

# XML Content Management: Challenges and Solutions

Dikran Meliksetian<sup>1</sup>, Louis Weitzman<sup>1</sup>, Sara Elo Dean<sup>1</sup>, Jeff Milton<sup>2</sup>  
Nianjun Zhou<sup>1</sup>, Peter Davis<sup>3</sup>, Jessica Wu<sup>1</sup>,  
Advanced Internet Technology, IBM  
Southbury, Connecticut

<sup>1</sup>(meliksd1, louisw, saraelo, jzhou, jessicaw)@us.ibm.com,  
<sup>2</sup>jmilton@razorfish.com, <sup>3</sup>pdavis@alum.mit.edu

## ABSTRACT

E-commerce presupposes dynamic and personalized presentation of information. In order to support an effective e-commerce environment, content on the Internet must be flexible and reusable. These characteristics can be supported if content on the Internet is highly modularized and richly tagged. We propose to use XML as the framework upon which to build a web content management system that supports these features. However, the design of such a system, where content is highly modularized and reusable, creates new challenges that have to be addressed. These challenges include finding relevant information fragments on demand, keeping track of the dependencies between fragments, transforming combinations of those fragments into viewable pages available to multiple device types, and designing a content creation tool that does not overwhelm the contributor with the details and the complexities of the underlying system. In this paper we describe these challenges and the solutions that we developed in the design of a web content management system based on XML, code-named Franklin.

## Keywords

Content publishing, cross-media publishing, XML content, XSL stylesheets, reusable fragments, DB2, WebDAV

## INTRODUCTION

Content on the Next Generation Internet needs to be highly adaptive. New interfaces and devices are emerging, the diversity of users is increasing, machines are acting more and more on users' behalf, and net activities are possible for a wide range of business, leisure, education, and research activities. To achieve maximum flexibility and reuse, content needs to be broken down into richly tagged fragments that can be combined and rendered appropriately for the user, task, and context. These features are also essential for an efficient e-business environment. XML is an emerging standard that can be leveraged to design a

content management system that supports these features. A content management system based on XML along with XSL enforces separation of content and presentation, thus allowing flexible rendering of the content to multiple device types and customized presentation to different audiences. Similarly, such a content management system will allow maximal reuse of information and data through the composition of XML fragments.

The design of a content management system based on those premises introduces new challenges. First, there exists the need to maintain information about the functional and semantic role of each fragment. This information describes what the content is about, who the target audience is, and its relationship to a taxonomy or other fragments. The same mechanism should support efficient searches of particular fragments. Second, an efficient method should be devised to track the effects of changes in a particular fragment and propagate those changes throughout the information space. Third, the user interface should be designed in a manner that it shields the content contributor from knowing the underlying syntax and complexities of the XML documents.

In this paper we describe the challenges that we identified and the corresponding solutions that we devised in the design and implementation of a prototype of a content management system, called Franklin. The prototype provides an end-to-end process from content creation and metatagging to quality assurance and publishing.

This paper is organized as follows. In the next section, we present the overall architecture of the system, following it with a section on the data model used in the system. We then present successively the issues associated with the design of the metadata store, the fragment dependency store, and the automated user interface creation features. We then present our conclusion.

## ARCHITECTURE

The system consists of five main components:

1. A database that stores the meta-information about the assets: the metastore.
2. A java application that maintains the dependency information between assets: the fragment dependency store.
3. The file system where the assets are stored.
4. A client application that allows content creators to interact with the system: the content editor.
5. A web application running within the scope of a web application server that coordinates the activities between the other components: the server.

The metastore maintains information about the functional and semantic role of each item of content. The metastore also supports fast searches of content and maintains state information. The functionality of the metastore is described in more detail in the Metadata Store section.

The fragment dependency store builds upon the Daedalus (a.k.a. Trigger Monitor) technology from IBM Watson Research [3,4,5,6]. Daedalus is designed to manage high numbers of rapidly changing content fragments. By maintaining an Object Dependency Graph, and by detecting changes to content, it manages pages on a web server or cached in a network router in a timely manner. Daedalus allows the loading of specialized handlers to perform tasks specific to a particular application, we have designed and implemented handlers that are described in greater detail in the Fragment Dependency Store section.

The content documents are stored in the file system as XML documents. Images, style sheets and other assets are also stored in the file system.

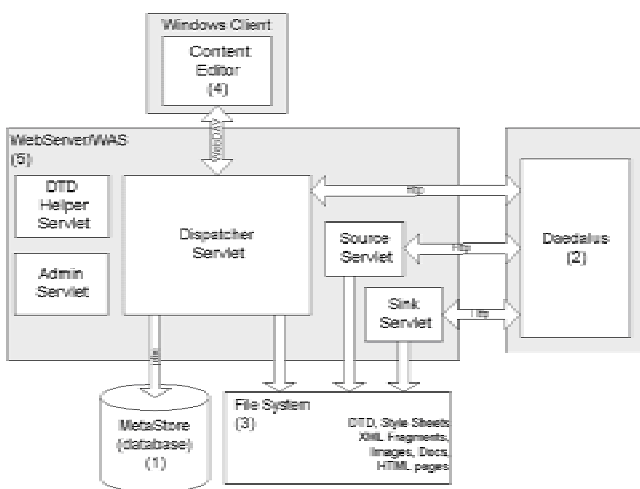


Figure 1. System Architecture.

The client application is a standalone java application that allows the content creator to interact with the system. Through the client application, the user can create new documents, search for existing documents, check-out documents, and check them back in after modification. In addition, the client application allows for previewing of the web pages that will be created from the XML documents. The communication protocols between the different components are based on industry standards: WebDAV, DASL, and HTTP. XML is used not only for content, but also for system configuration documents at startup and as the language for information exchange between the different parts of the system.

The server consists of four servlets and three subcomponents. The main servlet is the dispatcher that coordinates the activities of all subsystems and interfaces with the client application. The source and sink servlets allow Daedalus to retrieve fragments from the file system and write assembled pages to it. The admin servlet provides for administration and monitoring functionality. The three subsystems interface with the metastore, the fragment dependency store and the file system respectively.

## DATA MODEL

Franklin manages two types of content objects, *fragments* and *servables*.

A fragment is a content object that can be reused on several pages:

- A *simple fragment* is a self contained XML file containing text data and metadata – for example, a product specification
- A *compound fragment* is an XML file that contains metadata and points to an accompanying file such as a video or image file, an XSL style sheet, or a hand-crafted HTML page

A servable is an XML file that contains the text and metadata for one or more final published pages and imports reusable content from one or more fragments. It also points to one or more style sheet fragments.

Each fragment type and servable type has an associated DTD that describes the structure of the XML document. The DTD specifies both metadata elements and content elements. The DTD must abide to some constraints imposed by Franklin. The root element must have a child node that is common to all documents called SYSTEM with the children:

FRAGMENTID, CREATOR, MODIFIER, CREATIONTIME, LASTMODIFIEDTIME, PAGETYPE and CONTENTSIZE.

These elements are shared across all documents and comprise the common metadata used in searches. These elements are not displayed in the interface, since their value can be inferred from the context. Additional metadata, such as KEYWORD and CATEGORY, are provided by common DTD

elements to allow functional and semantic categorization of the content.

The metadata elements are used both at author-time and run-time. At author-time the metadata elements are used for categorization of fragments and for efficient searches of subfragments. At run-time, the same metadata elements can be used to perform personalization in a dynamic web site.

A fragment can include other fragments as subfragments. We call a fragment that includes other fragments a *composite fragment*. This leverages the reusability of content. If so, the entity reference that defines all subfragment types must be included in the DTD. Currently, the declaration of a subfragment must contain the SUBFRAGMENTTYPE attribute set to the appropriate document type, as illustrated in the following example:

```
<!ENTITY % SUBFRAGMENTTYPES SYSTEM
"http://server/franklin/dtd/subfragmenttypes
.txt">
<!ELEMENT SUBFRAGMENT (#PCDATA)>
<!ATTLIST SUBFRAGMENT SUBFRAGMENTTYPE
(%SUBFRAGMENTTYPES;) "IMAGEFRAGMENT" #FIXED>
```

This piece of a DTD specifies that a particular type of subfragment, IMAGEFRAGMENT, is needed as content for the element SUBFRAGMENT. The subfragment syntax will be replaced by the XLink syntax as it becomes a W3 recommendation and XML parser and XSL transformation engines support the syntax.

In the system, servables always result in one or more final published pages. The DTD must indicate the names of the XSL style sheets that can be used for layout for that particular type of document.

Because the servable includes content from subfragments, the style sheet must be written to work on the so-called *expanded servable*. Before page assembly, a servable is temporarily rewritten to include the content of all its subfragments. Thus the system implements a temporary solution that mimics the XLink functionality by expanding the servable

In addition, for each DTD, a Universal Format Transformation (UFT) document is defined that specifies the mapping of the metadata elements of the XML document to the columns of a relational database. The UFT is itself an XML document that abides to a particular DTD. Each UFT defines the relationship between the hierarchical structure of the XML document and the columns and tables of the relational database. In summary, the addition of a new document type to the system requires the definition of a DTD and the corresponding UFT. If the document type is a servable, style sheets defined in XSL are also required.

## METADATA STORE

The metastore is used to maintain information about the functional and semantic role of each fragment. The meta-

information stored in the metastore is grouped into system-generated tags and non-system generated tags. The values of the system-generated tags are generated by the dispatcher when a check-in is successful. The values of the non-system generated tags are specified by the content creator during the creation of the corresponding document.

The system generated tags correspond to the children of the SYSTEM element defined in every DTD, as described in the previous section. The non-system generated tags correspond to additional elements in the DTD's that are necessary for maintaining the functional and semantic role of the fragments. These tags can be further grouped into two parts: 1) the tags which are used for describing the XML object, such as keywords, categories and publishing information; and 2) the tags which are the necessary part of the XML object, such as TITLE and SUMMARY.

The metastore is implemented as a DB2/UDB database. The current implementation of the metastore is based on a fixed set of DB2 tables for all fragment types, but can be extended to include specific table(s) for different fragments.

DB2 is a relational database, and thus cannot be used directly to store an XML object, because the XML object has a hierarchical data model. We have to create one-to-one map from XML data model to a set of database tables. Currently, we are using a proprietary scripting language (UFT) to map the XML document elements that correspond to the metatags into a set of pre-defined DB2 tables.

For a content management system that will potentially have a very large number of interrelated documents and fragments, finding and locating a particular fragment or servable efficiently becomes one of the major challenges. We have concluded that such an operation based on a directory structure browsing operation is both inefficient and unreliable. We have replaced the browsing operation with a search operation that leverages the meta-information that is stored in the metastore. One of the essential functions of the metastore is to enable this search paradigm.

The search feature requires implementation at both client and server sides. At the client side, the UI provides a dialog that allows easy construction of search queries. The search query consists of the conjunction of elementary search conditions. The search conditions are created based on an initial XML specification sent from the server that specifies the searchable elements, the relational operators that can be used with each element, and in some cases the set of values that that element can assume. The client converts the query into a DASL query. As it receives the response from the server, the search dialog parses the results and displays them in a tabular format. From the table, the editor can select items that can be used in the editor.

At the server side, when the dispatcher receives the search query, it invokes the search module within the MetaStore Manager. The search module converts the DASL query into an SQL query dynamically, and queries the database. It then

converts the search result into DASL format and returns it to the client.

In order to ensure the scalability of the application, a number of techniques have been used to streamline database access operations. First, we use a database connection pool to maintain a set of active connections, instead of creating a new connection for each access. Second, we index the search fields in the database to speed up search operations. Third we cache the search results to minimize repeated access to the database for the same query from the same client.

### **FRAGMENT DEPENDENCY STORE**

The fragment dependency store uses IBM Research's Daedalus system to automatically propagate fragment changes to all affected fragments and servables, and to allow for multi-stage publishing to accommodate quality assurance. The fragment dependency store does this by creating an Object Dependency Graph (ODG), a directed acyclic graph within Daedalus, which represents the relationships of all fragments in the system. Several Daedalus stages are chained together to allow for multistage publishing.

### **Daedalus**

Daedalus is written in pure Java and implements *handlers* as pre-defined actions performed on the various configurable resources. Flexibility is achieved via Java's dynamic loading abilities, by more sophisticated configuration of the resources used by Daedalus, and through the use of handler preprocessing of input data. Most entities defined in a configuration file implement a public Java interface. Users may create their own classes to accomplish localized goals, and specify those classes in the configuration file. This permits run-time flexibility without requiring sophisticated efforts on the part of most users, since default classes are supplied to handle the most common situations.

For Franklin, we have created our own classes to implement three types of handlers: Extension Parser, Dependency Parsers, and Assemblers.

### **Extension Parser**

Within Franklin, Daedalus manages different types of files differently based on their extensions. Servables, simple, compound, and index fragments; style sheets and multimedia assets are all treated slightly differently in the publishing flow.

The Franklin Extension Parser takes in a name of a fragment, and returns an extension used in the Daedalus configuration files to specify actions to take during the publishing process.

The appropriate behavior for each type of fragment is defined in the Daedalus configuration files. These behaviors

include moving assets to different stages within the system as well as assembling the servables into the *expanded* mode described in an earlier section and invoking the XSL transformation to create viewable pages.

### **Dependency Parsers**

The Franklin Dependency Parsers analyze an XML object and updates the ODG maintained by Daedalus accordingly. Based on the Daedalus configuration file, the appropriate dependency parser is invoked for different types of fragments. The ODG maintains the dependencies between fragments. Currently we have defined two types of dependencies: *composition* and *style*. The *composition* dependency maintains structural information between fragments and between a complex fragment and its associated asset. The *style* dependency maintains information about the relationship between servables and stylesheets.

The composition dependency is unidirectional, it points from the subfragments to the fragments that include them. Thus allowing an efficient traversal of the dependency graph when a change occurs in one of the subfragments.

The ODG also maintains the dependency between a compound fragment and the associated media asset. In this case the dependency points from the fragment to the associated media asset.

### **Assemblers**

Daedalus is configured to invoke the appropriate Franklin Assembler for composite fragments and servables. The assembler executes the appropriate operations for the particular type of fragment. For the case of an index fragment, it makes a call to the metastore to retrieve the list of fragments or servables that satisfy a condition specified in the index fragment itself, and then creates an *expanded* version of this fragment. For the case of a servable, the Page Assembler assembles the servable into the *expanded* mode by including the contents of all included subfragments, and then invokes the XSL transformation engine to produce viewable output pages. As discussed in an earlier section, the first step of creating an expanded XML is a method that we are using in the absence of a final XLink standard and the lack of tools that handle XLink constructs.

The type of the viewable page, as well its target device, is determined from the stylesheet. The assembled XML and all the resulting viewable pages are written to one file, which is later split up, and the these pages are written to the appropriate directories on the server.

### **Chaining of Daedalus Stages**

Currently, two Daedalus stages are used in the publish process. Each Daedalus stage maintains a separate ODG,

and the sink of the first one is the source of the second, creating a publishing chain. The following diagram shows the set-up of the Fragment Dependency Store its relationship to the source and sink servlets, as well as a production web server where the composed viewable pages are stored.

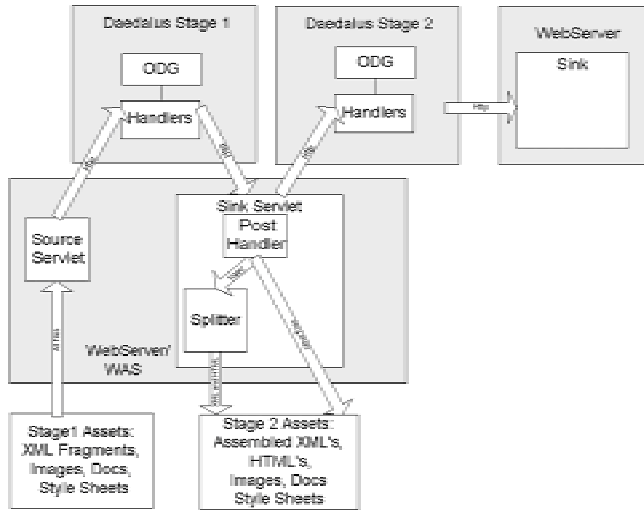


Figure 2. Fragment Dependency Store Architecture.

When a fragment is checked into the Content store, it is added to the first ODG, and a publish command is issued to the corresponding handler. Daedalus reads the fragment XML from the source servlet, uses the extension parser to find its extension, and then uses the dependency parser to find dependencies to add to the ODG. The appropriate assembler then pulls in the contents of the fragment's subfragments, and if the fragment is a servable, combines it with its stylesheets to produce the output pages (e.g., HTML files). The servable XMLs, output pages, binary files, and stylesheets - all fragments affected by the check in - are sent to the servlet specified as the sink of the first handler.

When a servable has been approved, a publish command on the servable fragment is issued to the handler of the second Daedalus. The corresponding ODG is updated, and all dependent servables are reassembled and recombined with their corresponding XSLs. The output pages are published to the production web server through a second sink servlet. Binary files (such as images) are also published to the second sink. This is where the web server pulls the final HTML and image files from.

The chaining of Daedalus modules can be extended to include any number of stages, as required by the particular workflow. The workflow for the particular implementation described in this section consists of two stages, a staging server and a production server.

## AUTOMATED USER INTERFACE CREATION

One of the biggest challenges of any publishing system is to remove as much complexity from the users' tasks as possible. When dealing with a relatively new technology like XML/XSL this aspect of the system becomes even more important. By hiding the syntax of XML from the editors and authors, domain experts can take on the role of creating and modifying the content without worrying about the syntax of a particular markup language.

One goal of Franklin, therefore, is to never present the tagging syntax to the user. Instead, Franklin creates a set of input forms that the user can easily fill out. However, some users insisted on placing simple HTML markup into text fields. Franklin does allow a small subset of HTML tags to be processed. However, this defeats many of the reusability and cross-platform publishing opportunities and is not a recommended strategy.

Users are assigned roles in the system and each role, in turn, is assigned specific document types. A user assigned to a role can only create or modify a document assigned to that role. When the user selects a document type to create or edit, Franklin reads in the DTD and automatically constructs an interface based on that document structure.

## DTD to Interface

The UI creation algorithm first constructs the basic interface from the DTD. This recursively adds widgets to the display as necessary. If we are creating a new document we simply display the widgets. If we are creating an interface from an existing XML instance, however, we insert the content into the appropriate widget. In addition, we must create additional widgets for elements that are repeated within the XML document.

Franklin makes a number of assumptions in handling DTDs and the automatic creation of the user interface. Most notably, the interface widgets are created for DTD elements, not attributes. Special attributes are then used to assist in the transformation of the element into an appropriate interface widget.

Until schemas become widely used, there is no standard way to provide typing for elements in the DTD. Franklin solves this problem by including the attribute, DATATYPE, whenever an element is to be displayed in the interface. If an element does not contain a DATATYPE attribute no input is allowed for that element. Children elements, however, may still contain DATATYPE attributes to specify their user interface. In addition, whenever an element has the DATATYPE attribute, it must contain a child of type PCDATA. This enables the DTD to specify, for example, whether a one line input, a medium text area or a large text area is required.

In the partial DTD shown here, TITLE, SHORTDESCRIPTION, and BODY each specify different text input widgets to use.

```

<!ELEMENT TITLE                (#PCDATA)>
<!ELEMENT SHORTDESCRIPTION    (#PCDATA)>
<!ELEMENT BODY                (#PCDATA)>

<!ATTLIST TITLE                DATATYPE
(%UITYPES;)                  "STRING"   #FIXED>
<!ATTLIST SHORTDESCRIPTION    DATATYPE
(%UITYPES;)                  "SHORTTEXT" #FIXED>
<!ATTLIST BODY                DATATYPE
(%UITYPES;)                  "LONGTEXT"  #FIXED>

```

The external entity *UITYPES* contains the list of all UI widgets known to the editor. These data types include:

```

DATE, INTEGER, STRING, SHORTTEXT,
LONGTEXT, CHOICE, BROWSESERVER,
BROWSELOCAL, LABEL, NOLABEL, ASSOCLIST,
ASSOCLISTDYNAMIC

```

Another typical interface widget is the drop-down menu. To accomplish this, the *DATATYPE* attribute is set to the *UITYPE CHOICE*, *ASSOCLIST* OR *ASSOCLISTDYNAMIC*, and a *CHOICES* attribute to a default value from a list of options. The options can be defined as an external entity for reuse across many DTDs. For example,

```

<!ENTITY % CATEGORYDEFS SYSTEM
"http://server/franklin/dtd/categorydefs.txt"
">

```

defines an external entity for a set of category choices. These choices could be defined as the types of Netfinity servers:

```

NONE | Netfinity_8500R |
Netfinity_7000_M10 | Netfinity_5500_M10 |
Netfinity_5600 | Netfinity_5500

```

The definition for *CATEGORY* in the DTD might then be:

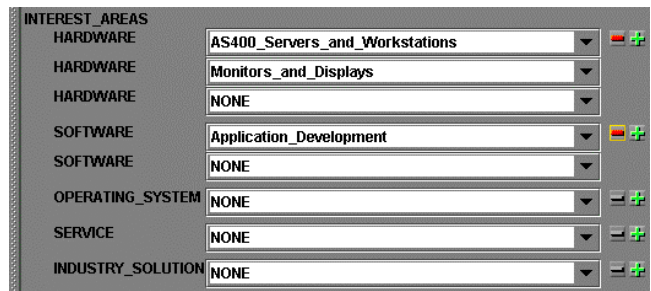
```

<!ATTLIST CATEGORY
DATATYPE (%UITYPES;)          "CHOICE" #FIXED
CHOICES (%CATEGORYDEFS;)     "NONE" #REQUIRED>

```

The editor assumes that if the first word in the set of *CHOICES* is the string *NONE*, and the user selects it, the XML element will not appear in the document.

In a DTD, elements can either be required, optional, or occur 1 or more or 0 or more times. If an element can appear more than once buttons appear next to the widget or group of widgets for replication, as shown in Figure 3.

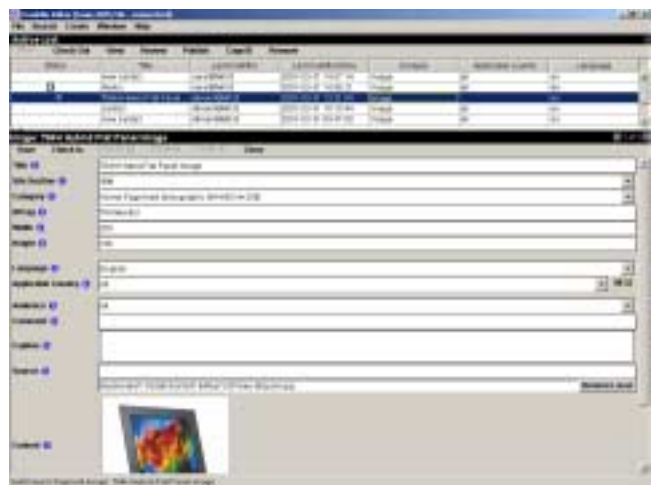


**Figure 3.** This illustrates how elements in the interface can be replicated to include 1 or more pieces of content. The -/+ buttons are used to add and remove widgets from the interface, and as a result, elements in the XML file.

Because of the strict way that the interface is constructed, each widget knows whether or not it is required and whether or not more elements can be added to an XML instance. If an element in the DTD is required, the widget will be highlighted (e.g. colored brightly) to allow the user to distinguish which fields must be filled in before submission. Therefore, only well-formed and valid documents are submitted to the server.

### Client user interface

Figure 4 illustrates the current Franklin client. This version of the client is a Java standalone application. The interface is broken into two main regions. The bottom of the interface displays the current document being edited. On the top, active documents are listed. This list can be created via a database search or the results of other creation operations within the current session. These active documents are used in constructing subsequent composite documents.



**Figure 4.** The Franklin interface with the current document being edited on the bottom and the active documents listed at the top.

## Object oriented UI

Each Java widget is encapsulated in a set of classes that include additional functionality. This object-oriented approach allows for modular design and future extensions to the set of interface widgets. Inheritance and generic methods are used throughout the class hierarchy for the definition of the interface widgets.

Each UITYPE may also provide very specialized functionality. For example, `BROWSELOCAL` and `BROWSESERVER` provide a button which, when clicked on, opens a dialog to choose a file on the local system or a directory on the remote server, respectively. This functionality is encapsulated within these particular classes.

`BODY` element tags are also handled specially within the system. The system assumes that `BODY` tags are composed of one or more `PARAGRAPH` tags. This is realized in the `LONGTEXT` widget in the user interface. Blank lines in the input are interpreted as paragraph separators. When constructing the XML document, these `PARAGRAPH` tags are automatically composed within the outer `BODY` tag. This functionality is inherited through the text widget class hierarchy.

## RELATED SYSTEMS

Other systems/tools that relate to Franklin's editor include markup languages that use XML to declaratively specify user interfaces, fully functioning editors, and systems that publish XML documents.

Bluestone Software's XwingML [9] enables the creation of Java Swing user interfaces without coding. The GUI is declaratively specified in XML and is translated into working Java code. This approach separates the GUI code from the application logic. Their DTD specifies the entire set of classes and properties for all of Swing components. Our goal is slightly different. Franklin doesn't want to create arbitrary interfaces in a declarative fashion, a much harder task. Franklin's goal is to create specific interfaces that reflect the document types for a given publishing environment.

XmetaL, from Softquad, [10] is a very flexible XML editor that supports three views into XML files. These views include raw XML mode, Tags-On mode that provides a WYSIWYG presentation with direct access to elements and attributes, and a full WYSIWYG mode in a word-processor like environment. The problem with this approach is that separate style sheets need to be used to support the editing vs. the publishing process. In addition, one stylesheet may not include all of the elements that would be used on other platforms or for different uses. The separation of content from presentation and the reusability of that content on different delivery environments make the use of WYSIWYG interface tools somewhat questionable.

Interwoven [11] is a complete publishing system that supports HTML as well as XML. It provides an end-to-end

solution from content creation to promotion and publishing. It also has a templating tool that provides the means to produce form-based pages. However, its support of reusable fragments within the environment is rather limited and the publishing to viewable pages is performed using non-standard methods.

## CONCLUSION

We have presented in this paper our experiments with the design and implementation of an XML content management system. The system is based on the following two fundamental ideas. First, separation of content and style: Information stored in the content management system is independent on how it is going to be presented. The presentation style is encapsulated elsewhere and can be used to customize the look and feel based on the end-user preferences as well as the delivery methods and devices. Second, reusability of information content: By encapsulating common information in fragments and subfragments and making these fragments insertable in other fragments, we avoid scattering and duplication of information. This allows us to restrict the edit operations to a limited number of relevant fragments, to affect global changes.

The implementation of the system is based on the following concepts:

1. Standards based design: The different components of the system interact through well-defined API's based on industry standards, such as: XML, XSL, WebDAV, HTTP, DASL.
2. Pervasive use of XML: We are using XML not only as the content model but also as the language in which information is transferred between the different parts of the system.

In order to demonstrate these ideas, we had to use existing technologies and standards. Newer standards or standard track proposals, such as XLink and XML Schema, and technologies based on those can be leveraged to improve the design and the implementation. These are future directions that we are contemplating. We have also identified a number of features that we intend to add to subsequent versions of the prototype. These include automated extraction of keywords and automated translation. We have also considered the implementation of a more web-centric client that requires no installation and can easily be accessed from any browser

## ACKNOWLEDGMENTS

We would like to thank the Internet Technology group and in particular David Grossman, Maria Hernandez, Bill Sweeney and Abel Henry for their help and support.

## REFERENCES

1. World Wide Web Consortium, 2000. Extensible Style Sheet Language (XSL), <http://www.w3.org/Style/XSL/>
2. World Wide Web Consortium, 2000. Extensible Markup Language (XML), <http://www.w3.org/XML/>
3. Jim Challenger, Paul Dantzig, and Arun Iyengar. "A Scalable and Highly Available System for Serving Dynamic Data at Frequently Accessed Web Sites" In *Proceedings of ACM/IEEE SC98*, November 1998.
4. Jim Challenger, Arun Iyengar, and Paul Dantzig. "A Scalable System for Consistently Caching Dynamic Web Data." In *Proceedings of IEEE INFOCOM '99*, March 1999.
5. Arun Iyengar and Jim Challenger. Improving Web Server Performance by Caching Dynamic Data. In *Proceedings of the 1997 USENIX Symposium on Internet Technologies and Systems*, December 1997.
6. Arun Iyengar, Jim Challenger, Daniel Dias, and Paul Dantzig "Techniques for Designing High-Performance Web Sites," Submitted to *IEEE Journal of Internet Computing*, October, 1998.
7. IBM XML4J XML Java parser <http://alphaworks.ibm.com/tech/xml4j>
8. Lotus LotusXSL, <http://alphaworks.ibm.com/tech/LotusXSL>
9. Bluestone Software XWingML <http://www.bluestone.com>
10. SoftQuad Xmetal <http://www.xmetal.com>
11. Interwoven TeamSite, OpenDeploy and Templating <http://www.interwoven.com>
12. IBM DB2 XML Extenders <http://www-4.ibm.com/software/data/db2/extenders/xmlext/docs/v71wrk/dxxawmst.htm>