Xaraya
Request for Comments: 0054
Obsoletes: 0018
STD: 2
Category: Standards Track

M.R. van der Boom
Xaraya Development Group
January 2006

Exception handling

Status of this Memo

This document specifies an Xaraya standards track protocol for the Xaraya community, and requests discussion and suggestions for improvements. Please refer to the current edition of the “Xaraya Official Standards” (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

Copyright Notice

Abstract

The exception handling in PHP applications has been problematic to say the least. PHP version 4 has no support for real exceptions. RFC-0018 documents the Xaraya solution to this problem. As we are moving into the new series of releases for Xaraya and PHP 5 is very commonplace now, this RFC documents the replacement of the Error handling system we had. The replacement is a true exception handling system, still limited by what is supported in PHP but nevertheless much more useful than the older system.

This RFC documents the design, implementation and usage of the new Exception Handling System (EHS) for Xaraya.

Table of Contents

1 Introduction and concept overview 3

2 Exception components description 4

2.1 Organisaton of exception types [Classes] 4

2.2 Handlers 5

3 Implementation 6

3.1 Interfaces 6

3.2 Class implementation 7

4 Using exceptions in your code 8

4.1 Throwing exceptions 8

4.2 Catching exceptions 8

5 Advanced topics 10

5.1 Handler implementation 10

5.2 Defining extra groupings 11

6 Compatibility notes 12

7 Revision history 13

Author's Address 14

Intellectual Property and Copyright Statements 15

1. Introduction and concept overview

We will assume you are familiar with the concept of exceptions in general. There are great resources on the internet which talk about how exceptions work and how they can or should be used with different languages. For reading this RFC, you as a developer, a basic knowledge of the exceptions documentation in the PHP manual is sufficient to understand the exception handling subsystem described in this document.

In this RFC we use the words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL". They are to be interpreted as described in (the IETF) RFC-2119[1].

[1] http://rfc.net/rfc2119.html

2. Exception components description

The exception handling system (to be called EHS hereafter) consists of 2 parts:

1. Classes which model certain types of exception, and of which object instances contain the exception information.
2. Handlers which deal with exceptions in different ways, depending on the context, type of exception or other parameters.

2.1 Organisaton of exception types [Classes]

Every exception ever raised within xaraya MUST ultimately be an instance of the native PHP Exception class. All other exceptions have that class in their ancestry.

Xaraya extends the native exception class in two stages. The first caters for the integration with the outside world and PHP itself, while the second stage takes care of presenting an exception library to other Xaraya components.

The first stage exposes exceptions which are either from outside Xaraya or low level exceptions. Xaraya itself defines `PHPException` and its own `XARExceptions`. Third party libraries fill the third and further up ranks in the first stage. They are REQUIRED to expose exactly one top level class to xaraya. It is assumed that if this is not natively the case it is trivial to do so ourself by defining a wrapper class in a convenient place. Top level classes like this SHOULD have a 3 letter capital prefix, so they are easily distinguishable as either being a low level type of exception (ironically, at the top of the hierarchy) or as being defined outside Xaraya.

The first stage inheritance looks like this:

```
Exception            - native PHP class
  |--> 3RDException  - 3rd party exceptions are assumed to fill this space
(example:SQLException from creole)
  |--> PHPException  - the PHP error handler raises instances of this exception
class
  |--> XARExceptions - All Xaraya exceptions inherit from this class.
         |--> ...second stage inheritance...
```

[rfc.comment.1]

Figure 1: First stage inheritance

The second stage inheritance defines logical groupings for the internal exceptions that Xaraya raises. The EHS itself does only provide those groupings (with some exceptions, which we will see later on). These groupings are useful to catch groups of exceptions at once and to define interfaces for them as a whole, because they somehow signal the same sort of unexpected situation.

Starting at the `XARExceptions` class the second stage inheritance tree looks like this:

```
|--> XARExceptions
       |--> ConfigurationExceptions
       |--> DebugExceptions
       |--> DeprecationExceptions
       |--> DuplicationExceptions
       |--> NotFoundExceptions
       |--> ParameterExceptions
       |--> RegistrationExceptions
       |--> ValidationExceptions
       |--> OtherExceptions
```

[rfc.comment.1] Note that it is XARExceptions (plural) whereas the others are singular. This is to signal that Xaraya can make a distinction between different types of its own Exceptions, but only sees the top level of third party exceptions and the top level Xaraya exceptions.

Figure 2: Second stage inheritance

Components within Xaraya MAY derive new classes as long as they have the `XARExceptions` class in their ancestry. The details and interfacing of the classes are documented in Section 3.2.

2.2 Handlers

Once an exception is raised, either on purpose while running some code, or by some subsystem, it MUST be handled. If an exception is not handled by anything PHP will issue a message saying so. What an exception handler actually does is dependent on the context and the type of Exception raised. In most cases, a (specifically formatted for the context) error message SHOULD be displayed at least.

We distinguish two sorts of handling:

1. global exception handlers
2. local exception handlers

The global exception handlers are replacements for the default handling of PHP with something more appropriate. The default handling of PHP is to just display a message that the exception is unhandled. Handlers are activated in php by calling the function `string set_exception_handler(callback exception_handler)` .

At any given time, there MUST only be one active exception handler. The function above returns the name of the previously registered handler as extra information, but PHP itself keeps track of the handler stack too. Implementors MAY set exception handlers to a handler they prefer in their context, but MUST call `restore_exception_handler()` to restore the previous handler when their subsystem goes out of context.

Handlers are responsible for handling the exception raised. This usually means making sure Xaraya can end the request in a useful way. Displaying an error message or sending an error code to the client are two examples. The details of the handler implementation are documented in Section 5.1.

3. Implementation

3.1 Interfaces

The base interface definition for all exceptions in the Exception Handling System is as follows:

```
interface IException
{
   /* Constructor */
   function __construct($message = null, $code = 0);

   /* Fixed, not overrideable */
   final public function getMessage();        // message of exception
   final public function getCode();           // code of exception
   final public function getFile();           // source filename
   final public function getLine();           // source line
   final public function getTrace();          // an array of the backtrace()
   final public function getTraceAsString();  // formatted string of trace

   /* Overrideable */
   public function __toString();              // formatted string for display
}
```

[rfc.comment.2] [rfc.comment.3]

Figure 3: Base Exception interface

The exception class `PHPException` does not further extend this interface but uses it directly. In practice this exception SHOULD NOT occur. If it does there is a certain bug in the code, always.

The interface definition for exception handlers:

```
interface IExceptionHandlers
{
    /* Overrideable */
    public static function defaulthandler(Exception $e);
    public static function phperrors(Integer $errorType, String $errorString,
String $file, Integer $line);
}
```

Figure 4: Exception handler interface

The exceptions raised by Xaraya itself, i.e. the instances of `XARExceptions` and its descendents have a modified interface definition:

```
interface IXARExceptions extends IException
{
    /* Constructor */
    final public function __construct(Array $vars = NULL, String $message = NULL);

    /* Overrideable */
    public function getHint();    // if available return a hint on how to pursue
solving the exception
}
```

[rfc.comment.2] The name IException is made up here, it is not actually documented as such in the PHP documentation. See The PHP manual for detailed information.

[rfc.comment.3] The PHP language does not actually allow specifying final in an interface declaration. It is used to clarify the declaration here. In the actual PHP file, the final keyword appears before the implementation of the method.

This extension of the interface further restricts Xaraya exceptions to the point where we safeguard their construction to one specific way. All Xaraya exceptions MUST be constructed in exactly the same way. The inherited `__toString()` method stays overrideable. (Note that this is a so called 'magic' method.)

Figure 5: XAR Exceptions extended interface

The parameters have the following meaning:

$vars - array of variable parts in the message (optional)

$message - string which holds a descriptive message for the exception raised. (optional)

We will see later on that each exception type defines a default message, possibly having variable parts. In most cases when throwing an exception it is RECOMMENDED to only specify the variable parts, so all exceptions of the same type use the same description of the Exception information.

3.2 Class implementation

The classes defined directly as part of the EHS, that is, the classes in the first stage of inheritance and the grouping classes described above, are declared `abstract`. To be able to throw specific exceptions a subsystem MUST define it's own derivative of one of the classes.

If your exception falls into one of the grouping classes and you want to belong to a certain group, you can **declare** your exception as (taking a configuration exception as example):

```
class BrainConfigurationException extends ConfigurationExceptions
{
    protected $message   = 'Your brain (part: #(1)) has been misconfigured.';
    protected $variables = array('Visual stimuli');
    protected $hint      = 'Not much we can do, we suggest to try wearing glasses
first';
}
```

Put this declaration of your exception in a place where it is available in the scope where this exception type may be raised. Other than this, nothing more is needed to implement your exception class for a minimal setup. Exceptions can be raised by code at that point and caught by try/catch constructs. The registered handlers will deal with this exception type from there on.

The three variables declared are the default values for the message, variable parts in the message and the accompanying hint for this type of exception. The 3 variables SHOULD specify values which are applicable for all situations where this exception can occur. The $message and $variables can be overridden when throwing an exception, but the $hint is inherently coupled to the declaration and cannot be changed.

4. Using exceptions in your code

It is important that your subsystem uses code which is suitable for exception oriented working. While it is not possible to give exact rules, there are a number of common practices which you may find useful.

Dealing with exceptions consist of two parts:

1. Raising or throwing exceptions
2. Catching or handling exceptions

4.1 Throwing exceptions

Once you know something has gone wrong by testing for a certain condition and you want to signal an exceptional situation or an error condition, the first thing to do is to determine as precisely as possible what type of exception to raise. Your code SHOULD NOT raise a PHP native exception but only exceptions derived from the `XARExceptions` class or further down the hierarchy. This can initially require some digging through the code, but since all exceptions defined SHOULD have meaningful names, over time it will get easier to know what type of exception to raise.

```
Some examples:

-- throwing exception using the defaults.
if($vision == 'fuzzy')
    // Probably not wearing glasses.
    throw new BrainConfigurationException();

-- adapting the message and variable parts
if($vision == 'none' and $glasses->state == 'on') {
    // Hmm, not sure what's going on here.
    $vars = array($glasses->getStrength());
    $msg = 'No visual stimuli, despite the fact that  you are wearing glasses of
strenght: "#(1)"';
    throw new BrainConfigurationException($vars,$msg);
    echo "This never shows"; // never gets executed.
}
```

Throwing an exception immediately stops running the code at that point and transfers control to the active exception handler at that point. In the second example, the line after the throw will never get executed by PHP

4.2 Catching exceptions

Assuming every part of Xaraya uses exceptions, what to do if one is thrown? (mostly by code of others) Since exceptions stop executing code immediately and the handler takes over, how to keep control of things in your code?

The idea is to identify *risky* parts of your code and enclose them in a so called "try/catch" block to keep you in control instead of an exception handler. By keeping that control you are also responsible for handling any exception raised inside that try/catch block.

The handling MAY consist of ignoring the exception if that is appropriate, or inspecting the situation and decide to let the handler take care of it after all. It all depends on the situation. Some examples:

```
-- non issue test
$rfc = new rfc('0054');
try {
    $person->read($rfc);
    $person->interpretReading();
} catch(BrainConfigurationException $e) {
    // If person is awake, it's real, let the handler deal with it
    // otherwise not sure what this person is doing in his sleep, but lets ignore it.
    if($person->state != 'asleep') throw $e;
}
```

```
-- repair
try {
    $person->WakeUp();
    $person->PickUp($glasses);
    $person->PickUp($coffee);
    $person->read($rfc);
} catch(ConfigurationExceptions $e) {
    // Something went wrong, repair and notify
    $person->FreeHands();
    throw $e;
} catch(UnderstandingException $e) {
    // Be easy, it's morning, ignore if the person doesnt understand a thing
}
// Any other exception raised will get caught by the default handler.
```

The first example tests a condition and rethrows the exception if the person is not asleep, since only then it constitutes a real exception.

In the second example, the catch clause specifies the whole `ConfigurationExceptions` group to be caught. Any exception deriving from that class will get caught by that line, not only the `BrainConfigurationException` example we used earlier. Further more the example contains an extra catch, which catches the person not understanding what he is doing, and just ignores that, since it's early morning.

If your code does not use a try/catch clause but an exception is raised anyway, the then active handler will pick it up and deal with the exception. In many cases, compared to existing code, you can just (try to) do in your function what you want to do, instead of testing for every possible situation first. Inserting the try/catch block in strategic places only is usually enough, assuming the code you use from other parts is reasonably well behaving.

It is not advisable to suppress exceptions if you are not sure you are able to handle the situation completely. Especially if your code is a library, you may never want to suppress the exception. It is the callees responsibility to use a proper try / catch construct and decide what to do if your library raises an exception.

5. Advanced topics

5.1 Handler implementation

An exception handler implementation is a special piece of code where the amount of freedom allowed is very limited. The code in an exception handler is guaranteed to be the last in the request, no code execution takes place after the code ends up in a handler.

The Xaraya core defines one class (called: ExceptionHandlers) which has no predefined interface. The class defines the following handlers (methods):

bone - a handler which is as simple as possible, functions as fallback for the other handlers.

defaulthandler - normal handler, displays errors based on a xar template, falls back to the bone handler when something goes wrong

phperror - this is actually an 'error' handler for php errors occurring. The handler collects information and throws a `PHPException`, after which the 'normal' exception handlers take over. Assertion failures (caused by the `assert()` function) also get handled by this handler.

The handlers defined by default assume that a (X)HTML capable client is making the request. [rfc.comment.4] If a part of Xaraya can not use the normal handlers in its context, it SHOULD define it's own exception handler and make sure it gets activated and deactivated at the appropriate time.

All situations where Xaraya produces other content than XHTML are candidates for dedicated exception handlers. (rss, xmlrpc, pdf etc.). In those areas, special considerations are needed to handle exceptions. In some situations an error condition is clearly definied within the area. For example, when acting as an xmlrpc server, the exception handler would make sure to produce a properly formatted xmlrpc error response and send that back to the client. Similar actions would be needed when acting as a webdav or flash server.

If a dedicated exception handler is needed, the implementation MUST derive from the `ExceptionHandlers` class defined and it is RECOMMENDED that the class includes the plural 'ExceptionHandlers' as a postfix for the classname. For example, exception handlers for the xmlrpc protocol SHOULD be defined in the `XMLRPCExceptionHandlers` class. The class MUST define its handler methods as `public static` in the class. The signature for an exception handler is as follows:

```
public static function handler(Exception $e)
```

The handler gets passed in the Exception object when it gets called and your handler can use the interface described in Section 3.1 to extract the proper information from that object to handle it.

In the implementation of your handler you MAY use the Xaraya API from core or other modules, but in that case you MUST also provide a fallback handler and enclose the use of the API in a try/catch block.[rfc.comment.5]

```
Example of a very simple dedicated handler:

class TestExceptionHandlers extends ExceptionHandlers
{
    public static function testhandler(Exception $e)
    {
        echo "In my exception handler Error:". $e->getMessage();
    }
}
// Make sure we use it.
set_exception_handler(array('TestExceptionHandlers','testhandler'));
```

[rfc.comment.4] This is VERY likely to change in the future.

[rfc.comment.5] This may seem a bit counterintuitive to catch and raise exceptions within an exception handler, but this does work. See the implementation of the default handlers to see how this works.

Note that in the example the default handler of Xaraya can still be used. If you want to make sure the default handler is NOT being used, which would be the typical situation, make sure the 'defaulthandler' method is overridden by your class in the appropriate way. In this way both extending and replacing the way exceptions are handled is possible. [rfc.comment.6]

While possible, dedicated handlers SHOULD NOT override the 'phperror' handler. That handler just collects information and raises a new exception which will be handled by the active exception handler. (like the one made active in the example, all exceptions, including php errors and assert failures end up in your handler)

5.2 Defining extra groupings

If your xaraya subsystem is larger and you want to be able to group your exceptions like xaraya itself does, this is possible. This can be comfortable if you want more control over the exceptions raised **within** your own subsystem and react differently to them. If you feel this is appropriate for your subsystem, you can create a derivation of the `XARExceptions` class (or any of the existing groupings)

```
abstract class MySpecialConfigurationExceptions extends ConfigurationExceptions
{}
```

Other than the declaration, nothing else is needed. Obviously your own exceptions should extend your own grouping for it to be useful. The class declaration MUST be abstract to avoid that instances of that grouping class are instantiated.

[rfc.comment.6] We actually wanted to call the method 'default' instead of 'defaulthandler' but for some reason PHP does not allow that

6. Compatibility notes

This section contains an assortment of notes comparing earlier Xaraya code with the news Exception Handling System.

1. The `xarErrorSet()` API function which was used to signal error conditions, often in combination with a return NULL; should not be used anymore. The function will continue to work until all code has been ported. The effect of using xarErrorSet() is that a special exception will be raised which, apart from handling the error being set, will also display a deprecation note.
2. A null return in old code used to signal an error condition, which was our way to let errors bubble up until our errorhandler could pick them up. In general it is now not needed anymore to take nullreturns into account. This means that functions and methods from this time onwards may have a null return type as their signature, but it also means you SHOULD NOT use the test for that null value anymore to check for error conditions. In practice, this will resolve itself, since xarErrorSet() raises an exception and the test for null is never executed.
3. Related to the above, it is now possible and RECOMMENDED that functions / methods either return a value of the type their declaration specified OR throw an exception and nothing else. In other words, dont return -1 or 0 if something can not be found, but raise a NotFoundException and let the callee handle that.
4. Since at any given time, there can only be one pending exception, the error stacks we maintained earlier do not exist anymore. If your code relies on building a stack of errors and dealing with them, you will have to rewrite those parts of your code.
5. The API functions like xarCurrentErrorType, xarCurrentErrorID, xarCurrentError, xarErrorFree, xarErrorHandled, xarErrorGet and xarErrorRender have no meaning anymore in the new system. They were related to and dependent on the error stacks, which are not there anymore. They have been temporarily provided in core as wrappers which just return and dont do anything.

7. Revision history

Instead of documenting every revision in here, we refer to our monotone repositories, which can give your great details on what has been changed in this RFC over time.

2006-01-20: initial revision.

Author's Address

Marcel van der Boom
Xaraya Development Group
EMail: marcel@xaraya.com
URI: http://www.xaraya.com

Intellectual Property Statement

The DDF takes no position regarding the validity or scope of any Intellectual Property Rights or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; nor does it represent that it has made any independent effort to identify any such rights. Information on the DDF's procedures with respect to rights in standards-track and standards-related documentation can be found in RFC-0.

The DDF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights which may cover technology that may be required to practice this standard. Please address the information to the DDF Board of Directors.

Acknowledgement

Funding for the RFC Editor function is provided by the DDF