NOTE 243 – XML-based Documentation for AIPS++

Raymond Plante

16 April 2001

Abstract

This note examines some of the current problems and deficiencies of the AIPS++ documentation and considers how an XML-based approach might be used to address them. I describe an over-all framework based on two key features: the use of the DocBook markup for expository documents and the use of in-line documentation for glish-based tools and functions. This framework will continue to supply users with both HTML and high-quality printable versions of all documentation. I describe the tools needed to support this framework and outline a plan for migrating from our present system. An important goal of this framework is to make it easier for developers to create and maintain high quality documentation that will improve the user experience. It is expected that this note will serve as the basis of one or more formal change proposals aimed at adopting this framework.

Contents

1 Introduction: The Current State of AIPS++ Documentation

1.1 Current Principles and Framework

Note 215 describes the current framework of documentation in AIPS++. In general, there are three types of documentation in AIPS++:

- Expository documents (e.g. *Getting Started, Getting Results*, Notes, Memos), usually in LaTeX format;
- Glish tool documentation (i.e. the *User Reference Manual*);
- C++ Class documentation, using SGML markup in-lined into C++ header files and extracted with cxx2html.

The first two types are the most important to users. The *Getting* * documents are meant to instruct users in the general skills and approaches used to process data in AIPS++. It's expected that users will read large parts of it in whole. The *Reference Manual*, on the other hand, is meant primarily as a definitive guide to specific tools and functions; users consult this document to locate specific bits of information.

The current documentation framework is based (in part) on the following principles:

1. Users must have access to documentation on-line.

2. Users must be able to print out documentation in a high-quality form.

3. It must be possible to build the documentation anywhere AIPS++ is installed.

4. The above features are be enabled via three "cornerstone" formats: LaTeX, HTML, and PostScript.

LaTeX is the source format for the documents aimed at end users. In particular, developers use specialized macros to markup components of the documentation that makes up the *Reference Manual*.

1.2 Room for Improvement

This section lists some of the problems with the current system which this study hopes to address.

1.2.1 General

Documentation build system is fragile and difficult to maintain. Because the system depends on complex, external packages which must be installed seperately (LaTeX and LaTeX2HTML), setting up the system up to build documentation automatically is not easy. UIUC does not build documentation locally because of the problems we've had trying to configure the system. Instead, we import the built documentation from Socorro.

LaTeX2HTML is certainly the weakest link in the documentation system. Since importing docs from Socorro, we have found that the HTML version of the *Reference Manual* regularly does not format argument lists properly. Problems formatting equations and images are common but often go undetected by the make system.

LaTeX is sufficiently complex as to make creating documentation non-trivial. Many of the formatting problems that currently exist can be traced to mark-up errors on the part of the author. Common errors like the "missing brace" can have drastic effects on the formatting. It's also important to check the translation to HTML; a document that processes fine with LaTeX can be mangled in a variety of ways when passed through LaTeX2HTML. The AIPS++ system does provide developers with tools checking their documentation (`make` *file*`.dvi` and, more recently `make` *file*`.html`). Nevertheless, formatting errors are still quite common. This may be largely due to developer laziness; however, it may reflect in part on the difficulty in getting the tools to work properly.

This author's own difficulties are primarily due to the fact that the UIUC installation is not properly configured to build the documentation. However, I was also unable to process my documentation on `tarzan` at the AOC. If this is a common experience among developers, then the supporting tools are not sufficiently helpful in checking our documents.

It is difficult to adjust the presentation of documentation. This is due to the inherent complexities of LaTeX and LaTeX2HTML. Significant skill and experience is needed to fiddle with LaTeX style files to adjust the document layout. It is arguably even more difficult to adapt LaTeX2HTML to variations in presentation. This barrier of complexity not only discourages experimenting with presentation but also the resolution of formatting problems.

As an example of experimenting with presentation, some developers have expressed a preference for layouts that maximize the amount of content that can be viewed at once. This might call for smaller margins in the hardcopy version. It might also call for a re-arrangement of information presented in a *Reference Manual* page. Small changes as these could have big effects on the documentation's user–friendliness; thus, it would helpful if it were easier to modify the layout and try new presentations.

1.2.2 Reference Manual

The overall organization of the *Reference Manual*, with its hierarchy of packages, modules, and tools, is good. A significant portion (if not the majority) of the content is of good quality. Nevertheless, it can still be difficult to find specific information on how to use specific tools and functions. This is not always due to a paucity of words, but often due to structure.

Process for creating module documentation is laborious After creating a tool implementation, the developer must create two separate files (*tool*`.help` and *tool*`_meta.g`), each to some extent containing redundant information both between them and the Glish source code. As separate files and with different formats, they are not conducive to being developed in parallel with the software. Thus, they are usually done as a separate chore, after the implementation. This can tend to discourage completeness, particularly in describing allowed values to functions or caveats to use special circumstances; this is because, since the developer is somewhat more removed from the inner workings of the code, including this information is more difficult.

Structure of function description makes it hard to find information related to specific parameters. It is common for a user to want to discover the meaning and use of a specific parameter. The obvious place to look is the table of function parameters; however, this contains limited information (see next item). For additional details, the user must sift through the general function description. (New users may not know to expect additional information to be hidden there.) It would be better if all the information related to a parameter could be found in one, easy-to-find place. Repetition of this information in the general description is fine and often useful when describing the role of the most important parameters in the overall use of the function.

The presentation of the parameter description discourages complete descriptions. Function parameters are listed and described in a structured table. In the HTML version of the manual, this table is quite narrow (perhaps one half the width of a typical browser window), so very little information can be contained in this table before it becomes difficult to read. As a result, developers must go to the effort of including more complete information in the general description of the function, which sometimes does not happen. Thus, parameter descriptions are limited to a single clause or sentence, and default and allowed values sometimes go undescribed. And because the content of the general description section is unstructured, the fact that this information is missing goes unnoticed (except by those specifically looking for it).

1.3 The Case for a New Approach

It is worth noting that a LaTeX–based system can probably be made to work and our overall system improved for users with additional effort, perhaps even comparable to the effort of converting to a new framework.[1] However, the inherent complexities of LaTeX and LaTeX2HTML discourages us from improving and enriching the system. In contrast, an XML-based framework offers new possibilities for developing documentation features (e.g. two-way

[1] The author notes his own experience and the experience of others in our community supporting LaTeX–based proposal submission systems and conference proceedings created from LaTeX documents contributed by participants.

links, automated indexing, context-based help, ASCII displays for Glish, integration with Tool Manager metadata).

More important than new capabilities, a new approach to documentation offers the opportunity to make authoring and delivering documentation easier. This note posits that if this process can be made less laborious, authors will create higher quality documentation–that is, more comprehensive and with the necessary information where users can easily find it. Indeed, substantial improvement in documentation quality for the end user must be a requirement for any substantial change in the documentation system.

2 An XML-based Framework

This proposed framework addresses the creation, maintenance, and delivery of the two types of documentation aimed at end users: expository documents and the *Reference Manual*. The author feels that the current framework for documenting C++ code in-line is pretty good. It's backed by a formal quality assurance process that is reasonably effective. Furthermore, the large amount of C++ code documentation already in place makes substantial changes to it hard to justify. Documentation for the end user is more important. In particular, this author feels that changes to how the *Reference Manual* is constructed will result in the greatest benefits for users.

This new framework is based on the following principles:

1. Users must have access to documentation on-line.

2. Users must be able to print out documentation in a high-quality form.

3. It is not necessary that documentation built locally from source files as part of a standard AIPS++; installing pre-built documentation should be sufficient.

4. Authors must be able to *easily* build and validate their documents for the different versions supported by the system.

5. HTML is the best format for the on-line version as it offers the greatest flexibility to the user for presentation (e.g. font style and size, window size, foreground and background colors).

6. It is easier to enable automated machine-processing of metadata and documentation using XML than LaTeX.

The proposed framework is characterized by two key features:

- DocBook, a system for structuring documents in SGML or XML, is used to create and maintain expository documents.
- The *Reference Manual* as well as the Tool Manager metadata files (i.e. *tool*`_meta.g`) are generated from in-lined comments using a markup similar to JavaDoc but which is converted to XML upon extraction.

XML will serve then as the base format for conversion into other formats. HTML is used for browser-based access. Plain text can be used for access within an interactive Glish session. Hardcopy can be provided in either PostScript or PDF format.

2.1 Use of DocBook

DocBook is perhaps best known as the most widely used SGML markup language available as open source and freely available. It is maintained by the DocBook Technical Committee of the Organization for the Advancement of Structured Information Standards (OASIS)[2]. DocBook began its life in 1991 as an SGML DTD; however, with Version 4.1, an XML version is now available and supported by the most common DocBook-compliant tools. The XML version includes MathML as an extension for marking up mathematical equations. The DocBook DTD is modular in design and has an explicit framework for extending it to add new elements or further control attribute values.

2.1.1 Advantages of using DocBook

DocBook is well supported

- **O'Reilly publishes a book on DocBook**[3].
- **LyX, a free WYSIWYG editor,**[4] **can create and export DocBook-format documents.** Successful use of this application could minimize the amount authors need to learn about DocBook as a markup language. It is possible to add new Layout classes to LyX that make

[2] http://www.oasis-open.org/

[3] Walsh, N & Muellner, L. 1999. *DocBook: The Definitive Guide*, (O'Reilly: Cambridge)

[4] This is not strictly true according to the LyX web site, http://www.lyx.org/, which calls it WYSIWY*M*, as in "what you see is what you mean." This is more appropriate for our purposes, since presentation is separated from content; nevertheless, the on-screen rendering gives the author a visual indicator about how the document is organized.

it easier to create documents that must conform to particular styles. (In fact, in addition to its DocBook layout classes and templates, LyX also ships with support for other styles, including one for the AAS journal manuscripts.) An easy–to–use word processor of this sort may make documentation authoring more attractive to non–developers.

- **Several packages are freely available for converting DocBook to other formats**. Jade provides extensible DSSSL style sheets for converting to LaTeXand RTF. (Jade can also be used for validation.) nwalsh.com supplies free, customizable XSL stylesheets for converting to HTML. Novell has produced a UNIX script for converting DocBook to HTML.
- A few commercial products such as **Adobe's FrameMaker+SGML** and **WordPerfect 9.0 for MS Windows** support DocBook "out of the box."
- The wide variety of generic SGML/XML tools can be used with DocBook documents.

DocBook is widely used.

- The **Linux Documentation Project**[5] uses DocBook for authoring the various HOWTOs, FAQs, and Project Guides it maintains. In fact, there are a few aspects of the LDP model that we can borrow:
  - availability of a HOWTO-HOWTO document[6] for new authors,
  - style guides,
  - standard style sheets,
  - recommended authoring tools.
- The **K Desktop Environment (KDE)**[7] uses DocBook for its documentation. They provide a very nice "crash course" document entitled Writing Documenation using DocBook[8].

DocBook is a cross-platform format.

[5]http://www.linuxdoc.org/
[6]http://www.linux.org/docs/ldp/howto/HOWTO-HOWTO/
[7]http://www.kde.org/
[8]http://www.caldera.de/ eric/crash-course/HTML/

DocBook documents can be edited with any editor.

As with all SGML/XML formats, content and presentation are seperated. This has two important ramifications. First, the presentation can be evolved without having to alter the source documents. Second, it is not necessary that all authors have the specific skills for manipulating presentation.

XML offers greater flexibility for controling presentation. It's worth noting that in principle, LaTeX separates content and presentation as well; however, in can be argued that this does not happen in effect. The expertise necessary to alter a style file and subsequently support it with LaTeX2HTML discourages experimentation with presentation. In this author's opinion, *manipulating XML style sheets is much easier than LaTeX style files.*

2.1.2 Potential Pitfalls

Very few actual experiments using DocBook for AIPS++ have been done to date, and many of the details concerning how we might use it have not been worked out, yet. Here are some ways that DocBook may not live up to its promise:

- The effort for adapting style sheets for use with AIPS++ may be more extensive than expected.
- Extending DocBook for handling AIPS++–specific structures (analogue to `aips2defs.tex`) may be more difficult than expected, or doing so makes generic DocBook tools less helpful. (It appears at this time, however, that we can make effective use of DocBook with little or no extensions to the DocBook DTD.)
- It may be necessary but difficult to adapt LyX for creating AIPS++ documentation. Without LyX or similar product, authors will have to learn the details of the DocBook DTD which defines about 100 different elements.
- It is too difficult to convert existing documents into DocBook when necessary. This is expected to apply only to documents like *Getting Results* that will continue to evolve and expand.

2.2 In-line Documentation for Glish

2.2.1 The Case for In-line Documenation

Given the proper supporting tools, in-line documentation of software can provide tremendous advantages:

- Much of the information that needs to appear in the documentation can be extracted directly from the code itself. This includes:

  - tool and function names and indices
  - function signatures
  - parameter names
  - default values

  As a result, quite a bit of information can be extracted with little or no markup included. Furthermore, this reduces the amount of redundant information the developer has to type in.

- In-line documentation aids fellow programmers in understanding what the code does when the code is examined directly.

- It enables a convenient regimen for developing documentation simultaneously with the development of the code itself. Otherwise, documenting the code can seems more like a separate chore, usually started after the development work is done.

- It is easier to keep documentation up-to-date with code changes, since the relevent information can be kept close to the actual implementation.

- In-line documentation favors a format that is well suited for a reference manual. Since in-line documentation appears close to the thing it is describing (e.g. tool or function definition), the information tends to aggregate to predictable locations in the documentation. This is important to users when they are looking for specific information.

- Well-structured in-line documentation can encourage completeness. Tags that mark up such things as default and allowed values make it more convenient to include full descriptions of these items. Furthermore, because some information can be extracted automatically, such as default values, it can be made more obvious to the developer when these descriptions are not present.

2.2.2 Features of the In-line Documentation Framework for Glish

For the purposes of discussion, this note will refer to the framework for in-lining documentation into Glish code as GlishDoc. An implementation of this framework as described here may result in a software tool of the same name.

This section gives an incomplete description of how the proposed framework for in-lining documentation will work. An extended example of documented code is given in Appendix A.

- Developers markup Glish source code files using specialized markup tags (see below) within Glish comments. It should be possible to filter out these comments as part of the AIPS++build system if they pose a performance problem.

- To process the documentation, GlishDoc comments are extracted from the Glish source code and converted into XML using a custom extraction tool (implemented in Glish).

- The XML DTD used for GlishDoc documentation may either be an extended version of DocBook or a custom DTD for Glish documentation. If the former is chosen, it will be necessary to define a number of tags to describe Glish tool interface components and characteristics. In the latter case, we may want to enable a further conversion to the DocBook DTD in order better to integrate the reference manual with the rest of the documentation. A hybrid solution (e.g. akin to the way DocBook supports MathML) may also be possible.

- The GlishDoc extraction tool constructs the basic XML document primarily from the Glish code itself. Markup tags added by the developer further guide and augment the output document's structure and content.

- A GlishDoc comment block has a special format that identifies it as such. The GlishDoc comment block starts with a line that contains only a comment starting with the characters `#@`. Subsequent comment lines are included in the block. The block ends prior to the first line that contains either Glish code or the start of another GlishDoc comment block. For example:

```
#@
# Writes a summary of the properties of the imager to the
# default logger.
public.summary:=function() {
```

- Markup tags use the JavaDoc style syntax of the form `@`*tag_name*. The marked text appears after the tag and continues either until the next tag or the end of the block; interpretation depends on the tag. This style of tag is quicker to type and less prone to errors than XML or LaTeX–style tags because it requires a minimum of special characters and does not require a closing element or brace. For example,

```
#@
# Summarize the measurement set.
#
# This function will print a summary of the measurement set to the
# system logger. The verbose argument provides some control on how
# much information is displayed.
#
# @outparam header  Selected header information returned as a Glish
#                   record.
# @inparam  verbose If true, produce a verbose summary.
public.summary:=function(ref header=[=], verbose=F) {
```

- XML tags can also be used for special types of markup within descriptions. This will be useful for including links to other documents, inserting tables or figures, or including mathematical formulas. The GlishDoc extraction tool would pass this markup unchanged.

- Many markup tags can be assumed based on the context of the GlishDoc comment block. For instance, the first example above could also be written as:

```
#@tfunction
# Writes a summary of the properties of the imager to the
# default logger.
public.summary:=function() {
```

However, the fact that the text describes a tool function can be assumed because it appears within a tool definition and just before a function definition.

- Among the available GlishDoc tags will be an include directive (e.g. `@include` *filename*). This will allow one to easily include longer descriptive sections that may be more easily authored externally using DocBook markup and LyX.

- All the functionality provided by the current LaTeX markup (defined in `aips2help.sty`) would be duplicated in the GlishDoc framework.

- Developers can also include information needed by the Tool Manager to build GUIs automatically. Initially, this information could be transformed from XML into Glish code of the form currently supported via the *tool*`_meta.g` files. In the future, this information could be accessed directly from Glish via a Glish XML parser client.

- Prior to checking in documented code, developers can run it through a validation tool that will alert them of syntax errors and warn them about missing markup that is required by the style guide.

3 Supporting Tools and Documents

This section lists the tools (in the more generic sense) and documents that will be needed to support the proposed framework.

- GlishDoc extraction tool. It is expected that this will be the only tool that will have to be developed "from scratch."

- Style sheets for validating XML documents and converting them to other formats. It is expected that these would be adapted from existing style sheets that come with Jade or some similar existing package.

- Style sheet application tools. These would be trivial scripts that call an existing program for applying our customized style sheets. In particular, we would have:

  - a GlishDoc validation tool. This would be used by Glish developers to validate the syntax and completeness of their in-lined documentation.
  - a GlishDoc format conversion tool. This would be used by Glish developers to preview their Glish tool documentation in various formats. It would also be used by the AIPS++ make system to convert the documentation into supported formats.

  - DocBook format conversion tool. This would be used by both authors and the AIPS++ make system to convert the documents into the various supported formats. Preferably, this tool would be the same tool used for converting GlishDoc documentation, differing only in the style sheets it uses.

- Lyx, a WYSIWYG editor for creating DocBook documents, adapted for AIPS++ via a pluggable layout class.
- A Note describing how to use GlishDoc tags and tools to document Glish code.
- A Note describing how to create expository documents (e.g. Notes, *Gettting Results* chapters, etc.) using DocBook and LyX.
- A Style Guide for GlishDoc documentation.

4 Implementation Plan

This section describes a phased approach for adopting and implementing the proposed change to the documentation system. Because XML is easily converted into other formats, this framework is well-suited for gradual adoption. This is helpful as there are many details that have to yet to be worked out. As each phase is completed we will have the opportunity to re-evaluate the framework in light of the implementation and tools available to date. It is expected that the steps listed below would be turned into development targets.

Phase I: Initial GlishDoc implementation

1. Adopt and/or create a DTD for encoding tool descriptions that will be extracted from the source code. Define necessary GlishDoc-style tags for documenting tool source code.
2. Implement an initial version of the GlishDoc document extractor in Glish. (It is expected that a Glish implementation will be easier for a typical AIPS++ developer to maintain than one based on, say, Perl. It also allows the possibility for further integration of the documentation extraction into the AIPS++ system in the future.)
3. Develop (or adapt) style sheets to convert XML tool descriptions into currently supported `.help` and `_meta.g` files. Install wrapper script

that applies style sheet transformations to XML descriptions. (Augment the EMACS Glish mode to add keyboard macros for inserting commonly used GlishDoc markup.)

4. Create an AIPS++ note describing how to use GlishDoc tags to mark up in-lined documentation and how to use associated tools to extract the documentation and convert it to currently support forms.

5. Encourage developers to try in-lining documenation. Evaluate current state of framework for ease-of-use, robustness, adaptability to the AIPS++ system, and effects on the quality of documentation and software process. Adjust plan for later phases accordingly.

Phase II: Initial Use of DocBook

1. Decide on the specific relationship between DocBook and the XML used by GlishDoc as described in §2.2.2, p. 11 (if not already determined in Phase I).

2. Define any extensions to the DocBook DTD needed for supporting AIPS++ documents. Ease of use by authors should be an important factor in defining new tags.

3. Adapt the LyX DocBook layout class as necessary and helpful for creating AIPS++ expository documents.

4. Create (or adapt) style sheets converting DocBook documents into LaTeX in the traditional AIPS++ style.

5. Create a DocBook-formatted template for creating AIPS++ notes.

6. Create an AIPS++ note using DocBook that describes how to use DocBook, LyX, and associated tools for authoring expository documents for AIPS++.

7. Encourage developers to try DocBook and LyX for creating documenation. Evaluate current state of tools for ease-of-use, robustness, adaptability to the AIPS++ system, and effects on the quality of documentation. Adjust plan for last phases accordingly.

Phase III: Final Integration Assuming the first two phases are successful and the framework is determined to be useful, this phase completes the adoption of the new framework.

1. Adapt existing validation tools to check the syntax and completeness of GlishDoc documentation.

2. Create (or adapt) style sheets for used to convert XML-based documentation into other needed formats, including HTML, PostScript/PDF, and (if necessary) plain text. (Unless a pre-existing tool for direct conversion to PostScript or PDF is available, it is expected that XML will be converted first to LaTeX for subsequent processing into the hardcopy format.)

3. Fully integrate tools that can be used by authors for validating and formatting DocBook and GlishDoc documents into the AIPS++system.

4. Adapt the AIPS++ make system as appropriate to build new XML-based documentation.

5. Determine how much of the documentation should continue to be built locally to the AIPS++ and how much should be preformatted and delivered to an AIPS++ site on demand.

6. Convert limited number of existing AIPS++ documents to DocBook format. This conversion should be limited to those documents that will be edited further into the future.

7. Convert all existing LaTeX-formatted Reference Manual documents to XML (DocBook) documents.

8. Create a style guide for creating complete, high-quality, inlined Glish tool documentation.

5 Conclusions

The most important aim of the framework described here is to make it easier for developers to create and deliver documentation that is high-quality not only in presentation but in content. Employing the widely-used DocBook XML markup has the potential for providing greater flexibility and reliability in formatting and presenting our expository documents than LaTeX. The use of the LyX editor can greatly reduce the overhead of learning to use DocBook markup and will hopefully encourage contributions from people outside the core development team.

The proposed system for in-lined documentation of Glish code will make it much easier to create complete, well-organized descriptions of tools and functions. Specifically, the simple markup syntax and the ability to extract some information automatically from the glish code itself reduces the amount the developer has to type. Furthermore, in-lined documentation encourages the practice of developing documentation simultaneously with implementation: simple descriptions can be entered when the tool is first prototyped; details can be added as the implementation crystalizes; once the interface is set, the function parameters can be described and the necessary metadata for the Tool Manager, provided; examples can be added during testing; and finally, as the tool is tweaked and debugged, the changes can be reflected right away into the documentation. The structure can encourage completeness, and the use of a validation tool can help ensure that all necessary infomation has been encoded.

Despite our best efforts toward software solutions, there is no cure for the lazy programmer in all of us. Thus, the key to making any documentation framework successful is in the conscientious efforts of developers. A style guide is important for advising us on how to make the best of the system. At the same time, the documentation system can make the best of our efforts by providing tools that make it easier pass our expertise onto users.

A Example of In-line Documentation Markup

The following example is provided to give the flavor of in-line documentation proposed through this note. It does not contain all the tags expected to be supported nor all the documentation that should be included as a matter of style. However, it should illustrate the basic structure of the in-line documentation and how certain information can be assume by the comments context.

```
pragma include once;

include 'unset.g';
include 'xmlstorage.g';

#@tool
# Buffer for building an element using "easy" storage.
#
```

```
# This tool is used for loading XML data into a metadata storage
# object which can be returned with a call to
# <tmlink>getelementrep</tmlink>.  The storage model supported by this
# buffer is optimized for access to XML data from the application/user
# level.  XML Elements are stored as Glish records, and their attributes,
# as Glish attributes.
#
# @constructor
# create an "easy" element buffer.
# @inparam element  previously built element to edit
#                   @type record
#                   @default unset  create a new element
xmleasyelementbuf := function(ref element=unset) {

    #@toolrec public
    public := _xmlstorage();
    private := [=];

    #@
    # set the name of the element
    # @inparam elname  the element name.  This must begin with a letter
    #                  and should should not end in a number.
    #                  @type string
    #                  @default none
    # @inparam clear   if true, clear all previously added data
    #
    public.setname := function(elname, clear=T) {
        # ...
    }

    #@
    # get the name of the top element in this buffer
    public.getname := function() {
        # ...
    }

    #@
    # set an attribute value
    # @inparam attname   the attribute name.  The name must begin with
    #                    a letter.
```

```
#                   @type string
# @inparam value    the value of the attribute
# @inparam default  if true, this value should be considered a default.
#                   This can used to affect conversion to XML--e.g. one
#                   may not want to have attributes with default values
#                   printed out.
public.setattribute := function(attname, value='', default=F) {
    # ...
}

#@
# add some text to the value of the current element
# @param  text     the text to add
#                  @type any
#                  @default an empty string
# @param  which    replace the which-th text value, if it exists;
#                  otherwise, just add it.
#                  @default 0, which means append a new processing
#                           instruction
# @fail if input text is not a string
public.addText := function(text='', which=0) {
    # ...
}

#@
# return a the buffer of a child element for editing
# @param elname   the child element name.  If the element has not
#                 yet been added, it will be.
# @param  which    replace the which-th element, if it exists;
#                  otherwise, just add it.
# @return tool of type xmleasystorage
public.getChildElementBuf := function(elname, which=0) {
    # ...
}

#@
# return a copy of the constructed element record
# @param options  a record containing options for formatting the
#                 element representation.  Currently supported
#       options include:
```

```
    # <pre>
    #        cleanup  if true, element content will be adjusted to aid in
    #                 user access.  All non-element children of the same type
    #                 (text, comments, or processing instructions) adjacent
    #                 to each other will be gathered into a single array.
    #                 Furthermore, if an element contains only text child
    #                 nodes, the element's children will be "pulled up"--that
    #                 id, replaced with a single array containing all the text
    #                 node values.  The default is T.
    #        nopullup a list of elements whose child text nodes should not be
    #                 "pulled up" as described above.  This might contain a
    #                 list of elements that can but not always contain mixed
    #                 (elements and text) content.
    # </pre>
    # @return the storage object in the form of a Glish record
    public.getElementRep := function(val options) {
        # ...
    }


    #@
    # shut down this tool
    public.done := function() {
        # ...
    }

    # ...

    return ref public;
}
```

B Draft Change Proposals