XML Support to Design for Testability

Daniel Deveaux[1], Guy St.Denis[2], and Rudolf K.Keller[2]

[1] Lab. VALORIA (AGLAE) – Université de Bretagne Sud
VANNES FRANCE – Tel: (33) 297 463 175
Daniel.Deveaux@univ-ubs.fr
[2] Lab. GELO – Université de Montréal
MONTREAL CANADA – Tel : (1-514) 343 7474
g.stdenis@computer.org - keller@iro.umontreal.ca

Abstract. The sophisticated document management needs of present and future Web-based environments have spawned the XML specification as well as a host of related technologies. In software engineering, XML to date has mostly been used to support three sub-activities: documentation management, data interchange and lightweight data storage. In this position paper, we propose using XML technology as the infrastructure for the integrated management of all core software development information. For several years now we have been developing the concept of *design for testability* based on a self-documented and self-testable class model. The aim of this paper is to propose a master document type that captures all relevant information for a class, i.e. documentation, contracts, tests, and so on. This document is defined by an XML DTD and we have tentatively named the resulting markup language OOPML: *Object-Oriented Programming Markup Language*.
After presenting our DTD prototype and describing the dedicated software development framework that we are currently building, we explore the enhanced work and control activities made possible by XML technology. Related software project activities (design with UML, documentation, project management) are also explored.
Keywords: object-oriented development, docware approach, XML, SE tools achitecture

1 Introduction

The emergence of the XML (Extensible Markup Language) standard, recommended in the fall of 1998 by the W3C (World Wide Web Consortium) and supported by the majority of document producers and consumers, has cast many application domains in a new light. Because XML technology facilitates the production, management and exchange of all structured documents, it is natural to exploit its strengths within software project management where most if not all artifacts are documents (textual or other). At the time of this writing, several software engineering projects that employ XML are under development and they can be classified into three general categories: documentation support, data interchange and lightweight data storage. We discuss these applications in section 5.

In this position paper, we propose applying XML technology to software development in a manner akin to the basic philosophy of the literate-programming community [Coa98,Cov98]; essentially, XML provides the technical infrastructure to enable the integrated management of all core software development information. Our approach relies on the observation that the majority of forward and reverse-engineering software tools are built around a similar internal model: a more or less sophisticated abstract syntax tree with various representations (e.g., serialized as tuples, semantic nets, trees of objects, and so on). Unfortunately, the proprietary and incompatible nature of these representations is a major obstacle to tool interoperability. In particular, a considerable gap exists between analysis and design tools which are increasingly UML-based, and code management tools which rely on source code parsing.

For CASE tool developers, this state of affairs results in a disproportionate amount of time and energy being spent managing a given tool's basic technical infrastructure rather than enhancing the functional capabilities that will distinguish it from competing products. For software developers, the incompatibility of software system representations inhibits CASE tool interoperability (e.g., because of time-wasting and error-prone data imports and exports via non-standard exchange formats) and constitutes a limiting factor in the evolution of fundamental areas such as traceability.

In this proposal, we draw upon our teams' combined experiences in this domain: the LSDoc tool of the AGLAE project [Dev99a] uses an SGML engine (recall that XML is a simplified version of SGML), and the SPOOL framework developed by the GELO lab [KSRP99] now uses an XML-based data interchange format [SSK00].

Following a brief overview of XML, we present the details of our exploration. We then develop a few examples illustrating the usefulness of XML technology and examine related works. We conclude with an outline of an architectural framework for software engineering.

2 XML: an overview

The unprecedented success of the World Wide Web has provided the incentive to define a Web-enabled document management infrastructure: the resulting XML metalanguage specification [W3C98,Mic99a,St.98,Meg98] retains the extensibility of SGML but is considerably simpler to handle. The arrival of this exciting technology prompted the question: "Can XML technology and its related tools be used to support object-oriented software development and maintenance?". In our view, there is a considerable advantage in adapting existing solutions provided by document management specialists to serve the needs of software engineers rather than developing from scratch and maintaining parallel technologies and tools.

The underlying principles of modern document management systems consist in separating content from presentation and storing information in plain text documents. These objectives are fulfilled by marking up documents with

meaningful tags that delimit and identify the data elements that they comprise. A given set of tags and their allowable arrangements constitute a markup language. XML is a markup metalanguage specification that is used to define concrete markup languages. These markup languages in turn define the structure and content of a class of documents, either through the use of the DTD (Document Type Definition) or the XML Schema mechanisms. An example of a marked-up Java source code fragment is given in figure 1

```
<Method id="meth.item">                  | <ReturnType>Object</ReturnType>
 <Ident>                                 | </MethSign>
  <Name id="item"> item </Name>          | <BlocCode plang="java"
  <Role>                                 |           visibility="priv">
   get the value on top of the stack     |  <Comment>
  </Role>                                |   the top of the stack is the last
 </Ident>                                |   element of the Vector
 <Description>                           |  </Comment>
  <Para>                                 |  <Code>
   Define abstract method                |   <![CDATA[
   inherited from Container              |     st().profile() ;
   (contracts already defined)           |     return (coll.lastElement()) ;
  </Para>                                |   ]]>
 </Description>                          |  </Code>
 <MethSign>                              | </BlocCode>
  <Visibility> public </Visibility>      | </Method>
```

Fig. 1. XML sample: a Java method marked-up with OOPML

The W3C is currently in the process of standardizing numerous related technologies that extend and enhance the usefulness of XML:

- XPath (XML Path Language), XPointer (XML Pointer Language) and XLink (XML Linking Language) [W3C99d,TEI99,McG98] make it possible to define, manage and exploit advanced hyperlinks
- XSL (Extensible Stylesheet Language) and XSLT (XSL Transformations) [W3C99b] are complementary technologies that define stylesheets and a transformation language respectively. The style-sheets are used to specify how a given XML document should be presented in a particular context, and the transformation language is used to transform one XML document into another.
- To facilitate the automated machine-processing of data contained in XML documents, the W3C has defined the RDF (Resource Description Framework) specification [W3C99c,OCL99,Bra98] which provides a meta-data definition and processing framework. RDF makes it possible to define domain-specific vocabularies that describe resources of interest for a given community.

– Finally, there are currently two APIs that define how applications access and manipulate XML documents: DOM (Document Object Model) is a tree-based data model [W3C99a] whereas SAX (Simple API for XML) is an event-based interface. The existence of these standard APIs encourage the construction of generic tool-building libraries and components that XML-based applications can reuse, giving them immediate access to XML documents. Many useful tools in use today (e.g., parsers, editors, ...) are built from such libraries.

The rapid success of XML has resulted in a flood of specifications, documentation and tools which can easily overwhelm an unsuspecting newcomer. To alleviate the information overload, we have summarized our first bibliographic study and related tool exploration in an *"XML world map"* which is available at the XML4SE Web site (XML for Software Engineering[1]).

3 The *Object-Oriented Programming Markup Language* project

Our team has been collaborating for several years with IRISA's Pampa Project (see acknowledgements) to develop a class design methodology, called *design for testability* [DFF+00], that integrates documentation, contracts and tests to improve the trustability of software components. This approach is based on a self-documented and self-testable class model that has been implemented in several object-oriented languages (Eiffel, Java, Perl) [DJ99,TDJ99]. This model has been used for several years now for teaching purposes [DFF99] and experiments are currently under way to use it in professional developments. These products are freely available from the Web and are distributed under the GNU General Public License under the names STclass[2] and LSDoc[3].

As a first exploration of XML support for software engineering, we have studied the feasibility of implementing our self-documented and self-testable class model as an XML document. We were able to formalize an in-memory DOM that captures all class-related information (e.g., source code, contracts, many levels of documentation, test suite, etc.) upon which it is possible to apply editing and/or validating operations. This DOM can be built from an existing XML document or from a source code file in a given programming language documented with LSDoc or another similar documentation tool, and it can subsequently be saved as an XML document. Additionally, we can extract from this DOM compilable source code and various documentation views.

The DOM structure is defined by the DTD for our proposed Object-Oriented Programming Markup Language, or OOPML. Our initial framework prototype supports Java-based source code, but we stress the fact that OOPML is independent of any programming language [4]. Likewise, our framework is extensible to handle

[1] URL: http://www.iro.umontreal.ca/labs/gelo/xml4se
[2] URL: http://ww.iu-vannes.fr/docinfo/STclass
[3] URL: http://ww.iu-vannes.fr/docinfo/LSDoc
[4] In fact, an OOPML structure can simultaneously accommodate implementations of a given class in different programming languages.

other programming languages as well. Our XML-Java framework is depicted in figure 2.

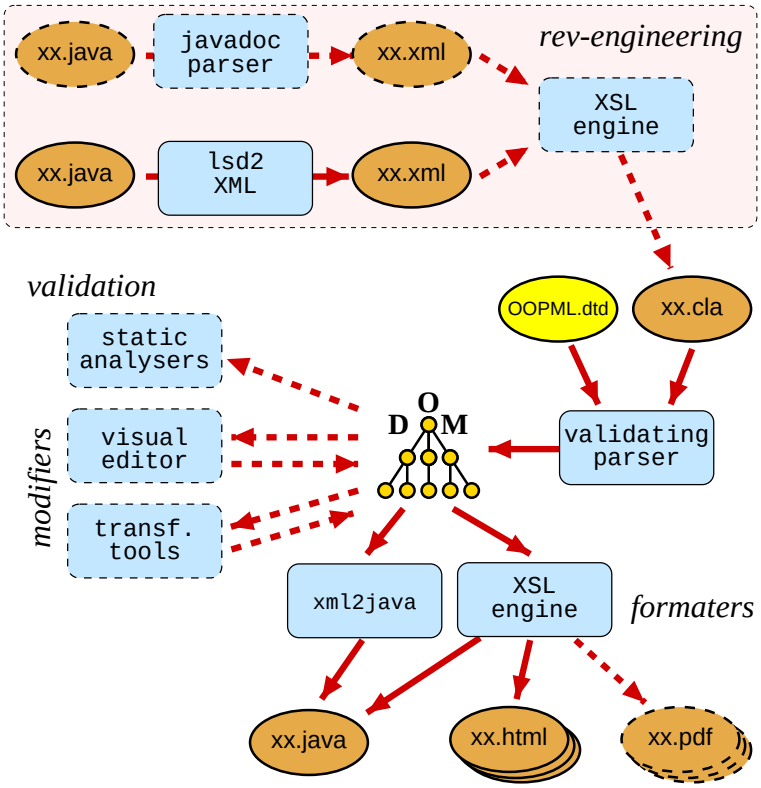

Fig. 2. A framework to develop Java code with XML

In this framework, the primary storage unit for classes are *.cla files which are in fact XML documents, not Java files; the Java to XML conversion (top of the figure) is a reverse engineering operation. In a typical software development process, the classes are created with a visual editor or are extracted from XMI documents.

As mentioned by G. J. Badros [Bad00], all modification and verification tools operate on the *Abstract Syntax Tree* (or the DOM in our case) because they can efficiently exploit a well-defined and well-understood structure. Although our framework tolerates incompleteness and modifications to the DOM, we can rest assured that the resulting AST remains valid at all times by using a DOM validating parser.

The bottom right branch of the figure shows the code generation and documentation processes that are based on an XSL engine. The left branch, xml2java, corresponds to our initial attempt to extract Java code from the DOM using a Perl script: this path has been abandoned. This portion of the figure clearly shows how the source code has become a by-product of our proposed development process, much in the same way as technical documentation is extracted from source code in typical development processes today.

In the figure, the steps shown in solid lines have been carried out using tool prototypes, which we will describe shortly. The rest of the process appearing in dashed lines constitutes current or future work. The development of tool prototypes relied on several XML 'accessories' that are freely available on the Web. The majority of components were libxml-perl [DMPW99] and IBM-Alphaworks [IBM99] products: XML4J parser, Xeena editor, LotusXSL transformer.

Our first step was to produce a non validated XML instance of one of our Java classes (Stack.java); this was achieved by extending our LSDoc translation tool, lsd2, with a new formatting XML option. From this instance we constructed the OOPML DTD which we designed to leverage XML's new capabilities, and we recast the Stack class based on this new document type. The DTD OOPML.dtd and the XML form of the class (Stack.cla) were then validated with IBM's Xeena validating editor. For the final extractions we initially developed a Perl script (xml2java.pl) to extract a compilable Java source file from the XML document. Afterwards, we used an XSL engine (LotusXSL) and developed XSL scripts to extract Java code and technical documentation (in HTML and Postscript) with various views. All of this material is readily downloadable from our oopml[5] Web page.

At present, several tasks are under way:

- validate the OOPML DTD with many applications,
- make use of intermediate DTDs and XSLT to transform non valid XML class material (from javadoc or LSDoc) to valid OOPML documents,
- make use of XLink to improve intra and inter-class browsing and to manage class hierarchies,
- develop a dedicated editor, based on the DOM, to create and edit classes, features and tests.

Our goal is to build a framework that supports the development and testing process of self-documented and self-testable classes with enhanced automatic validation and control possibilities. This framework should allow the development of new classes as well as the integration of existing classes through reverse-engineering operations. The functional prototypes that we have developed clearly indicate the feasibility of such a project.

4 Improved quality control with XML

XML provides several mechanisms that can improve the software development process:

[5] URL: http://www.iro.umontreal.ca/labs/gelo/xml4se/oopml/

- The robust definition of well-structured documents by DTD or XML Schema enable structure validations on all documents during the design process.
- XSL defines a generic transformation engine that can drive the overall document production for the project, including activities such as converting a model to code or extracting a documentation view. .
- XLink defines very advanced linking techniques that can be used to manage inheritance and implement traceability and consistency control.
- Meta-data support through RDF should facilitate information interchange between project, versioning and process management.

The interest in XML as a support for software engineering environments isn't limited to its efficient and portable data structures managed by standard and available tools. As a matter of fact, an XML environment offers infinitely more document structuring possibilities than those available from the source code of current programming languages. In this section we will show, for illustration purposes, two situations where XML technology provides elegant and easily implemented solutions to problems that current tools don't handle very well.

4.1 XLink applications

One of the interesting possibilities provided by the XLink specification consists in using extended out-of-line links to dynamically create relationships between resources without having to modify the actual resources themselves. We describe in the following text the *linkbase* concept and afterwards present a usage example.

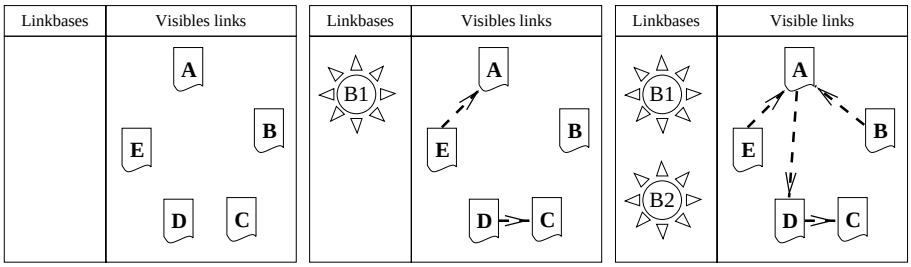

Fig. 3. Visible links in a browser after reading of linkbases

A *linkbase* can be implemented using an ordinary database that feeds link information to an XML browser's dedicated link engine. A simpler alternative however involves encoding the *linkbase* as an XML document that contains a list of extended links; these links define a relationship network between arbitrary resources. Such a document is called a "hub" because it serves as a starting point for a browsing session. Figure 3 shows how a *linkbase* can be used. Images

represent a browser's successive depictions of the link network between resources A through E. This example illustrates the great flexibility of `XLink` technology and the possibility of creating particular views of a resource set. Because the link management takes place independently of the linked resources, the stability of the resources is maintained and the reuse and adaptability potential is considerable.

Within a software engineering context, `XLink` facilitates the automated management of vital link information between a system's components, such as class hierarchies, interface inheritance, and client-supplier relationships. Let's explore the simple class hierarchy relationship to see how automated link management can take place. The `OOPML DTD` contains an extended link element called `Inherit` which is used to specify a given class' superclasses. When an `XML` browser and its link engine process a class document, the link information between this subclass and its parents, as well as the links flowing in the opposite direction, will become visible to the user. However, it would be interesting (and useful) to be able to navigate from a superclass to all of its descendants without having to preload all the class documents defining the subclasses.

Without `XLink`, this type of navigation would have required modifying the superclass document in order to add references to all of its children. This approach is error-prone because it relies on manual intervention. With `XLink`, a more practical solution consists in creating "structural" hubs, that is, *linkbases* which contain structural information about a software system. The hub is created and maintained automatically by parsing new or edited class documents, extracting the appropriate link information (figure 3), and updating a *linkbase* (which is independent of the class documents) that contains up-to-date relationship information between the superclasses and their dependent subclasses. This scenario effectively eliminates the need for laborious and error-prone manual updates. Afterwards, we can immediately view all inheritance information for any class in the hierarchy by loading the single *linkbase* in the browser. Naturally, this example may be applied to any type of relationship: containment and class collaborations are two additional examples.

By generalizing the structural hub idea even further, we can envision the existence of *linkbases* offering custom views of a system adapted to the documentation needs of the user.

We'll round out this section on `XLink` with a final example. The possibility offered by `XLink` to navigate to several destinations from a single starting point offers interesting solutions for a source code browser. Let's consider the case where a developer wishes to obtain information for a given class member, say, a method. Using `XLink` technology, the developer can click on the member's name and then be presented a menu of possible destinations: typical choices include the member's definition, its declaration, all references made to it, and so on. In fact, any number of links can be easily provided because a class document is never edited as such; all that is required is to create appropriate *linkbases* defining the desired views and to selectively load them into the browser based on the type of information that is required.

4.2 Towards progressive validation

In the current state of software development practices the documented class structure (i.e., our `OOPML DTD`) described previously remains inapplicable for the most part: few development processes preserve as much information in the source code. Moreover, there are real-life cases where this abundance of information is unjustified: educational situations and throw-away programming for prototyping are two such cases. Furthermore, in maintenance or reverse-engineering situations, we are required to work with the software as it currently exists.

These observations underline the fact that software engineering tools must be able to handle project documents regardless of their quality level and their completeness with respect to an ideal model. The handling of incomplete models is in reality a constant need: when new components are being developed, every related document is incomplete because it is still being drawn up.

Luckily, `XML` accepts a document which is simply well-formed (i.e., it is syntactically correct as per the `XML` specification, but its structure does not conform to a known `DTD`). In this context, it is possible to convert any structured document to an equivalent `XML` version and to load it into a `DOM` repository, whether the document's structure was or wasn't defined in a `DTD`. `XML` tools (e.g., filters, editors, etc.) can analyze and edit the document, and at any moment, validate it against a model by activating a `DOM` parser. This approach is particularly well suited in teaching situations, where a model's detail level can adapt rapidly be adapted to suit evolving student skills.

In our teaching environment, we require that students use the self-documented and self-testable classes model [Dev99b,DFF99]. During the initial phases, imposing the complete model at once is obviously out of the question. In fact, over the two year initiation period, our students are exposed to four successive versions of the model (code names are in parentheses):

1. (`ini`) Initially, the only requirement is a compilable and operating program. The use of adequate and well-placed comments is explained and recommended, but not imposed.
2. (`jdb`) In the second phase, the `javadoc` documentation is introduced and the distinction between interface and implementation views is explained. At this point, `javadoc` comments become mandatory.
3. (`jdc`) Then, contracts are added.
4. (`full`) Finally, the whole model comprising contracts, testing facilities and multiple documentation views in `LSDoc` is put into action [Dev99a].

In fact, each of the first three cases corresponds to an increasingly reduced subset of the full model and can therefore be placed in a document structure in which certain elements are simply absent; we can easily add them afterwards, thereby improving the quality of our document.

Using simple constructs (a description follows), `XML` allows us to decide whether to consider or ignore certain elements of a `DTD` (e.g., ignore the 'Test'

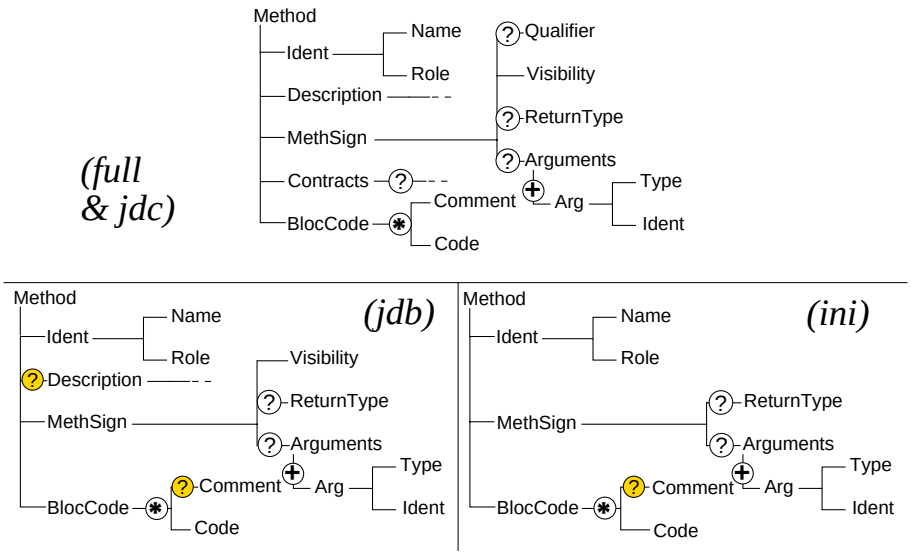

Fig. 4. Creating `DTD` levels through tree pruning and constraint relaxation

element and its descendants) and to activate or deactivate the mandatory nature of any element . Figure 4 illustrates this level creation mechanism for the 'Method' substructure in the `OOPML DTD`.

Thus, using parameterized `DTD`s, it is possible to validate arbitrary program documents of varying levels of quality and/or completeness. In reverse-engineering activities, this progressive validation may also be used as a *measure* (e.g., the amount of reverse-engineering effort required); the software under study would be loaded without validation in a `DOM` repository and then validated against models of increasing completeness levels.

`XML`'s `INCLUDE/IGNORE` mechanism ([Mic99a] p.75) allows us to control the visibility and the constraints over the document through the definition of entities. The selection of the desired validation level is determined with a switch inserted (automatically, eventually) at the top of the source file to be processed.

Moreover, our initial attempts have shown that it is possible to exploit the structural constraints imposed by a `DTD`. For example, we can verify and enforce programming style guidelines such as nesting limits in alternative or iterative structures, and forbidding multiple 'return's within a method. We are convinced that by defining sufficiently detailed `DTD`s a large percentage of the checks currently (not) carried out by cross-validation or reviews could be made automatically by a validating `XML` parser.

The enhanced functionality described above should be considered as a free side effect of `XML` use. It can moreover be particularly useful in teaching or reverse-engineering situations.

5 Related work

Documentation support – Over the years, the SGML community has proposed standards such as DocBook [WM99] for the structure and presentation.of technical documentation for software. The Linux system documentation (LinuxDoc) uses a similar standard.

Because of SGML's complexity, these projects proposed a single yet rather complicated DTD that could manage every possible document type. Such an approach assumes that the DTD is a *standard* (thus implying a slow evolution), and it imposes relatively heavyweight tools for document management.

XML's big advantage over SGML is that it permits small DTDs that are dedicated to one particular purpose. We believe that document interchange support cannot be obtained using a single worldwide monolithic structure; rather, document translations will be achieved using XSL transformations in collaboration with appropriate vocabularies defined with RDF. Moreover, many document structures proposed in these smaller DTDs are highly interesting.

An XML DocBook DTD[6] has been available since 1999. In order to ease the transformation to and from DocBook, we have chosen to align element names in OOPML with the corresponding elements in DocBook.

Recently, several projects (e.g., Apache Cocoon [Apa99], JDox[7]) have used XML documents as the target for javadoc extracted class documentation. A javadoc DTD is defined and doclets are developed to produce valid XML documents. Afterwards, XSLT scripts are used to generate final presentations on Web or paper.

The Apache project includes another noteworthy item: it aims to define a Web server that stores all of its information in XML format and dynamically extracts views in XML, HTML, or PDF for Web clients. Such a server can play a central role in the construction of software design team support, particularly if it is associated to a WebDAV[8] application.

Data interchange – This facet has been explored mainly by the UML (Unified Modeling Language) community which has proposed several data interchange formats such as XMI [OMG98], Microsoft's XIF [Mic99b], and UXF[9].

The standardization of interchange formats in software engineering (SE-EDI) is certainly necessary but we contend that these interchange documents cannot contain all the information required for validation or process control, unless all participants use the same methods and tools, which is rather unlikely in reality. Instead, in order to meet a developers' needs, an interchange format should be handled by the development tool at hand by standard input parsing and output filtering techniques.

[6] URL: http://www.nwalsh.com/docbook/xml/
[7] URL: http://www.componentregistry.com/javadox/
[8] URL: http://www.webdav.org/
[9] URL: http://www.yy.cs.keio.ac.jp/~suzuki/project/uxf/uxf.html

Lightweight data storage – An XML document is a suitable candidate to store working data for many applications. In software engineering, a common activity is UI generation. In an XML context, an interactive tool is first used to construct the interface and to save its definition in an XML configuration file. Filters are then used to generate application classes for different programming languages (Proto[10], Glade[11]). Similarly, XML lies at the heart of the *Open source* Gnome[12] project. And finally, an emerging application domain for XML is beans construction, as evidenced by the BML[13] = *Bean Markup Language* from IBM-Alphaworks, and Koala[14] from INRIA.

With respect to the literate programming with XML approach [Coa98,Cov98], our goal is more ambitious: we wish to formalize all project information (textual documents, models, code) as XML documents. Standard XML tools will support the design evolution and transformation process and will allow quality control throughout. However, we have not adopted the literate programming paradigm because it requires a radical change in point-of-view with respect to the design activity. We believe that the next step is to propose a model and a development process that permits a gradual re-engineering of existing source code.

Besides, we believe that it is important to have different document structures to store classes, design models, textual reports and project management data. Each level has to manage its own information: for example, implementation description and verification is strictly local to each class and should never appear in conceptual models. In that way, OOPML representation of software application is complementary to the UML model: it is possible to generate OOPML from an UML case tool and research is under way to implement backward traceability from class to UML models. This traceability should leverage XLink's capabilities.

Recently, two works closely-related to OOPML have been published: G. J. Badros' JavaML [Bad00] and the SDS open development initiative [Fou00]. OOPML and these works complement each other. In OOPML, the emphasis is placed upon generic class structure and documentation support, whereas in the other projects the central goal is to capture all code structure; the source code file remains the central code storage mechanism. The SDS platform can be adapted to a large collection of programming languages, whereas JavaML is dedicated to Java. The greatest shortcoming of these proposals however is the poor handling of comments (e.g., javadoc) in the sources; it should be possible to drastically improve information content by including comment analysis during the parsing of source code.

Email discussions are currently under way with G. J. Badros: we have proposed using JavaML (extended with javadoc comments manipulation) as an intermediate step between an OOPML representation of a class and Java compilable source code. The resulting schema is displayed in Figure 5.

[10] URL: http://www.pierlou.com/prototype/
[11] URL: http://glade.pn.org/
[12] URL: http://www.gnome.org/
[13] URL: http://www.alphaworks.ibm.com/formula/bml
[14] URL: http://www.inria.fr/koala/kbml/

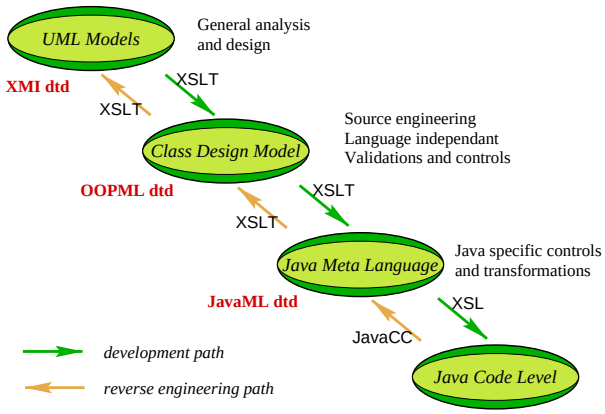

Fig. 5. DTD levels in Java development process

With these two levels of representation, it is possible to develop Java-specific tools for the JavaML model and generic, reusable for other languages, tools at the OOPML level. Nevertheless, a question remains unanswered: should we use a specific, slightly modified DTD for each language, or a generic model like CSF in the SDS project? Actually, we favor the former approach because it simplifies translation tools such as the language parser used in the first reverse engineering step.

6 Conclusion and outlook

Towards an XML-based architecture: The schema in Figure 6 illustrates the kind of framework organization that we intend to construct.

This figure presents four families of tools built around a data core which is based on the DOM API; the actual data may reside in live memory or off-line in some type of database. The typography of figure 6 distinguishes the general tools (described in the preceding paragraphs) from the tools that must be developed specifically for software engineering. The proposed architecture is not in itself revolutionary: several software engineering frameworks, including SPOOL and LSDoc, rely on a common data representation at their core. However, we believe that the proposed framework contains a few new twists and deserves a closer look. Firstly, the framework's core is based on the recognized DOM API for which several proven and robust browsers can be readily obtained, often at minimal or no cost. Additionally, several of the suggested tools in figure 6 (in the sans serif font) already exist or are presently being developed by electronic document specialists: software engineering tool developers are thus free to concentrate their efforts on their own specialized applications. Finally, and of greater importance than the preceding considerations which are, after all, only of a material na-

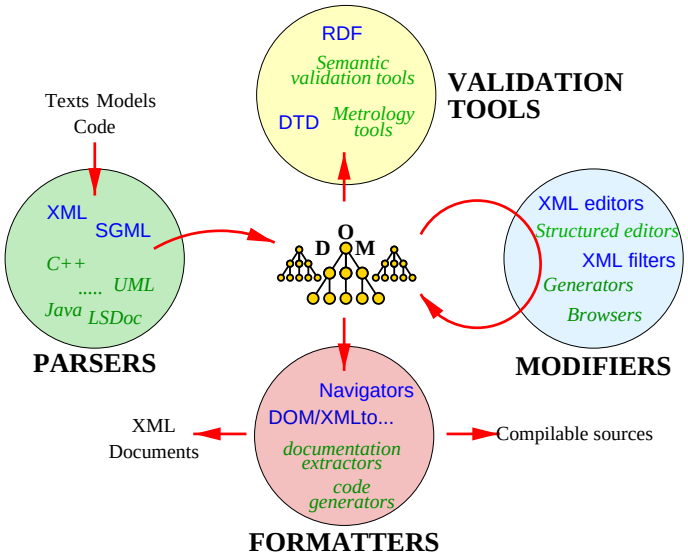

Fig. 6. `XML`-based software engineering environment

ture, `XML` offers a new and fundamental paradigm: the unified management of all project documents (requirements specifications, design models, source code, tracking logs or maintenance reports). This last point is crucial and constitutes a basic requirement towards greater coherency and traceability in the software development activity.

Figure 6 defines a general architectural schema. We believe it is important, in this schema, to carefully distinguish the four families of applications: parsers, validators, editors and formatters. This distinction assures maximal flexibility and adaptability and allows for the reuse of tools which haven't been designed explicitly for software engineering.

Moreover, the biggest difficulty at this time with `XML` and its related tools is to sort through all of them in order to understand which needs are best fulfilled by each one. After having analyzed numerous articles and proposals, we suggest the layered organization shown in Figure 7 for data structures and their related processing by various `XML` tools. We have associated each of the four layers with `XML` specifications or tools that appear best suited to provide the required solutions to the challenges that are stated.

In conclusion: the `DTD` and tools that we have implemented, the scenarios that we have developed to manage traceability and to exploit progressive validations are all prototypes; nevertheless, our results allow us to claim that we must seriously consider `XML` as a fundamental technology in the design of software engineering applications. Firstly, significant time and effort savings are possible

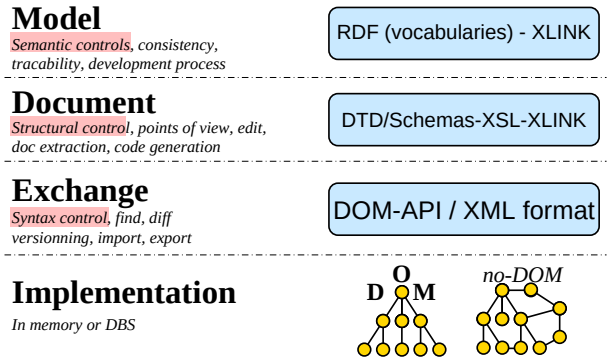

Fig. 7. Use of XML technologies in software engineering tools

thanks to the tool reuse that XML allows. Secondly, XML provides original and innovative solutions to recurring problems in our discipline.

Based on the results of this preliminary study, we will attempt to build stable document models for the whole spectrum of project-related information and study more thoroughly the use of XLink technology to manage a project's traceability and to provide browsing solutions for interactive tools.

Acknowledgements: The *design for testability* approach and self-testable class concept have been developed in *"the trusted components initiative"*[15] in collaboration with Jean-Marc Jézéquel and Yves Le Traon from IRISA's *Pampa project*[16]. Thanks also to Patrice Frison and Régis Fleurquin from the AGLAE team for their help in developing the OOPML specification.

References

[Apa99] Apache. Cocoon. 'http://java.apache.org/cocoon/', November 1999. World Wide Web site - Java Apache Project.

[Bad00] Greg J. Badros. Javaml: a markup language for java source code. In *WWW9, Ninth International World Wide Web Conference*, Amsterdam, May 2000.

[Bra98] Tim Bray. Rdf and metadata. http://www.xml.com/xml/pub/98/06/rdf.html, June 98. World-Wide Web document.

[Coa98] Anthony B. Coates. XML and literate programming. 'http://www.ems.uq.edu.au/Seminars/XML_LitProg/', 1998. World-Wide Web document.

[Cov98] Robin Cover. SGML/XML and literate programming. 'http://www.sil.org/sgml/xmlLitProg.html', 1998. This document

[15] URL: http://www.trusted-components.org/

[16] URL: http://www.irisa.fr/pampa/

includes links to other literate-programming-in-SGML documents and software packages.

[Dev99a] Daniel Deveaux. Distribution de LSDoc-4.6 sur internet. 'www.iu-vannes.fr/docinfo/LSDoc', December 1999.

[Dev99b] Daniel Deveaux. Distribution de STclass sur internet. 'www.iu-vannes.fr/docinfo/STclass', June 1999.

[DFF99] Daniel Deveaux, Régis Fleurquin, and Patrice Frison. Software engineering teaching: a 'docware' approach. In ACM, editor, *ITiCSE'99*, Cracow, June 1999. ACM - ITiCSE'99 Symposium.

[DFF+00] Daniel Deveaux, Régis Fleurquin, Patrice Frison, Jean-Marc Jézéquel, and Yves Le Traon. Composants objet fiables : une approche pragmatique. *L'Objet*, April 2000.

[DJ99] Daniel Deveaux and Jean-Marc Jézéquel. Des classes auto-testables. In Jacques Malenfant and Roger Rousseau, editors, *LMO'99 : langages et modèles à objets*. Hermes, January 1999.

[DMPW99] Eduard Derksen, Ken MacLeod, Eric Prud'hommeaux, and Larry Wall. libxml-perl. 'http://bitsko.slc.ut.us/libxml-perl/', August 1999. World Wide Web download.

[Fou00] Software Development Foundation. Sds. 'http://sds.yi.org/', February 2000. World Wide Web site - Open source project.

[IBM99] IBM. alphaworks. xml parser for java, version 1.1.9. http://www.alphaworks.ibm.com/tech/xml, April 1999. Web site.

[KSRP99] Rudolf K. Keller, Reinhard Schauer, Sébastien Robitaille, and Patrick Pagé. Pattern-based reverse engineering of design components. In *Proc. Twenty-First Conference on Software Engineering (ICSE'99)*, pages 226–235, Los Angeles, CA, May 1999. IEEE.

[McG98] Sean McGrath. XLink: The XML linking language. *Dr. Dobb's Journal of Software Tools*, 23(12):94–101, December 1998.

[Meg98] David Megginson. *Structuring XML documents*. The Charles F. Goldfarb series on open information management. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1998.

[Mic99a] Alain Michard. *XML : langage et applications*. Eyrolles, Paris, 1999.

[Mic99b] Microsoft. Xml interchange format (xif) now available. http://msdn.microsoft.com/repository/technical/xif.asp, may 1999. Web based article.

[OCL99] OCLC. Metadata. http://purl.oclc.org/metadata/dublin_core, June 1999.

[OMG98] OMG. Xml metadata interchange (xmi. ftp://ftp.omg.org/pub/docs/ad/98-10-05.pdf, October 1998. Document ad/98-10-05.

[SSK00] Guy Saint-Denis, Reinhard Schauer, and Rudolf K. Keller. Selecting a model interchange format. the SPOOL case study. In *Proceedings of the Thirty-Third Annual Hawaii International Conference on System Sciences*, Maui, HI, January 2000. to be published.

[St.98] Simon St.Laurent. *XML a primer*. MIS:Press - IDG Books, Foster City - CANADA, 1998.

[TDJ99] Yves Le Traon, Daniel Deveaux, and Jean-Marc Jézéquel. Self-testable components: from pragmatic tests to a design-for-testability methodology. In *Proc. of TOOLS-Europe'99*. TOOLS, June 1999.

[TEI99] TEI. Tei extended pointer notation. 'http://etext.virginia.edu/bin/tei-tocs?dib=DIV2;id=SAXR', June 1999.

[W3C98] W3C. Extensible markup language (xml) 1.0. http://www.w3.org/TR/1998/REC-xml-19980210, February 1998. W3C Recommendation.

[W3C99a] W3C. Document object model (dom). http://www.w3.org/TR/PR-DOM-Level-1, June 1999. W3C Recommendation.

[W3C99b] W3C. Extensible style language (xsl). http://www.w3.org/TR/WD-xsl, June 1999. W3C Recommendation.

[W3C99c] W3C. Resource description framework (rdf): Schema specification. http://www.w3.org/TR/1999/REC-rdf-syntax-19990222, March 1999. Proposed recommendation.

[W3C99d] W3C. Xml linking specification (xlink). http://www.w3.org/TR/WD-xlink, June 1999. W3C Recommendation.

[WM99] Norman Walsh and Leonard Muellner. *DocBook: The Definitive Guide*. O'Reilly & Associates, Inc., 1999.

A `OOPML DTD` a quick tour

To present the `OOPML` structure in a concise way, we have chosen to use screen shots of an `XMLviewer` window. This choice does not provide an exhaustive view of a `DTD` but instead facilitates overall comprehension.

As displayed in the first tree (left figure), an `OOPML` document contains a general *project document* structure with a meta-data element (`MetaData`) which comprises the usual identification elements (title, author, summary, history, legal notes, ...) and a processing directives element (`TrtCommands`) that we will return to later.

The class description itself is contained within the two elements `ClsPackage` and `Class`. Note that it is not possible to define the `package`, container of classes, within a class structure. After an identification field, the class is organized in chapters:

- general textual documentation of the interface (`ClsGenDoc`);
- class *signature* (`ClsSign`);
- textual documentation of the implementation (`ImplDesc`);
- class *links* , inheritance and clients (`ClsLinks`);
- several `Features` chapters, which group primitives (variables and methods);
- finally, a `Test` chapter.

Links (e.g., `XLinks`) may be used within textual documentation zones and the `ClsLinks` element to point to the final design documents or to other classes, thereby ensuring information unicity and the ascending traceability. This envisaged functionality was not formalized nor tested yet because we do not have an `XLink` processor.

The `ClsSign` element leads us to a more technical description of the class: a `Qualifier` element (non visible here) makes it possible to manage the `Java` qualifiers such as `static` or `final`. It is expected that this class model can simultaneously manage several programming languages (see below): consequently, the `PLangList` element manages the declaration of the supported languages.

Fig. 8. `OOPML` snapshot

The implementation documentation element (`ImplDesc`) holds a description of the general philosophy behind an implementation; it has a `priv` visibility by default.

The `ClsLinks` element contains all external relations involving the class, including clients and class (and interface) inheritance. Each block is optional; however, it must have content if it is defined. These three blocks use `XML` links which provide the fundamental navigation mechanism in the technical documentation of classes.

In the next zone (see the middle tree), the class primitives (i.e., methods, variables, constants) are contained in `Features` elements (which were inspired by the `Eiffel` language). These sections enable the logical grouping of primitives based on their role, their visibility or the nature of the methods. Methods and variables can be combined arbitrarily. Some attributes have a particular role:

`category`	in `Feature` makes it possible to have several views of the interface
`visibility`	idem as above (`pub`, `pack`, `child`, `priv`)
`family`	method *family* (`constructor`, `modifier`, `accessor` or `services`)

The variable and method elements are similarly structured with an identity, a description, a signature and contracts. The code consists of a `BlocCode`(s) file, each one comprising a comment block and a code zone. For example, in `Java`:

```
/*
 *  this is the comment block for the code below
 */
has = 5;
```

In a code block, the `plang` *attribute defines the programming language used for the code. If it is not defined, it takes the default value defined in the mandatory* `default` *parameter of the* `PLangList` *element.*
This strategy allows implementations expressed in different programming languages to coexist within the same class document.

The `Features` element cardinality is unconstrained: it is also possible to express the *multiple interface views* concept (once again, as in the `Eiffel` language) and/or to group the functionalities according to logical design criteria (e.g., the components of a *design pattern*).

The `Test` block which appears in the right hand tree is optional since it is only pertinent to self-testable classes. It comprises two sections: the first is mandatory (`TestUnits`), the second (`TestLaunch`) is not, because abstract classes and interfaces cannot have launch tests.

A.1 General comments

This proposal is a feasibility prototype, not a final standard. Indeed, the initial version of our `OOPML DTD` had a strong bias towards the use of element attributes rather elements. However, after revising the `DTD` production rules stated by A. Michard ("XML Language et Applications" p45 to 52), with a particular attention paid to rule 4 on semantic marking and to rule 5 on the correct use of attributes, we believe that the current version of the `DTD` exhibits an improved balance in this area. The basic rule that we currently follow is:

To reserve attributes for information normally **hidden** when using text data in a normal fashion.

A.2 Naming and attributes

We chose to name the elements *" à la Java"* (i.e., each word in a string is capitalized, without separators) and to name the attributes in lowercase letters.

Elements which have semantic equivalents in the `DocBook-V3.1 DTD` were named identically, whereas every other element name was chosen to avoid clashing with a name defined in the `DocBook` standard.

Some attributes are generic and can be explained immediately:

`name(ID)`	name of the element, declared like *`XML` identifier*, the element thus marked can be a target for a link.
`id(ID)?`	optional identifier to define whether this element should be a target for a link
`role`	element role = short description (less than one line)
`visibility`	marks the *structural visibility* of an element, can be `pub` (public), `pack` (visible of the package), `child` (visible to the children = `protected`), `priv` (private)
`plang`	indicates the programming language target, mandatory in `Class`, optional in `Code` where it is implied by the value defined in class.
`edstat`	edition status, can be *empty* (stable), `new` (recently created), `modif` (recently modified) or `obs` (obsolete)

A.3 Some feature choices

The `TrtCommands` element contains `Exec` elements that define commands which can be applied to the class to carry out various development operations. The options of `Exec` (in particular `require` and `target`) make it possible to document the dependencies between classes:

Eventually, a specialized `XML` processor will be able to automatically invoke or run these commands in the manner of `make`, but by using information distributed within the various classes themselves.

Moreover, it is expected that the client and inheritance relationships will be established with the help of extended `XLink` links. This provision should make it possible to create sophisticated hyperlink-based navigation systems.

The inheritance description elements, in association with a document inheritance engine should allow the constitution of *flat* documents that describe all of the primitives (local and inherited) of a class.