid Ent must all match.

I've omitted all but the `validate()` method for the sake of clarity.

```
class PL_LicenseCodesController
{
    public function validateSave(&$form)
    {
        $filter = new Zend_Filter_Builder;

        $filter->add(new Zend_Filter_StringTrim, '/./', Zend_Filter_Builder::PATTERN);
        $filter->add(new Zend_Filter_HtmlSpecialCharsDecode, '/./', Zend_Filter_Builder::PATTERN);
        $filter->add(new Zend_Filter_StripTags, '/./', Zend_Filter_Builder::PATTERN);
        // This runs the Digits filter all the fields that end with "_ent" or "_acct"
        $filter->add(new Zend_Filter_Digits, '/^\[a-z_]+?_(ent|acct)$/',
Zend_Filter_Builder::PATTERN);
        $form = $filter->filter($form);

        // Reuse validators
        $notEmpty  = new Zend_Validate_NotEmpty;
        $isNumeric = new Zend_Validate_IsNumeric;
        $string50  = new Zend_Validate_StringLength(50);
        $string10  = new Zend_Validate_StringLength(10);

        $validator = new Zend_Validate_Builder;

        $validator->add($isNumeric, 'id', Zend_Validate_Builder::OPTIONAL);
        $validator->add($isNumeric, 'code');
        $validator->add($string50, 'title');
        $validator->add($string10, 'general_statute');

        // The user can use either the 'amount' or 'rate_schedule' field, but not both
        $validator->add(new Zend_Validate_AtLeastN(1, $notEmpty),
                        array('amount', 'rate_schedule'),
                        Zend_Validate_Builder::PASS_GROUP);
        $validator->add(new Zend_Validate_AtMostN(1, $notEmpty),
                        array('amount', 'rate_schedule'),
```

```
                Zend_Validate_Builder::PASS_GROUP);
$validator->add($isNumeric, 'amount', Zend_Validate_Builder::OPTIONAL);
$validator->add(new PL_Validate_RateSchedule, 'rate_schedule',
                Zend_Validate_Builder::OPTIONAL);

// Cash Ent/Account and Revenue Ent/Account are required
$validator->add($isNumeric, array('cash_ent', 'revenue_ent'));
$validator->add(new GL_Validate_AccountNo($form['cash_ent']), 'cash_acct');
$validator->add(new GL_Validate_AccountNo($form['revenue_ent']), 'revenue_acct');

// The other account fields are optional
$validator->add($isNumeric, array('ar_ent', 'reserve_ar_ent', 'prepaid_ent'),
                Zend_Validate_Builder::OPTIONAL);
$validator->add(new GL_Validate_AccountNo($form['ar_ent']), 'ar_acct',
                Zend_Validate_Builder::OPTIONAL);
$validator->add(new GL_Validate_AccountNo($form['reserve_ar_ent']), 'reserve_ar_acct',
                Zend_Validate_Builder::OPTIONAL);
$validator->add(new GL_Validate_AccountNo($form['prepaid_ent']), 'prepaid_acct',
                Zend_Validate_Builder::OPTIONAL);

$validator->add(new Zend_Validate_Equals($form['cash_ent']), 'due_to_ent',
                Zend_Validate_Builder::OPTIONAL);
$validator->add(new Zend_Validate_Equals($form['revenue_ent']),
                array('ar_ent', 'reserve_ar_ent', 'due_from_ent', 'prepaid_ent'),
                Zend_Validate_Builder::OPTIONAL);

// If we have an AR Account, and the Cash Ent is different from the AR Ent,
// then the Due To and Due From Accounts are required. Otherwise, if the
// Cash Ent and Revenue Ent are different, the Due To and Due From Accounts
// are required. Two different conditions that end in the same result.
if ((!empty($form['ar_acct']) && ($form['cash_ent'] != $form['ar_ent'])) ||
    $form['cash_ent'] != $form['revenue_ent']) {

    $validator->add($isNumeric, array('due_to_ent', 'due_from_ent'));
    $validator->add(new GL_Validate_AccountNo($form['due_to_ent']), 'due_to_acct');
    $validator->add(new GL_Validate_AccountNo($form['due_from_ent']), 'due_from_acct');
}

if (!$validator->isValid($form)) {
    $this->_forward('editPage');
    return;
}

$this->saveData();
```

```
        $this->_redirect('/pl/license_codes', array('code'=>303));
    }
}
```

UC-10

The fluent interface. This demonstrates the fluent Facade by re-writing UC-07. A lot of the "fluency" of the interface depends quite a bit on the naming of the filter and validator classes.

```
// The filter structure for the contact "record"
$zfb1           = new Zend_Filter_Builder;
$filterContacts = new Zend_Filter_Builder_FluentFacade($zfb1);
$filterContacts->glob('/./')->stringTrim()->htmlEntityDecode()->stripTags();
$fitlerContacts->phone_no->digits();

// Use the Array filter to validate all the "records" in the set
$zfb2       = new Zend_Filter_Builder;
$filterForm = new Zend_Filter_Builder_FluentFacade($zfb2);
// Zend_Filter_Array is the Filter analog to Zend_Validate_Array (look in the Laboratory)
$filterForm->contacts->array($filterContacts);
$post = $filterForm->filter($post);

// The validator structure for the contact "record"
$zvb1        = new Zend_Validate_Builder;
$valContacts = new Zend_Validate_Builder_FluentFacade($zvb1);
$valContacts->name->stringLength(50);
$valContacts->phone_no->isNumeric();
$valContacts->email->emailAddress();
$valContacts->notes->stringLength(2000)->optional();

// Use the Array validator to validate all the "records" in the set
$zvb2    = new Zend_Validate_Builder;
$valForm = new Zend_Validate_Builder_FluentFacade($zvb2);
$valForm->contacts->array($valContacts);
if ($valForm->isValid($post)) {
    $this->saveData($post);
}
```

UC-11

And finally, the big one, re-written with the fluent Facade. A bit less big now.

```
class PL_LicenseCodesController
{
    public function validateSave(&$form)
    {
        $zfb    = new Zend_Filter_Builder;
        $filter = new Zend_Filter_Builder_FluentFacade($zfb);

        $filter->glob('/./')->stringTrim()->htmlSpecialCharsDecode()->stripTags();
  $filter->glob('/^\[a-z_]+?_(ent|acct)$/')->digits();
        $form = $filter->filter($form);

        // The 'namespaces' option is passed to the Validator Factory, so it can find
        // validators other than the default Zend_Validate_* ones.
        $options['namespaces'] = array('GL_Validate', 'PL_Validate');

        $zvb      = new Zend_Validate_Builder;
        $validate = new Zend_Validate_Builder_FluentFacade($zvb, $options);

        $validate->id->isNumeric()->optional();
        $validate->code->isNumeric();
        $validate->title->stringLength(50);
        $validate->general_statute->stringLength(10);

        // Note the shortcut used to create instances of Zend_Validate_NotEmpty
        $validate->each(array('amount', 'rate_schedule'))
                 ->atLeastN(1, $validate->notEmpty())
                 ->atMostN (1, $validate->notEmpty());
        $validate->amount->isNumeric()->optional();
        $validate->rate_schedule->rateSchedule()->optional();

        // Cash Ent/Account and Revenue Ent/Account are required
        $validate->each(array('cash_ent', 'revenue_ent'))->isNumeric();
        $validate->cash_acct->accountNo($form['cash_ent']);
        $validate->revenue_acct->accountNo($form['revenue_ent']);

        // The other account fields are optional
        $validate->each(array('ar_ent', 'reserve_ar_ent', 'prepaid_ent'))->isNumeric()->optional();
        $validate->ar_acct->accountNo($form['ar_ent'])->optional();
        $validate->reserve_ar_acct->accountNo($form['reserve_ar_ent'])->optional();
        $validate->prepaid_acct->accountNo($form['prepaid_ent'])->optional();

        $validate->due_to_ent->equals($form['cash_ent'])->optional();
        $validate->each(array('ar_ent', 'reserve_ar_ent', 'due_from_ent', 'prepaid_ent'))
                 ->equals($form['revenue_ent'])->optional();

        // If we have an AR Account, and the Cash Ent is different from the AR Ent,
        // then the Due To and Due From Accounts are required. Otherwise, if the
        // Cash Ent and Revenue Ent are different, the Due To and Due From Accounts
        // are required. Two different conditions that end in the same result.
        if ((!empty($form['ar_acct']) && ($form['cash_ent'] != $form['ar_ent'])) ||
            $form['cash_ent'] != $form['revenue_ent']) {

            $validate->each(array('due_to_ent', 'due_from_ent'))->isNumeric();
            $validate->due_to_acct->accountNo($form['due_to_ent']);
            $validate->due_from_acct->accountNo($form['due_from_ent']);
        }

        if (!$validate->isValid()) {
            $this->_forward('editPage');
            return;
        }

        $this->saveData();
        $this->_redirect('/pl/license_codes', array('code'=>303));
    }
}
```

Namespaces and Class names
The factories used by the fluent Facades attempt to find classes by searching through an array of namespace prefixes. If you have filter or validator classes with the same "base name" in two different namespaces, only the first one found will be returned. There's no way to pick which namespace to use on a call-by-call basis. You can easily replace the factories with your own in order to implement different instantiation rules. The standard Builder classes are not affected by this problem.

9. Class Skeletons

Zend_Validate_Builder

```
class Zend_Validate_Builder implements Zend_Validate_Interface
{
    const OPTIONAL   = 1;
    const PATTERN    = 2;
    const PASS_GROUP = 4;

    protected $_validatorTable;
    protected $_errorManager;
    public $dataSet;

    public function __construct(array $dataSet = null) { }
    public function clear() { }
    public function getErrorManager() { }
    public function setErrorManager(Zend_Validate_Builder_ErrorManager_Interface $em) { }
    public function setFlags($id, $flags = 0) { }
    public function addFlags($id, $flags = 0) { }
    public function getFlags($id) { }
    public function add(Zend_Validate_Interface $validator, $fieldSpec, $flags = 0) { }
    public function isValid($dataSet) { }
    public function getMessages() { }
    public function getErrors() { }
}
```

Zend_Validate_Builder_FluentFacade

```
class Zend_Validate_Builder_FluentFacade implements Zend_Validate_Interface
{
    protected $_builder;
    protected $_options;
    protected $_factory;

    public function __construct(Zend_Validate_Builder $builder, array $options = array()) { }
    public function getBuilder() { }
    public function setBuilder(Zend_Validate_Builder $builder) { }
    public function getFactory() { }
    public function setFactory(Zend_Validate_Builder_ValidatorFactory_Interface $factory) { }
    public function getOptions() { }
    public function setOptions(array $options) { }
    public function glob($pattern) { }
    public function each() { }
    public function group() { }
    public function __get($field) { }
    public function __call($name, $args) { }
    protected function _getAdder($fieldSpec, $flags = 0) { }
    public function isValid($data) { }
    public function getMessages() { }
    public function getErrors() { }
}
```

Zend_Validate_Builder_FluentAdder

```
class Zend_Validate_Builder_FluentAdder
{
    protected $_builder;
    protected $_factory;
    protected $_fieldSpec;
    protected $_flags;
    protected $_rowIds;

    public function __construct(Zend_Validate_Builder $builder,
                                Zend_Validate_Builder_ValidatorFactory_Interface $factory,
                                $fieldSpec,
                                $flags = 0) { }
    public function optional($value = true) { }
    public function __call($name, $args) { }
}
```

Zend_Validate_Builder_ValidatorFactory

```
class Zend_Validate_Builder_ValidatorFactory implements
Zend_Validate_Builder_ValidatorFactory_Interface
{
    protected $_options = array();

    public function __construct(array $options = array()) { }
    public function getOptions() { }
    public function setOptions(array $options) { }
    public function create($name, $args) { }
    public function namespaceLoad($basename, array $namespaces = null, $dirs = null) { }
}
```

Zend_Validate_Builder_ValidatorFactory_Interface

```
interface Zend_Validate_Builder_ValidatorFactory_Interface
{
    public function create($name, $args);
}
```

Zend_Validate_Builder_ErrorManager

```
class Zend_Validate_Builder_ErrorManager implements Zend_Validate_Builder_ErrorManager_Interface
{
    protected $_messages;
    protected $_raised;

    public function __construct() { }
    public function setTemplate($field, $valClass, $reason, $message) { }
    public function setTemplates(array $errors) { }
    public function getTemplates($field = null, $valClass = null, $reason = null) { }
    public function raise($field, Zend_Validate_Interface $val) { }
    public function getMessages($field = null, $valClass = null, $reason = null) { }
    public function hasMessages($field = null, $valClass = null, $reason = null) { }
    public function clear($field = null, $valClass = null, $reason = null) { }
    protected function _createMessage($message, Zend_Validate_Interface $val) { }
}
```

Zend_Validate_Builder_ErrorManager_Interface

```
interface Zend_Validate_Builder_ErrorManager_Interface
{
    public function raise($field, Zend_Validate_Interface $val);
    public function getMessages($field = null, $valClass = null, $reason = null);
    public function clear($field = null, $valClass = null, $reason = null);
}
```

Zend_Filter_Builder

```
class Zend_Filter_Builder implements Zend_Filter_Interface
{
    const PATTERN = 1;

    protected $_filterTable;
    public $dataSet;

    public function __construct(array $dataSet = null) { }
    public function clear() { }
    public function setFlags($id, $flags = 0) { }
    public function addFlags($id, $flags = 0) { }
    public function getFlags($id) { }
    public function add(Zend_Filter_Interface $filter, $fieldSpec, $flags = 0) { }
    public function filter($dataSet) { }
}
```

Zend_Filter_Builder_FluentFacade

```
class Zend_Filter_Builder_FluentFacade implements Zend_Filter_Interface
{
    protected $_builder;
    protected $_options;
    protected $_factory;

    public function __construct(Zend_Filter_Builder $builder, array $options = array()) { }
    public function getBuilder() { }
    public function setBuilder(Zend_Filter_Builder $builder) { }
    public function getFactory() { }
    public function setFactory(Zend_Filter_Builder_FilterFactory_Interface $factory) { }
    public function getOptions() { }
    public function setOptions(array $options) { }
    public function glob($pattern) { }
    public function each() { }
    public function __get($field) { }
    public function __call($name, $args) { }
    protected function _getAdder($fieldSpec, $flags = 0) { }
    public function filter($data) { }
}
```

Zend_Filter_Builder_FluentAdder

```
class Zend_Filter_Builder_FluentAdder
{
    protected $_builder;
    protected $_factory;
    protected $_fieldSpec;
    protected $_flags;
    protected $_rowIds;

    public function __construct(Zend_Filter_Builder $builder,
                                Zend_Filter_Builder_FilterFactory_Interface $factory,
                                $fieldSpec,
                                $flags = 0) { }
    public function __call($name, $args) { }
}
```

Zend_Filter_Builder_FilterFactory

```
class Zend_Filter_Builder_FilterFactory implements Zend_Filter_Builder_FilterFactory_Interface
{
    protected $_options = array();

    public function __construct(array $options = array())
    public function getOptions()
    public function setOptions(array $options)
    public function create($name, $args)
    public function namespaceLoad($basename, array $namespaces = null, $dirs = null)
}
```

Zend_Filter_Builder_FilterFactory_Interface

```
interface Zend_Filter_Builder_FilterFactory_Interface
{
    public function create($name, $args);
}
```

Get the Code!
The full implementation of these classes can be checked out from the Zend Laboratory SVN repository: http://framework.zend.com/svn/laboratory. The source code is in the `library/Zend/Filter` and `library/Zend/Validate` directories. The unit tests are in `tests/Zend/Filter` and `tests/Zend/Validate`.

]]></ac:plain-text-body></ac:macro>
]]></ac:plain-text-body></ac:macro>