Xaraya
Request for Comments: 0050
Category: Best Current Practice

M. Lutolf
Xaraya Development Group
February 2004

Xaraya Component Hierarchy

Status of this Memo

This document specifies a Xaraya Best Current Practices for the Xaraya Community, and requests discussion and suggestions for improvements. Distribution of this memo is unlimited.

Copyright Notice

Abstract

This RFC presents a proposal for a universal system of defining component objects ("xarlets") in Xaraya and managing the relationships between them.

```
Caveat:
Although it makes sense to define such components as OO objects, their properties
and the relationships
among them as defined are Xaraya specific. This means that although similar
concepts such as inheritance
exist among Xaraya components as they do between OO objects, the two are not
identical.
```

This RFC does not look at content issues within the components themselves. Rather, it covers the way different components in Xaraya can be related to each other, and how they can interact in creating, modifying and sharing content.

The proposal is conceptually divided into two parts. In the first we define the formalism to manage the diverse types of components that Xaraya supports in a consistent fashion. This includes setting up a standard API that all components adhere to.

In the second part we look at the way in which components can interact with each other through their API.Some basic structures for the resulting object hierarchy are defined, as well as mechanism for dynamically creating richer structures.

Table of Contents

1. Definition of terms

- Xaraya Component or Xarlet: a general term to describe components objects in the Xaraya system with a common API.
- Content component: any object in Xaraya that contains content that can be accessed, modified. displayed and executed.

Example: users, articles, privileges and xarBB posts are all content components.

- Code component: any xarlet in the Xaraya codebase that has an object wrapper ,that adheres to the component API and contains content that can be accessed, modified. displayed and executed. The terms content and code component are used to underline their respective natures, but in practice the distinction is not always clearly defined.

Example: xarModAPIFunc('roles','user','getall') can be defined as a code components.

- Component Hierarchy: A formal structure in Xaraya that manages definitions of xarlets and how they interact.
- Assignment: refers to a unidirectional ternary operator between 2 xarlets, according to a definition by a third xarlet.

Examples: a privilege is assigned to a role according to a definition of "privilege assignment", or an an article is assigned to a role according to a definition "author".

- Content Types: defines the type of a xarlet. This is a way of classifying xarlet. A xarlet can have many types, The type definitions themselves are xarlet.

Example: "privilege", "sales order", "category" and "type" are xarlet (content component) types

2. Introduction

- The main points of this RFC are:
    - Data and code in Xaraya can be cast as components (xarlets) and managed in a consistent manner through standard APIs.
    - This model initially focusses on "end-user content", i.e. content objects that a user of the system interacts with directly, such as articles, posts or groups. But it can also (hoprfully) be extended to components that the system uses internally, such a module functions.
    - The concepts of xarlets and assignments can be used as the basis for a model describing the relationships between different types of components used in the Xaraya framework.
    - By defining a standard Xaraya API for assignments, it becomes possible for conponents to interact in a standardized fashion. In particular it becomes possible to create a standard for data exchange between components of different types. At the lowest level this amounts to just restating (in a different terminology) how API function calls are currently performed. Looking a bit further, it is giving a standard description to higher level mechanisms such as hooks and dynamic data. At the limit it means *extending* such mechanisms to the point where any two modules, or more generally any two components, can "talk to each other".
- Benefits:
    - Standardization of data exchange between modules
    - Xarlet "owned" privileges
    - Elimination of redundant code (hierarchies in roles, privileges, categories, comments).
    - Elimination of redundant tables (xar_security_acl, xar_privmembers, xar_rolemembers, others)
- This RFC does not cover:
    - How content is handled within modules or xarlets. It does however attempt to define a standard interface these objects need to show to the world.
    - What should or should not be considered an component/xarlet. This is an issue to be decided by each module (e.g. is a pubtype a component?). At the limit everything in Xaraya could be considered a component, but this may not always be feasible or necessary. This is another way of stating that I believe the component model and the "traditional" 3-part function call model can coexist and complement each other. The initial point to be reached is one where the API part is largely component based, while the GUI part is (probably) largely "traditional".

3. Opening Remarks

There are a number of reasons for putting forth this proposal, but most of them in some way come under the heading of "trying to make it easier for parts of Xaraya to work together".

The philosophy underlying the Xaraya applications framework stresses the importance of small, modular, reusable components that can be chained together to form larger units of enhanced functionality. Where previously developers would create large, mission-specific modules, Xaraya has been built in a way that lets them draw from a pool of common components to do the job.

However, the success of this approach has also shown its limits. People are finding more sophisticated ways in which they would like modules to interact, and finding it increasingly difficult to do so in a clear and easy way. This document tries to sketch out a way to do just that.

Illustration 1: Many modules want to interact with the articles module in order to create new functionality: forums, blogs etc. One of the problems to be solved is finding a way to make this easy, and automatically adjusting the privileges of the modules so that the privilege interface is seen to be that of a single complex component, rather than a collection of modules glued together.

Illustration 2: E-commerce modules such as shopping carts and payment (e.g. PayPal) modules want to link up with the ledger module in order to incorporate an accounting backend. Here the problem is one of a single standard interface for exchanging relatively complicated types of data from different sources.

This is an attempt to establish a common way of describing different functionalities of the Xaraya framework, which is hopefully extended to new functionality as it is added. While the implemention at a code level may vary, what is important is that the concepts are commonly understood and can be applied to different situtions in the framework.

If we look at the way Xaraya is structured today we have three discrete areas:

1. The 'production' part -> based on a state of a storage (database) produces the "content objects" to be assembled. The content is decided based upon the input parameters supplied by human or process.
2. The 'assembly' part-> takes the information delivered by the guts of xaraya and manipulates and assembles it into an information set to be cast into a given from through a rendering process.
3. The 'rendering' part-> takes the asseembled information and creates an output document.

Schematically:

```
           Production     --->     Assembly     --->     Rendering

              API                    GUI                 Blocklayout
```

This RFC deals mainly with the 'production' part, and can hopefully be extended to the assembly part.

The formalism developed here is cast in a way to take advantage of both PHP's OO syntax and Xaraya's 3 part function syntax. On the one hand the latter is more familiar and accessible to Xaraya developers, on the other we want to highlight that the object definitions in this proposal are not objects in the OO sense. Rather, they are components specific to the Xaraya applications framework, with their own rules and structures.

The structures proposed here do not (hopefully) call for a rewrite of the codebase. [MrB: "famous last words"] Rather they represent a compatible extension of what is already available. The proposal should also be seen as a generalization of those instruments of data exchange already present, such as hooks and dynamic data.

4. Concepts of the Component Hierarchy

Defining xarlets is a way of treating the different things that Xaraya processes in a uniform way. The specifics of how the content is arranged within the xarlet are not of interest to the hierarchy. Nor is the way the data is managed or called from the UI, which will be specific to each module.

Instead, the hierarchy is interested in the ways xarlets deal with each other. Within a given module, components today have module-specific ways of interacting. To a certain degree this may persist into the future. Towards the hierarchy, however, they expose standard properties and have a standard API. The definition of this API needs further work, as it must incorporate a dynamic element that allows two components to decide whether and to what degree they are compatible and what sorts of interaction is possible. The hierarchy therefore offers a bridge mechanism through which components from different modules can deal with each other.

4.1 The Component API

For convenience the component hierarchy code is put into a new module called System. (This arrangement can be changed in the future without impacting anything) The module's API can be accessed via calls of the type:

```
        $result = xarModAPIFunc('base','object',$function);
```

4.2 List of API Functions

The following functions are available via the syntax above:

```
        Function                Description
        ---------------------   -----------------------
        createobject            adds a new object definition to the register
        updateobject            updates an object definition in the register
        deleteobject            deletes an object definition in the register
        getobject               returns an object in the register
        getallobjects           returns all objects according to specific criteria

        createassignment        adds a new assignment to the assignment table
        updateassignment        updates a assignment in the assignment table
        deleteassignment        deletes a assignment in the assignment table
        getassignment           returns a assignment in the assignment table
        getallassignments       returns all objects according to specific criteria
```

The first group of functions deals with managing objects themselves, the scond groups deals with managing relationships between the available objects, which is how the hierarchy is created.

5. Xaraya Components

5.1 Content and Code Components

Xaraya components can be catalogued in various types for the purposes of the hierarchy. As such they have a standard interface toward the outside world. From a structural point of view they can be distinguished as content or code components.

Content components are, loosely speaking, structured datasets. In a simplest form a content components is an object wrapper around a record from a database table. More complex content components can contain data from different data tables or different data sources.

Code components, on the other hand, are snippets of code from the Xaraya codebase. These themselves may or may not be objects in the PHP OO sense. A simple example of such an object would be a Xaraya GUI or API function, such as

Note: The distinction between content and code components is a conceptual one, namely that the two types are different "kind of things", coming from different sources. But from the API perspective there really should be only one kind of component.

```
            xarModAPIFunc('roles','user','getall');
```

Components can be called from the object register using the getobject call above.

```
            $object = xarModAPIFunc('system','object','getobject',
                          array('module'  => 'roles',
                                'type'    => 'user'
                                'function => 'getall'
                                )
                      );
```

or an equivalent shorthand form:

```
            $object = xarGetObject('roles','user','getall');
```

Once available, the component can then be manipulated via its interface, as outlined below.

5.2 Component Interface

Every Xaraya component implements the following interface:

```
            Method                  Description
            ---------------------   -----------------------
            getID                   returns the component's ID
            getOID                  returns the component's local ID reference
            getName                 returns the component's name
            getDisplayName          returns the component's display name
            getResult               returns the result of executing a code component,
or null
            getContent              returns an array of the component's content values
            getContentNames         returns an array of the names of the component's
content values
            toArray                 returns the components public (PHP5 concept) fields
in the
                                                form of an array

            setContent              sets content in the component
            run                     excutes the component's operator function

            (to be completed)
```

6. The Component Hierarchy

Once we have component and their interface in place, we can combine them to create complex extensible structures.

Fundamental to the description of any hierarchy is the concept of relationships. By defining rules for the way relationships can be built and how they create interaction between component will define the richness of the hierarchy that can be built.

6.1 Assignments

Assigments in Xaraya represent unidirectional relationships between two xarlets,whose nature is defined by a third xarlet. A simple example is assigning a privilege to a role. The corresponding object can be gotten with the following call:

```
        $object = xarModAPIFunc('system','object','getassignment',
                      array('from'    => $privilege,
                            'to'      => $role
                            'operator => $privilegetype
                            )
                  );
```

or an equivalent shorthand form:

```
        $object = xarGetAssignment($privilege,$role,$privilegetype);
```

Assignment definitions can be freely added by module developers for their own purposes. Such "assignment objects" can in general be created, modified or removed from the system. However, Xaraya has several pre-defined assignment definitions, as listed below. Other assignments are specific to modules, such as categories and comments.

1. Property: A property is an assignment of one xarlet to another that can be best described as "ownership". Through the assignment one xarlet "owns" another. A xarlet can have zero, one or many properties, and similarly a property can "be had" by zero, one or many xarlets.

   Example: a privilege assigned to a role is a property of the role.

2. [MrB: Where is the bordercase? a privilege assigned to a role could also be a construction of a new object. When is something a property and when is it a new object? Something like singular or atomicity comes to mind, not sure where to draw the line though. I tend to favor towards new object construction and staying within that untill the atomic data in the object itself.]
3. By convention every xarlet automatically has a "Self" property assigned to it. Self is an assignent of identity, and simply refers to the xarlet itself
4. Member: A member is an assignment of one xarlet to another that can be described as a "subordinate". Through the assignment one xarlet "is a child" or "is a member" of another. A xarlet can have zero, one or many children/members, and similarly a child can "be assigned" by zero, one or many xarlet, who are referred to as the child's parents.

   Example: a category "Benelux" can be defined as a member of a category "EU Countries" and itself can contain members "Belgium", "Luxemburg" and"Netherlands".

   Example: "Property" and "Member" are members of "Object".

5. Type: A type assignment is a special assignment of a xarlet to itself. The assignment defines the type of xarlet a xarlet is.

   Example: the top level role xarlet (Everybody) is assigned the type "role".

6. Hook: A hook assignment between two xarlet (module components) corresponds to creating a Xaraya hook beetween two modules.

Example: Hooking comments into the articles module creates a hook assignment between those two modules.

[MrB: This is too vague i think. The concept of hooks can be reformulated if we go the 'object way' by having a dynamic object construction articles, comments -> responses Done by discovering that articles and comments have a compatible interface for construction and exposing this in the interface. When flagged the 'reponses' object is brought to life, whether that is done by a hook internally or not isn't very relevant i think.]

6.2 Inheritance

Assignments can be simply passive entries in the relationships registry that document the relationship between two xarlet and its characteristics. But, going one step further, relationships can also be made to actively affect the xarlet themselves.

Inheritance is one such way in which relationships affect xarlets. The most general definition of inheritance in the formalism described above is this:

"Inheritance is the way in which one xarlet in a relationship passes on certain of its assignments (properties? or any assignments?) to another."

In Xaraya, inheritance passes on certain assignments of the from-xarlet to the to-xarlet, as illustrated below. Another way of looking at it is that inheritance allows us to create temporary assignments at run-time.

The framework will pass on any assignment types that have been defined as properties of the assignment in question. So for example, assigning Type and Privilege as properties to the Member assignment will let the Roles hierarchy pass on types and privileges, as described below.

6.3 Examples of Inheritance

6.3.1 Categories

This is the simplest type of inheritance. each for-xarlet in a category relationship passes the Self property to the to-xarlet.

6.3.2 Roles

Roles inheritance passes on two types of assignements. Each for-xarlet in a member relationship passes on the Type and Privilege properties to the to-xarlet.

This is a way of saying that if the for-xarlet is of a type Role (e.g. the Users group), the to-xarlet (a member of the Users group) will also have the same type, and any privilege assigned to the group will also be assigned to its members.

6.3.3 Comments

Each for-xarlet in a comment relationship passes on the Type property inherited from the article it refers to.

6.4 Hierarchy Overview

The following diagram shows the topmost portion of the object hierarchy (core modules). *Only member assignments are shown.* [rndom: this is still largely in flux.]

```
      xarHierarchy
      |
      |--Object
      |   |
      |   |--ContentObject
```

```
        |  |
        |  |--CodeObject
        |
        |--Assignment
           |
           |--Property
           |
           |--Member
           |
           |--Type
```

7. Variable Sets

A variable set is a set of discrete variables of a component in a common name space. Grouping variables in variable sets lets us manipulate them in a consistent fashion for all components.

Example: The set of all variables exposed by a xarlet for data exchange is a variable set. This variable set can be called by the components getContent() method.

Manipulating variable sets, rather than single variables, is conistent with and an extension of the PN/Xaraya practice of passing variables to functions as an array $args.

A variable in a variable set has the form :identifier:_:varname:

The identifier is a prefix that identifies the name space. The varname can be freely chosen. The name space identifier currently in usee are:

- *xardata*: identifies the variable set used in data exchange with other components. These are the public fields of the component that can be gotten with the getContent() method. The definition of this method is:

```
    function getContent()
    {
        return $this->getVarSet('xardata');
    }
```

- *register*: identifies the variable set of the component's basic data. Ingeneral this will be distinct from the previous varset, ass not all these variables are public. These values can be gotten with the toArray() method.

```
    function toArray()
    {
        return $this->getVarSet('register');
    }
```

- *save*: identifies the variable set of the component's data that is stored in the database

The base object class incorporates a number of methods for manipulating varsets. [random: these will presumably be protected methods in PHP5.] They are:

- *getVarSet($str)*: gets an array of variables where $str denotes the name space ideentifier. The array elements are of the form :varname: => :varvalue:.
- *getVarNames($str)*: gets an array of variable names where $str denotes the name space identifier. The array elements are of the form :varname:.
- *setVarSet($str, $array)*: sets the values of a varset where $str denotes the name space identifier. The elements of $array are of the form :varname: => :varvalue:. If the varset contains variables that are not contained in $array, those variables are untouched.

The code sample below shows how one of these methods works.

```
    function getVarSet($str) {
        $vars = get_object_vars($this);
        $vararray = array();
        $len = strlen($str)+1;
        foreach ($vars as $key =>$value) {
            if(substr($key, 0, $len) == $str .'_') $vararray[substr($key,$len)] =
$value;
        }
        return $vararray;
    }
```

8. Examples Using the Component Hierarchy

8.1 Types

Type is a useful, all-round assignment. Assigning Types to xarlet and having them passed on makes it possible to "slice" sets of objects by type. Since objects can have more than one type, this means you can create views by type, with each view displaying only the data of the xarlet pertinent to that type.

8.2 Categories and Types

By assigning the Type property to the Category assignment, categories will inherit not just Self, but also types. This is a simple way of defining so called "base" categories in the categories hierarchy.

8.3 Hooks and Masks

By assigning the Mask property to the Hook assignment, modules linked by hooks can pass on privilege masks. In this way it becomes possible to split up the current complex mask instances and distribute their parts to the modules that have knowledge about the relevant mask data.

8.4 Roles and Dynamic Data

DD properties in roles (or other modules) can be viewed as property assignments of DD "objects" to roles xarlet. By tying the assignment of such properies to the *type* of role, rather than to any xarlet in the roles hierarchy, it becomes possible to have different DD properties attached to e.g. users and groups.

8.5 Content as properties

In theory the content of a xarlet can also be considered a property of the xarlet, and can be passed on to other objects. This is akin to the OO concept of inheritance, but somewhat more flexible, since more than one parent is possible and since the type of inheritance (in this case what type of data is passed on, perhaps depending on the type) can be dynamically programmed.

[MrB: Given a certain object hierarchy the final object is a 'leaf' in the tree and the instances at runtime based on the request are passed to the assembly line. In its simplest form this would be an 'unfolding' type of functionality, which unfolds all instances needed by the request delivering the instance content-object to the assembly which takes that object and merges itwith the templates into out output representation.]

This concept can be used to create "distributed objects" between modules. An example is a customer xarlet that is referenced by both the ledger (accounting data) and paypal (transaction data) modules. The actual data of the customer resides in different modules, but the reference is the same for all of them.

9. Database Tables

The component hierarchy requires two database tables, one to hold component definitions and one for the assignments between xarlet These are:

9.1 Register Table

N.B.: This still needs some more thought and proof-of-concept. In particular either xa_oid or xar_sourceidentifier could be superfluous.

```
            CREATE TABLE xar_system_register (
              xar_id int(11) NOT NULL auto_increment,
              xar_oid int(11) NOT NULL default '0',
              xar_name varchar(100) NOT NULL default '',
              xar_sourceid int(11) NOT NULL default '0',
              xar_sourceaddress varchar(255) NOT NULL default '',
              xar_sourcecontainer varchar(100) NOT NULL default '',
              xar_sourceinstance varchar(100) NOT NULL default '',
              xar_sourceidentifier varchar(100) NOT NULL default '',
              PRIMARY KEY  (xar_id)
            ) TYPE=MyISAM;

          Table: xar_objects_register

            Field                      Description
            ----------------------     --------------
            xar_id                     unique index assigned by the system
            xar_oid                    object id for external reference
            xar_sourceid               designates source type of the object
            xar_sourceaddress          location of the object
            xar_sourcecontainer        specific container
            xar_sourceinstance         instance within the container
            xar_sourceidentifier       unique reference in the instance
```

Sourceid: values are (for instance) ids representing db connections, and 100 and 200 for internal or external code components.

Sourceaddress: For *Nuke-like code components: module reference. For db objects: table reference.

Sourcecontainer: For *Nuke-like code components: type reference. For db objects: record reference.

Sourceinstance: For *Nuke-like code components: function reference. For db objects: field reference.

Sourceidentifier: For db objects: record id reference.

9.2 Relationship Table

```
            CREATE TABLE xar_system_assignments (
              xar_id int(11) NOT NULL auto_increment,
              xar_from_id int(11) NOT NULL default '0',
              xar_to_id int(11) NOT NULL default '0',
              xar_operator_id int(11) NOT NULL default '0',
              PRIMARY KEY  (xar_id),
              KEY i_from_xar_id (xar_from_id),
              KEY i_xar_to_id (xar_to_id),
              KEY i_xar_operator_id (xar_operator_id)
            ) TYPE=MyISAM;

          Table: xar_objects_assignments

            Field                      Description
            ----------------------     --------------
            xar_id                     unique index assigned by the system
            xar_from_id                reference to the from object's oid
            xar_to_id                 reference to the to object's oid
            xar_operator_id           reference to the operator object's oid
            xar_sourceaddress         location of the object
```

From: a xarlet reference corresponding to the xarlet the assignment "goes from".

To: a xarlet reference corresponding to the xarlet the assignment "goes to".

Example: Assigning a privilege to a role

```
          From                    To                 Operator
          ---------------------   ----------------   ------------------
          privilege id            role id            privilege type id
```

The reference privilege type id is a special reference. It is the privilege oid reference given in the table entry:

```
          From                    To                 Operator
          ---------------------   ----------------   ------------------
          type id                 privilege id       type id
```

10. Revision history

Version 0.3.0, Jan 18, 2004: First preliminary draft of this document

Version 0.3.1, Feb 4, 2004: after review by Marcel van der Boom and Joanna Dalle Nogare

11 Reference title

Author's Address

Marc Lutolf
Xaraya Development Group
EMail: marcinmilan@xaraya.com
URI: http://www.xaraya.com

Intellectual Property Statement

Acknowledgement

Funding for the RFC Editor function is provided by the DDF